Neuro-Fuzzy Techniques for Intelligent Control

A dissertation presented to

The School of Electrical and Electronic Engineering

North-West University

In partial fulfilment of the requirements for the degree

Magister Ingeneriae

in Electrical and Electronic Engineering

by

Morné Neser

Supervisor: Prof. G. van Schoor

February 2006

Potchefstroom Campus

Neuro-Fuzzy Techniques for Intelligent Control

Abstract – In this study the utilization of neuro-fuzzy techniques is investigated for the realisation of intelligent control. Neuro-fuzzy systems combine the learning capabilities of neural networks with the rule based system description offered by fuzzy logic.

Techniques are evaluated by means of a simple two-dimensional simulation of a three-segment robotic manipulator. The inverse kinematics and path planning required in such a system provides all the complexity needed for testing and evaluation.

In complex systems, obtaining sufficient training examples also prove to be a problem. To address this problem an automated process of 'action evolution' was implemented, through which a genetic algorithm is used to collect examples as training data.

A generic modular controller architecture is developed in order to simplify the comparison of different neural and neuro-fuzzy controllers. This architecture unifies numerous controller architectures into a single controller, capable of representing and combining classical, adaptive, intelligent and reinforcement learning controllers and it exposes the presence of various cognitive attributes.

Neural and neuro-fuzzy systems are implemented and evaluated for local trajectory tracking and for global path planning. A serious problem of contradicting solutions encountered in the examples produced by the genetic algorithm, is solved through reinforcement learning. A modified fuzzy clustering algorithm is used to estimate the system's state values and control commands are derived by optimising this value. Modifications included negative reinforcement of prohibited states and concepts borrowed from ant algorithms for establishing solution paths.

This study highlights the effect of ill-posed problems on the training of intelligent controllers. It shows how the problem can be simplified by basing the control policy on a value function and it implements neuro-fuzzy techniques to rapidly construct such a function. Proposals are made on how memory based search algorithms can be used to improve training data integrity and how evolving self-organising maps might prevent erroneous policy interpolation.

This study contributes valuable conclusions on the implementation of intelligent controllers for the control of complex non-linear systems.

Neuro-Wasige Tegnieke vir Intelligente Beheer

Opsomming – In hierdie studie word die gebruik van neuro-wasige tegnieke vir die realisering van intelligente beheer ondersoek. Neuro-wasige stelsels kombineer die leervermoë van neurale netwerke met die reëlgebaseerde stelselbeskrywing van wasige logika.

Tegnieke word geëvalueer deur middel van 'n eenvoudige twee-dimensionele simulasie van 'n drie-segment robotarm. Die inverse kinematika en roetebeplanningsvermoë wat verlang word van so 'n stelsel voorsien al die kompleksiteit benodig vir evaluering.

In komplekse stelsels is die verkryging van voldoende leerdata egter 'n probleem. Om hierdie probleem aan te spreek word 'n outomatiese proses van 'aksie evolusie' geïmplementeer om leerdata deur middel van 'n genetiese algoritme te versamel.

'n Generiese modulêre beheerderargitektuur is ontwikkel om die vergelyking van neurale en neuro-wasige beheerders te vereenvoudig. Hierdie argitektuur verenig verskeie beheerderargitekture in 'n enkele beheerder in staat om klassieke, aanpasbare, intelligente en insentiefleer beheerders voor te stel en te kombineer. Verder word die teenwoordigheid van verskeie kognitiewe eienskappe deur die argitektuur uitgelig.

Neurale en neuro-wasige stelsels word geïmplimenteer en geëvalueer vir plaaslike trajekvolging en vir globale roetebeplanning. 'n Ernstige probleem is ervaar met teenstrydige oplossings in die voorbeelde wat die genetiese algoritme genereer. Hierdie probleem is opgelos met behulp van insentiefleer. 'n Aangepaste wasige groeperingsalgoritme word gebruik om die stelsel se toestandswaardes te skat en bevele word afgelei deur hierdie waarde te optimeer. Modifikasies sluit penalisasie vir ongeldige toestande en konsepte uit mieralgoritmes vir die vestiging van oplossingsroetes in.

Hierdie studie beklemtoon die effek van swakgedefinieerde probleme op die opleiding van intelligente beheerders. Dit illustreer hoe die probleem vereenvoudig kan word deur die beheerstrategie op die waardefunksie te baseer en hoe om neuro-wasige tegnieke te implementeer om vinnig so 'n funksie te skep. Voorstelle word gemaak oor hoe geheue-gebaseerde soekalgoritmes aangewend kan word om die integriteit van leerdata te verbeter en hoe evolutionêre self-organiserende netwerke foutiewe strategie-interpolasie kan voorkom.

Die studie maak waardevolle gevolgtrekkings oor die implementering van intelligente beheerders vir die beheer van komplekse nie-lineêre stelsels.

"*This was the biggest breakthrough of all. Vast wodges of complex computer code governing robot behaviour in all possible contingencies could be replaced very simply. All that robots needed was the capacity to be either bored or happy, and a few conditions that needed to be satisfied in order to bring those states about. They would then work the rest out for themselves.*" – **Douglas Adams, Mostly harmless**

Acknowledgements

Thanks to Prof. George for guidance and support more than he realises.

Thanks to my friends and family for their encouragement and contributions.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AI	artificial intelligence
AIS	artificial immune system
AL	artificial life (A-life)
ANFIS	adaptive neuro-fuzzy inference system
DP	dynamic programming
EA	evolutionary algorithm
EC	evolutionary computing
EFC	evolutionary fuzzy clustering
ESN	echo state network
FFN	fuzzy feedforward network
FIS	fuzzy inference system
FL	fuzzy logic
FS	fuzzy system
GA	genetic algorithm
GAFRL	GA fuzzy reinforcement learning
GD	gradient descent
GFS	genetic fuzzy systems
IA	intelligent agent
LMS	least mean square
ML	machine learning
NNC	nearest neighbourhood clustering
NF	neuro-fuzzy
NN	neural network
RBF	radial basis function
RL	reinforcement learning
SOM	self-organising map
SVM	support vector machine
TD	temporal difference
TDGAR	TD GA reinforcement

Chapter 1

Introduction

In this study the efficiency of neuro-fuzzy techniques are evaluated for intelligent control. After an overview of computational intelligence, different neural and neuro-fuzzy controllers are tested on a simulated robotic system. Finally suggestions are made on how performance can be improved for intelligent control of complex systems.

1.1 Background

The full automation of nonlinear processes requires intelligent control. Fuzzy systems and neural networks have unquestionably demonstrated their efficiency in the modelling and control of such nonlinear processes. These mechanisms are function approximators which are set up to produce the desired input-output mappings, but they both have different capabilities and shortcomings.

Fuzzy systems are very convenient for describing and modelling nonlinear systems in an intuitive rule based fashion. It allows the user to construct nonlinear systems in terms of sets of fuzzy rules. Simple rules can be set up following common sense reasoning or when the systems become more complex, rules can be obtained from experts. In the absence of experts to provide the logic, the modelling of complex systems can become problematic.

Neural networks are powerful universal nonlinear function approximators based on the physiology of the biological brain. Algorithms have been derived to train such a network for desired behaviour from a set of examples. This is very convenient when models and controllers need to be developed of which the policy is unclear. Unfortunately typical neural networks suffer from the 'black box' effect where the inner dynamics of a trained network can become very complex leaving users unable to verify the logic on which the system's behaviour is based.

These two mechanisms are at opposite sides of the spectrum of the interpretability of operating policies. Conventional fuzzy systems lack the ability to be trained from examples while neural networks lack simplicity.

Neuro-fuzzy systems are hybrids, usually designed to combine the training capabilities of neural networks with the rule based reasoning of fuzzy systems. Various architectures have been developed to construct such adaptive rule based systems.

Mobile robots are often required to operate autonomously in highly nonlinear environments, which bear the need for intelligent control. In this study a simulated robotic manipulator is used to evaluate the efficiency of neuro-fuzzy techniques for intelligent control. For effective training of the controller large amounts of good examples of desired behaviour is required as training data. In complex systems such information is often not freely available. Therefore, to assist in the training of the controller, the acquisition of training data is automated through guided exploration of the environment.

The objective of this study is to put forward techniques that can be applied to achieve rule based control in complex systems.

1.2 Problem statement

The driving force of automation is continuously increasing the demands put on control systems. Ever more complex nonlinear systems are expected to be controlled autonomously. The desired control policies of such systems are frequently not known and have to be derived empirically from examples or policies have to be derived through experience to adapt to changing systems.

In addition, in many cases it is required that the applied policies are accessible for examination and verification. Although the use of neuro-fuzzy systems seems to be an appropriate method to follow to satisfy these requirements, the performances achieved are often unsatisfactory and the rules that are produced are incoherent. In complex systems, finding adequate quality training data might be part of the problem.

Further research is required in the use of neuro-fuzzy systems for creating adaptive rule based policies for intelligent control. Techniques and architectures need to be evaluated in order to further enhance the capabilities of automated system. A target application is required for evaluation of the controllers. A generic system framework has to be constructed through which different controllers can be implemented and compared.

1.3 Issues to be addressed

Before any controller architecture or optimisation techniques can be evaluated, an evaluation platform should be developed and a convenient test environment should be created. Furthermore sufficient training data should be acquired. These issues are discussed here.

1.3.1 Evaluation platform

A simulated system is required for the evaluation of the neuro-fuzzy controllers. A robotic manipulator is chosen for its simple graphic representation, scalable level of complexity and unlimited degrees of freedom. The robot should interact with the environment by means of sensors and actuators. The robot's goal will be to navigate to a target co-ordinate. The environment will contain obstacles around which the robot will have to negotiate its trajectory. This will form the platform for evaluation of control techniques and system architectures.

1.3.2 Software design standards

By following design standards in the development of software, the integration time and the reuse of software can be optimised. The design of the software will depend on the choice of the development environment. The environment should be chosen to allow the following:

- rapid prototyping
- sufficient processing power and speed
- code reusability and inheritance
- importing of 3rd party tools
- portability
- graphic interfacing

By making use of 3rd party tools, development and prototyping time can be significantly reduced.

1.3.3 System architecture

Multiple different controller configurations will have to be evaluated. By defining generic interfaces in such a system, the substitution of controllers in different parts of the system can be simplified to streamline the process of comparison of techniques. A general framework and templates for the interfacing of different parts of the system is therefore required.

1.3.4 Environment exploration

For an adaptive control system to be entirely autonomous it has to be able to explore its environment automatically to find optimal control policies. Especially in a virtual environment or where a model is available, the environment should be thoroughly explored to collect a good representation of the state space. Optimisation techniques will be required to find optimal solutions to all the states encountered. This acquired data can be used for training the control systems.

1.3.5 Controller implementation

Various neural and neuro-fuzzy systems have to be developed. They should be implemented in different controller architectures. They will be trained with data acquired through exploration and tested to control the robotic manipulator under different conditions and starting positions. The different controller architectures will be evaluated in terms of performance.

1.4 Research methodology

The following active research steps are taken to address the issues listed in section 1.3 in order to solve the problem identified in this study.

1.4.1 Evaluation platform

A multi-segment robotic manipulator simulation is developed. The manipulator is centred in a 2D grid-world as depicted in Figure 1.1. An open software interface allows different controller modules to control the robotic manipulator by giving joint angle commands to modify the joint angles in either direction. The angular commands for all joints are scaled to be executed over an equal amount of steps - higher command values resulting in faster relative movement. The goal is for the controller to supply angular commands to steer the manipulator to a particular target position.

Obstacles and walls can be set up in the environment. The manipulator detects contact with obstacles on the grid and is prevented from passing through them. Joint angle or joint position feedback as well as the distance to the target are reported back to the controller. Additionally, the manipulator can be set up to a specific joint angle state and the target position can be set. The obstacles can be set up through the graphical user interface or loaded from file. The software interface is given in Table 1.1. 3D game programming techniques are implemented for collision detection.

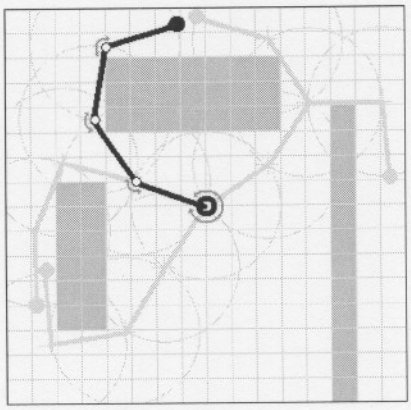

Figure 1.1 *Virtual environment*

The simulated multi-segment robotic manipulator negotiates obstacles to reach a target in an grid-world environment

Table 1.1 *Manipulator software interface*

Direction	Property
input	3 joint angles (manipulator position)
input	3 angular commands
input	goal x-y coordinate
output	3 x-y coordinates of 2 movable joints and tip
output	3 joint angles
output	reward (combination of Manhattan distance to target and distance travelled)

1.4.2 Software design standard

A library of software modules are developed in Java™. Java™ has become a recommended prototyping and demonstration language for engineering software over alternatives like MATLAB® and Visual C++ for the following qualities:

- object-oriented
- light weight stand alone capability
- platform independence
- simple GUI & event model
- seamless integration with MATLAB®

These modules can be used for compiling different control systems as stand alone applications or through MATLAB®. MATLAB® is used for its toolboxes, ability of rapid prototyping of architectures and its graph plotting functionality.

1.4.3 System architecture

A compound intelligent control system architecture is constructed as shown in Figure 1.2. Different controllers, function approximators and optimiser modules are developed and implemented as functional blocks of this architecture. Elaborate modular intelligent agents can be constructed through the use of adaptive functional blocks.

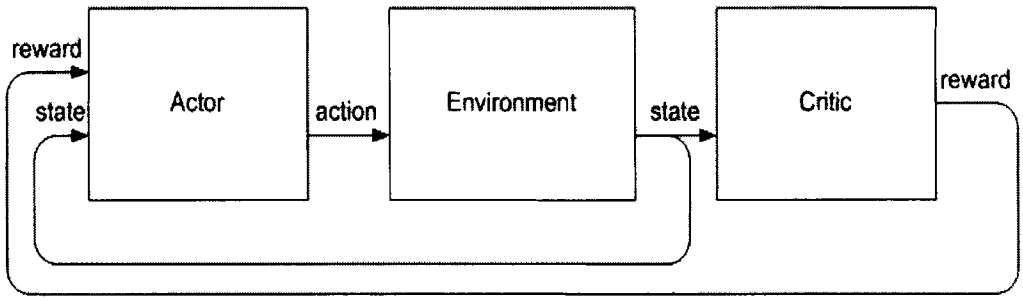

Figure 1.2 *Modular Control System*
The controller is based on the actor-critic architecture for reinforcement learning

Modules inherit interfaces from templates, allowing them to be slotted into a framework of the architecture. This makes them seamlessly interchangeable, which simplify the substitution of controllers for comparison of controllers and techniques.

1.4.4 Environment exploration

A genetic algorithm (GA) is implemented for optimising the joint angle commands to be given to the manipulator. It searches an offline copy or model of the environment. The best single action or the best sequence of actions can be determined. It uses the distance the manipulator tip is from the target as a fitness function. When a solution or set of solutions are found it is reported to the controller

1.4.5 Controller implementation

Various neuro-fuzzy controllers, control techniques and architectures are investigated. They are trained and tested for the control of the robotic manipulator under different conditions.

Some of the primary concerns with the control of autonomous mobile robots are that of path finding and obstacle avoidance. It should be possible to achieve dynamic obstacle avoidance or close target approach with a static, reflex-type control policy. A local controller architecture is implemented to investigate the feasibility of such a control scheme for a robotic manipulator.

A global controller architecture is implemented for controlling the movement of the manipulator from the starting state to the target along a path, circumventing static obstacles in the environment.

1.5 Project evaluation

The implemented architectures and techniques will be compared according to the controller sizes, the training errors achieved, the distances from the target that were achieved and the number of unsuccessful trials.

These implementations are also compared to other approaches to the multi-segment manipulator control problem.

1.6 Dissertation overview

A background study of the fields of computational intelligence and neuro-fuzzy hybrids are presented in Chapter 2. In Chapter 3 an account is given of the most popular approaches that are currently followed in robotic manipulator control. Chapter 4 establishes the requirements needed for an evaluation platform for the research of behaviour of intelligent controllers. It also presents the selected platform and its operating interface. In Chapter 5 the object oriented software design philosophy which will be followed in the software development in the subsequent chapters is discussed. Chapter 6 puts forward a modular system architecture for intelligent agents while Chapter 7 explains the automated process of collecting training data. With the simulation, training data and a system framework in place, different controller configurations are implemented for consideration in Chapter 8. Final conclusions and suggestions are made in Chapter 9. This is followed by the appendices consisting of a list of the software modules developed and a draft copy of a paper presented at an IEEE symposium.

Chapter 2

Computational Intelligence

This chapter study presents a background study on the field of computational intelligence. It focuses on neuro-fuzzy systems and fuzzy clustering techniques for their application to intelligent control.

2.1 Research domain

'Intelligent control' implies the use of artificial intelligence techniques for the realisation of adaptive nonlinear control [1]. Artificial intelligence is located at the intersection of studies in several disciplines. Different approaches originated from the fields of mathematics, computer science, psychology, biology and mechanics. This intersection of disciplines is illustrated in Figure 2.1 and briefly discussed in the following paragraphs.

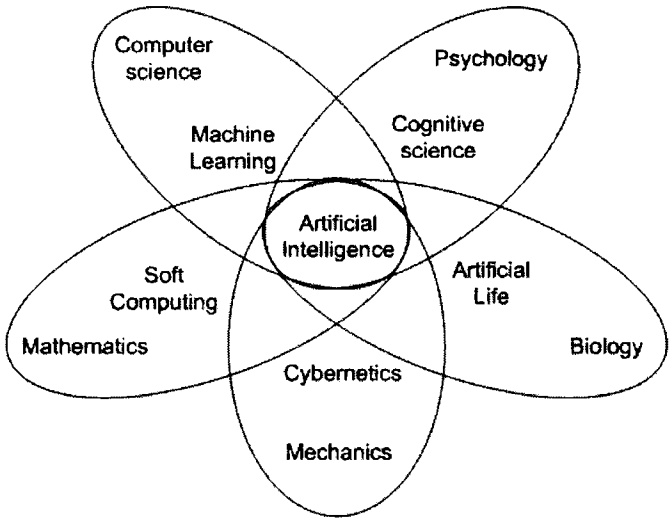

Figure 2.1 *Intersection of disciplines*
Artificial intelligence lies at the intersection of soft computing, machine learning, cognitive science, artificial life and cybernetics.

2.1.1 Artificial Intelligence

Artificial intelligence (AI) defines a subset of computer science which encapsulates all concepts and methods which involve automation and intelligent behaviour such as reasoning and learning [2]. In its early history AI was considered as the ability to simulate human behaviour. In 1950 the Turing-test was devised through which such abilities could be demonstrated [3]. A human judge engages in a text based conversation with a human and a machine. The machine passes the test if the judge cannot tell which is human.

Since then the definition of AI has changed and linguistic capabilities are no longer considered a prerequisite for the label of intelligent behaviour. Instead, the ability to adapt and improve as information becomes available is sought after. This extends the application of AI to the general problem of pattern recognition. AI techniques have over recent years become very effective in solving complex, nonlinear problems of classification, modelling and control.

AI research was very heavily funded by US government in the 1980's but failure to produce immediate results led to large cutbacks in funding and a so-called AI winter. However, during the Gulf War the scheduling of deployment of US forces were calculated with the use of AI methods, which resulted in savings more than the entire investment made in AI research over the preceding 30 years.

The classification of different branches and techniques in AI are the source of some controversy and is fuzzy at best. The classical categorisation is diagrammed in Figure 2.2. Historically AI is divided into what is now called conventional AI and the newer computational intelligence (CI). Conventional AI mainly involves reasoning systems and logic search methods such as case based reasoning, predicate logic and expert systems. Computational intelligence boasts adaptability through parameterised learning in systems which encompass neural networks (NN), fuzzy systems (FS) and evolutionary computing (EC). More recently many hybrid systems have been developed, combining the different branches of CI and even combining them with those of conventional AI.

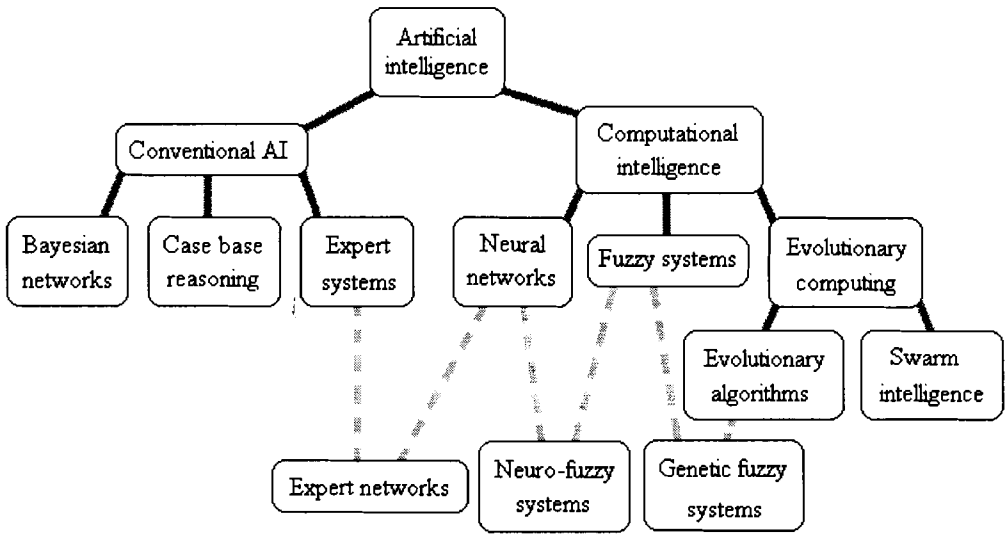

Figure 2.2 *Categorisation of AI*

The branches of AI can be categorised under the headings of conventional AI and computational intelligence. Combinations of techniques have lead to various hybrid systems.

2.1.1.1 Strong AI vs. Weak AI

Strong AI is a philosophy which holds that there is no reason why human level intelligence cannot be achieved by computational systems. Alternatively, supporters of weak AI claim that true intelligence and consciousness require non-computable non-algorithmic processes [3]. Resent research has shown than the dynamics of microtubules in the organic brain rely on quantum processes. It is proposed that such processes might be essential for realizing true consciousness [4].

In response to this, in an ongoing debate, strong AI devotees argue that with the realization of quantum computers AI will also be able to tap into these quantum processes. This holds a great promise for the field of AI. It has already been shown that search techniques could be drastically accelerated with the use of quantum computers and quantum search algorithms.

Strong AI states that intelligence is an attribute which is independent of medium or hardware [5]. Its supporters criticize those of weak AI of constantly moving the goalposts and effectively defining intelligence as 'whatever humans can do that machines cannot'. Nevertheless, these perspectives are both aimed at improving intelligent behaviour in machines and their differences are only of philosophical matter.

2.1.1.2 Neat AI vs. Scruffy AI

Another matter in AI of greater importance is one regarding the approaches towards its goal. There are two methodological schools of thought: the 'neats' and the 'scruffies'. Neats emphasize the formal establishment of theory and logical representation of knowledge while scruffies apply ad-hoc methods driven by empirical knowledge about the problem.

Conventional or neat AI (also called classical, logical or symbolic AI) attempts to mimic human intelligence through symbolic manipulation of abstract concepts. It usually attempts to build logical systems resembling human reasoning and behaviour. Predicate interpreters like PROLOG, expert systems and case based reasoning systems are typical examples. Reasoning is based of inference of a type of database and learning often involves extension of the databases. The methods used to train these systems are collectively referred to as machine learning (ML)

Modern or scruffy AI generally builds connectionist systems using soft computing techniques with parameter optimisation. It involves automated and often incremental parameter tuning as opposed to systematic design.

A lot of the terminology used overlap to a great extent and can be the cause of some confusion. In Table 2.1 terms roughly describing the same idea are grouped together. It is noticeable that computational intelligence plays a predominant role in scruffy AI.

Table 2.1 *Schools of thought with opposing terminology*

conventional AI	computational intelligence
neat AI	scruffy AI
machine learning	incremental learning
symbolic AI	sub-symbolic AI
hard computing	soft computing
expert systems, case based systems	connectionist systems

2.1.2 Soft Computing

A whole range of mathematical problem solving methodologies is grouped together under the term 'soft computing' [6]. These methodologies are roughly based on models of human thought and natural emergence and are studied and used by many different disciplines like computer science, neuroscience, psychology, biology, mathematics and philosophy. The most notable of these methods are:

- neurocomputing or neural networks
- support vector machines
- fuzzy logic
- probabilistic reasoning
    - fractal and chaos theory
    - evolutionary algorithms
    - belief networks
    - artificial life

Many hybrids of these methodologies have also been developed. Techniques are combined in various configurations as indicated in Figure 2.2, mainly to improve learning speed. These

include not only soft computing techniques such as neuro-fuzzy systems, fuzzy neural networks, genetic fuzzy systems and evolving neural networks, but also neat-scruffy hybrids like expert networks and fuzzy expert systems.

2.1.3 Cybernetics

Cybernetics is defined as "the study of communication and control in animals, humans and machines" [7]. There is no prioritization of artificial over natural systems and unlike in AI, much less concern about modelling the one on the other. In practice cybernetics involve the investigation of machine behaviour against the reference point of human and animal behaviour.

While AI tends to follow a theoretical top-down approach in the modelling of learning, cognition, reasoning and planning, cybernetics follows a pragmatic bottom-up approach, attempting to achieve advanced machine control through an emergent process of machination of mind.

2.1.4 Cognitive Science

The science of cognition, knowledge acquisition, representation and utilization has emerged in an overlap of the fields of psychology, neuroscience and computer science.

Cognitive science is currently faced with two conceptions labelled as the broad and the narrow conception [8]. The broad conception describes cognitive science as a domain of investigation with the goal to understand the principles of intelligence and cognitive behaviour. The narrow conception describes cognitive science as a doctrine of representational and computational capacities of the mind and brain.

2.1.5 Artificial Life

Although artificial life is classically independent of AI, these two subjects are now closely related and it is therefore worth mentioning. A-life, as it is also called, developed in the theoretical research of biology [9]. Especially the development of cellular automata has shown the emergence of complexity as Figure 2.3 depicts. It is a field which is now strongly correlated with the work in evolutionary computing and swarm intelligence.

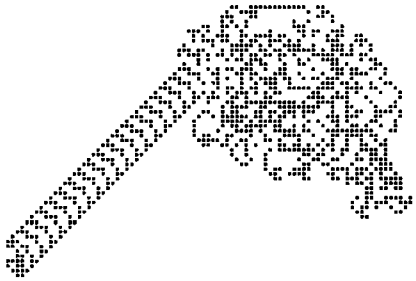

Figure 2.3 *Langton's ant*

Following a few simple rules of moving and changing cell states, starting with an empty grid, after about 10 000 iterations of chaos, a pattern emerges.

2.2 Machine Learning

Machine learning involves the adaptation of classifiers and control systems. In classifiers the system predicts the class to which an input belongs. In controllers recommended actions are inferred by the system.

Learning can be divided into three types, distinguished by the source of the data used [10]. They are supervised learning, unsupervised learning and reinforcement learning as Figure 2.4 indicates. Although the term 'machine learning' has lately become strongly associated with classical AI, the classification of applications and learning types applies to the whole field of AI.

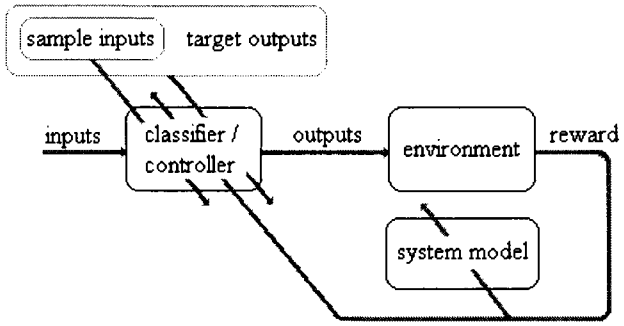

Figure 2.4 *Source of learning data*

Supervised learning uses input-output data pairs for training of the controller. Unsupervised learning uses only sample inputs for classification. Reinforcement learning can be used to train the control policy or a system model, based on a reward from the controlled system.

2.2.1 Supervised learning

When the target values are available for given inputs, it can be used as training data for supervised learning of an adaptive system. The most commonly used supervised learning method

for non-symbolic systems is the backpropagation algorithm. This algorithm adjusts the system parameters, based on the output errors, as to minimize this error. This topic will be revisited under neural networks in section 2.3.1.2.

Other computational methods such as combinatorial optimisation and self-organising can also be used for minimizing the system error. These methods will be discussed in section 2.3 on computational intelligence. In classical AI based systems such as expert systems output is based on cases in a sample database. Learning consists of incorporating training data as exemplars into this database.

2.2.2 Unsupervised learning

A great amount of pre-processing can be performed on data without having the desired output available. This is called unsupervised learning. One approach in unsupervised learning uses cluster analysis, in which input data is grouped based on the Euclidean distance the data is apart. An extension of this approach is to allow a sample to be a member of different clusters to different extents, instead of assigning it to one specific cluster. This is known as fuzzy clustering and forms an essential part of neuro-fuzzy computation. It will be further discussed in section 2.4 on hybrid intelligent systems.

Another approach is through probabilistic models. In this approach statistical analysis is used for classification. One of the simplest probabilistic models is the ''mixture model' in which the probability is calculated for data having been generated by a mixture of Gaussian distributions [11]. Learning is done by adjusting the parameters of the model to maximize the likelihood that data was generated by a specific source.

Unsupervised learning can be used to identify rules or reduce the amount of exemplars in a sample database or the dimensionality of inputs. It can also be used for data compression or for self-organizing. Self-organising maps are discussed under neural networks in paragraph 2.3.1.4.

2.2.3 Reinforcement learning

In many control problems there is no expert knowledge of the desired outputs available. Therefore there is no training data available for supervised learning. What is available instead is a reward signal. The reward is a measure of the controller's success or failure. When learning is based on such a reward feedback signal as in Figure 2.5, it is called reinforcement learning (RL). When learning takes place autonomously, the system can be called an intelligent agent (IA) [12].

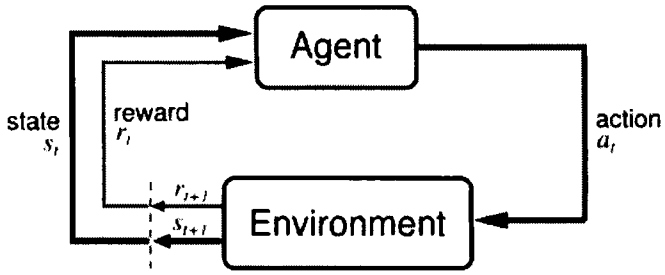

Figure 2.5 *Reward feedback*

The environment returns new states as well as associated reward values [13].

The agent explores its environment to gather information for RL. After taking an action, the agent receives feedback from the environment including an immediate reward. Often the system needs to pass through a sequence of neutral states before the desired goal state can be reached. To overcome this problem of no rewards for intermediate states, the agents implement a scheme for estimating future rewards. The agent uses experience about system states and rewards received, to construct an internal value function which estimates future rewards.

Agent actions are based on internal control policies. The aim is to derive an optimal policy that maximizes the sum of rewards over time. The three primary methods for creating value functions for action reinforcement are dynamic programming (DP), Monte Carlo methods and temporal-difference (TD) learning [13].

2.2.3.1 Dynamic Programming

DP refers to the collection of algorithms to compute optimal policies given that an accurate differentiable model of the environment is available. This requirement and the fact that these algorithms are computationally intensive limit the use of DP in reinforcement learning.

2.2.3.2 Monte Carlo

Unlike DP the Monte Carlo method bases its calculations on experience data. State-action-reward sets acquired through interaction with the system or a model can be used. Only a simple model generating state transitions is required. However, this method only updates the policy after a complete sequence of actions has been taken and a reward has been issued.

2.2.3.3 Temporal difference

The TD(λ) variation of this method is very popular for it seamlessly integrates and optimises the previous two methods. Although it is the most complex method, it requires no model and is

fully incremental. Because of its convenience TD learning has gained much popularity. This method can be implemented with an adaptive or fixed model or with no model at all.

Through an iterative process the temporal difference between the expected value for the current state and the previous state is reduced. Gradually the reward issued is propagated back in time in training sequences to states leading up to successes of failures as illustrated in Figure 2.6.

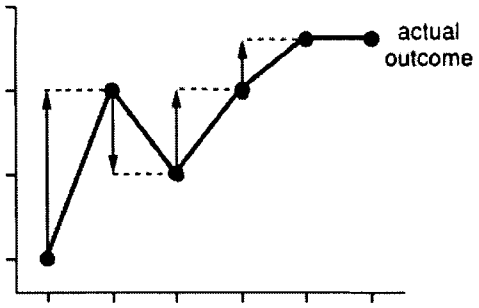

Figure 2.6 *TD learning*

After every time the predicted reward for the previous state is adapted to estimate the current predicted reward [13].

2.3 Computational Intelligence

Computational intelligence can be described as the branch of artificial intelligence based on soft computing techniques. Unlike in conventional AI these techniques generally disregard statistical analysis and systematic design. Instead, it makes use of gradual training algorithms for parameter tuning based on empirical data. Systems commonly form connectionist architectures and learning is usually an iterative and automated process.

The techniques used in computational intelligence are

- neural network, which tacitly ignores statistical analysis,
- fuzzy logic, which explicitly rejects statistical methods, and
- evolutionary computation, which implements meta-heuristic methods.

All of these will be discussed here.

2.3.1 Neural networks

The connectionist philosophy is most notable in the architecture of neural networks. Artificial neural networks were developed to simulate the workings of natural neural networks in an attempt to achieve more complex input-output mapping capabilities. A neural network consists of a multitude of similar, interconnected neurons.

Artificial neurons are generally relatively simple multi-input, single output units. The neuron output is typically a weighted sum of the inputs,

$$y(\underline{x}) = \sum_{i=1}^{n} w_i . x_i \tag{2.1}$$

where $\underline{w} = (w_1, ..., w_n)^T$ is the weight vector, $\underline{x} = (x_1, ..., x_n)^T$ is the input vector and n is the amount of elements in the input vector.

The values by which the inputs are multiplied before summation, are properties of individual neurons. By changing these values the characteristics of the neuron can be adapted so that any linear function can be modelled [11].

After an explanation of the operation of the basic perceptron some different neural network architectures are discussed.

2.3.1.1 Perceptron

A perceptron is a neuron with a threshold value. In such a neuron the output is only activated when the weighted sum of inputs reaches the built-in threshold value. This is accomplished by adding a bias value to the weighted sum of inputs and feeding the sum through an activation function. The output of perceptron k is then calculated as

$$y_k(\underline{x}) = f\left(\left(\sum_{i=1}^{n} w_{ki} . x_i\right) + w_{k0}\right) \tag{2.2}$$

where $\underline{w_k} = (w_{k1}, ..., w_{kn})^T$ is the weight vector, $\underline{x} = (x_1, ..., x_n)^T$ is the input vector, and w_{k0} the bias value for perceptron k. The amount of elements in the input vector is given by n and f is the activation function [14].

This more closely resembles the operation of its organic counterpart. A perceptron can easily be utilised to realise a function for binary classification of data. Such a function is called a discriminant function. Supervised learning algorithms have been derived for tuning the weights and threshold values.

2.3.1.2 Multi-layer perceptron

The shortcoming of the single neuron perceptron is that it can only be used to classify linearly divisible data. However, by linking up multiple neurons in a multi-layer network, nonlinear discriminant functions can be constructed. This is known as the multi-layer perceptron (MLP).

The outputs of an array of neurons in a single layer can be represented through matrix multiplication as

$$\underline{y}(\underline{x}) = \underline{f}\left(\underline{w}^T.\underline{x} + \underline{w}_0\right) \tag{2.3}$$

where $\underline{w}$ is a matrix consisting of a weight vector column for every neuron, $\underline{x} = (x_1,...,x_n)^T$ is the input vector, and $\underline{w}_0 = (w_1,...,w_n)^T$ is the bias vector. f is the activation function for all the neurons in the layer [14].

When two layers are connected with the outputs of one layer connected to the inputs of the next, the output function becomes

$$\underline{y}(\underline{x}) = \underline{f}_2\left(\underline{w}_2{}^T.\underline{f}_1\left(\underline{w}_1{}^T.\underline{x} + \underline{w}_{b1}\right) + \underline{w}_{b2}\right) \tag{2.4}$$

where $\underline{x} = (x_1,...,x_n)^T$ is the input vector, $\underline{w}_k$ is the weight matrix, $\underline{w}_{bk}$ the bias vector and f_k the activation function for all the neurons in layer k [14].

The superior capabilities on a MLP can be easily demonstrated with the XOR classification problem. No single line formed by a single neuron can sufficiently divide the data of the XOR function in a two dimensional binary input space. By connecting three neurons as shown in Figure 2.7, two lines are combined to correctly classify the data as shown in Figure 2.8.

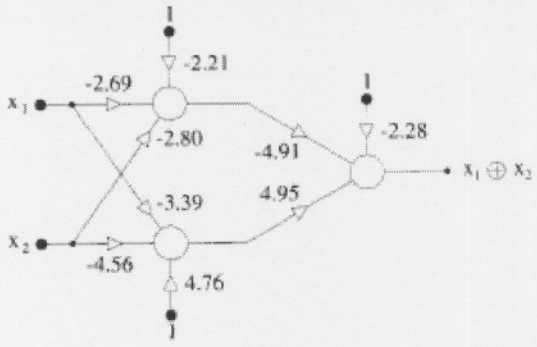

Figure 2.7 *XOR neural network*

If input x1 and x2 is -1, the first neuron is activated. If input x1 or x2 is -1, the second neuron is activated. When the first neuron not active and the second neuron is active, the output neuron is activated, resulting in an exclusive OR logic function [14].

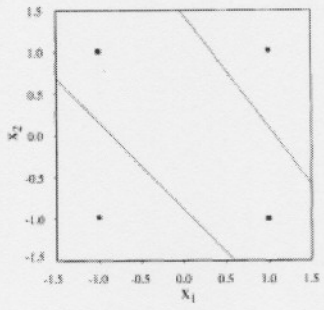

Figure 2.8 *XOR classification*

A multi-layer perceptron is capable of representing a discrete exclusive OR logic function by isolating and combining two true cases with three neurons [14].

Since the XOR problem has binary inputs any function can be represented by a two-layer network, independent of its dimensionality. If the input space becomes real-valued a two-layer network is only capable of classifying convex areas. A three-layered network is required to classify concave areas. This can be explained by noting that a convex area can be described by a set of AND operations (in the second layer) of straight line classifiers (in the first layer), while a concave area requires a further OR operation (in a third layer). Refer to Figure 2.9.

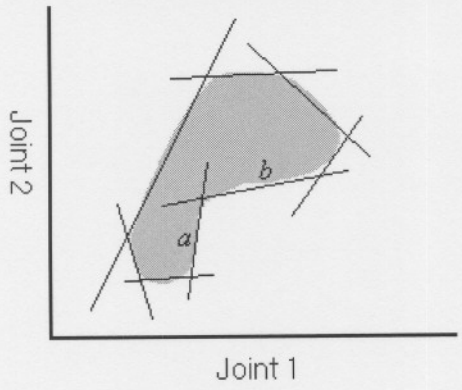

Figure 2.9 *Convex and concave isolation*
Any binary condition can be isolated with one neuron. The convex area in (b) requires a hidden layer. The concave area in (c) requires two hidden layers.

Function approximation can be done by allocating real values to classified areas in the input space. It has been shown that any arbitrary function can be approximated to any accuracy through a network with 3 layers of neurons.

Learning is defined as the process of optimization of network parameters to best fit the training data. The goal of optimization algorithms would be to minimise an error function, which is the difference between the network's predicted outputs for the training data and the target values of the training data. This value is a function of the network parameters.

One of the most popular approaches is the gradient based approaches, of which the gradient descent method is the most common used. In this method the gradient of the error function is determined and the parameters are adapted in small steps as to decrease the error value. This is an incremental process and training can take several iterations, depending on the step size, which is a training parameter.

The MLP implementation of this method is called the backpropagation algorithm. The network error is systematically backpropagated from the outputs through the output layer and the hidden layers towards the inputs. Updating of the parameters can be done in two distinct manners: batch learning and incremental learning. In batch learning, the error is calculated over all training samples and then the parameters are updated all at once. This is significantly faster than incremental training, but it cannot be used for online training, since all the training data is required to be available beforehand.

Since the backpropagation algorithm requires the error derivative, the step threshold activation function is insufficient. This function is replaced by a sigmoidal or ARCTAN activation function shown in Figure 2.10, which also has the effect of smoothing the network output which is better for generalization. The sigmoidal activation function is given by

$$s(x) = \frac{1}{1 + \exp(\alpha - x)} \tag{2.5}$$

where α is the function threshold value [14].

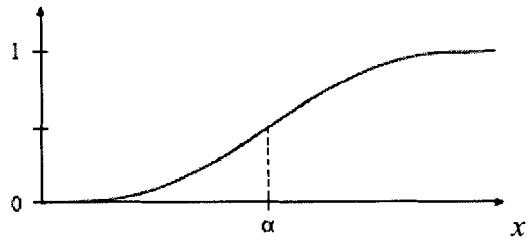

Figure 2.10 *Sigmoid function*

There is a gradual activation of the output for inputs larger than α [15].

2.3.1.3 Radial Basis Function

The radial basis function (RBF) is another connectionist function approximator. This is a two-layer network where the second layer forms a weighted sum of outputs of the first layer, similar to those of the MLP. The first layer however makes use of somewhat different neurons. In the case of the perceptron a weighted sum of the inputs are calculated, whereas for the radial basis neuron the difference between the input and an internal vector is calculated.

Put another way, instead of multiplying the input vector with a weight vector, the Euclidian distance between these vectors are calculated. Furthermore where the perceptron uses a sigmoidal activation function, the radial basis network uses a Gaussian function shown in Figure 1.1. The result is that the neuron is activated only when the input is close to the neuron's internal vector or basis value. The Gaussian activation function is given by

$$g(x) = \exp\left(-\frac{(x-\alpha)^2}{2\sigma^2}\right) \tag{2.6}$$

where α is the function centre and σ is a parameter determining the width or smoothness of the function [11].

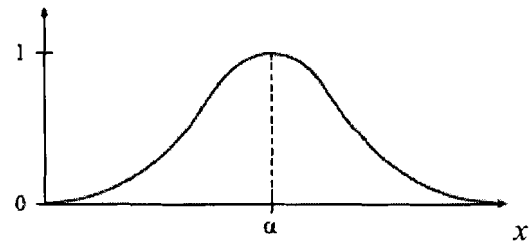

Figure 2.11 *Gauss function*

The output is activated for inputs around α [15].

By appropriately selecting the weight vectors, the input space can be covered in such a manner that the network can approximate the target function simply by tuning the second layer weights. This is shown in Figure 2.12. Different methods can be implemented to find appropriate basis vectors. This may range from normal distributions to multi-pass iterative unsupervised clustering algorithms. These methods will be covered in section 2.4.2. Finally the least mean square (LMS) [14] supervised learning algorithm can be used to derive the second layer weights.

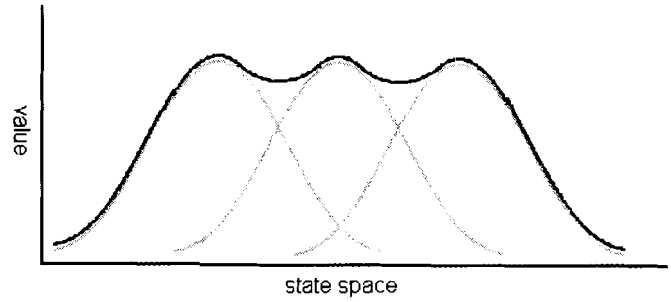

Figure 2.12 *Gaussian coverage*

Approximating arbitrary surfaces by the linear combination of Gaussian functions.

The weight vectors define the hidden layer nodes. The output of node j is the Gaussian function for a multi-dimensional input space, calculated as

$$g_j(\underline{x}) = \exp\left[-\frac{(\underline{x}-\underline{w}_j)^T(\underline{x}-\underline{w}_j)}{2\sigma_j^2}\right] \tag{2.7}$$

where $\underline{w}_k = (w_{k1},\ldots,w_{kn})$ is the weight vector for node j, $\underline{x} = (x_1,\ldots,x_n)$ is the input vector, and σ_j is a normalization parameter calculated as

$$\sigma_j^2 = \frac{1}{M_j} \sum_{x \in \Theta_j} (\underline{x} - \underline{w_j})^T (\underline{x} - \underline{w_j}) \tag{2.8}$$

where Θ_j is the training set, $\underline{w_k}$ is the cluster centre vectors for j nodes and M_j is the amount of training samples in the training set [14]. The network output for neuron k is given by

$$y_k(\underline{x}) = \underline{w_k}^T . \underline{g}(\underline{x}) + w_{k0} \tag{2.9}$$

where $\underline{w_k} = (w_{k1}, ..., w_{kn})^T$ is the weight vector, $\underline{x} = (x_1, ..., x_n)^T$ is the input vector and w_{k0} the bias value and $\underline{g}$ is the Gaussian activation function for the individual elements of $\underline{x}$.

The clustering algorithms as well as the LMS for a single layer are significantly faster than the backpropagation network. In addition, because of the shape of the Gaussian activation function, modifications to weights only have localised effects, meaning that trained RBF networks can be adapted to changed training data without the need for global retraining. The downside of the RBF is that it requires a significant amount of first layer radial basis neurons to cover the input space, which increases exponentially with the dimensionality of the input space.

2.3.1.4 Self-Organizing Maps

The Kohonen self-organising map (SOM) is a type of unsupervised learning network. Similar to the RBF, it implements competitive learning for data clustering. Cluster centres are represented by the weights of the neurons. However, the neurons are arranged into a linked array or grid as can be seen in Figure 2.13, and every time a neuron is updated, so are its closest neighbours [16].

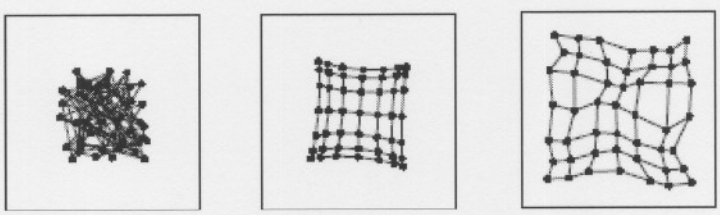

Figure 2.13 *Tuning of a self-organising map*
The initially randomised neurons on the left gradually get organised to cover the complete input space as shown on the right [17].

This linking of the neurons has the effect of preserving possible structure and metric of the input space. The output neurons are topologically ordered in such a way that neighbouring neurons

tend to correlate to similar regions in input space. Furthermore, regions with higher density in input space are mapped to larger areas in output space. These features make the SOM useful for simplification through dimensionality reduction, mapping data from arbitrary high dimension into one or two dimensional space.

2.3.1.5 Recurrent neural network

All the network architectures discussed above (MLP, RBF and SOM) are strictly feedforward networks. Outputs of one layer always connect to inputs of the next layer until it reaches the network outputs. Such networks are not capable of modelling internal system dynamics. Outputs of the network solely depend on the current input values and the previous state of the network has no effect.

The recurrent neural network (RNN) architecture feeds outputs from internal or output neurons back into the network. Feedback can have different time delays as Figure 2.14 suggests. This makes the network dependent on its own internal state and allows the network to build up a memory and have responses based on historic events [16].

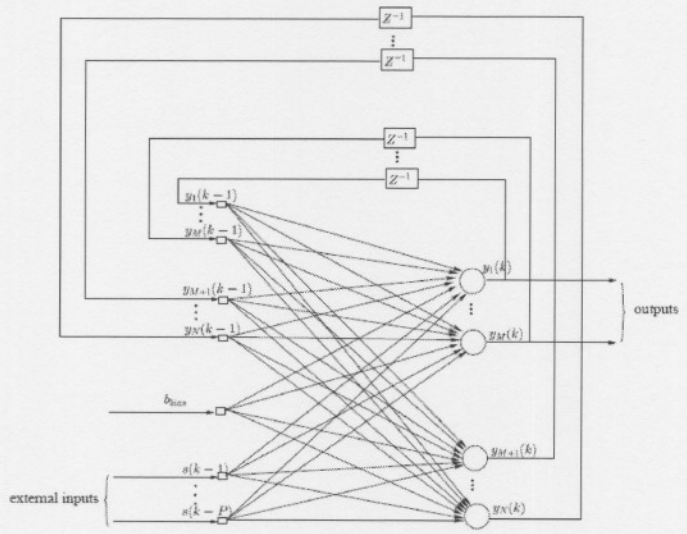

Figure 2.14 *Recurrent neural network*

The recurrent neural network has internal connections linking neuron outputs back to the same of previous neurons [18].

This architecture allows the network to represent system dynamics internally and better model complex processes. Although such networks can be very useful for time series function prediction and process modelling, learning is very difficult.

2.3.1.6 Hopfield network

Hopfield networks are special forms of binary RNNs. They are trained to stabilise at only specific output patterns. Any input is guaranteed to converge to one of the trained output patterns. If an untrained pattern is supplied as input, the network will converge to the closest matching trained pattern, stabilising at a local minimum. Therefore if only a partially correct input pattern in supplied, the network will be able to restore the trained pattern [16].

2.3.1.7 Echo State Network

The Echo State Network (ESN) is a relatively new variation of recurrent neural networks [19]. The traditional RNN allows modelling of system dynamics through feedback connections at the cost of complicated backpropagation training.

The ESN implements a very large amount of hidden neurons which are sparsely connected as shown in Figure 2.15. Input connection, recurrent interconnection and output connection weights are initialised at random. Then during training only output connection weights are updated.

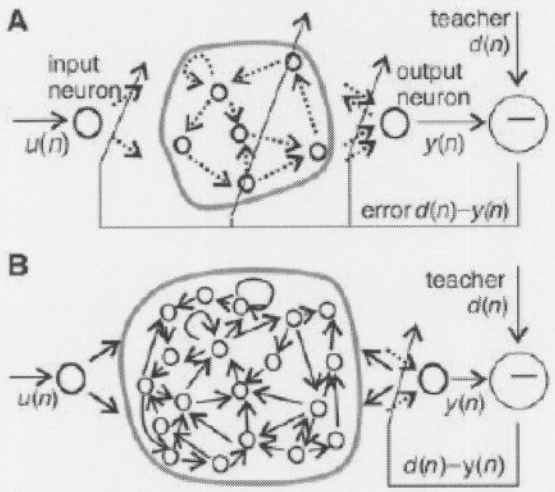

Figure 2.15 *Recurrent neural network (A) and echo state network (B)*
The ESN employs a high volume of hidden neurons with random fixed weight connections and a few adaptive weight connections to the output neurons. Dotted lines indicate trained connections and grey lines indicate the training process

This approach promises the same dynamic capabilities as RNNs, but with a very simple training algorithm.

2.3.1.8 Adaptive Resonance Theory

If a new pattern needs to be added to an already trained neural network, the network would normally have to be trained with the new pattern as well as the previous patterns; otherwise the previously trained data would be lost. Adaptive Resonance Theory (ART) addresses this problem. This RNN variant followed a study of the processes occurring in the organic brain [16].

When an input vector which has been learnt by the network is presented to the network, the output will resonate between two layers of the network where reinforcement of the associated stored pattern will occur. If a new input vector is presented to the network, the network will enter a new resonant state where the new pattern will be stored.

2.3.1.9 Instantaneously trained neural networks

The need for fast learning neural networks motivated the modelling of short-term memory and the development of the corner classification family of networks. These networks learn instantaneously and exhibit good generalization, but suffer the disadvantage of operating only on discrete data and requiring considerable amounts of neurons.

Fast classification networks are a generalisation of corner classification networks, capable of operating on real data, adding considerable flexibility and reducing the amount of neurons required [20]. The distance between hidden layer neurons and the input vector is calculated and fed to a rule base, which calculates a membership vector. The output is produced by taking the dot product of this value and the output vector.

Other instantaneously trained neural networks include many variations of diluted networks as well as Willshaw networks and Advanced Distributed Associative Memory (ADAM) networks which operate on binary data.

2.3.2 Fuzzy Logic

Fuzzy logic introduced an alternative to formal discrete binary logic. Whereas binary logic demands absolute classification, allowing a statement only to be true or false, fuzzy logic presents a way to deal with partially true conditions. Using binary logic for classifying a person as 'tall', a subject would be classified as 'tall' if his length is more than a threshold value and 'not tall' otherwise. Fuzzy membership allows a person to be a member of the class 'tall' to some degree: being tall then becomes a gradual change-over with increase in length as indicated

by the fuzzy membership function in Figure 2.16. The degree of membership is usually a real value in the range of zero to one.

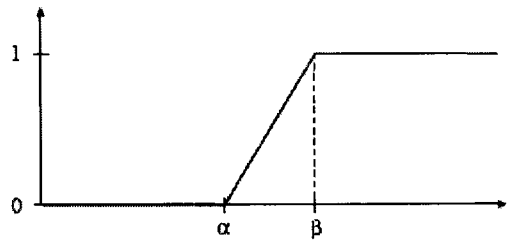

Figure 2.16 *Fuzzy membership*

With fuzzy logic the man is assigned to the class of "tall" with a degree of membership gradually increasing between lengths α and β [15].

Converting crisp or real values to fuzzy membership values are called fuzzification. Fuzzy logic rules from a knowledge base are applied to the fuzzy values in the fuzzy inference unit as shown in Figure 2.17. Finally fuzzy results are defuzzified to create crisp output values.

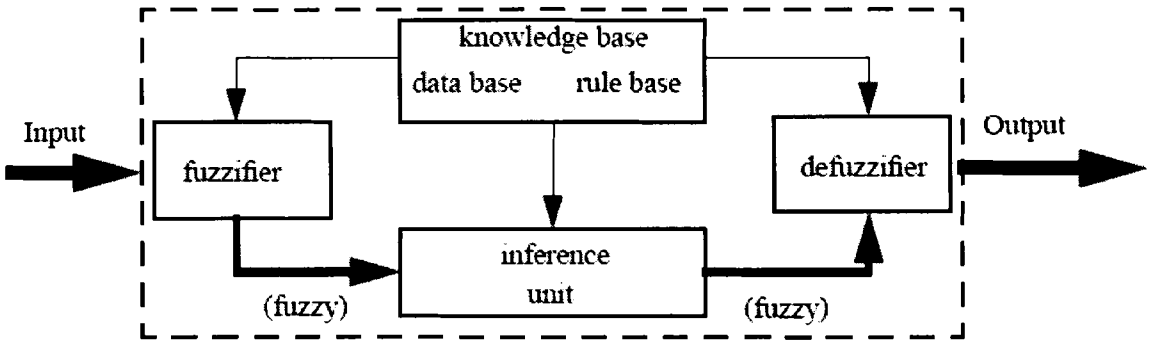

Figure 2.17 *Fuzzy logic system*

The system consists of a fuzzifier, converting crisp values to fuzzy values, an inference unit for executing fuzzy rules and a defuzzifier, converting fuzzy results back to crisp values [21].

2.3.2.1 Fuzzy inference systems

Fuzzy logic supplies a set of operators to deal with logic statements [22]. A fuzzy inference system consists of a set of fuzzy 'if' statements operating on fuzzy input values. The fuzzy operands are the following:

- The AND operator is given by a function called the T-norm which can simply be the minimum of all the inputs. Alternatively it could also be calculated as the product of inputs.

- The OR operator is given by a function called the N-norm which can be the maximum of the inputs.
- The NOT operator is simply one minus the input value.

There are mainly three different types of representations of the 'then' clause in the fuzzy statements namely Mamdani, Sugeno and Tsukamoto [23].

Mamdani

The Mamdani-type fuzzy is the most popular type of rules for fuzzy control problems. Outputs are allocated to fuzzy classes:

If (input X1 is small) and (input X2 is medium) then output Y is large

where small, medium and large are defined classes for Xi and Y.

Sugeno

The Sugeno (also known as TSK – Takagi, Sugeno, Kang) type fuzzy rules are more complicated than Mamdani type rules. This type of rules is best suited for fuzzy classifier problems. Outputs are declared as a linear combination of input value:

If (input X1 is small) and (input X2 is medium) then output Y is k1*x1 + k2*x2

where small and medium are defined classes for Xi, xi are the real values of Xi and ki are constants.

Tsukamoto

Tsukamoto type rules define outputs through a membership function which then result in crisp inference values. By inferring a crisp output value this method avoids the tedious defuzzification process required by the other two methods. This method is however not commonly used. Outputs are declared as membership functions

If (input X1 is small) and (input X2 is medium) then output Y is C

where small and medium are defined membership functions for Xi, xi are the real values of Xi and C is a membership function.

2.3.3 Evolutionary computation

Evolutionary computing (EC) encompasses a broad range of combinatorial optimization methods. These methods can be described as loosely fitting the following criteria:

- soft computing
- iterative progress, growth or development
- population based solutions
- guided random processes
- usually meta-heuristic approach
- often biologically inspired

Unlike neural networks and fuzzy logic systems which form connectionist systems with multi-variable input-output mappings, EC are merely search algorithms. Therefore EC cannot in itself be used as classifiers or controllers but it can be used for finding and optimizing parameters in classifiers and controllers.

Evolutionary methods are capable of global search, finding near optimal solutions in a nonlinear piece-wise differentiable search space. This makes them very efficient for training neural networks and fuzzy logic systems, especially in highly nonlinear conditions.

The methods used in EC can be divided into three groups.

- Evolutionary algorithms mimic natural reproduction, mutation and survival of the fittest. Traditional methods include evolutionary strategies, genetic algorithms, evolutionary programming and genetic programming.
- Swarm intelligence mimics flocking behaviour and path finding. Primary methods are ant colony optimization and particle swarm optimization.
- Non-biologically inspired techniques such as simulated annealing and tabu-search.

Many hybrids and variations of these methods exist. Only the relevant methods will be further discussed here. It is worth noting that the swarms and colonies can be considered as a form of artificial life, and that A-life programmers often also make good use of methods such as genetic algorithms for evolving their subjects.

2.3.3.1 Evolution strategy

Evolution strategy is one of the earlier evolutionary algorithms [24]. Optimisation parameters are represented in a single chromosome as a list of real values. The goal of the problem is defined as finding parameters for the chromosome that maximizes a predefined fitness function.

The chromosome is initialised with random values. In every generation or iteration the chromosome is duplicated and mutated by making small random changes to the parameters. The fitness of the new chromosome is compared with that of its parent and the stronger of the two is preserved. This is repeated, allowing the parameters to gradually improve through random variation. The process ends when a certain fitness threshold or amount of generation has been reached or if the fitness becomes stagnant.

2.3.3.2 Genetic Algorithms

Unlike in evolution strategy, in genetic algorithms (GAs) a large population of potential solution chromosomes are preserved [25]. The population is initialised with random chromosome values so as to span the complete solution space. In every generation the fitness of all the chromosomes in the population is evaluated. The fittest chromosomes are selected as parents from whom a multitude of offspring is cultivated. This involves a process of crossover as well as mutation. This cycle is illustrated in Figure 2.18. Like with evolutionary strategies, the process is repeated until the maximum fitness level of the population has reached a threshold, a specific number of iterations have been reached, or the fitness level has stagnated. Note that the fitness function needs to be maximised.

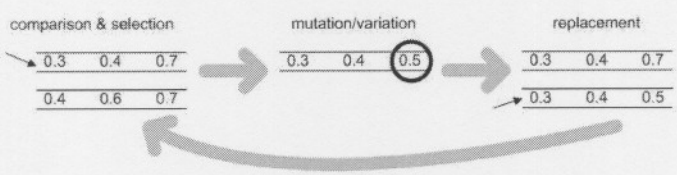

Figure 2.18 *Evolutionary cycle*
GAs cycle iteratively through comparison and selection, crossover, mutation and replacement.

Usually a preset percentage of the population gets replaced by new chromosomes in every generation, keeping the population size constant. Selection of the chromosomes to be replaced is done on the grounds of their fitness; the fitter part of the population is preserved.

For every new offspring chromosome two parent chromosomes are selected. Parent selection is random, with higher probability to fitter individuals. Many different methods can be used to accomplish this. The most popular is 'roulette wheel selection' illustrated in Figure 2.19. The chance of selecting parents can be adjusted by assigning different shares of the selection range to the different chromosomes.

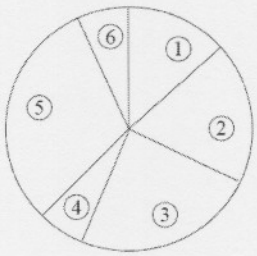

Figure 2.19 *Roulette wheel selection*
By assigning different sector sizes of the wheel to different chromosomes, the likelihood of using better candidates can be increased [25].

New chromosomes are created from a combination of the parent chromosomes through crossover. The chromosomes are split at randomly selected crossover points and parts are exchanged between the parents to create two new offspring chromosomes as indicated in Figure 2.20. In some variations multiple crossover points are selected to exchange only segments of the chromosomes.

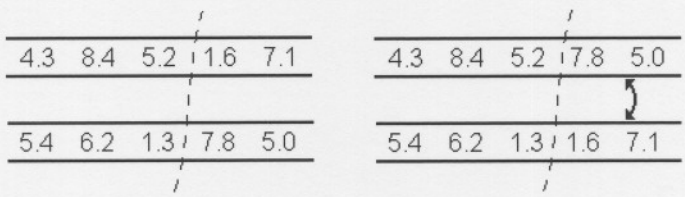

Figure 2.20 *Genetic crossover*
During crossover genes to the one side of a random sectioning point of two chromosomes are interchanged, converting the two chromosomes on the left to the two on the right.

In classic GAs the parameters are encoded in the chromosome as binary strings. Mutation is simply the 'flipping of a bit', changing a random 0 in the chromosome to a 1, or vice versa. This event occurs with very low probability.

Many variations of the GA have been implemented, including variable-length chromosomes and multi-objective GAs [1]. The most commonly implemented change is the use of real numbers in

the chromosomes. This allows more systematic mutation and eliminates the need for the back-and-forth conversion of chromosomes to real numbers for fitness evaluation.

2.3.3.3 Ant Colony Optimization

Ant algorithms, more formally known as ant colony optimization have especially been popularised in path finding problems in discrete 2D grid-based environments.

A multitude of ants roam an environment with obstacles in search of an optimal path to the goal state [26]. Initially roaming is completely random. Whenever the goal state is reached by any ant, the path it followed is laid with pheromones, reinforcing the route. When other ants cross that path roaming gets biased towards that path as illustrated in Figure 2.21.

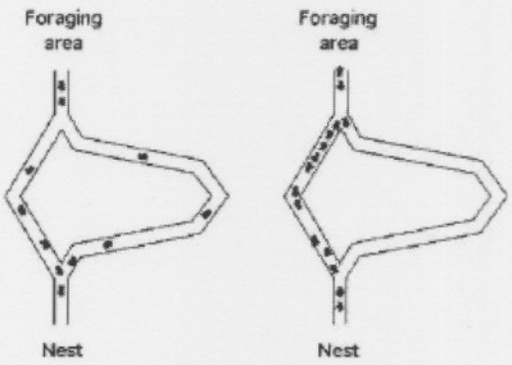

Figure 2.21 *Reinforcing the shortest path*
The shortest path to the foraging area is laid thicker with pheromones, biasing successive ants to follow that path [27].

Since the ants are allowed to stray from the path, especially at early stages when the path has not been reinforced much, opportunities exist to discover alternative routes. Figure 2.22 illustrates how a successful path is laid with pheromones. Pheromones are allowed to decay gradually to remove non-optimal solutions. As more ants reach the goal along the most optimal route, that route gets reinforced, eventually swaying the whole colony towards the better solution. This makes it a type of reinforcement learning.

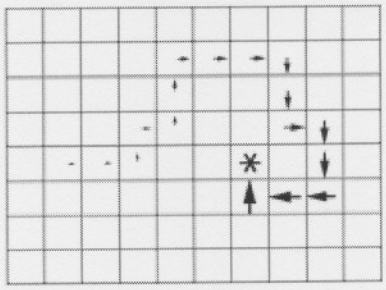

Figure 2.22 *Ant pheromone trail*

Simulated directional pheromones are laid at cells of a grid-world or states visited during a successful trial. This improves the likelihood of the same trail being followed again and being further reinforced.

Although this technique is relatively effective, as with GAs, absolute optimality cannot be guaranteed in finite time. It can be used with equal effectiveness to discover sequences of actions in alternative discrete state spaces. Methods have been suggested to combine this with fuzzy logic in continuous space, but no such methods have been widely implemented.

2.3.3.4 Particle Swarm Optimization

By simulating flocking behaviour, particle swarm optimization (PSO) techniques are capable of effectively searching a vast solution space [26]. Individual particles of the swarm exhibit random behaviour, but unlike GAs they are influenced by the average movement of the total population. This movement is driven by momentum as well as a history of probable solutions. The effect is a somewhat more systematic search than with GAs with completely random crossover and mutation. Swarms tend to 'sweep' the solution space and then start oscillating around possible solutions and eventually settle at a solution.

2.3.3.5 Simulated annealing

Simulated annealing is not biologically inspired, but it still emulates natural processes.

In the metallurgic process of metal annealing, a metal is typically softened by heating it up and cooling it down slowly, allowing the atoms to settle without internal stresses and crystal defects. Similarly, in simulated annealing a set of parameters is excited and then allowed to settle in a more desired state. When excited, values of a set of parameters are allowed to change by some probability factor, even if it means that the fitness of the set decreases. Gradually the allowed margin of allowed decrease in fitness is reduced, forcing the parameters to settle at a minimum point. The idea of excitation is to enable the parameter set to escape from possible local minima.

2.3.3.6 Artificial Immune Systems

A new approach in combinatorial optimization is through simulation of the immune system. The human immune system is receiving much attention in the research of computational intelligence due to its powerful information processing capabilities **[28]**.

The purpose of the immune system is to identify and destroy foreign cells or molecules (e.g. bacteria and viruses). They are called pathogens. The immune system consists of B-cells and T-cells. B-cells are capable of classifying other cells in the body as self or non-self. Non-self cells are then further categorized to induce the appropriate defensive mechanism.

B-cells have receptors (antibodies), capable of matching antigens of other cells. This is shown in Figure 2.23. B-cell receptors learn to recognise non-self antigens through a process of hypermutation and clonal selection. B-cells which offer better recognition capabilities attain a longer lifespan and become memory cells. Memory cells allow the system to deal with previously encountered cells instantly.

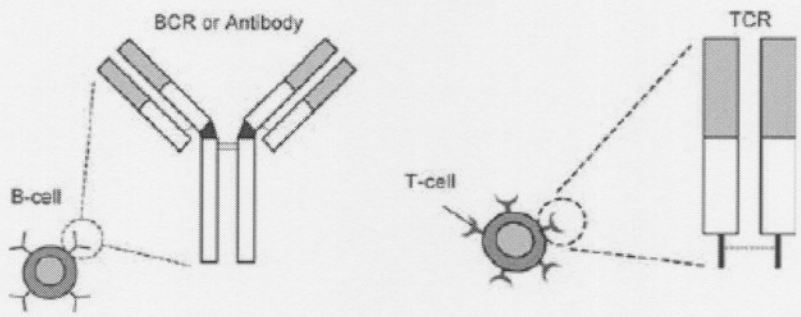

Figure 2.23 *Cells of the immune system*
The antibody or B-cell receptor is shown on the left and the T-cell receptor on the right. The receptors act as pattern matchers to recognise pathogens [29].

The immune system is considered as a remarkable parallel and distributed adaptive system from an information-processing perspective. Pattern recognition and classification tasks are solved through an emergent learning process of evolution, memory, and association.

2.4 Hybrid intelligent systems

The AI techniques described up to this point have been developed and applied rather independently. It has been realised that these techniques have different strengths and weaknesses, and instead of opposing one another, they are complimentary and can be used together. Integration of the separate techniques in many different arrangements has become an active area

of research in AI in recent years. Such hybrids can combine the best features of these technologies:

- the strict flow diagrams of expert systems
- the trainability of neural networks
- the fuzzy rule based logic of fuzzy logic
- the parallel global optimisation of evolutionary computation

2.4.1 Neuro-fuzzy systems

The term neuro-fuzzy denotes all techniques which draws aspects from both neural networks and fuzzy logic. It does not imply any specific configuration and can, among others, be any of the following combinations:

- representing fuzzification, fuzzy inference and defuzzification through multi-layer feedforward connectionist networks
- realising fuzzy membership through clustering algorithms in unsupervised learning
- deriving fuzzy rules from trained RBF networks
- fuzzy logic based tuning of neural network training parameters
- fuzzy logic criteria for increasing a network size

The last two combinations implement pure neural networks with fuzzy logic only to accomplish supporting tasks. Such combinations will not be discussed here. The other implementations can be considered as special cases of neural or connectionist networks through which fuzzy logic functions can be represented.

2.4.1.1 Feedforward fuzzy network

The power of neural networks lies in their ability to be trained by means of gradient descent. The down side of neural networks is that it leaves the development engineer with a complex nonlinear 'black box' system of which the logic is difficult to interpret, understand and verify.

Fuzzy logic systems make use of fuzzy sets to describe nonlinear systems in linguistic terms. This allows the development engineer to model the operation of the system in terms of humanly interpretable rules. These rules may be intuitive or they may be obtained from an expert. They can easily be modified and optimised, but these systems lack the automated trainability of neural networks. Furthermore there is not always an expert with a set of rules at hand.

By representing a fuzzy system as a multi-layer fuzzy feedforward network (FFN) as shown in Figure 2.24, the fuzzy system inherits the trainability of neural networks and becomes an adaptive rule based system [21].

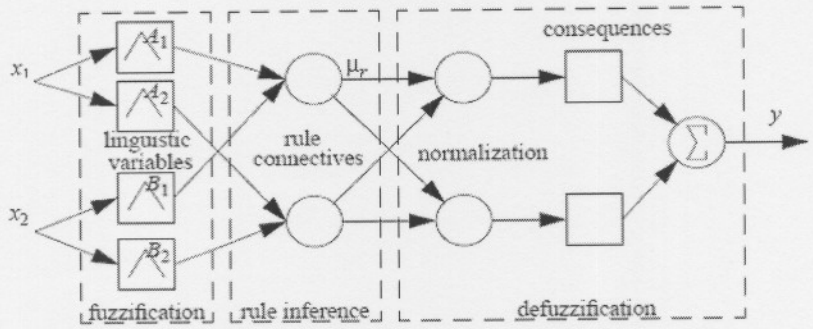

Figure 2.24 *Feedforward fuzzy network*

A fuzzy system can be represented as a 5-layer feedforward connectionist network with the layers implementing fuzzification, intersection, union and normalization respectively [21].

To sustain linguistic interpretability of the internal logic some restrictions are imposed on such systems. It results in reduced performance and the requirement for more neurons compared to standard neural networks. Various architectures have been proposed to acquire the best of both systems. The FFN with centre average defuzzification, product inference, singleton fuzzifier and Gaussian membership are represented as

$$f(\underline{x}) = \frac{\sum_{l=1}^{M} \bar{y}^{l} \left[\prod_{i=1}^{n} \exp\left(-\left(\frac{x_i - \bar{x}_i^{l}}{\sigma_i^{l}} \right)^2 \right) \right]}{\sum_{l=1}^{M} \left[\prod_{i=1}^{n} \exp\left(-\left(\frac{x_i - \bar{x}_i^{l}}{\sigma_i^{l}} \right)^2 \right) \right]} \tag{2.10}$$

where $\underline{x} = (x_1, \ldots, x_n)^T$ is the input vector, M is the amount of rules, n the amount of terms per rule and $\bar{y}^{l}$, $\bar{x}_i^{l}$ and σ_i^{l} are adjustable variables of the system. These terms can be trained through the backpropagation algorithm [22].

2.4.1.2 ANFIS

The neuro-fuzzy technique implemented in MATLAB®'s fuzzy toolbox is the Adaptive Neuro-Fuzzy Inference System (ANFIS) [23]. The ANFIS architecture is explained in Figure 2.25. It also implements a 5-layer feedforward connectionist network.

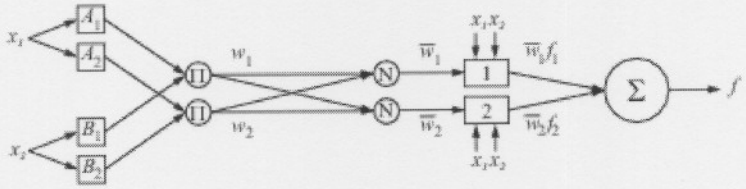

Figure 2.25 *Adaptive Neuro-Fuzzy Inference System*

ANFIS is a 5-layer feedforward connectionist network [23]. *The layers function as follows:*

1st layer: bell-shaped membership functions perform fuzzification

2nd layer: T-norm functions carry out "AND" clause operations

3rd layer: normalization of firing strength (to sum of all rule's firing strength)

4th layer: calculate rule consequence or scaling (Sugeno or Tsukamoto)

5th layer: sum all consequences

2.4.2 Fuzzy clustering

Cluster analysis and clustering algorithms are used not only for unsupervised data categorization as depicted in Figure 2.26, but also for data compression, feature extraction, model construction and fuzzy rule extraction from training data.

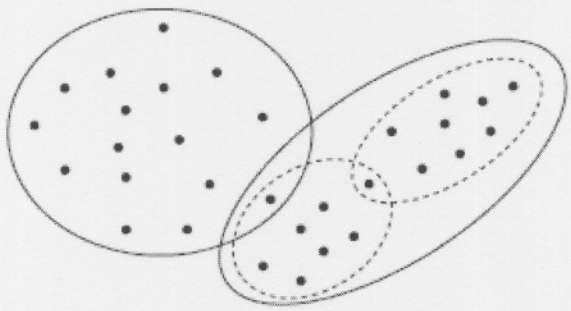

Figure 2.26 *Fuzzy clusters*

Depending on predefined cluster shape and size and amount of clusters allowed, data can be classified in different configurations [15].

Fuzzy clustering algorithms allow partial classification and are used extensively for neuro-fuzzy system modelling and control. Cluster centres are encoded into weight vectors of neurons and

used to represent the 'if' clause of fuzzy rules. This is typically the case in RBF networks where some fuzzy clustering algorithm is implemented for unsupervised learning of the basis vectors.

Many different clustering algorithms and variations exist of which the most prominent will be discussed here.

2.4.2.1 K-mean clustering

The family of K-mean clustering techniques are very popular and is typically used by MATLAB® for hidden layer training RBF and ANFIS networks [14].

Cluster centres are initially selected randomly or received from some faster but less accurate method, or simply taken directly from the first training samples. Training data is then classified according to these cluster centres. New cluster centres are then repeatedly calculated as the mean value of its members.

In the Fuzzy C-mean version training samples are allocated to different clusters only to a degree, depending on its distance from the centre. The cluster centre calculation algorithm is adapted accordingly. Many variations of this techniques exist, including K-median and Genetic K-mean.

2.4.2.2 Mountain clustering

The mountain clustering method is a simple technique for rapid cluster centre estimation [23]. It can be used as stand-alone or as starting conditions for more sophisticated methods. A mountain function is generated through a data density measure at grid-points over the input space. Mountain height is calculated as the proximity of data points to the grid points. The mountain function is then sequentially destructed by selecting the next cluster centre at the highest grid-point and then eliminating this cluster's effect on the mountain function. This is done by subtracting a Gaussian function from the mountain function at the selected cluster centre. In this way clusters are incrementally removed from the mountain function.

The grid-points are generally uniformly distributed over the input space. This results in an exponential expansion of grid-points and computation with increase in dimensionality. To solve this problem a variation of this approach, called subtractive clustering, was introduced.

In the subtractive clustering method data points are used instead of grid-points as potential cluster centres. In this way the points of density calculation is independent of the input dimension.

2.4.2.3 Growing Neural Gas

Growing self-organising networks is a subset of unsupervised competitive learning which is the neural network approach to cluster analysis [30]. Each node of the network represents a vector in input space. Nodes are interconnected by edges to form a graph structure as depicted in Figure 2.27. A node's neighbourhood is defined as those nodes connected to it by edges.

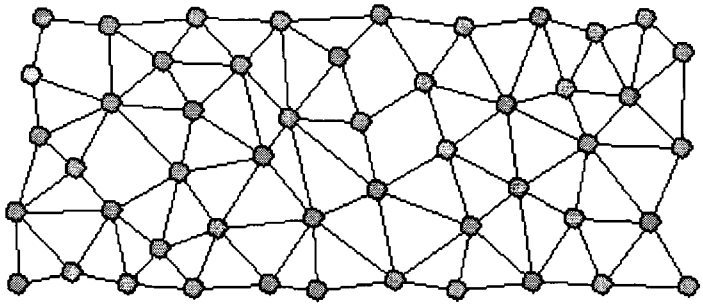

Figure 2.27 *Growing neural gas*
Nodes are connected with closest neighbours to form a network of mutually influencing units, covering the whole input space.

During learning only the nodes closest to the input vector are updated. The network structure can also be altered by adding or removing nodes and edges. Edges are produced by competitive Hebbian learning. This has the effect of a dynamic graph structure.

A family of growing neural gas (GNG) algorithms has emerged from an original growing cell structure algorithm. The name originated from the gas-molecule resembling vibration of its nodes. The original algorithm grows a k-dimensional network, usually of fewer dimensions than the input space. Therefore the model performs a dimensionality reduction similar to SOM. Node deletion is based on a measure of node activity and contribution. GNG does not impose such a strict network-topology preservation rule and edges are deleted based on age, resulting in a more dynamic system.

Many versions of the GNG algorithm have been developed including Hierarchical GNG, GNG with utility, and growing grids.

2.4.2.4 Evolutionary methods

Since the clustering problem can be described as a parameter optimisation problem, it is notable that most of the classical techniques discussed in the section on evolutionary computation can also be applied directly to fuzzy clustering.

Since the vectors of all fuzzy clusters need to be optimised simultaneously, it can result in a search space of very high dimensionality. The chromosomes in a GA, the state of every individual in a swarm, or the system state in simulated annealing would typically consist of a list of parameters of size input dimension times the allowed amount of clusters.

2.4.2.5 Evolving classifier functions

These methods should not be confused with methods of evolutionary computation. In evolving classifier functions a structure is developed systematically over time as training data becomes available. Training is supervised and training data consists of an input vector and the associated class. Nodes are added to the system to represent clusters. Each node consists of a centre point and a variable size. Multiple nodes can be used to represent the same class. Training involves the follows basic procedure:

- If the size of a node of the same class can be increased to include the input vector without covering any node of a different class, or exceeding maximum allowed size, resize the node accordingly.
- Otherwise, if a node for a different class includes the input vector, reduce this node's size to exclude the input vector. Create a new node at this point with minimum allowed size.

Different evolving connectionist systems have been developed [31] that follows this process. Dynamic evolving neuro-fuzzy inference systems create rectangular clusters in a single pass through the training data. Evolving fuzzy neural networks and evolving self-organising maps are hybrid classifiers combining the features of GNG and SOMs.

The nearest neighbourhood clustering (NNC) function can be considered as a much simplified evolving classifier since new nodes are added to the system at data points not already covered by other nodes. However nodes are fixed in size and location after instantiation. Node output values are calculated as the average of the target values for the samples allocated to the cluster. The classifier function output is given by

$$f(\underline{x}) = \frac{\sum_{l=1}^{M} A^l . \exp\left(-\frac{\left| \underline{x} - \underline{x}_0^l \right|^2}{\sigma^2} \right)}{\sum_{l=1}^{M} B^l . \exp\left(-\frac{\left| \underline{x} - \underline{x}_0^l \right|^2}{\sigma^2} \right)} \tag{2.11}$$

where $\underline{x} = (x_1, ..., x_n)^T$ is the input vector, M is the amount of rules and $\underline{x}_0^l$ are the cluster centres. σ is a cluster smoothing parameter, A^l is a sum of target output vectors for samples and B^l is the amount of samples in cluster l [22].

2.4.2.6 Support Vector Machines

This relatively new method of learning classification functions originated from statistical learning theory. Support vector machines (SVM) were introduced in the structural risk minimization framework.

The SVM uses a linear hyperplane to separating data into two classes with a maximum margin. 'Vapnik Chervonenkis' bounds provide the probability of errors which is minimized when the margin is maximized [6].

When classes are not linearly separable in input space, the SVM transforms the input space into a feature space by means of one of many different mapping techniques. With appropriate transformation, the data can be linearly separated and finding the optimal hyperplane is trivial.

2.4.2.7 Alternative methods

Most of the soft computing techniques have been applied to the clustering problem. Prominent techniques include Gustafson-Kessel, Gath-Geva and the LBG (Linde, Buzo, Gray) clustering algorithms. Other techniques include learning vector quantization, deterministic-annealing, outer product based networks, agglomerative clustering as well as a variety of ART variants, only to name a few.

2.4.3 Genetic fuzzy systems

When GAs are used for the parameter optimization of fuzzy system parameters the system becomes known as a genetic fuzzy system. By representing the fuzzification and defuzzification

membership functions in terms of some generic parameterised function, and listing the rule base in terms of all possible combinations of input membership functions, it is possible to describe any part or the whole fuzzy system through the chromosomes of a GA. For the optimisation of such a system some criteria or fitness function is required. This can only be done by evaluating the effectivity of the system proposed by the GA as a whole. It can be seen that this process is a form of reinforcement learning since adaptation is based on an evaluation value obtained through interaction with its environment.

Two approaches have originally been followed in the encoding of fuzzy systems in GA chromosomes. Recently an alternative third approach, called the incremental approach has since been developed to combine the Michigan approach and the Pittsburgh approach [32].

2.4.3.1 Michigan approach

The most straight forward method of optimising fuzzy systems through GAs is called the Michigan approach. In this approach the complete list of parameters representing the system is encoded into every chromosome. A population of such chromosomes, or in effect a population of different fuzzy systems, compete for survival.

The complete population is initialised with random-valued chromosomes. Every fuzzy system needs to be evaluated individually for its fitness. Evaluation requires every system to be tested over a reasonable part of the input space. Doing so for a complete population of fuzzy systems over several generations proves to become very time consuming. All the other facets of the GA are implemented in the standard fashion.

2.4.3.2 Pittsburgh approach

An alternative to Michigan is the Pittsburgh approach. In this approach only the fuzzy rule base is encoded and optimised. Every rule is encapsulated in its own chromosome. The rule base is represented by the complete population. Rules compete against one another for survival. Evaluation is done on the ground of every individual rule's contribution to the success of the resulting system.

2.5 Intelligent Agents

An intelligent agent (IA) is an intelligent control system interacting with its environment. IAs exhibit the following capabilities [33].

- learning quickly from large amounts of data
- adapting online and in real time as new data is encountered
- accommodating new problem solving rules incrementally
- data storage and forgetting in short and long term memory
- learning and improving through interaction with the environment

IAs are characterised for being situated inside their target problem domains with the ability to control their own destiny to some extent. Embodiment of the controllers is another important feature which allows them to directly interact with and learn from their environments. With the ability to learn online and in real time, IAs can support life-long learning and adaptation to dynamic environments.

Therefore it is required that IAs can assimilate large amounts of data, make feasible deductions and incorporate into their control policies in a relatively short time. Further, they should be able to discover new solutions to problems in changing environments [34].

2.5.1 Exploration and exploitation

IAs gather knowledge of their environments to construct offline models of their environments. These models are used to estimate expected future states or future rewards for given actions. In simple systems with small discrete state and action space these models can be represented in tabular form. The action with the highest reward for a given state would suggest the action to be taken. When the state space becomes significantly large or continuous, this option is no longer feasible. Function approximators like neural networks are used to estimate expected results. The function estimating the expected reward is called the value function. A function approximator can also be used to produce the desired control actions. Such a function is called the policy.

By taking the action proposed by the policy, the agent exploits its policy. The agent also has to explore the environment in order to gather data and discover optimal actions. The gathered information can then be used to update the value function, the state model and the policy. Note that it is not essential that all of these are implemented explicitly. It is for example possible to derive actions from the value function or state model.

2.5.2 Actor-critic architecture

In the popular actor-critic architecture [13] of reinforcement learning the policy and value functions are separated as depicted in Figure 2.28. While the critic typically applies TD learning

for updating the value function, there are different methods for training the policy. These methods can generally be divided into two approaches. One approach is to use gradient based methods. The other approach is to use a meta-heuristic search algorithm to find optimal values for the parameters of the policy.

Examples of each of these approaches are described here. Both these examples implement adaptive networks for the actor and the critic. This property makes it very attractive for use in agents operating in continuous state and action space. Gradient methods are used in the adaptive critic design (ADC) [35] while the system described in paragraph 2.5.2.2 uses a GA to optimise the policy.

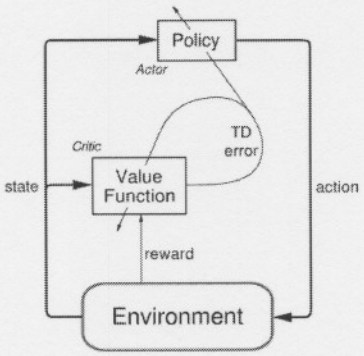

Figure 2.28 *Actor-critic architecture*

The actor implements the policy that generates the action. The critic implements a value function for estimating the reward for updating the policy [13].

An addition to the actor-critic architecture is the dyna-Q system, implementing and adapting a model of the environment for offline exploration. This drastically decreases the amount of online exploration required.

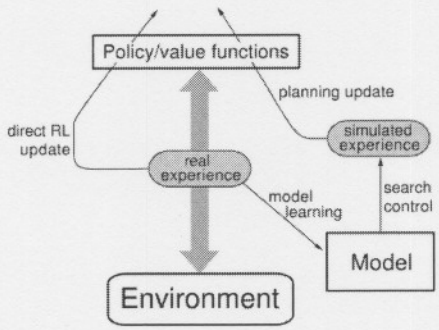

Figure 2.29 *Dyna-Q system*

With the use of a model, the dyna-Q system is capable of offline exploration and learning based of simulated experience [13].

2.5.2.1 Adaptive Critic Design

A family of ACD controllers has been developed [35]. Actor training uses adaptive networks for both the actor and the critic. Training of the actor involves backpropagation of the derivative of the value function with respect to the actions $\partial J(t)/\partial A(t)$. This in turn might require a system model [36]. The model requirements for different variations of the ACD are listed here:

- Heuristic Dynamic Programming – A constant value is back-propagated through the critic and then through the system model to obtain $\partial J(t)/\partial A(t)$ for which a model is required.
- Action Dependent Heuristic Dynamic Programming – The critic receives actions as input, allowing $\partial J(t)/\partial A(t)$ to be obtained by back-propagation only through the critic. No models are required.
- Dual Heuristic Programming – The critic estimates the derivatives of the cost function with respect to the system states. This requires a full system model. $\partial J(t)/\partial A(t)$ is acquired by back-propagating the critic's outputs through the system model. Models are required for training of both the actor and critic. This is illustrated in Figure 2.30.
- Action Dependent Dual Heuristic Programming – Critic training is the same as for Dual Heuristic Programming. The critic estimates $\partial J(t)/\partial A(t)$ directly. A model is required only for critic training.

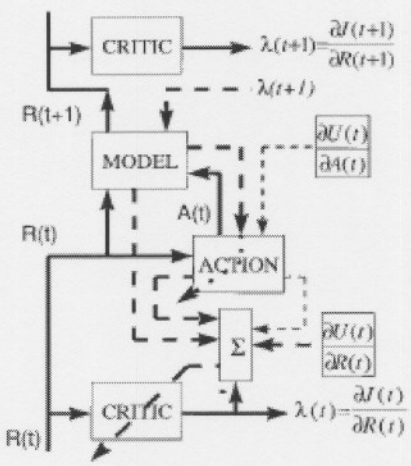

Figure 2.30 *Dual Heuristic Programming in the ACD*

The critic is shown for two consecutive steps in time. Backpropagation is indicated with dashed lines for training the critic and the actor [36].

Deriving accurate system models can be very time consuming. Dynamic systems require adaptive models. Although differentiable models can be updated online, accuracy is important since actor errors are generally a result of inaccurate system modelling.

2.5.2.2 TDGAR

The Temporal Difference Genetic Algorithm Reinforcement learning (TDGAR) architecture shown in Figure 2.31 uses a GA to search for optimal parameters for the actor [37]. Each chromosome represents the parameters for a complete policy. Each potential policy is evaluated by controlling the system for a finite time and measuring the success.

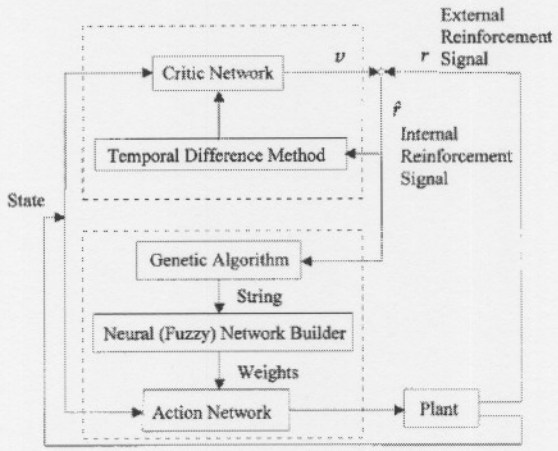

Figure 2.31 *TDGAR architecture*

The actor is trained via a GA with reinforcement feedback from the critic. The Critic is trained with external reinforcement feedback from the plant [37].

Since the controller needs to be tested over a range of control conditions to give a reliable evaluation, this process is very time consuming. TDGAR is therefore not suitable for online learning. It has however been implemented successfully in various applications with offline learning.

2.5.2.3 Cognitive robotics

Cognitive robotics generally denotes the physical embodiment of intelligent agents. Unlike software agents operating on the internet or in data mining, robots are physical systems, interacting with the physical objects in the real world.

The control of simulated robots in simulated physical environments for research and development can also be considered as cognitive robotics. It concerns real-world problems such

as path planning, obstacle avoidance, trajectory tracking and computer vision. These robots also have to interact with their environments by means of sensors and actuators.

2.6 Concluding remarks

A vast array of AI approached and techniques are discussed in this chapter. An intelligent agent will be developed for this project. An expansion of the actor-critic architecture will be proposed. Various controllers, based on feedforward neural networks and neuro-fuzzy systems, will be constructed for implementation in this architecture. Backpropagation and clustering algorithms will be used to train the controllers. A genetic algorithm will be implemented to gather data for training these systems.

Chapter 3

Manipulator Control

In the field of cybernetics the multi-segmented manipulator is a common device. This chapter gives an overview of research done on the control of manipulators on humans, frogs and robots. It then discusses current approaches followed for the solving of the path planning problem in the multi-segment manipulator domain.

Manipulators are used to pick up, manoeuvre or work on various items. They often exhibit a high level of degrees of freedom (DOF) with redundancy, in order to be able to grasp and reach objects in a semi-obstructed environment as illustrated in Figure 3.1. The major problems concerning mobile robot movement control are path planning, trajectory tracking and obstacle avoidance. This study focuses on path planning. Trajectory tracking will only be described briefly. Since obstacles are static, obstacle avoidance can be incorporated as part of path planning.

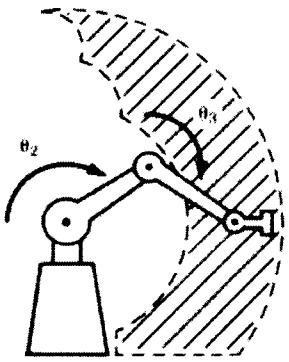

Figure 3.1 *Robotic manipulator*
The multi-segment manipulator can be used to reach a large range of positions in different postures [38].

3.1 Motor control

It has been shown that in humans and primates two distinct motor control phases are invoked for arm movement. An initial ballistic phase entails rapid, big movements. This is followed by correcting phase which takes the hand precisely to the target.

3.1.1 Ballistic phase

The initial rapid movement traverses a large distance to take the hand to the vicinity of the target. It is characterised by a bell shaped velocity curve [39]. The arm is gradually accelerated and

decelerated as can be seen in Figure 3.2. Movement is too fast for neuronal feedback control, typically less than half a second. The action sequence for the complete movement of this phase is calculated prior to execution. In control theory this equates to an offline path planning strategy.

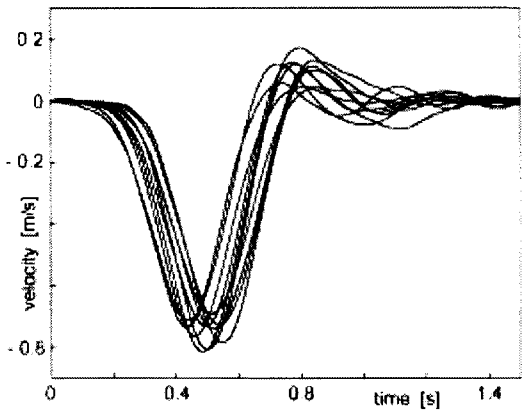

Figure 3.2 *Ballistic and adjusting arm movement*
The gradual acceleration and deceleration for 10 trials correlates in the ballistic phase. This is followed by the feedback controlled adjustment phase [39].

3.1.2 Adjustment phase

During the adjustment phase sensory feedback is used to adjust the position of the hand with slower correcting movements. Humans rely on visual feedback for control of the arm in this phase.

3.2 Spinal fields

Experiments have proposed how sensory-motor systems might be organized. Electrical stimulation of circuitry in the spinal cord of frogs and rats imposes muscle activation which directs the hind leg towards an equilibrium point. Recording the applied force at different positions as illustrated in Figure 3.3 revealed a well-structured spatial pattern. The observed force fields are called spinal fields [40]. Each spinal field directs the limb towards the equilibrium point, regardless of the initial condition.

Noting that simultaneous activation of multiple spinal fields result in vector summation, which led to a hypothesis that movement are based on the combination of a few motion primitives stored in the spinal cord.

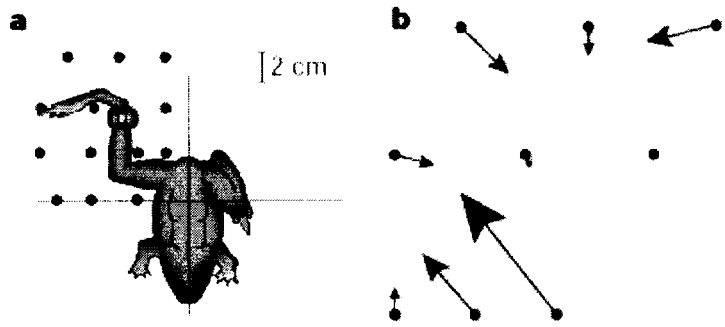

Figure 3.3 *Spinal field measurement*
By measuring the applied force vectors (b) at different positions (a) a complete force field can be reconstructed [40].

3.3 Path planning

Global navigation concerns the establishment of a path through space and negotiating obstacles in order to reach a required goal. For a robotic manipulator path planning involves finding a valid trajectory in state space. In its simplest form, using Aristotelian physics [41], the manipulator's state space is a volume of the same dimension as the manipulator's DOF. Figure 3.4 demonstrates this case for a 2 DOF system. This model disregards parameters such as velocity, momentum of variable payload and the dynamics of changing environments.

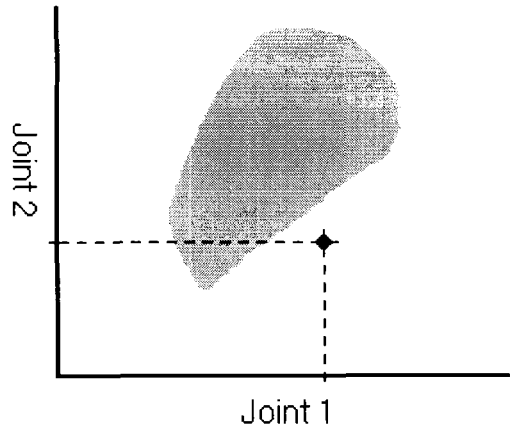

Figure 3.4 *Manipulator state space with 2 degrees of freedom*
Any static state of the manipulator in can be described by a unique vector in a state space with dimensions equal to the amount of joints of the manipulator.

Due to redundancy, different manipulator postures can result in reaching the target position. As a result, the target point and obstacle translate into a multiple of disjoint 'closed' hyper-surfaces in state space. Obstacles and physical limitations of the manipulator render many areas in the state space invalid. The goal is to find the optimal state space trajectory from the current state to a state of the target surface.

The trajectory is highly dependent on the environment and the goal. A trajectory-dependent policy therefore needs to be updated every time that any of these factors changes. Three different approaches have been identified to accomplish this. These approaches involve geometric analysis, path reinforcement and the construction of 'roadmaps'. Methods applied in these approaches will be discussed in this section.

3.3.1 Geometric analysis

The state space of navigation of a free roaming robot normally includes a position vector. Therefore the state space closely resembles the physical environment and path planning problem can be solved through analysis of the geometry of the physical environment.

The most popular way of doing this is through construction of a potential field simulation [42]. In a model of the physical environment all open space is filled from the goal outwards with a vector function pointing towards the target as the example in Figure 3.5 shows. The optimal geometrical trajectory can then be found by simply following the vector function.

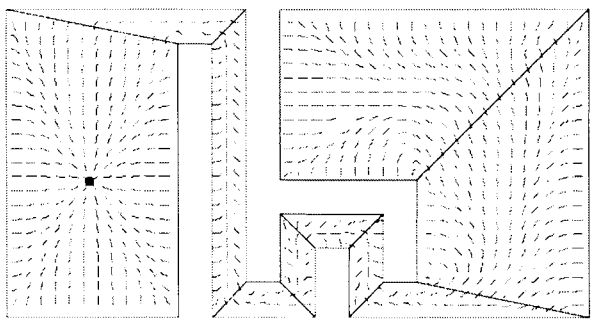

Figure 3.5 *Potential field simulation*
By following the arrows of the simulated potential field the target can be reached via an optimal path from any initial state [42].

The geometrical trajectory can easily be utilised by a free moving mobile robot, but since for a robotic manipulator, the posture of the whole manipulator has to be taken into account, it is not that simple. One notion is to track the trajectory by the tip, followed by consequent sections. Inverse kinematics is applied to calculate the required joint angles. Obstacle avoidance is implemented to ensure the manipulator avoids contact with obstacles during this process. A hyper-redundant manipulator is shown following the geometrical path in Figure 3.6.

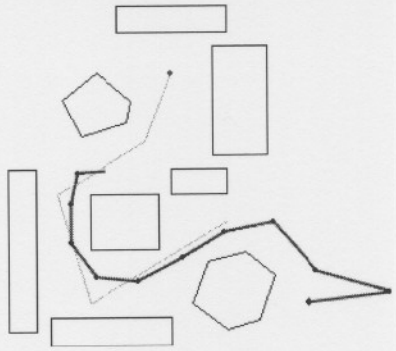

Figure 3.6 *Hyper-redundant manipulator*

The robotic manipulator follows a pre-defined path in geometric space through the obstacles

With a static environment the potential field is fixed and capable of providing a solution path from any position in real time. However, every time the environment changes the potential field must be recalculated.

3.3.2 Path reinforcement

Reinforcement learning techniques are very often applied to the path finding problem, especially in grid-world environments such as computer games [13]. State space is explored for solution paths through application of random actions. The exploration of a maze with techniques such as Q-learning and ant-algorithms is demonstrated in Figure 3.7.

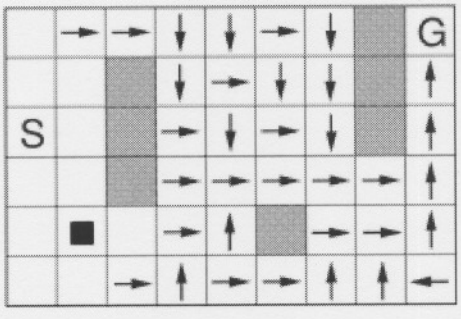

Figure 3.7 *Maze exploration*

Exploration with the dyna-Q algorithm as discussed in section 2.5.2 [13].

State space does not necessarily resemble geometrical space and can be of much higher dimension for instance in the case of manipulators. An increase in dimensions increases the amount of possible states exponentially and finding a target state through a sequence of random actions can easily become unrealistic.

3.3.3 Roadmap approach

The most successful approach to date for controlling hyper-redundant robotic manipulators in an environment with obstacles uses the probabilistic roadmap method [43]. This involves an offline pre-processing phase, creating a roadmap and an online phase for finding a route.

In the first phase the set of valid unobstructed manipulator states is acquired through random selection. These states form nodes in state space. The nodes are then evaluated in pairs to see if the manipulator would be able to move unobstructed, following a direct line in state space from one node to the other. If this is true, a valid connection is made between these nodes. In this way a network or roadmap of legal moves between various node states are created.

In the second phase the current state and goal state is defined. The closest node to the current state with which the current state can form a valid connection is selected as the first node. In the same way the last node is selected in relation to the goal state. The remaining task is then to find the shortest sequence of nodes connecting the first node to the last. This problem is very similar to the travelling salesman problem [44] shown in Figure 3.8 and similar search techniques can be applied to solve the problem.

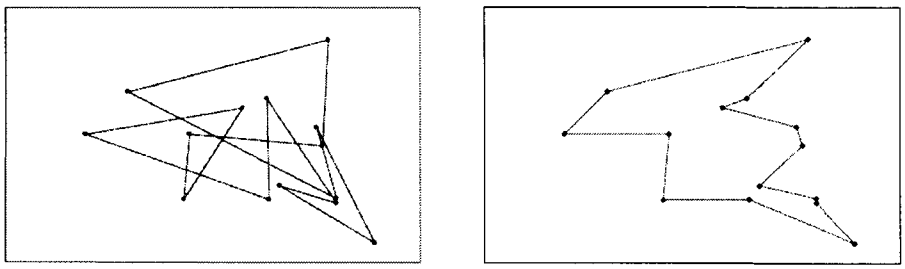

Figure 3.8 *Travelling salesman problem*
Nodes are connected at random on the left. The same nodes are connected optimally along the shortest path on the right [44].

This approach assumes that enough nodes are sufficiently chosen to make the whole state space reachable through a direct move from some node, and that all nodes are interconnected. While a lot of processing can be done offline, the complexity of the online search phase increases exponentially with an increase in the amount of nodes.

3.4 Trajectory tracking

The problem of trajectory tracking involves the calculation of the appropriate actions to move a system along a known path. Since the desired next state is known, there is no navigation or planning concerned. Trajectory tracking involves inverse kinematics computation, including system dynamics such as inertia and momentum. The simplest models implement Aristotelian physics where movement is in direct relation to commanded force applied.

3.5 Obstacle avoidance

In a dynamic environment obstacles may move into the robot's path. In extreme cases this can invalidate the planned path, demanding the calculation of a new path. In less extreme cases it would necessitate only minor diversions from the planned path.

Such evasive manoeuvres can be handled through local navigation. This involves a trajectory independent control policy operating on local conditions only. Control inputs would typically include the distance from the manipulator segments to the surrounding obstacles.

3.6 Concluding remarks

There exist different problems regarding the control of robotic manipulators. This project focuses on the problem of path planning in an environment with static obstacles. The approaches generally followed to solve this problem either require intensive computation every time the target or the environment changes (potential field) or they require online processing. Online processing increases exponentially with the increase in DOF (Q-learning/ant-algorithms) or the increase in complexity of the environment (probabilistic roadmap).

In this project neuro-fuzzy techniques are applied to the path planning problem. Variations of the methods discussed in section 3.3 will be combined. A function resembling the potential field is constructed in state space through the use of reinforcement learning. Finally a notion from the roadmap approach is added to achieve viable performance.

Chapter 4
Evaluation Platform

In this chapter the implementation of a target application for the evaluation of neuro-fuzzy techniques is discussed. After the requirements for such a platform are given, a control problem is proposed and the associated problem space is discussed.

4.1 Requirements

Since the aim of the project is the evaluation of techniques for intelligent control, the specific application is not important. What is important is having a system which offers a sufficient amount of nonlinear complexity to require intelligent control, but at the same time is simple enough to be implemented and evaluated on a PC.

4.1.1 Embodiment

It is generally accepted that embodiment of the control system is an important step towards the emergence of true intelligence. The control system or agent is said to be embodied in the environment when the agent and environment have the capacity to mutually perturb each other's states. In other words, the agent should have a body through which it can interact with its environment. Effectors or actuators are used to influence the environment or the control system's relation to the environment. Sensors are used for feedback or for measuring or observing the changing state of the environment. This creates a closed loop through which the system can explore its environment and discover appropriate solutions.

4.1.2 Simulation

Embodiment does not imply that the agent have to operate in the physical world. An agent might as well be embodied in a virtual environment or a simulation. The advantages of a simulated body in a virtual environment are manifold.

Simulations allow for rapid development of robotic systems. It overcomes the mechanical and financial complications of building, powering and modifying physical robots. Additionally, simulations can usually be executed at much higher speeds than real-world control. They can also easily be set to required scenarios for re-runs, training and evaluation. Through proper design, the amount of segments of the manipulator can also simply be set as a parameter.

There is one disadvantage of using only a simulated environment. No simulation can fully encapsulate the complexity and richness of the real world. Even the best models can only approximate physical dynamics, drag, friction and wear. If a system is intended to eventually operate in the real world, it should be trained in the real world to ensure optimal adaptation.

However, the work done in this project is for theoretical purposes and therefore the physical laws governing the environment are not critical. Additionally, the simulation does not include a hand or gripper since the objective of this project is that of path planning. A simulated environment is an ideal evaluation platform.

4.2 Proposal

A target application offering the complexity required for research in intelligent control is proposed to serve as evaluation platform. Aspects of simulating the target application will be discussed here. It involves the control problem, the virtual environment, system dynamics and type of feedback available.

4.2.1 Control problem

The control of a robotic manipulator was chosen for the complex real-world problem it poses. As can be seen in Figure 4.1 the robotic manipulator control is faced with an ill-posed problem of mapping tip coordinates to joint angles. In an obstructed environment this becomes a highly nonlinear problem.

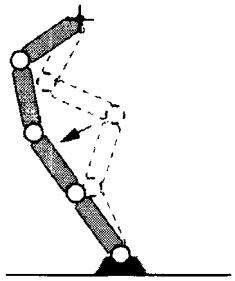

Figure 4.1 *Ill-posed problem of mapping tip to angles*
There is no unique mapping from target position coordinate to the joint angles required.

The multi-segment 2D robotic manipulator was favoured for its simple kinematics and ability to be represented visually. As mentioned, the aim of the controller is to steer the manipulator around obstacles to reach a given target with its tip.

4.2.2 Virtual environment

A 2D discrete world simulation as shown in Figure 4.2 supplies the required problem domain complexity, but still allows for simple computation of environmental response and graphic representation.

A grid represents a map of the environment. The cells of the grid can independently be set to form walls or obstacles. The map can be edited, saved and reloaded. The map is intended to be static during operation.

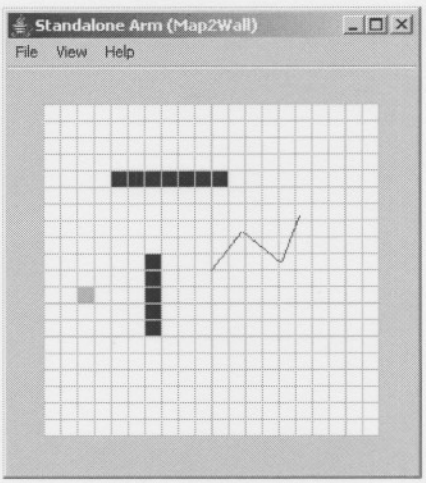

Figure 4.2 *Virtual environment of evaluation platform*
A robotic manipulator in a grid-world environment set up with (black) wall obstacles and a goal cell (grey).

The robotic manipulator is fixed to the grid at its one end. It can be manoeuvred by commanding angular movement of the joints between the segments. The aim is to move the tip of the manipulator to a target co-ordinate. The obstacles prevent the arm from moving through certain positions. No part of the manipulator is allowed to move through a filled cell or outside the edge of the map. Obstacles are rigid and cannot be moved by or moved over by the manipulator.

Refer to Table 1.1 for the evaluation platform interface.

4.2.3 System dynamics

Since the aim of this project is to research intelligent controllers in general, the accuracy of the laws of nature is not critical. In this virtual environment simple Aristotelian dynamics is implemented [41]. This approach is commonly used in computer games and simulations because of the simplicity of calculation.

Although it differs greatly from the real world, the dynamics in such a world is still quite intuitive. Speed of a body is directly proportional to the force exerted on it. When the force is discontinued, the body stops instantly. Collisions are non-elastic and there is no momentum, therefore objects cannot be thrown.

Note that it is possible to use a controller developed for this environment in the real world, by implementing independent low-level controllers to shield the dynamics from the high-level controller. This subject falls outside the scope of this project.

4.2.4 Reinforcement feedback

In a virtual environment basic reinforcement feedback can easily be achieved. By stating the goal of the controller, a reward value can be calculated by the simulator, representing the success of the actions taken. However, this feedback can greatly assist in finding solutions for the control of the manipulator.

The simulator computes a reward value as a combination of the distance the tip is from the target, and the amount of action performed to reach this distance. This combined value will have the effect of reducing the distance travelled as well as the proximity of the target.

4.3 Problem space

The control parameters of the system exist in different spaces with varying dimensions. These parameters and the related spaces are discussed here to prevent possible confusion.

4.3.1 State space

The state space is a set of all the state vectors that a system can assume at any time. Any state vector should be a set of values that describe the internal conditions of the system sufficient enough to allow successful control of the system, based on this value.

In the case of the simulated robotic manipulator the minimal state space would consist of all the possible independent joint angle combinations. With the selected target application the dimensionality of the system can be increased indefinitely by adding more manipulator segments. By significantly increasing the amount of segments the robot can become a hyper-redundant snake-like structure.

For this project a manipulator with three independently controllable segments is utilized. This allows for multiple solutions to exist. See Figure 4.3 for an illustration. For a manipulator with three DOF, the state space is three-dimensional accordingly.

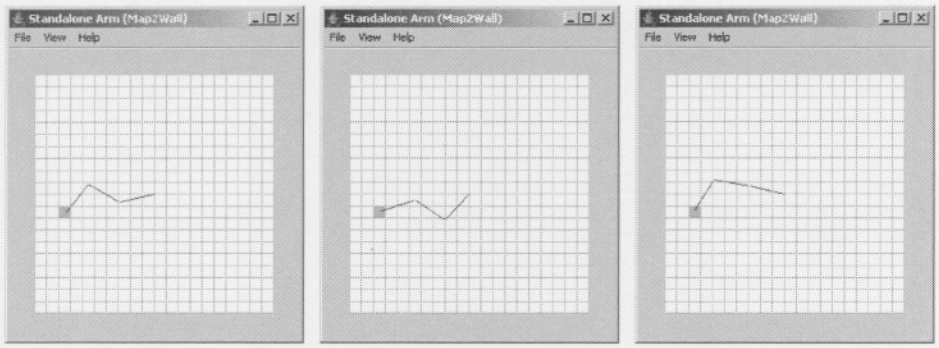

Figure 4.3 *Solution states for multi-segment manipulator*
In a 2D environment with three degrees of freedom an infinite amount of solutions are possible.

4.3.2 Goal surface

The target represents the point in space the manipulator is required to reach with its tip. The posture of the rest of the manipulator is not specified. Therefore a set of states exist which results in the tip reaching the target. This set of states is called the goal surface, which is contained within the state space. A fault tolerance is added by requiring the tip only to reach a target cell in the grid.

The dimension of the goal space is equal to the difference in dimension between the state space and the target. The state space is three-dimensional. Since the manipulator operates in a 2D environment, the target is also two-dimensional. Therefore the goal surface is one-dimensional. This is illustrated in Figure 4.4.

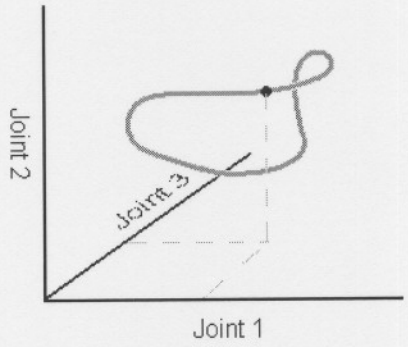

Figure 4.4 *The goal surface in state space*

A one-dimensional surface (or line) exists in the three-dimensional state space, which describes the set of joint angle combinations that places the manipulator tip at the target.

4.3.3 Action space

The action space is defined as the complete set of all possible actions that can be invoked by the controller. One command from the controller results in one action. In many cases the control problem is solved through a discrete action space. This can even be true for adaptive fuzzy model-based control. For instance, 'bang-bang' control has been successfully implemented for inverted pendulum control, where the possible actions that can be taken are either full forward or full backward.

In the case of this simulation the robotic manipulator can be controlled by giving independent commands of angular movement to the manipulator joints. An action is defined as a set of angular movement commands for all the different joints and can be represented as an action vector in action space.

The speed of movement of the joints is scaled so that the joint movement is synchronised to start and stop simultaneously. It is worth noting that since Aristotelian physics is implemented, if there is no collision, any action will result is a change of state equal to the action.

4.3.4 Solution space

Since there are obstacles in the environment which have to be circumvented, in many cases the target cannot be reached through a single action. A solution consists of a combination of actions executed consecutively to allow the tip to reach the target in such cases.

The solution can be represented as a chain of action vectors in state space as shown in Figure 4.5. Since action space is three-dimensional, solution space is nine-dimensional.

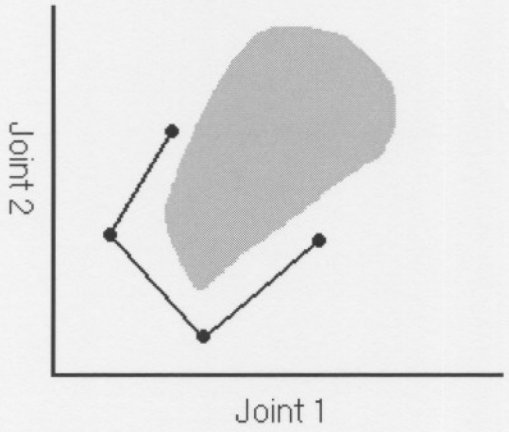

Figure 4.5 *Sequence of actions to reach target*

In this two-dimensional representation of state space, the manipulator needs to move to two intermediate states before a goal state on the opposite side of an obstacle can be reached.

4.4 Concluding remarks

A simulated three-segment robotic manipulator is implemented in a virtual environment. The environment can be preset with obstacles and a goal position. The posture of the manipulator can also be preset. The manipulator can be controlled by any controller through its command interface. The environment returns the manipulator state and a reward value as feedback. This allows the controller to adapt through reinforcement learning.

Chapter 5

Software Design Standards

This chapter discusses the standards followed in the design of the software for this project. It covers aspects concerning system modelling, selection of the development environments and the modular design and interfaces of the simulator and controller.

5.1 Universal Modelling Language

Universal Modelling Language (UML) is a third generation descriptive language, expressing constructs, relationships and data flow of complex systems in a formal, graphic top-down approach.

UML comprises the following elements:

- Objects – autonomous functional units including attributes and behaviour
- Attributes – member data of an object
- Behaviour – member functions or methods of an object, manipulating its attributes to perform specific tasks
- Messages – functional and procedural calls, interrupts and events of communication in and between object
- Responsibilities – the role an object plays in a system, through interfaces and behaviour
- Concurrency – management and synchronisation of separate running threads

A class is the implementation structure or the template for objects. Objects can be instantiated from the class in run-time. The class type or interface defines its accessible members, but it excludes its functionality. Consequently different classes can be of the same type by sharing a common interface. An object's behaviour should be sufficient to collectively accomplish its responsibilities, but doesn't have to define them.

Associations between class types are defined as use cases. Scenarios are instances of such use cases. Objects are symbolised graphically as regular shapes and relationships between them are represented by means of connecting lines. Objects can be related to each other by any of the following relationships:

- Association – definition of links between objects for exchange of messages, manifested in run-time
- Aggregation – one object logically contains another
- Composition – aggregation with explicit ownership and responsibility of object instantiation
- Generalisation – an 'is kind of' relationship with its parent class
- Dependency – when one class depends on another
- Multiplicity – a number denoting the amount of instances of an object (* for undefined)

5.2 Development environment

Java™ and MATLAB® were chosen to be used in synergy for control system prototyping and experimentation. Software development is done in Java™ while MATLAB® is used for testing and evaluation. These two development environments compliment each other with seamless integration of their specialised attributes. The key features of these environments are discussed here.

5.2.1 Java™

Java™ is a modern object-oriented programming language that has become a preferred prototyping language in the engineering community [45]. It is based on C and C++ which is a very popular engineering language. The Java™ language is a stable evolution of these languages. The choice of using Java™ over C++ is based of the added benefits it offers. The major benefits are platform independence, safe memory management and the convenience of rapid prototyping.

5.2.1.1 Platform independence

Java™ was designed with platform independence as its primary objective. Unlike its predecessors Java™ runs interpreted on a virtual machine, making the hardware transparent, allowing the compiled byte-code to be truly platform independent.

All but the most basic functionality are to the language through optional class libraries, allowing giving the language to bear a very small core footprint. The virtual machine is embedded in most PC operation systems, web browsers and development environments, including MATLAB®. Most attractive, is the fact that any class developed in Java™ can easily be instantiated and accessed from a web browser or the MATLAB® user interface.

5.2.1.2 Safe memory management

The most common source of run-time errors in C/C++ is the unsafe accessing of memory. Java™ solves this problem through strict type checking, range checking and built in allocation and de-allocation of memory. There are no pointers in Java™, only references to objects, which are all dynamically created in run-time. When an object is no more referenced, its memory gets reclaimed by the built-in garbage collector. References can only be cast to inherited or parent types, and arrays can only be accessed inside its declared boundaries. This prevents the unsafe accessing of memory.

5.2.1.3 Rapid prototyping

Java™ promotes rapid development of code by introducing the following features.

- omission of header - there is no separate files for class definitions, definition and implementation is done in one place
- implicit dynamic binding and polymorphism – any inherited class can override functions by simply defining a new implementation; every inherited object will automatically be associated with its corresponding methods
- dynamic linking of classes – library classes can be modified without the need of re-compiling the utilizing application
- extremely simple event model – standard editable code is implemented for the handling of user or system generated events
- multithreading – the spawning and synchronizing of different threads is inherent to the language
- extended class library for GUI – libraries of Windows user interface components and utilities are freely available

5.2.2 MATLAB®

MATLAB® is a leading mathematical tool from 'The MathWorks, Inc', optimized for matrix manipulation. Commands are given through coded instructions similar to other programming languages, but unlike most languages, MATLAB® is interpreted. Although MATLAB® script files can be compiled, its appeal lies in its interpreted user interface. With the functionality it offers, it allows for powerful manual data manipulation, fast concept testing and experimentation prior to coding. The support for graphic representation, the available toolboxes and the integration with Java™ make great savings in development time possible.

The modules described in section 5.3 can be instantiated in MATLAB® either as Java™ objects or through any of the available MATLAB® toolboxes. The system architectures discussed in Chapter 6 can then be tested within MATLAB®.

5.3 System design

Code is developed in this project for creating a simulator and for developing an intelligent controller. To improve reusability a modular approach is followed and standard interfaces are defined. An overview of these aspects of the system design is given here. A detailed list of all the classes developed for this project can be found in Appendix I.

5.3.1 Modularity

In this text a module refers to a superset of the object defined in UML. Modules form the specialised high level functional building blocks in a system architecture which is proposed in Chapter 6. The modules inherit interfaces from a common pattern class. This important attribute allows the modules to be interchangeable.

A library of modules implements various neuro-fuzzy control and path-finding techniques. All these modules share the same general interface. This allows for the evaluation and comparison of different techniques with only minor modification of the coordinating code.

5.3.2 Interfaces

Modules represent functional blocks and therefore their interfaces implemented one main data flow function. The patterns for the data flow of the functional blocks in Chapter 6 are listed in Table 5.1.

Table 5.1 *Manipulator interface in virtual environment*

Functional block	Input	Output
controller	state + goal	action
optimiser (evolutionary)	reward	action
optimiser (reinforcement)	action	controller / reward
environment	state + action	new state
evaluator	state + goal	reward
rewarder	state + reward	evaluator

Rather than defining a separate pattern class for each of these functional blocks, a simple, general function class is defined for all of these blocks. The primary function of this class receives a single input vector and returns an output vector.

All the modules inherit from this parent class. It implements a set of generic attributes and behaviours but the primary function is overridden by every module. While the function interface stays the same, its interpretation changes, depending on the functional block it is intended for. The advantage of this approach becomes apparent when it is realised that, for instance, a neural network might be utilised as a variety of different function blocks.

5.3.3 Simulator

The complete robotic manipulator simulation is implemented in Java™. It handles virtual environment obstacle setup, user interface commands and commands from any controller. It calculates manipulator segment positions and goal proximity. The simulator class named 'Arm' inherits from the 'Spirit' class, supplying it wire-frame modelling capabilities, and implements classes as indicated in Figure 5.1. 'Arm' is fully functional and can operate as a stand alone application or it can be instantiated and controlled from MATLAB®.

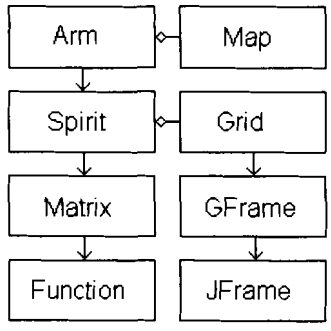

Figure 5.1 *Inheritance and of the manipulator class*
Arm inherits from Spirit which inherits from Matrix. Matrix inherits from Function. Arm implements an environment through Map and Spirit implements Grid, which supplies it with windowing capabilities.

5.3.4 Controller

Various combinations of controller types need to be evaluated. Therefore a general system representation for intelligent agents and controllers is required. By defining generic interfaces for modules in such a system, controllers can be replaced and compared with minimal modifications to the code.

Initially a simple environment will be set up and control will be attempted through MATLAB®'s built-in fuzzy inference system. This will serve as benchmark for alternative control systems.

For more advanced controllers, code is written in Java™. The specific controller inherit from the modullar controller architecture called 'Mocca' and implements different connectionist networks which inherits from 'ConnectNet'. Such an example is given in Figure 5.2. Policies have been implemented through a multi-layer perceptron (MLP), a radial basis function (RBF), a fuzzy feedforward network (FFN) and a nearest neighbourhood clustering (NNC) function. All the developed networks are listed in Appendix I.

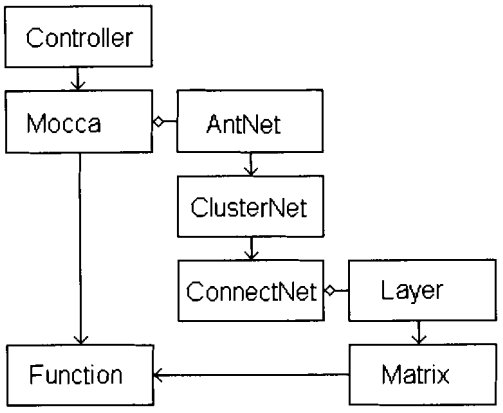

Figure 5.2 *Inheritance of controller class*
The controller inherits from Mocca which is a function. Mocca implements multiple approximators like AntNet shown here, inheriting from ConnectNet through ClusterNet. ConnectNet implements Layers which are extensions of the Matrix class.

A GA class is also developed function although it is not directly used as a controller. The GA is used to explore the simulation for solutions to be used as training data.

5.4 Concluding remarks

A programming standard has been set in this chapter. By following strict standards, development time is reduced and integration and testing is simplified. The UML design paradigm is followed, which is an industrial standard for software design. Standard interfaces and modular design makes it possible to reuse a substantial amount of code.

Chapter 6

System Architecture

An architecture is proposed in this chapter which serves as a framework in which various controllers and systems can be combined in different configurations. It discussed the functional blocks out of which the framework is build and the possible of data flow which can be employed between the functional blocks.

6.1 Controllers

Controllers induce actions on systems to modify their states. Their goal is to modify the states of the systems to reach target values. Control actions of feedback controllers are based on the systems' current states as well as the target values. Classic controllers can only control a single state parameter. Their actions are based on the error between the state parameter and the target parameter. These are linear controllers and they are not well suited for control of nonlinear multi-variable systems. Such systems require intelligent controllers with adaptive nonlinear capabilities to perform the mapping between system state variables and control actions.

Techniques from computational intelligence have been applied to this problem. Neural controllers, fuzzy controllers and neuro-fuzzy hybrids have been developed, capable to adapt to such nonlinear systems. To control more complex systems even more advanced controllers are required. Cognitive controllers are intelligent agents in which a substantial level of cognitive attributes becomes evident.

A cognitive controller should be knowledgeable of the effects of its actions on the environment and its relationship to the environment [46]. It should be able to plan ahead, anticipate, evaluate and optimise future results based on an internal representation of its environment. It should also be able to explore strategies, validate representations, do evaluations and adapt accordingly.

In this chapter a modular controller architecture is proposed. Various cognitive attributes can be observed in this architecture. The architecture consists of multiple functional blocks and the attributes are characterized by distinct patterns of data flow between the functional blocks. Although some attributes are still quite primitive, they bear the potential to develop into strong cognitive processes.

6.2 Framework

The framework of the proposed modular controller architecture consists of three primary units. It is based on the actor-critic design. An 'environment' unit has been added to complete the representation of the system. These units encapsulate a set of six functional blocks as indicated in Figure 6.1.

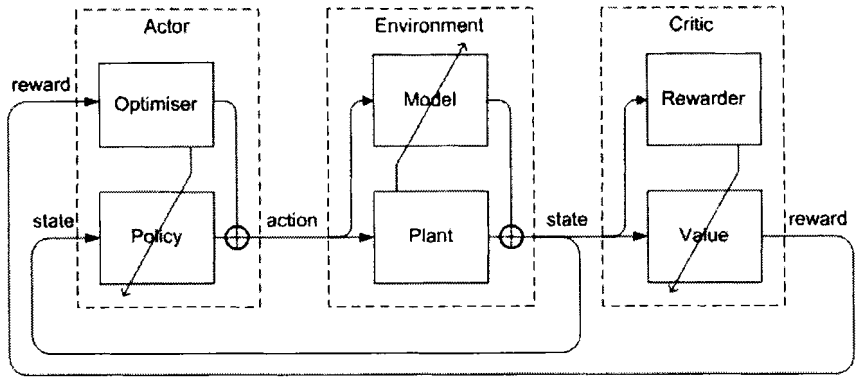

Figure 6.1 *Architecture of a modular control system*
The actor-critic architecture is extended to unify numerous controller architectures, ranging from classical controllers to intelligent agents.

The interfaces of these functional blocks are generic. This allows for different functions, techniques and algorithms to be implemented in modules which can slotted into the framework. In this way different modules can be developed and independently and different combinations of modules can be evaluated with minimal modifications.

The functionality of all these blocks is discussed here.

6.2.1 Actor

The actor houses the controller and additional independent units for exploring the environment and training the controller.

6.2.1.1 Policy

The policy unit is responsible for generating the control commands. It can range from classical controllers to expert system, neural, fuzzy, neuro-fuzzy or even evolutionary system. The policy receives the system state and user-defined goal as inputs and gives control commands as output. In the diagrams given here, the goal is assumes to be encoded as part of the state as an input to the policy.

6.2.1.2 Optimiser

This module distinguishes adaptive controllers. It is responsible for identifying appropriate control actions and it contains the functionality for training and tuning the controller. If training data is available, the optimiser can train the controller offline through supervised learning. If only rewards are available, the controller is trained through reinforcement learning.

Reinforcement learning can be divided into backpropagation based reinforcement, guided random policy reinforcement and 'action evolution' reinforcement [47]. For backpropagation based reinforcement the optimiser applies a reward which is backpropagated through the system. It then receives an action on which training is based. The latter two approached involve meta-heuristic optimisation such as genetic algorithms. For guided random policy reinforcement a controller parameter set is proposed and the reward is evaluated over time as an indication of the controller's fitness. In the case of action evolution reinforcement an action is applied (usually to the model) and the reward received is used as fitness function.

6.2.2 Critic

The critic is responsible for supplying the system with an estimated reward. It does so by means of a function approximator. If an adaptive critic is required, an associated training algorithm is utilised to update the estimator function.

6.2.2.1 Evaluator

Approximation of the expected reward is done through a value function. This function plays a role similar to the cost function commonly used in dynamic programming and the fitness function used by evolutionary algorithms. It can be realised by a Q-value look-up table or some form of universal function approximator. Note that the value function needs to be maximised.

In model-based control the value depends on the change in system state. In model-free control the value is based on the state and proposed action (action-dependent critic).

6.2.2.2 Rewarder

By recording successes and failures, the rewarder can generate an external reward which it uses for training the evaluator. Success and failure conditions are custom functions specific to the problem domain, but a success would typically occur when the state equals the goal. By moving the external reward generation from the environment to the critic, the outputs of the environment can be made independent of the goal and evaluation.

The rewarder implements any reinforcement learning techniques such as DP, Monte Carlo or Q-learning to propagate the reward to the evaluator. TD learning is used to train a neural network based evaluator.

6.2.3 Environment

The environment encapsulates the plant which is the system that is being controlled. It might be a real-world system or an interactive computational system. For model-dependent reinforcement learning systems the environment block is required to seat at least one model of the plant. The model enables the prediction of future states, which can be used for action evaluation.

6.2.3.1 Plant

The online real-time system or process that is being controlled is known as the plant. In the case of a robotic system it is the real-world environment in which the system is operating. Its internal state changes as a result of action commands received as inputs. It may also have some uncontrolled inputs or disturbances.

The ultimate goal of the control system is to enforce actions on the plant to change its state towards or keep it at a given goal state.

6.2.3.2 Fixed model

The fixed model is a working model of the plant. It serves two purposes. For testing, simulation and initial training, the fixed model takes the place of the plant. Alternatively the fixed model is used offline for the same purpose as the adaptive model.

In simulation the controller interacts online with the fixed model and attempts to control it. The advantages of using a model are that reinforcement learning of the controller can proceed in simulated time and that the plant can be set up in certain scenarios for the controller to become familiar with unlikely or dangerous conditions.

The working model receives the plant state and controller actions as input and calculates the expected response or next state as output. The model is not modified by the system.

6.2.3.3 Adaptive model

When a fixed model is not available or when it is not sufficiently accurate, an adaptive model can be implemented and trained online. The purpose of having an accurate model of the plant integrated into the control system is that it allows offline training, planning and verification. The adaptive model is not implemented in action-dependent reinforcement learning system.

The adaptive model is trained with action-response data as it becomes available from the plant. Similar to the fixed model, it takes the plant state and controller actions as input and gives a predicted response as output.

6.3 Data flow

Interaction between the modules of the framework follows predefined data flow patterns. These patterns govern all processes within the control system.

6.3.1 Online processing

While in operation, the controller interacts with the plant. In simulation the controller interacts with the working model of the plant. The controller does not distinguish between controlling the plant and controlling the working model. This is called online processing.

There are two possible modes when the system is online: exploitation and exploration. Selection between these two modes is done in the actor. An untrained system would be biased towards exploration. As the system becomes trained the bias would shift towards exploitation.

6.3.1.1 Exploitation

Policy to Plant (Figure 6.2): The actor follows policy to control the plant. It is a direct 'reflex' reaction without any planning or validation. It can be considered as 'instinct' which can gradually change over time.

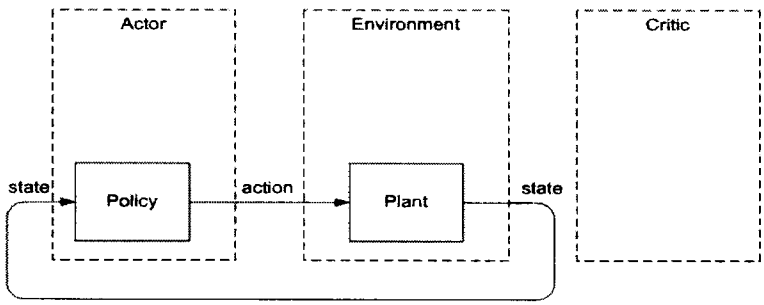

Figure 6.2 *Data flow for exploitation*

6.3.1.2 Exploration

Optimizer to Plant (Figure 6.3): The actor takes actions to get to know the plant. It could be purely random 'playing' or guided 'experimentation' in search of some real or imaginary goal.

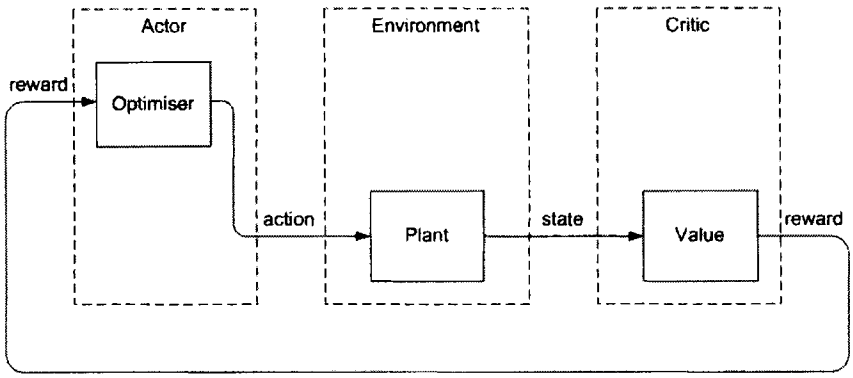

Figure 6.3 *Data flow for exploration*

6.3.2 Offline processing

If the system has a model it can do offline processing. Any processing which does not involve the plant (or the fixed model in the case of simulation) is considered as offline. The purpose of such processes is to reduce training time and increase system integrity.

6.3.2.1 Planning

Optimizer to Model (Figure 6.4): The actor 'plans' actions offline in search of required result before really taking the action. Some kind of 'creativity' can emerge here. This process is further discussed and implemented in Chapter 7.

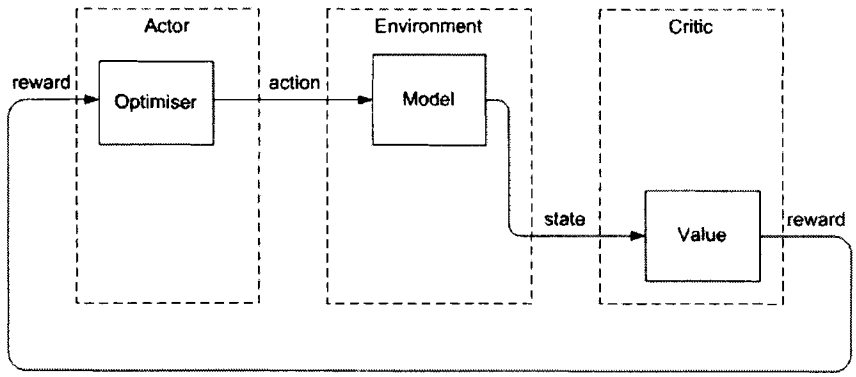

Figure 6.4 *Data flow for planning*

6.3.2.2 Validation

Policy to Model (Figure 6.5): The actor tests its policy offline against its internal model and evaluator to verify that actions which will be taken will lead to desired results. This integrity check leads to improved 'self-confidence'.

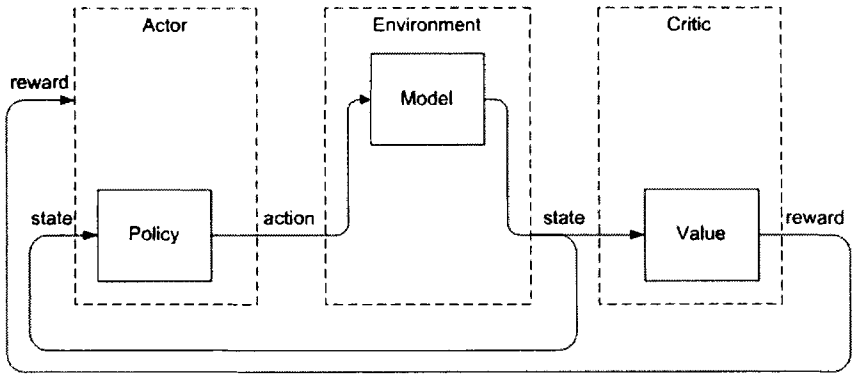

Figure 6.5 *Data flow for validation*

6.3.2.3 Perception

Model to Rewarder (Figure 6.6): When the evaluator is updated based on feedback from the model instead of from the plant, it creates a perception of which the accuracy depends on the model accuracy. Building perceptions can speed up adaptation, but false perceptions can seriously hamper performance.

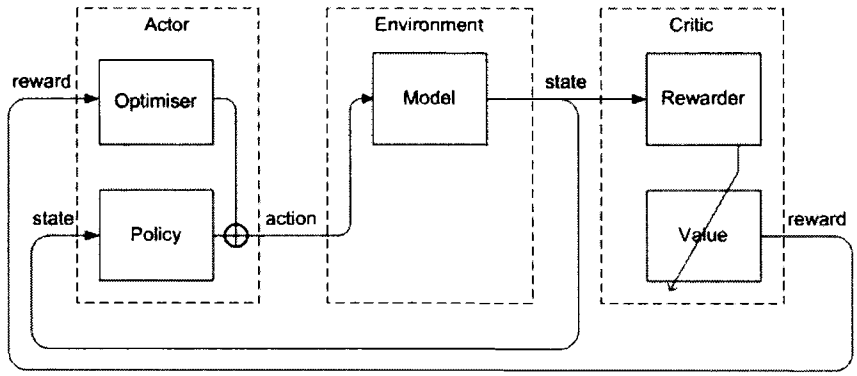

Figure 6.6 *Data flow for perceptions*

6.3.2.4 Emotion

When a system's evaluation is not based on the expected state of the system (model-free control), but instead on unverified values assigned to actions, it can result in 'emotional' actions.

6.3.2.5 Dreaming

The model can be set up to some random state and the controller can then operate on the model as if it was online. Random goals can be assumed and pursued. This can give origin to perceptions, ideas, plans and inventions.

6.3.3 Consciousness

For a system to be cognitive means for it to be aware of its environment and its relation to the environment. It should be able to make assumptions based on previous experience in order to take appropriate actions towards reaching a goal. This requires the system to have sufficient data available of not only the state of the environment but also of its own state. It also requires the system to preserve an internal model of its environment in order to anticipate responses and predict consequences. In order to continually adapt to a dynamically changing environment, this model should be adaptive in the same way as the controller is adaptive.

6.4 Concluding remarks

A versatile modular system architecture has been developed. It consists of a framework, into which any function or function approximator which complies with the functional block interface, can be inserted. The inserted module seamlessly gains access to some high level control attributed through the predefined data flow patterns. This allows for easy comparison of different controller and optimisation techniques.

Chapter 7
Environment Exploration

To improve the autonomy of the robotic manipulator, the system needs to gather data for training its controller automatically. In this chapter an explorer is developed for compiling training data. After selection of an appropriate search mechanism, the choice of the search space representation and the fitness function are discussed. After optimisation, the results are evaluated.

7.1 Search mechanism

Training data is not available for the control of the robotic manipulator. Instead, the control policy has to be derived from a value function indicating the effectiveness of states [13]. The distance from the manipulator tip to the target cannot be used as a value function, since there might be obstacles that prevent the manipulator from reaching the target.

Since a sufficient value function is not available, the environment has to be explored to find solutions for the control problem. In other words, the commands required to move the manipulator tip to the target has to be discovered.

Reinforcement learning could be considered for the estimation the value function, based on a reward received from the system. The reward indicates success if the target is reached or failure in the case of a collision. However, rewarded states are usually preceded by unrewarded states. Due to the delayed rewards and the fact that state space is continuous, the random exploration of reinforcement learning is not very effective.

As an alternative, a global search algorithm can be utilised. The simulation of the manipulator allows for parallel offline search and the distance from the tip to the target can be used as a fitness function. A simple real-valued genetic algorithm (GA) is implemented. A GA is capable of finding feasible solutions in partially continuous search spaces with multiple local maxima. Note that unlike a cost function, the value and fitness function need to be maximised.

The evaluator is requested to find solutions to move the manipulator tip to the target from various random initial manipulator states. The solutions found can then be used for training a

control policy. The chromosome representation of the solutions and the fitness function of the GA will be discussed next, followed by a discussion on the optimisation of the GA.

7.2 Chromosome representation

The GA has to find command solutions for the control of the robotic manipulator. Solutions are encoded into the GA chromosomes. Solutions consist of sequences of actions as discussed in section 4.3. For the training of some controllers the actions are fractured into smaller sections. These aspects of the solutions are discussed next.

7.2.1 Actions

An action is defined as a set of angular movement commands ranging from maximum clockwise to maximum anti-clockwise for every joint. In the case of this three-segment manipulator the action consists of three parameters.

7.2.2 Sequence

It has been shown in Chapter 3 and Chapter 4 that it might be necessary to apply a sequence of actions before the manipulator tip can reach a given target. It is assumed that any arbitrary target can be reached from any initial state within three actions. Since an action consists of three parameters, a solution requires nine parameters. Therefore, the explorer's GA chromosomes also have nine genes.

When moving around obstacles, it might be required that the tip of the manipulator sometimes move away from the target first, to circumvent the obstacles as illustrated in Figure 4.5. The solution, however allows for three actions in order to move past such intermediate states. Consequently the fitness of a chromosome will only be evaluated after completion of the sequence.

7.2.3 Fracturing

For the training of some controllers, pre-training transformations of the solution data is required. One such transformation is to break the actions of the solutions up into multiple smaller actions. By executing these actions, a large set of intermediate state-action training data pairs can be created for every solution found by the GA. The implementation of this transformation will be discussed further in Chapter 8.

7.3 Fitness function

For the best control, both the distance from the tip to the target and the amount of movement has to be taken into account. This is to prevent the manipulator from making unnecessary moves. The fitness function has to be set up accordingly. The distance and the efforts of movement are optimised in this section before they are combined to construct a final fitness function.

The Euclidian distance from the tip to the target is not a good estimate of the distance, since there might be obstacles in the way. The Manhattan distance takes obstacles into account, but since a multi-segment manipulator is concerned, this is still not an accurate account of the 'reachability' of the target. This is explained in Figure 7.1. However, since GAs execute a global search, this approximation is adequate.

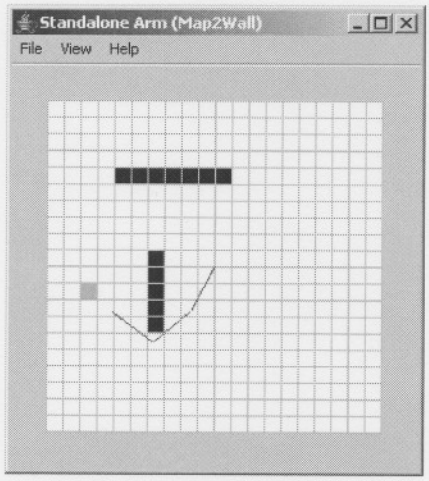

Figure 7.1 *Reachability*
Although the tip is close to the target in geometric distance, it cannot be reached along this path. In state space, this state is not close to the target.

7.3.1 Manhattan distance

The Manhattan distance between two points in space is defined as the sum of the independent differences in every dimension. In a grid-world with obstacles the Manhattan distance is modified to indicate the shortest path between two points around the obstacles. This is demonstrated in Figure 7.2. The Manhattan distance is calculated by flooding the open space, assigning distances to cells from the target cell outwards.

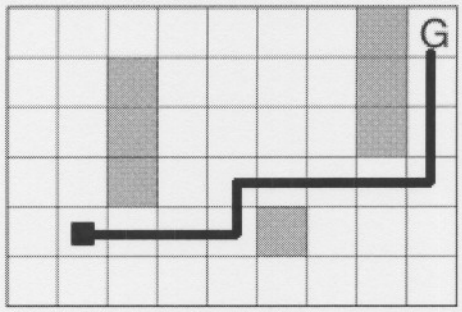

Figure 7.2 *Manhattan path with obstacle circumvention*
The Manhattan distance in a grid-world with obstacles measures the amount of cells in the shortest path between two points [13].

7.3.2 Manhattan smoothing

The grid-world Manhattan distance is a discrete function which implies a multiple-step function in the manipulator's continuous state space. Since GAs rely on gradual improvements, this discrete function hampered the performance of the GA and its output was somewhat erratic. The discrete distance function is replaced by a continuous smoothing function transformation as demonstrated in Figure 7.3.

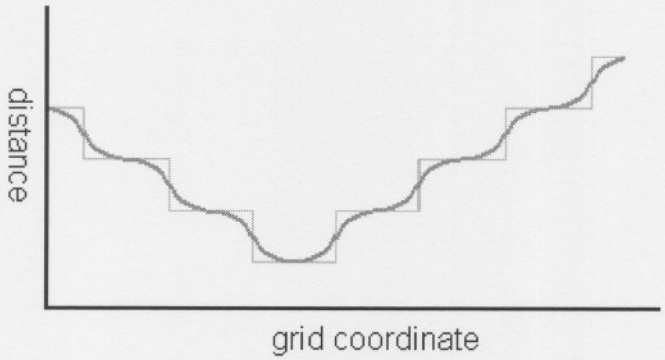

Figure 7.3 *Smoothing of the Manhattan distance*
By interpolating the values of surrounding cells, a continuous estimate of the distance is constructed.

This modification allowed a better convergence of solutions generated by the GA.

7.3.3 Effort of movement

Since the target can be reached through different sequences of actions as Figure 7.4 illustrates, the sequence requiring the least effort should be selected. The total absolute value of the joint commands is used as an indication of the effort.

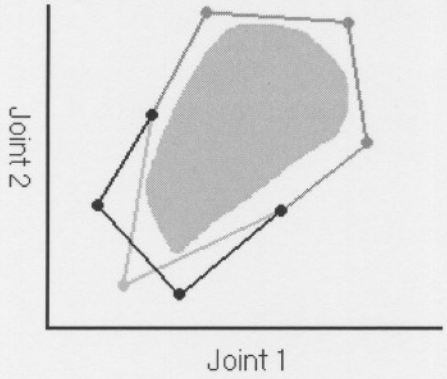

Figure 7.4 *Alternative action sequences*

Infinite possible sequences exist for reaching the goal state.

Still, the target can be reached through different sequences with the same effort. By allocating a greater cost to actions later in a sequence, the GA is biased to find a 'best first' solution effort as indicated in Figure 7.5. Reaching states closer to the target earlier in a sequence is considered as better control policy. In addition, this also attempts to prevent the fitness function from having multiple optimal solutions.

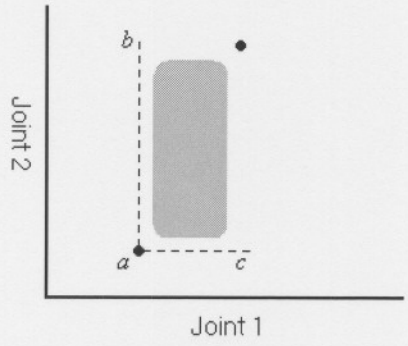

Figure 7.5 *Early effective acting*

Moving from a to the goal on the opposite side of the obstacle, will require two actions. Moving to b holds a greater advantage earlier in the sequence.

According to the same reasoning, a greater cost is allocated to movement of joints closer to the base of the manipulator. The final implemented calculation of effort is given in (7.1)

$$effort = \sum_{i,j=1}^{3} (4-i).j.\left|g_{ij}\right| \tag{7.1}$$

where *effort* is the weighted sum of actions proposed by any chromosomes of the genetic algorithm and g_{ij} is the gene in the chromosomes of the genetic algorithm, representing the suggested command for manipulator joint i for action j in the sequence of actions.

7.3.4 Combining distance and effort

The fitness function consists of a combination of the distance from the tip to the target and the total efforts of the actions taken. The coefficients of the function were chosen heuristically to balance the effort taken and the distance achieved. The final implemented fitness function is defined in (7.2).

$$fitness = \frac{1}{effort + 8.dist^2} \tag{7.2}$$

It is conceivable that conditions can exist for which there are different optimal solutions, e.g. two ways around an obstacle with equal distance to travel. Note that this is not a consequence of inaccurate modelling and is therefore considered as a problem to be resolved by the controller.

7.4 Optimisation

In this section some optimisation schemes are applied to the GA. It is optimised in for two reasons. To improve execution time, a simple parent selection scheme is implemented. To improve the consistency of solutions found by the GA, a range of schemes are implemented. These optimisation schemes have not been thoroughly evaluated in this project.

7.4.1 Parent selection

Parent selection is done with a normally distributed random selection over a fixed-sized fittest part the population. The rest of the population is replaced by offspring in every generation.

A sorting algorithm has been optimised for this parent selection scheme. Since the two portions only need to be separated, a relaxed version of the quicksort algorithm is implemented. This divide-and-conquer technique is well suited for the separation problem. Furthermore, the ineffectiveness of the quicksort algorithm to sort very small sections is complimented by the tolerance of GAs, and low scale sorting can simply be abandoned.

7.4.2 Solution consistency

The GA produces varying solutions for the same conditions in consecutive trials. This is referred to as the inconsistency of the solutions. Three contributing factors for this inconsistency are put forward. Firstly, the inverse kinematics of a multi-segment manipulator is an ill-posed problem. Secondly, GAs cannot guarantee finding the optimal solution. Thirdly, the multi-variable fitness function creates ambiguity in the solutions as described in paragraph 7.3.

Using such conflicting results as training data can seriously hamper the performance of the controllers. Attempts are made here to improve the chance of consistently discovering the optimal solution in a fitness function with multiple local minima.

Winner mutation

While mutation usually occurs at random points in the population with a low probability, a mutant copy of the winner is also inserted into every new generation. This combines evolutionary strategies with GAs and ensures exploration of the neighbourhood of the winner.

Multiple trials

As a quick alternative to a multi-populations GA, the GA is executed several times. The best solution over all these trials is used. Similar to the multi-population GA, every trial can produce a different solution. This reduces the chance of accepting sub-optimal solutions trapped in local maxima.

Solution seeding

In every consecutive trial for multiple trials, the best solution of the previous trial is inserted into the new population. This mimics the cross fertilization of the multi-population GA and adds a sense of memory seen in other meta-heuristic search algorithms, allowing the system to build upon existing solutions.

Randomized seeding

Random new chromosomes are continuously inserted into the population. This decreases the probability of the population getting stuck in local minima by always reintroducing new genes.

The specifications of the GA finally implemented are listed in Table 7.1.

Table 7.1 *Specification for explorer GA*

Property	Value
Population size	100
Chromosome size	9
Persistence	40%
Procreate	40%
Mutation	10 %
Recreate	10%
Timeout	Stagnant for 30 generations
Trials	5

7.5 Evaluation of results

The GA is successful at finding feasible solutions for problems requiring action sequences. Figure 7.6 shows the improvement of the GA. After the attempts to reduce inconsistency, the GA still produces greatly varying solutions for the same test conditions as indicated in Figure 7.7.

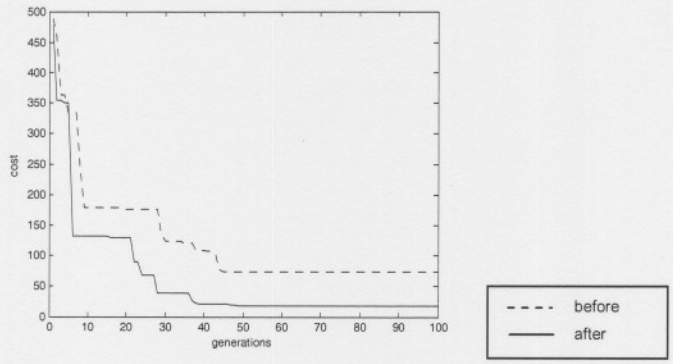

Figure 7.6 *GA optimization*

The population cost over 100 generations for solving a test problem, before and after GA optimisation. Cost is the inverse of the fitness. Note that the witnessed improvement is not always true and varies between trials.

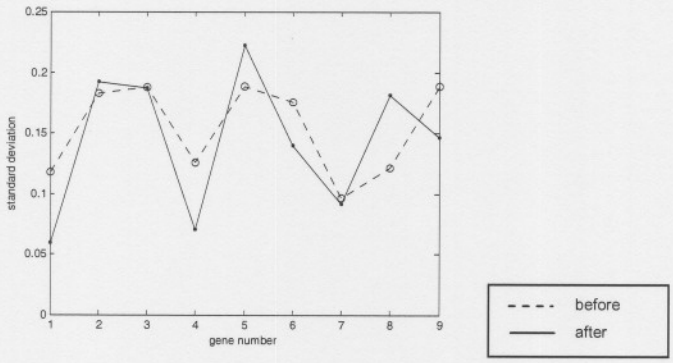

Figure 7.7 *GA results for sample solution*

The standard deviation in the chromosomes of the solutions found, measured over 100 trials, before and after GA optimisation. Note that there is no clear improvement.

7.6 Concluding remarks

A genetic algorithm is implemented for exploring the environment to find solutions for the robotic manipulator control problem. The GA is successful at finding good solutions to manipulator states requiring sequences of actions. However, the solutions found show a high level of variance which produces contradictions in training data. This is expected to create a problem for controller training. This problem is not unique to this system and it can be expected from the controller to be able to handle this condition.

Chapter 8

Controller Implementation

In this chapter various controllers are evaluated at the hand of three different approaches to the robotic manipulator control problem namely local control, global control and evaluator based control. Optimisation techniques are discussed and the resulting controllers are compared with regards to training errors and controller performance.

It is suggested that different control strategies be followed when the manipulator tip is in the close vicinity of the target and when considerable manoeuvring of the manipulator is still required. A global and a local control approach are derived accordingly. This correlates with the ballistic and adjusting movements of human arm control as discussed in Chapter 3. Finally a value based control strategy is implemented.

Different fuzzy, neural and neuro-fuzzy controllers are applied to all three of the mentioned approaches. The modular design of the controllers and the generic framework of the system architecture allow for easy reconfiguration for the different approaches as well as for effortless substitution of the different controllers for evaluation.

An overview of these approaches is presented in Table 8.1, Figure 8.1 and the following paragraph.

Table 8.1 *Control approaches*

Approach	Approximator type	Input space	Outputs
Local control	FLC, MLP, FFN	Manipulator orientation relative to target	Joint angle movement command
Global control	MLP, RBF, NNC	Absolute manipulator orientation	Joint angle movement command
Evaluator based control	NNC, fuzzy Ant-Q	Absolute manipulator orientation	State value approximation

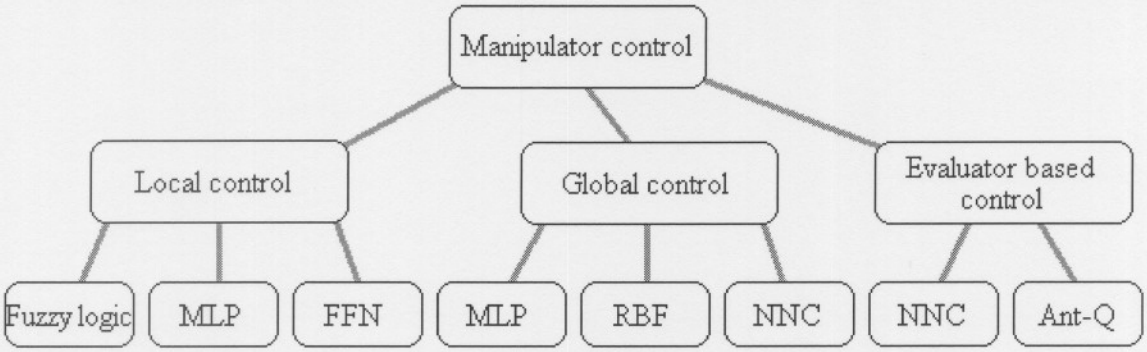

Figure 8.1 *Manipulator controllers*

Local control, global control and critic based control and their respective implementations.

A local controller controls the robotic manipulator in the close vicinity of the target. For this type of control a fuzzy logic system, a multi-layer perceptron (MLP) and a backpropagation learning fuzzy feedforward network (FFN) are suggested. For global control again a MLP, a radial basis function (RBF) and a nearest neighbourhood clustering (NNC) function are implemented for evaluation. Two additional modifications are made to the latter in an attempt to improve its performance.

Finally an alternative approach is proposed in which control is based on the best action found in a state evaluator function. This approach is implemented through a NNC function and through a modified fuzzy Ant-Q system. Various modifications are made to both these systems to improve performance.

Although fuzzy rules cannot easily be derived from MLPs, implementing and testing them on the problem can provide an indication of whether a neuro-fuzzy mapping between inputs and outputs is likely to be found. MLPs are more flexible and can often solve problems with fewer neurons. For this reason MLPs will also be implemented.

8.1 Local control

When the tip of the robotic manipulator is in close proximity to the target and there are no obstacles blocking the optimal transition states it is suggested that a control strategy is applied which is dependent only on the state of the manipulator relative to the target. By disregarding global conditions the controller does not need to adapt to changing environments.

This controller can be used for fine adjustment of the manipulator after the global controller has moved it into the vicinity of the target, thereby reducing the precision required from the global controller.

Local control could be activated when the tip reaches a threshold distance from the target. It is assumed that the global controller puts the manipulator in a position which gives it unobstructed direct access to the target as shown in Figure 8.2. The error can then simply be calculated as the Euclidean distance between the tip and the target.

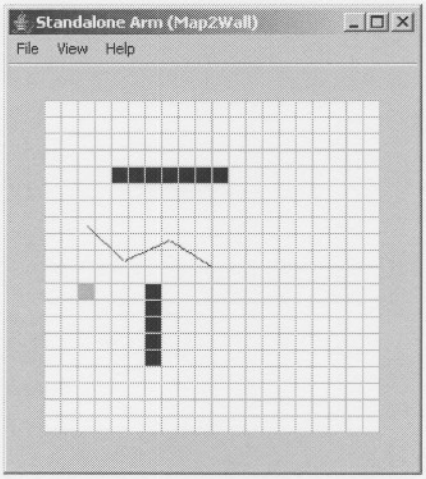

Figure 8.2 *Manipulator with unobstructed access to the target*
Unobstructed direct access to the target implies that all the joint angles can be freely changed to move the tip closer to the target with no obstacles in its path.

8.1.1 Fuzzy logic controller

A set of predefined fuzzy rules can be derived according to which the joint angle command for every joint can be calculated. It is suggested that each joint of the manipulator be controlled independently with limited inputs.

Method

Assuming the manipulator has unobstructed access to the target, the possibility is envisaged to derive feasible actions for every joint individually, by taking into account only the relative position of the next joint, the tip and the target.

By normalising the input data through a transform, the interpretation is made equivalent for all joints, allowing for the same controller to be used for calculation of control of all the joints. The transformation involves rotation and scaling of the data around the respective joint to put the

target at unity distance. By transforming the data to a polar coordinate system the rules can be simplified. The transformed input data is illustrated in Figure 8.3. The simple fuzzy rule base suggested in Table 8.2 has been implemented and tested in MATLAB®'s fuzzy inference system.

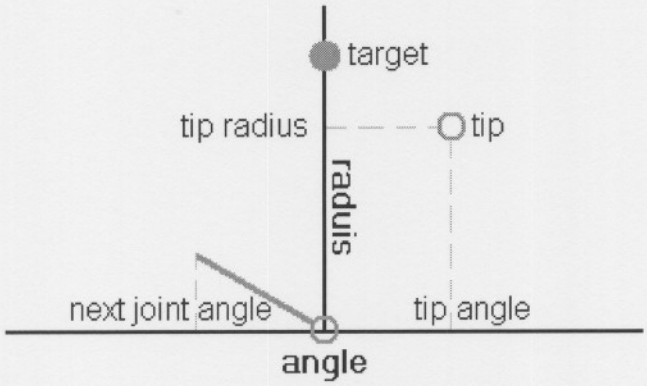

Figure 8.3 *Individual joint control input transformation*
Inputs required for deriving feasible joint movement commands are angle to the next joint, angle and distance to the tip and distance to the target. The posture of the rest of the manipulator is irrelevant to the movement of the current joint.

Table 8.2 *Fuzzy rules set up for individual joint control*

Next joint angle	Tip angle	Tip radius	Output
$< \pi/2$	< 0	< 0.8	-1
$< \pi/2$	< 0	> 0.8	+1
$< \pi/2$	> 0	< 0.8	-1
$< \pi/2$	> 0	> 0.8	+1
$> \pi/2$	< 0	< 0.8	-1
$> \pi/2$	< 0	> 0.8	-1
$> \pi/2$	> 0	< 0.8	-1
$> \pi/2$	> 0	> 0.8	+1

Results

Even with the suggested simplifications implemented, satisfying control could not be achieved with the manually set up rules. It seems that feasible manipulator control requires a significant amount of rules. Setting up such a set of rules becomes unfeasibly complicated.

Controller overloading might be worth developing for a hyper-redundant snake-like manipulator but for a three segment manipulator the complexity of overloading would outweigh any savings. Since the focus of this study is to develop a system that learns to control a three segment manipulator, further manual composition of fuzzy rules were abandoned. By automating the process of creating training data, more scenarios can be explored and a better mapping between states and actions can be created.

8.1.2 MLP local controller

A multi-layer perceptron (MLP) backpropagation learning neural network is used to learn the mapping between the manipulator's local states and appropriate actions to be taken. Good state-action training data pairs are generated automatically by exploring the action space offline on the model as described in Chapter 4.

Method

Assuming unobstructed access to the target, an obstacle free environment is considered as shown in Figure 8.4. This implies that the target can be reached through a single action. A simple GA is used to optimise the three parameters for the three joint commands. The fitness function is chosen to minimise the Euclidean distance between the tip and the target with the minimum action. Less weight is assigned to joints closer to the tip, implying the assumption that movement of these joints require less effort. For that reason, in the scenario depicted in Figure 8.4, it is preferred to adjust the joint closest to the target.

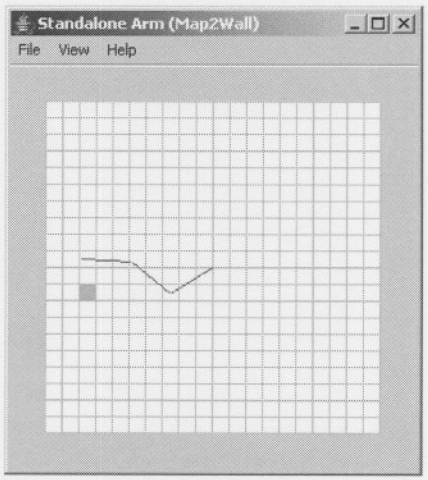

Figure 8.4 *Manipulator in obstacle free environment*
The manipulator's position is fixed at the centre of the grid-world. This is called the first joint. All the joint angles can be adjusted in order to move the tip of the manipulator to the goal cell. The

Training data consists of the local state space of the manipulator as inputs as listed in

Table 8.3 and the command parameters proposed by the GA as outputs. The specifications of the controller are listed in Table 8.4. After gathering sufficient training data the controller is trained through a standard backpropagation algorithm [14].

Table 8.3 *Input specifications for local controller*

Inputs	Range
1st segment angle	$[-\pi ; +\pi]$
2nd joint coordinates (x & y)	[-10 ; +10]
Tip coordinates (x & y)	[-10 ; +10]
Target distance	[0 ; +10]

Table 8.4 *Specifications for MLP local controller*

Property	Value
Outputs (range)	3 joint commands $[-\pi/100 ; +\pi/100]$
Hidden activation	Sigmoidal, B = 1.0
Output activation	Sigmoidal, B = 0.05
Training ratio	0.04

Results

The MLP controller was initially tested on a limited amount of training data. As indicated in Figure 8.5, with 5 hidden neurons the 5 training samples could be learnt successfully. However, the evaluation of the network with test data which was not part of the training data set, indicates an eventual increase in the error. This signifies overtraining, at which point training should be stopped [11].

To achieve feasible performance, sufficient training data must be added to span the whole input space. By adding significant new data, more neurons can be required. Insufficient neurons will prevent the network from achieving the desired training error. However, increasing the amount of neurons beyond requirement, increases the chance of overtraining. To prevent overtraining, a high data-to-neurons ratio should be maintained. Consequently, enough minimal neurons and ample training data should be added to improve generalization.

In Figure 8.6 neurons were added in two hidden layers to improve the controller performance. While there is enough training samples to prevent overtraining, a satisfactory error level is not achieved.

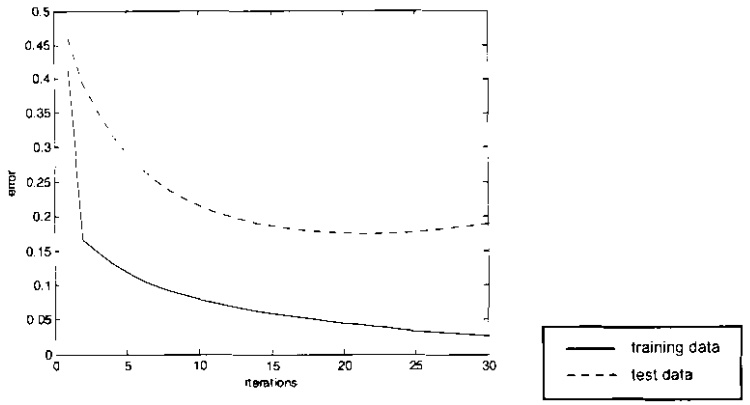

Figure 8.5 *Small MLP local controller trained with 5 samples*
Training error for MLP network with 5 hidden layer neurons. Overtraining commences after about 20 training iterations when the test data error starts to increase.

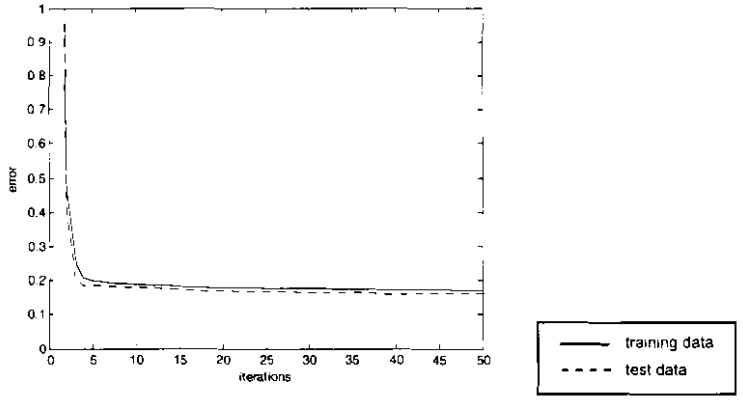

Figure 8.6 *Medium sized MLP local controller trained with 50 samples*
Training error for MLPnetwork with 20 first hidden layer neurons and 12 second hidden layer neurons. The training data error and the test data error show no overtraining divergence.

The trained system is capable of producing feasible results only in very close proximity to the target. With a further increase in network size the training data error can be improved, but not without overtraining as Figure 8.7 indicates.

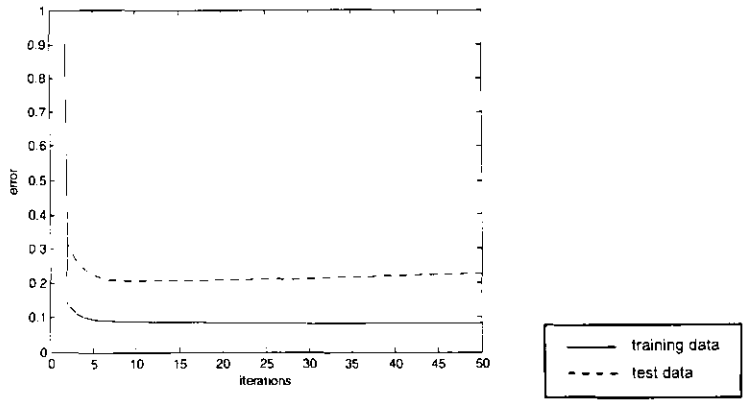

Figure 8.7 *Large MLP local controller trained with 50 samples*
Training error for MLP network with 100 first hidden layer neurons and 60 second hidden layer neurons. The test data error is significantly higher than the training data error.

Due to inconsistent solutions to similar states proposed by the GA when the tip is further away from the target, contradictions occur in the training data, resulting in unsatisfactory controller outputs.

8.1.3 FFN local controller

The fuzzy feedforward network (FFN) with backpropagation learning can be used to replace neural networks. The major disadvantage of neural networks like the MLP used in section 8.1.2 is their 'black box' property which is described in section 2.3.1. Strategies implemented by neural networks can easily become complicated and difficult to interpret, which deems them unsuitable for many control applications. In many cases this problem can be solved by implementing adaptive fuzzy controllers.

Method

By representing a fuzzy system as a three layer FFN, fuzzy membership and rules can be derived by training the network through a backpropagation algorithm [22]. Specifications for the controller are given in Table 8.5. The training data is exactly the same as for the MLP local controller in the previous section.

Table 8.5 *Specifications for FFN local controller*

Property	Value
Inputs	See Table 8.3
Outputs (range)	3 joint commands [$-\pi/100$; $+\pi/100$]
Hidden activation	Gaussian, B = 1.0
Output activation	Normalised linear
Training ratio	0.1

Results

The fuzzy controller was tested first with only 5 training samples. With 5 nodes the system could learn the training data successfully, but due to the lack of sufficient training data, it shows poor generalization. This is evident from the high error for test data in Figure 8.8.

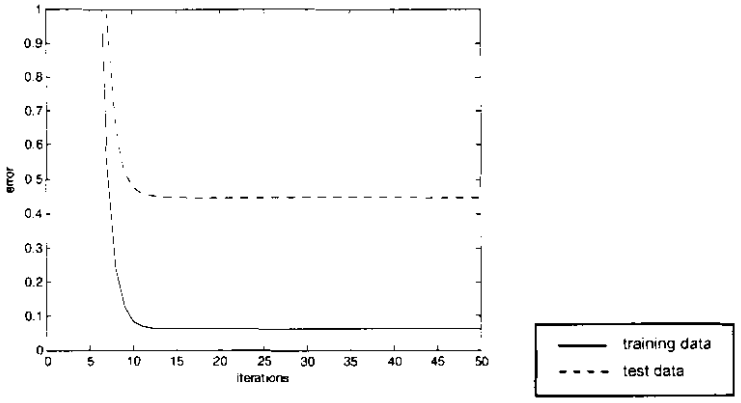

Figure 8.8 *Small FFN local controller trained with 5 samples*
Training error for adaptive FFN with 5 hidden layer nodes. The training data is learnt rapidly. The test data error stays high.

Generalization is improved by adding training data. The same performance as with the MLP controller can be achieved by adding sufficient nodes to the fuzzy controller network. In Figure 8.9 a system with 30 nodes is trained with 50 samples.

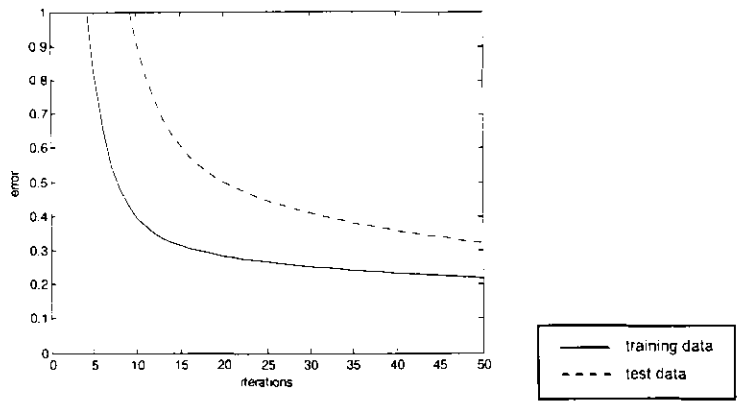

Figure 8.9 *Medium sized fuzzy local controller trained with 50 samples*
Training error for adaptive FFN with 40 hidden layer nodes. The test data error converges with the training data.

As for the MLP controller, the fuzzy controller only provides acceptable results when the tip of the manipulator is close to the target. As for the MLP controller adding more nodes to the network only improves the training data error as Figure 8.10 indicates.

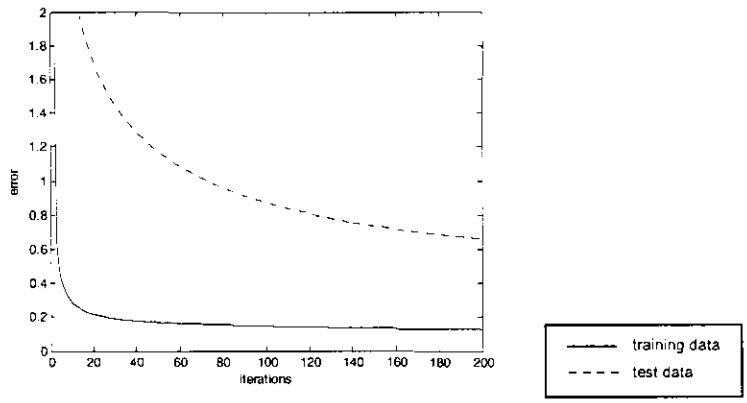

Figure 8.10 *Large fuzzy local controller trained with 500 samples*
Training error for adaptive FFN with 100 hidden layer nodes. The increased amount of hidden nodes reduces only the training data error.

8.2 Global control

In order to solve the problem of global control, the controller has to deduct a control policy for the complete environment. Control is based on the manipulator's global state. This implies the abandonment of the local state transformation and the use of absolute state descriptions. The manipulator state can be represented in terms of joint position coordinates or joint angles. In two-dimensional space as is the case here, joint position coordinates equates to two parameters per joint while an angular description only require one parameter.

The disadvantage of angular representation is the inherent discontinuity between 360 and zero degrees. While control at these two instances should be essentially the same, these states are located at opposite ends of the state space. Therefore outputs for these states have to be declared explicitly. However, due to the high level of discontinuity expected for control in an environment with obstacles, the added complexity is negligible. Joint angle state representation is employed since it significantly reduces the amount of inputs to the controller.

As with the local controller the generation of training data is automated with the use of a GA exploring the system model. Noting that in an environment with obstacles a solution might not be possible with only one action, a sequence of actions is allowed. Three consecutive actions with three joint angle commands each are assumed to be sufficient. The GA is accordingly extended to search a nine-dimensional solution space representing three actions. Due to possible obstacles, the Euclidean distance can no longer be used for calculation of the solution fitness. The Manhattan distance as described in paragraph 7.3.1 is used instead. Apart from this difference, the same equation for fitness as for the local controller is utilised.

Although the GA is allowed to return solutions consisting of a sequence of actions, there remains a great deal of redundancy in this data regarding training of a controller. As the manipulator is taken through a sequence of states many intermediate states may be shared among solutions to different starting conditions as is illustrated in Figure 8.11. The only data which is required to be learnt by the controller is the next action to be taken. By removing this redundancy in the trained actions, the amount of conflict in the training data can be reduced.

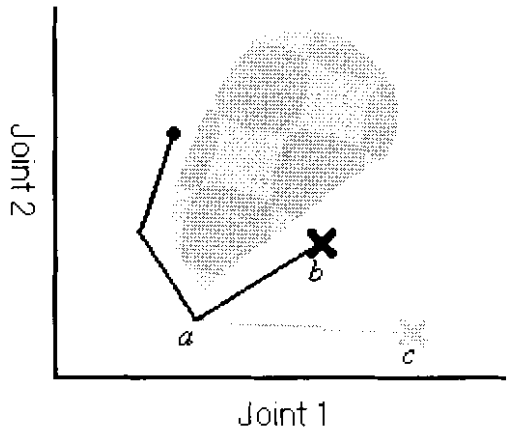

Figure 8.11 *Shared intermediate states*

Different starting states b and c shared an intermediate state a and the consecutive states and actions taken.

In addition to the next action for the current state, the solutions provided by the GA can produce a set of intermediate states with their associated action.

8.2.1 MLP global controller

An MLP backpropagation learning neural network, very similar to the one used for local control, is implemented. An attempt is made to create a mapping between global manipulator states and appropriate next actions.

Method

An environment is set up with fixed obstacles and a target as shown in Figure 8.12. The GA is used to generate training data as discussed in paragraph 8.2. It consists of the global state space of the manipulator as inputs and the next action commands as outputs. Gathering training data is a time consuming process. This is done only once and the data is saved. Future controllers can then make use of this pre-generated training data. The specifications of the controller are listed in Table 8.6. The controller is trained through backpropagation.

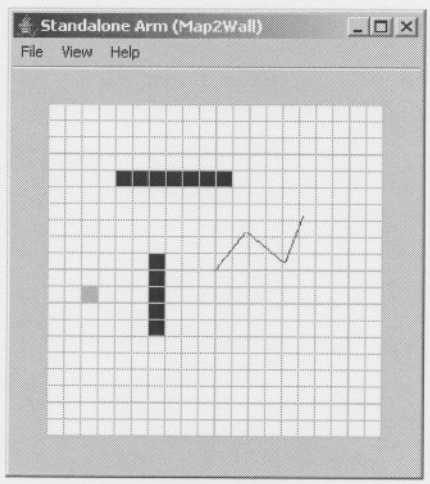

Figure 8.12 *Manipulator in environment with obstacles*

The manipulator is fixed at the centre of the grid-world. The first segment and joint angles can be adjusted in order to move the tip of the manipulator to the goal cell (grey) in the left side of the grid. The segments of the manipulator cannot pass over the (black) obstacles.

Table 8.6 *Specifications for MLP global controller*

Property	Value
Inputs (range)	3 joint angles [$-\pi$; $+\pi$]
Outputs (range)	3 joint commands [$-\pi/100$; $+\pi/100$]
Hidden activation	Sigmoidal, B = 0.05
Output activation	Sigmoidal, B = 0.05
Training ratio	0.04

Results

First the MLP controller was tested with a limited amount of training data. A network with 5 hidden neurons quickly overtrained on the 5 training samples, reducing the training data error and increasing the test data error as can be seen in Figure 8.13.

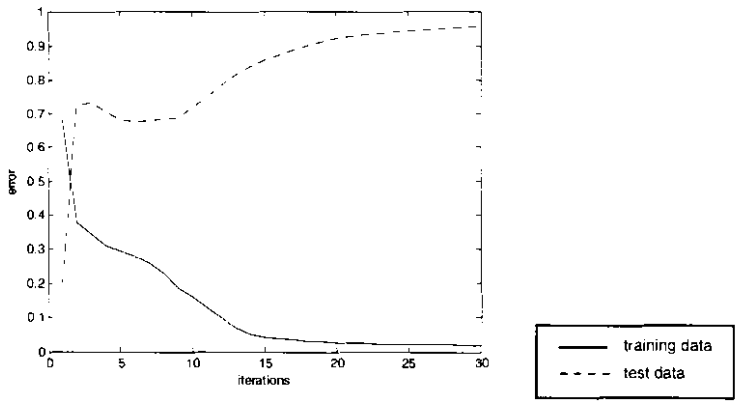

Figure 8.13 *Small MLP global controller trained with 5 samples*
Training error for MLP network with 5 hidden layer neurons. Quick overtraining of the data is clearly evident from the decreasing training data error and the increasing test data error.

To prevent overtraining, training data is added. Figure 8.14 shows the results for a network where neurons have been added in two hidden layers. With 50 training samples overtraining occurs before a satisfactory training error is achieved. By further increasing the amount of training data the test data error can be reduced, but the training data increases significantly as Figure 8.15 indicates. Even after increasing the network to include 50 neurons in a first hidden layer and 30 in a second the training data error could not be feasibly reduced.

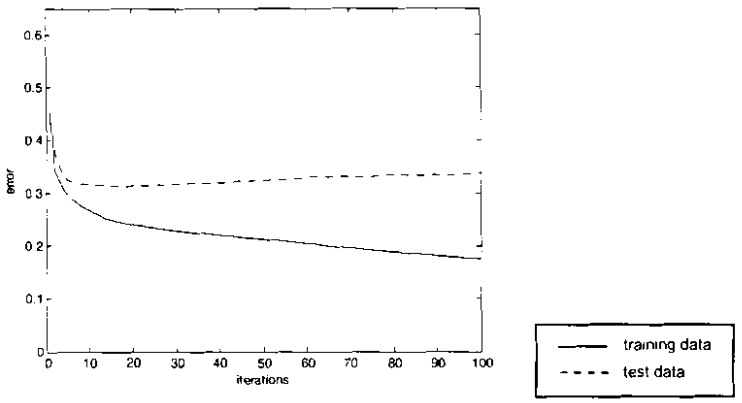

Figure 8.14 *Medium sized MLP global controller trained with 50 samples Training error for MLP network with 20 first hidden layer neurons and 12 second hidden layer neurons. The training data error steadily decreases while the test data error increases indicating network overtraining.*

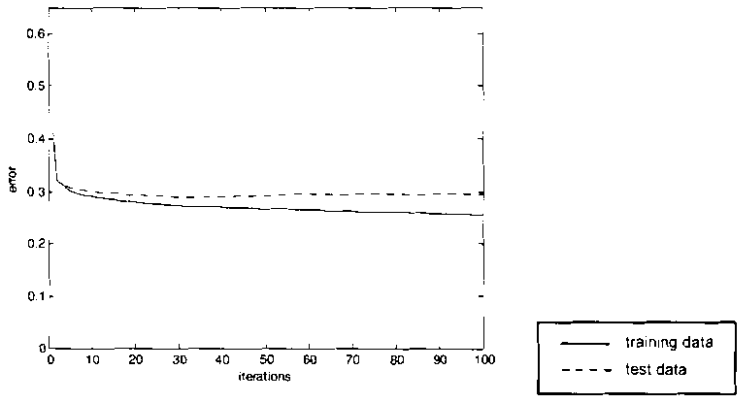

Figure 8.15 *Medium sized MLP global controller trained with 500 samples Training error for MLP network with 20 first hidden layer neurons and 12 second hidden layer neurons. The large set of training data reduces the test data error while increasing the training data error.*

At such large network sizes the backpropagation algorithm becomes very slow, encouraging the use of alternative faster training methods [11].

8.2.2 RBF global controller

The control of a manipulator in an environment with obstacles is highly discontinuous. Due to the unbound sigmoidal activation in MLPs, training of such a controller is expected to be somewhat problematic. The RBF makes use of unsupervised clustering techniques. Furthermore, training is bounded locally. For this reason much larger networks can be implemented more effectively.

Method

The MLP controller was replaced by a RBF controller. Although RBF networks require more neurons, training is significantly simpler. In addition, fuzzy rules can be derived directly from a trained RBF controller. The specifications for the networks implemented are listed in Table 8.7. The network is trained with pre-generated data as described in the previous section. Hidden layer neurons are trained unsupervised by means of K-mean clustering. The output neurons are trained through the least mean square (LMS) algorithm.

Table 8.7 *Specification for RBF global controller*

Property	Value
Inputs (range)	3 joint angles [$-\pi$; $+\pi$]
Outputs (range)	3 joint commands [$-\pi/100$; $+\pi/100$]
Hidden activation	Gaussian, B = 1.0
Output activation	Linear
Training ratio	0.04

Results

The RBF was tested with 5 training samples. Learning of the training data was possible with only 3 hidden neurons. Overtraining of the data is still a problem with this method, clearly visible in Figure 8.16. Increasing the amount of training data has the expected result of decreasing the test data error while increasing the training data error, shown in Figure 8.17.

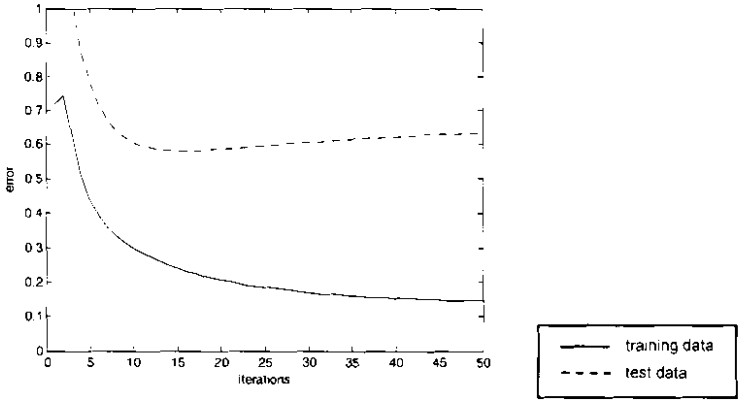

Figure 8.16 *Small RBF global controller trained with 5 samples*
Training error for RBF with 3 hidden layer neurons. Overtraining commences after about 15 training iterations when the test data error starts to increase.

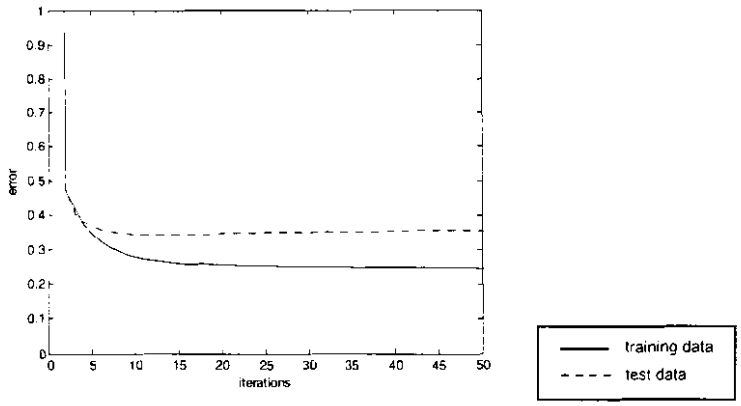

Figure 8.17 *Small RBF global controller trained with 50 samples*
Training error for RBF with 6 hidden layer neurons. The increase in amount of training samples reduces the test data error while increasing the training data error.

Unlike with the MLP, even with large training sample sets the training data error can be kept low by increasing the amount of hidden neurons accordingly. In Figure 8.18 a RBF with 40 hidden neurons was trained successfully with 50 samples.

The apparent fast initial improvement in the test data error seen in Figure 8.18 is a result of hidden neuron activation. The average output of the hidden layer neurons is significantly higher for the training data than for the test data, as shown in Figure 8.19. The low hidden neuron activation for the test data results in the network outputs defaulting to the bias value. The bias value is quickly changed to the output average. Since the training data has higher hidden layer activation, its error resembles the gradual improvement in the output layer error.

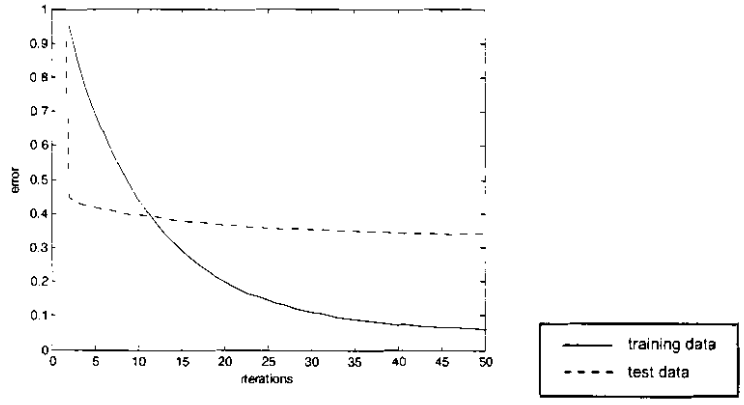

Figure 8.18 *Medium sized RBF global controller trained with 50 samples*
Training error for RBF with 40 hidden layer neurons. Training data is learnt successfully but the test data error remains unacceptably high.

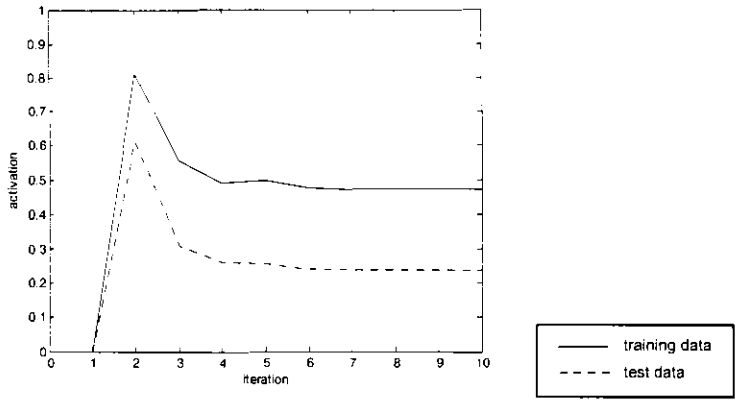

Figure 8.19 *RBF hidden layer activation*

The average output of the hidden layer neurons of the RBF with 10 hidden neurons trained with 50 samples. Activation for the trained data is significantly higher than for the test data.

It is worth noting that although generalisation is poor, the RBF network can successfully control the manipulator from states encountered in the training data. However, this closely resembles a look-up table. The RBF global controller is optimised by increasing the amount of training samples and adding sufficient hidden layer neurons to keep the training data down. Beyond 500 training samples and 200 hidden neurons, illustrated in Figure 8.20, the test data error could not be further improved.

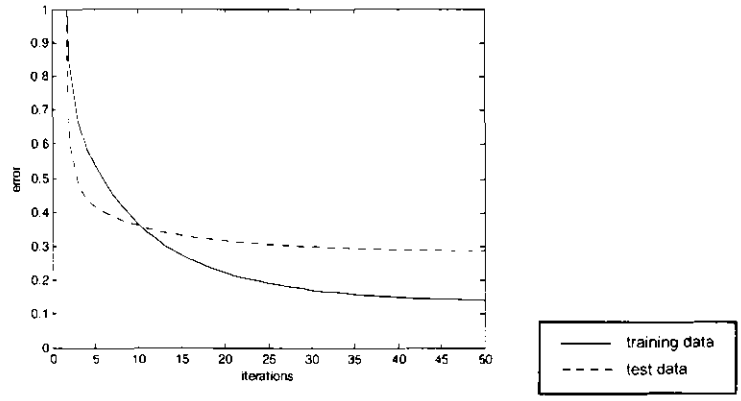

Figure 8.20 *Large RBF global controller trained with 500 samples*

Training error for RBF with 200 hidden layer neurons. The increase in amount of hidden layer neurons reduces the error for the training data while the test data error stays the same.

8.2.3 NNC global controller

An alternative global controller is constructed with the nearest neighbourhood clustering algorithm. By breaking the solutions found by the explorer up into sequences of small actions,

many training samples can be produced from a single solution. This provides better coverage of the state space for the same amount of solutions from the explorer. The notion of potential fields discussed in paragraph 3.3.1 is applied to state space. The controller resembles a potential field where a three-dimensional field vector represents the actions. The controller is trained with a very large set of small action training data. The aim is to instigate inference between contradicting data at a small scale to even out their effects as indicated in Figure 8.21.

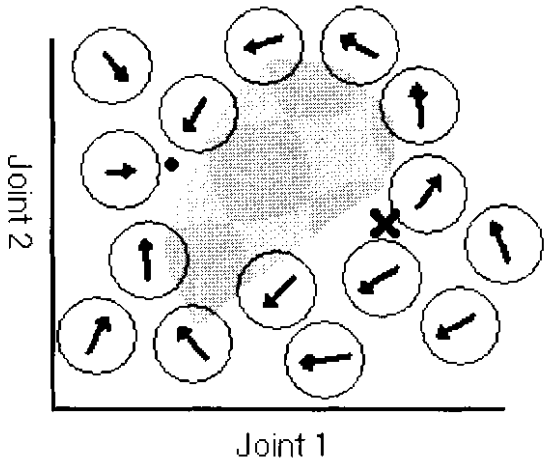

Figure 8.21 *Intermediate states and actions*

In this 2D representation of state space clusters produces actions resembling the vectors of a potential field pointing around the obstacles to the goal.

Method

Because of the increased amount of training data, a faster training algorithm is required. Nearest neighbourhood clustering (a single pass fuzzy training algorithm allowing for instantaneous training) is implemented. The actions pre-generated by the GA are fractured into consecutive small actions. These actions are used to generate a large set of intermediate states with associated actions. This data is used to train the controller.

Unlike all the previous controllers implemented, the amount of nodes or neurons in the fuzzy clustering controller is not predetermined. Clusters are generated during training as they become necessary and the amount is a function of the predetermined maximum cluster size. Therefore the success of training is a function of cluster radius and smoothing parameter (sigma) rather than network size and the amount of training iterations. The controller specifications listed in Table 8.8. Network parameters can be reduced by linking the cluster radius to the sigma value to add complexity as smoothness is reduced.

Table 8.8 *Specification for NNC global controller*

Property	Value
Inputs (range)	3 joint angles [$-\pi$; $+\pi$]
Outputs (range)	3 joint commands [$-\pi/100$; $+\pi/100$]
Hidden activation	Gaussian, B = 1.0
Output activation	Normalised linear

Results

The training error starts off small since it is calculated over only the limited amount of samples for which the network was trained. It increases as the network starts to generalise. At the same time the test error decreases. Figure 8.22 illustrates this with a sigma value of 1. The test data error stabilizes after about 500 training samples at which point 133 clusters have been created. Figure 8.23 shows the results for a sigma value of 0.6, creating 344 clusters and Figure 8.24, for a sigma values of 0.2, creating 801 clusters.

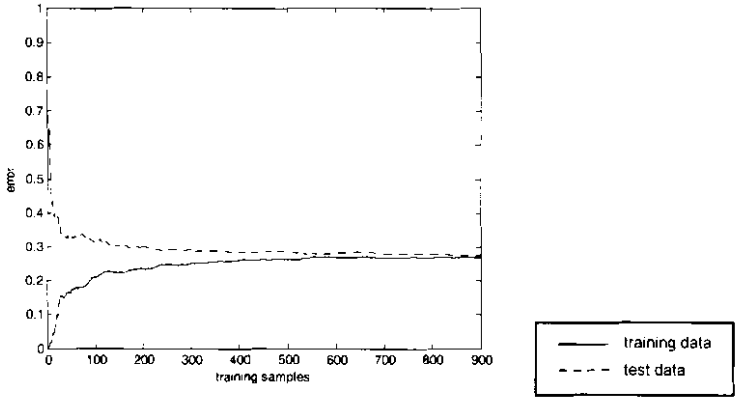

Figure 8.22 *NNC global controller with a sigma value of 1.0*

Training error for NNC function. The training data error and test data error stabilises after about 500 samples.

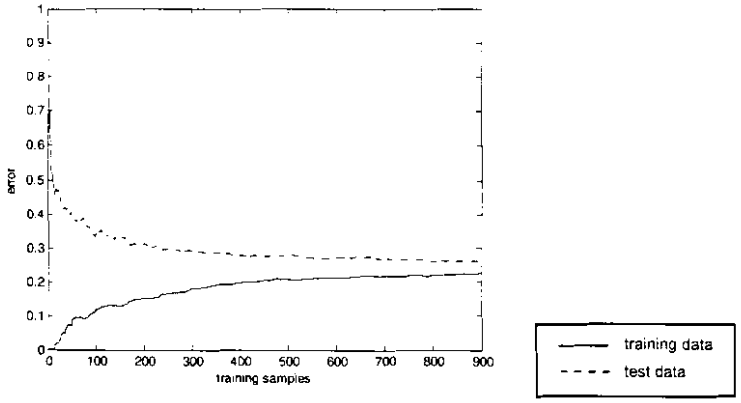

Figure 8.23 *NNC controller with a sigma value of 0.6*

Training error for NNC function. The training data error and test data stabilises after about 700 samples.

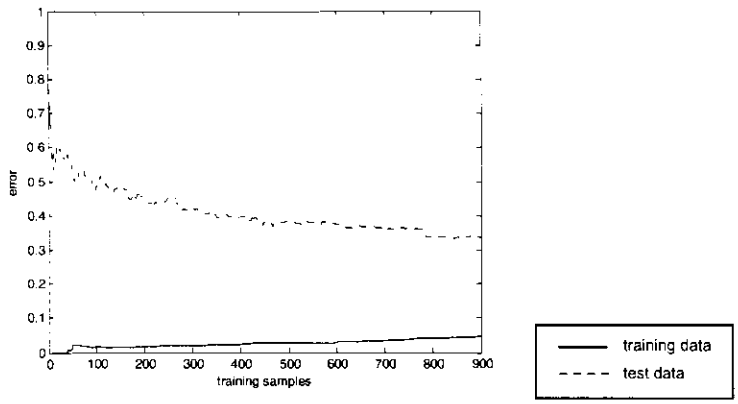

Figure 8.24 *NNC controller with a sigma value of 0.2*
Training error for NNC function. The training data error and test data stabilises after about 900 samples.

In Figure 8.25 the training and test data errors are indicated for different sigma values. It is noticeable that the training data error can effectively be eliminated by sufficiently reducing the sigma value, but at the cost of the test data error. For this data, optimal generalization occurs with sigma at approximately 0.8.

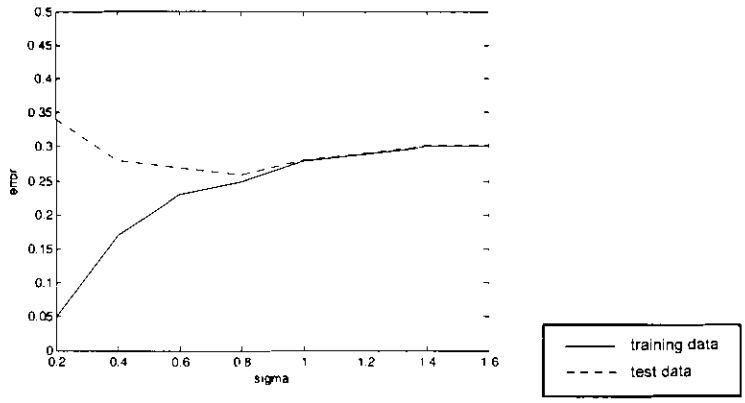

Figure 8.25 *Optimal sigma value for NNC controller*
Decreasing the sigma value until approximately 0.8 results in a decrease in both the training data error and the test data error. Beyond this point the test data error starts to increase again.

Due to conflicting actions proposed by the GA as discussed in section 7.4.2, discrepancies can occur in the training data and test data, making it impossible to learn the training data with good generalisation. This does not imply that feasible control is not possible through training of this data. It does however suggest that the training error is an inadequate measure of success. From here onwards controllers will be evaluated based on the Manhattan distance the controller is

capable of reaching. Unfortunately this method disregards the distance travelled to reach the target and therefore does not distinguish between optimal and non-optimal paths followed.

Figure 8.26 displays a histogram of the distances from the target that the tip of the manipulator could reach for a hundred trials within a finite amount of control steps, using the fuzzy clustering controller with optimal sigma value. The manipulator tends to stop in open space as illustrated in Figure 8.27 rather than to move close to the target. This is a direct consequence of the contradicting training data. The manipulator settles within 50 steps, at which point the distance is recorded.

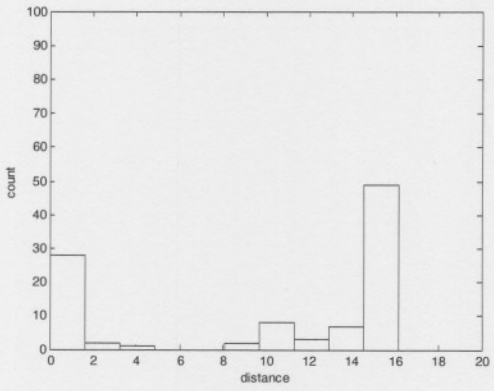

Figure 8.26 *Global controller results*

The controller reaches the target 28 % of the time and on average reaches a distance of 9.75 units from the target.

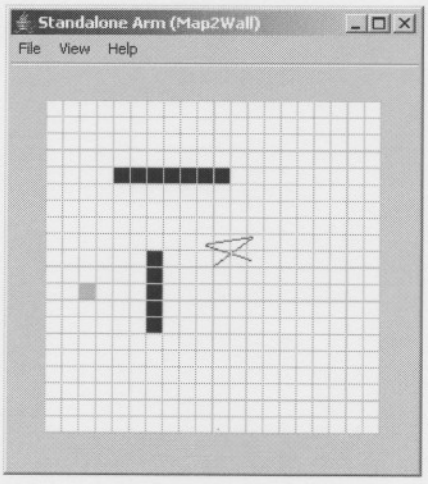

Figure 8.27 *Global controller stops in open space*

The manipulator stops between two options of what action to take next.

The following two paragraphs describe two modifications to the NNC function which was made in an attempt to prevent the manipulator from settling in open space.

One sample per cluster

When adaptive systems are trained with inconsistent data, the result is often the average of the target data. Contradicting data can result in a cancelling effect which is not always appropriate. In an effort to solve this problem only one training sample is used to calculate the effect of every node. Training data resulting in new nodes is added to the network while data that would normally be added to an existing node is discarded. In effect, only the first sample of any cluster is incorporated in the controller. This solves the problem of contradicting training data for any particular node, but not for the complete system. Contradicting neighbouring rules result in a similar cancelling of commands in areas between the rules. The state marked with **X** in Figure 8.21 indicates such a state of contradicting commands.

Winner-takes-all

By adopting a winner-takes-all approach for the activation of the rules, there can be no cancellation due to contradictions in simultaneously activated rules. The resulting controller loses the power of generalisation, but most noticeable is the effect of the manipulator oscillating between two contradicting neighbouring rules. These modifications showed no improvement on the performance of the global controller.

8.3 Adaptive critic

Due to the lack of success in the previous attempts to derive a stable controller, an alternative approach is implemented. The failures are assigned to the contradictions in training data. Since different actions can be found with equal fitness, the notion to train a critic with the state fitness, rather than training a controller with the actions is considered. This data is much less conflicting and of lower dimension.

This is the same approach which is followed in the actor-critic architecture in reinforcement learning system described in section 2.5.2. The adaptive critic design typically implements a neural network with a temporal difference training algorithm for realisation of a value function in the critic. However, for this study the use of fuzzy clustering is explored as an alternative. In accordance with section 6.2.2.1 this function is called the evaluator.

After fracturing the solutions provided by the GA as discussed in section 8.2.3, a great amount of data is available for training. A NNC function is implemented so that this data can be easily assimilated into the value function.

To derive actions from the evaluator, the processes of action deduction and action evolution is applied. These processes are discussed in this section, followed by the implementation of the evaluator and two additional adaptations namely Ant-Q and wall penalisation.

8.3.1 Action deduction

Having a value function based on solutions is much better than having it based on the original Manhattan distance since it takes into account the fitness of the state of the complete manipulator relative to the environment. The optimal path for moving the manipulator to the target would be the sequence of actions resulting in the steepest ascent in the value function.

In obstacle free space in an Aristotelian universe, and given that the manipulator state is represented in joint angles and actions are represented in joint movement, the resulting state for any action is simply the sum of the previous state and the action. Consequently the action required to move the manipulator from one state to another is simply the difference between the joint angles of two states. If the two states in question are in free space and close together it can be safely assumed that the area between these states are obstacle free and the statement above will also hold true in an environment with obstacles. This will be called the action deduction assumption.

A direct consequence of this assumption is that the optimal action to be taken is in the direction of the gradient of the value function.

Proposed method

Finding the gradient of the value function requires that the derivative of the value function can be calculated. Calculating the derivative of the evaluator can be significantly simplified if a RBF is implemented instead of the NNC function. Unfortunately the RBF is not capable of single pass learning. The possibility of converting a NNC function into a RBF after training was considered.

While the ultimate value function would suffer no problem of local maxima, the clustering function has inherent minima between cluster centres and consequently local maxima as demonstrated in Figure 8.28. As a result a strict gradient method will get stuck at these local maxima on cluster centres. For this reason the conversion to RBF was not implemented.

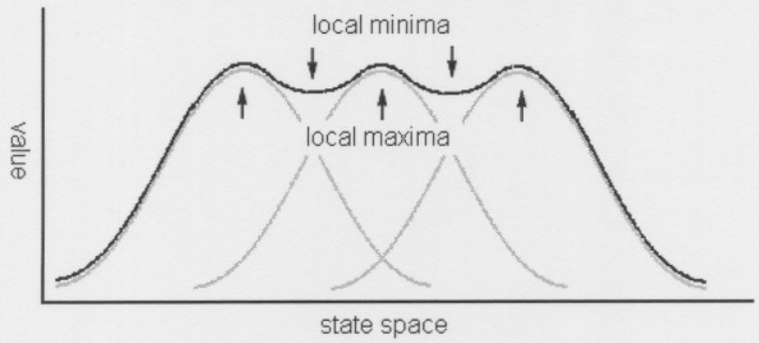

Figure 8.28 *Local maxima and minima*

The cluster centres create local maxima with local minima between them in the value function.

8.3.2 Action evolution

The optimizer needs to find an action which can improve the state value. Applying the action deduction assumption, it only needs to find a state in close proximity to the current state, which can improve the state value.

Method

Meta-heuristic search is applied to find such states. A simple GA is used to find a state with the highest value in close proximity to the current state. By making the range of the search space larger than the cluster size, solutions can be found beyond the minima discussed in section 8.3.1. This is illustrated in Figure 8.29. Since the search space is small, of low dimension and smooth, a good solution can be found quickly.

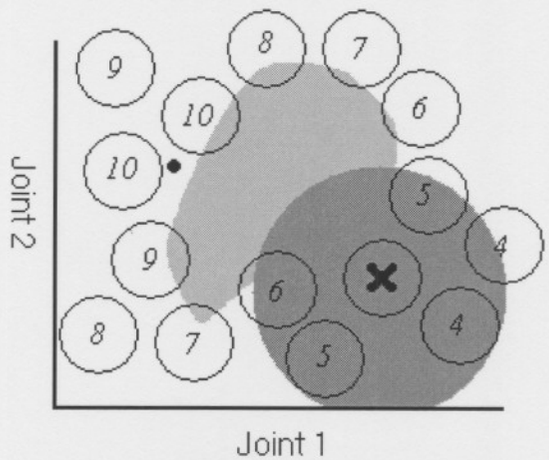

Figure 8.29 *Leaping across local minima*

The dark shaded area indicates the search area which spans clusters with higher values.

8.3.3 Evaluator

In the critic a function is constructed that represents the value of the manipulator state. This is a scalar value that gives no consideration to the action to be performed, making it an action independent critic.

Method

All the pre-generated GA solutions are fractured into sequences of actions, resulting in sequences of intermediate states when executed. Starting from the last state in such a sequence and working towards the first, decreasing values are assigned to all the intermediate states. The highest value assigned resembles the overall fitness of the combined sequence of actions as proposed by the GA. These values are then learnt by the evaluator. Figure 8.30 illustrates the value function representation in contrast to the controller function in Figure 8.21.

Table 8.9 *Specifications of evaluator*

Property	Value
Inputs (range)	3 joint angles [$-\pi$; $+\pi$]
Outputs (range)	1 state value [0 ; 100]
Hidden activation	Gaussian, B = 1.0
Output activation	Linear

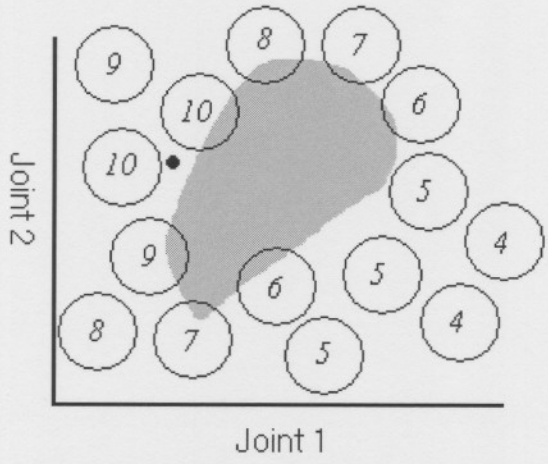

Figure 8.30 *Evaluator state values*

The evaluator is trained to estimate a value function with every cluster representing a state value. These values are a maximum at the goal and declines further away.

Results

Since a meta-heuristic search is implemented, non-optimal or even wrong actions can be taken. With sufficient training a path can be derived from any state and mistakes can easily be corrected. This approach results in feasible control rather than optimal control. It was observed that non-optimal paths around obstacles are often followed by the manipulator as illustrated in Figure 8.31.

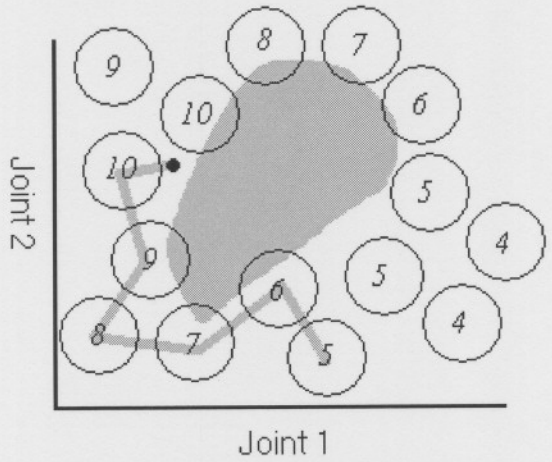

Figure 8.31 *Following a non-optimal solution path*
Non-optimal solutions of the action optimiser still lead to the gaol as long as the state value increases with every step.

Comparing Figure 8.32 with Figure 8.26 shows that the optimiser and the global controller exhibit roughly the same performance for reaching the target. However, notice that the average distance of the failures has moved closer. While the global controller settled in open space, as shown in Figure 8.27, the manipulator now stops against obstacles as indicated by Figure 8.33. This is a great improvement from settling in open space. It happens because there is no training data for the obstructed states. The fuzzy clustering function normalises rule activation, therefore interpolating between trained surrounding states, resulting in an erroneous high value assumed for moving through the obstacles.

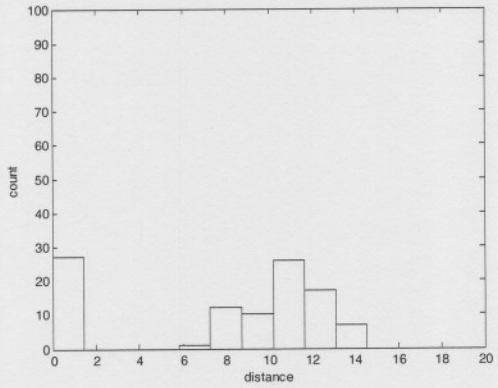

Figure 8.32 *Evaluator optimiser results*

The optimiser reaches the target 27 % of the time and on average reaches a distance of 7.90 units from the target.

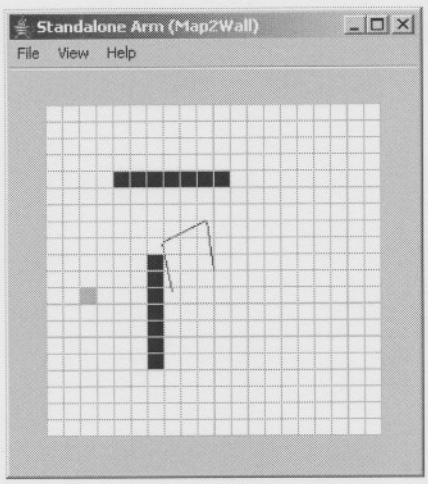

Figure 8.33 *Evaluator optimiser collides against wall*

In the process of moving closer to the target, the manipulator collides and then stops against obstacles.

In cases where no rules are significantly activated, extreme normalisation scaling occurs. By adding a small bias value to the total rule activation, this can be prevented. The function output is then drawn to zero in areas where no training took place, instead of interpolating between distant rules.

This modification results in a decrease in the value of the obstructed states. In this way obstacles are taken into account by the evaluator and negotiated by the optimiser. A major improvement is evident in Figure 8.34 for controlling the manipulator, reaching the target almost 70 % of the time.

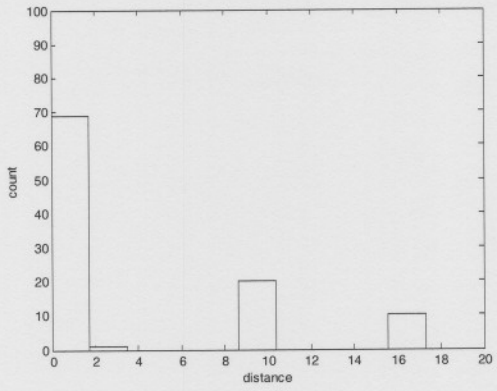

Figure 8.34 *Evaluator with biased activation*

The optimiser reaches the target 69 % of the time and on average reaches a distance of 4.05 units from the target.

8.3.4 Ant-Q

Observing that the manipulator sometimes still settles in open space, suggests that local maxima exists at which the optimiser some times stagnates. These maxima are created when less optimal solution paths in the training data intersect with better paths. If a less optimal path is trained over a better path, the better path is disrupted, leaving it with isolated islands in the value function as illustrated in Figure 8.35.

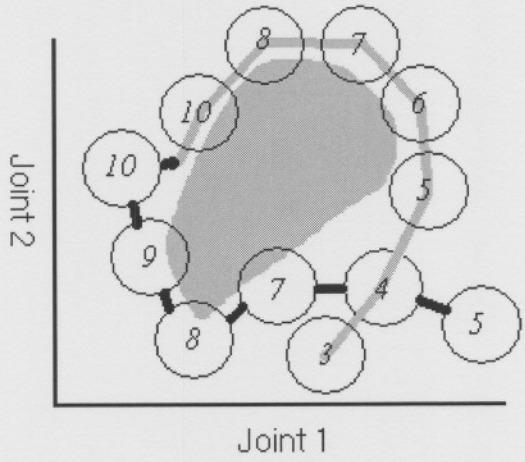

Figure 8.35 *Creating of value function islands*

After intersecting with a lower valued path (grey), the earlier better path (black) is disrupted with a gap (cluster value 4) that can not be breached (from 5 to 7).

The emergence of these local minima can be prevented if the value function assumes only the best value encountered during training for any cluster. If the manipulator now reaches an intersection of two trained sequences, the higher valued path is most likely to be selected by the optimiser. The value function starts to resemble the formation of pheromone trails of ants in

ACO as described in paragraph 2.3.3.3. Ant-Q denotes a system where pheromone trails are used to construct a value function [44].

Method

The NNC evaluator is trained with only the highest training value found for any cluster. If a training sample is assigned to a cluster that has a lower value, the cluster's value is replaced by the higher value; otherwise the cluster value is left unchanged. The consequent training samples in sequence of states will however follow the cluster value suggests. Nearest neighbourhood clustering makes this instantaneous modification to clusters possible.

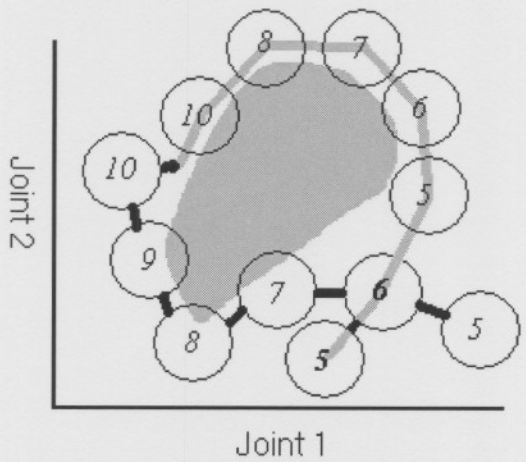

Figure 8.36 *Ant-Q maximum value*
After intersecting with a higher valued path, the training value now adopts the higher value for the consequent training samples.

Results

Performance of the evaluator and optimiser is further improved by preventing the manipulator from getting stuck at local maxima, reaching a success rate of 80 % as indicated in Figure 8.37. The manipulator however still gets stuck at obstacles for 20 % of the trials.

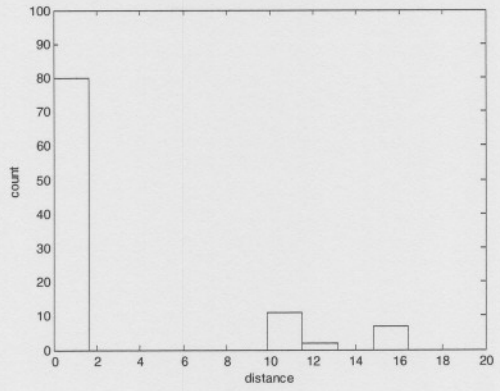

Figure 8.37 *Ant-Q evaluator*

The optimiser reaches the target 80 % of the time and on average reaches a distance of 2.58 units from the target.

8.3.5 Wall penalisation

A high trained state value on the target side of an obstacle can create a false high state value on the non-target side. If the manipulator needs to be moved from the non-target side of the obstacle it might try to move towards the trained value, getting stuck against the wall as illustrated in Figure 8.38.

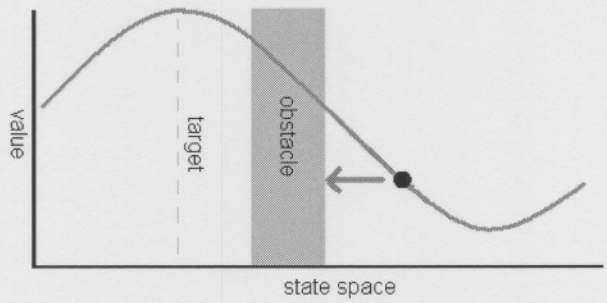

Figure 8.38 *Increased value towards wall*

The manipulator's state value can be increased by moving it closer to the wall until it gets stuck.

This happens because training data included only viable solutions and there was no negative reinforcement learning of illegal states. By explicitly penalising the value function at illegal states minima can be created around obstacles, forcing the manipulator to move around them. The effect of negative reinforcement is proposed in Figure 8.39.

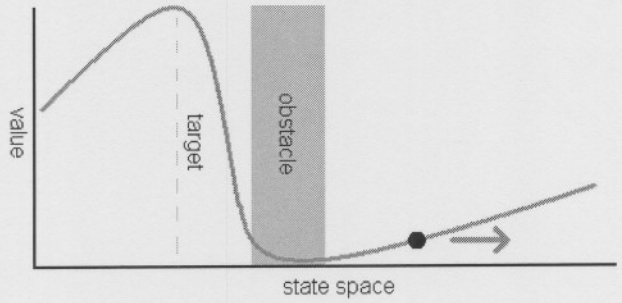

Figure 8.39 *Decreased value towards wall*

The manipulator's state value can only be increased by moving it away from the wall, allowing it to find the target.

Method

In addition to the set of solution training data received from the GA, the environment is randomly searched for illegal states. Nodes with zero value are created at these states. By gathering and training sufficient negative reinforcement data, the observed false state values can be removed.

Results

The evaluator is trained with about a hundred illegal states. By varying the sigma value of the zero value nodes the repulsiveness of the obstacles can be adjusted. With a large enough value the manipulator steers clear of any obstacles. However, when the manipulator needs to be manoeuvred through an opening in a wall, the repulsion of the walls prevents the manipulator from approaching the opening. As before, the manipulator gets stuck at a local maximum in open space which seriously hampers the system's performance. This is clearly visible in Figure 8.40.

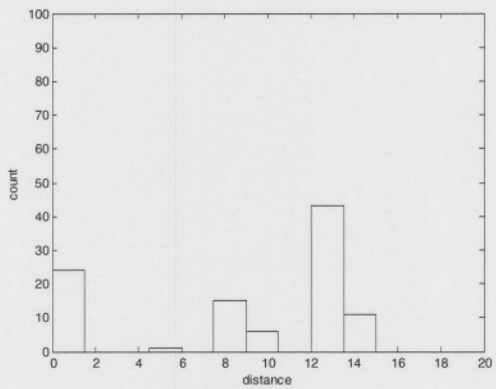

Figure 8.40 *Ant-Q evaluator with walls with sigma value 0.5*

The optimiser reaches the goal 25 % of the time and on average reaches a distance of 9.10 units from the target.

A balance needs to be found for the repulsiveness of obstacles. With careful selection of the negative reinforcement node sigma values, the evaluator is optimised to produce remarkably good results. Figure 8.41 shows a success rate of 88 %.

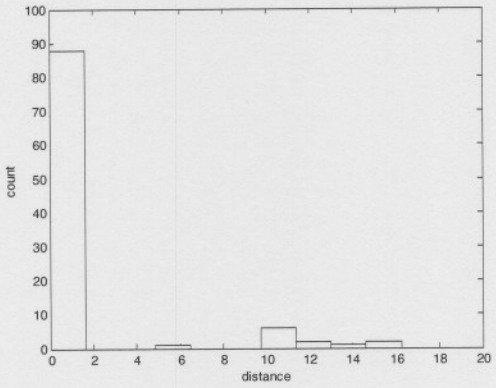

Figure 8.41 *Ant-Q evaluator with walls with sigma value 0.1*
The optimiser reaches the goal 88 % of the time and on average reaches a distance of 1.41 units from the target.

8.4 Concluding remarks

The inconsistencies in data generated by the GA during guided exploration produces a problem when used as training data for neural and neuro-fuzzy controllers. Such controllers are not capable of deriving accurate outputs when measured against the training data. Resulting controllers interpolate between conflicting data often resulting in undesirable behaviour like oscillating or becoming stagnant far from the target.

Local controllers disregarding obstacles in the global environment produces satisfactory results only when the manipulator is in close proximity to the target where conflicts in the training data are reduced. Global controllers were not able to feasibly control the manipulator. The manipulator frequently becomes stagnant in open space, starts oscillating or collides against obstacles.

With the introduction of the adaptive critic and optimiser, major improvements have been achieved. Through negative reinforcement and optimisation on the value function the manipulator can successfully navigate the environment. Although actions are not received directly from a feedforward controller, it can be derived fast enough to allow feasible real-time control.

Chapter 9

Conclusions and Suggestions

In this chapter final conclusions are drawn concerning the project and suggestions are made for future work and improvements to the system which was developed.

9.1 Final conclusions

The results of the study are assessed against the criteria of the original problem statement. The successes achieved, problems encountered and contingency resolutions are discussed here under corresponding headings.

9.1.1 Evaluation platform

The control of the simulated robotic manipulator forms an excellent benchmark application for testing and evaluating intelligent control systems. While it is simple to visualise the problems and solutions, the problem from a control point of view can be scaled to high complexity by increasing the amount of manipulator segments and the distribution of obstacles.

Due to the complexity of the problem, the target and obstacles were made static and a manipulator with only three segments was implemented. A hyper-redundant manipulator can be implemented equally well. By making use of a grid-world environment, collision detection is greatly simplified. Additionally, this makes editing of obstacles in the environment very easy. By increasing the grid size, the resolution can be improved and curving edges on the obstacles can be simulated.

9.1.2 Software design standard

It is essential to follow an object-oriented approach when developing integrated software. Encapsulation improved reusability and integration and consequently prototyping.

Java™ is a developer friendly object-oriented programming language, which allows the development of a class library to follow naturally. One great advantage is that any of the classes developed can be directly instantiated and accessed from MATLAB®. This allowed for seamless integration of the evaluation platform, search algorithms and any controllers with any MATLAB® functions or tools.

9.1.3 System architecture

The modular controller architecture which was developed is generic enough that control systems ranging from a classical proportional controller to a model-based intelligent agent can be constructed by simply plugging the required units into the framework. As new controllers were developed, they could be added to the system to add or improve functionality.

9.1.4 Environment exploration

The GA implemented for exploration is very successful in finding commands to move the manipulator to the target. Even when sequences of actions are required to reach the target, the GA can find feasible solutions based on the Manhattan distance to the target.

The problem experienced with the GA is that the solutions provided are inconsistent. However, this is not an unrealistic problem: it seems necessary to develop a controller which can handle this inconsistency.

9.1.5 Controller implementation

As a result of the contradicting training data produced by the explorer, the controllers evaluated could not successfully learn to model the training data. The training error was minimised by interpolation of contradicting data. In this complex system the average of two good solutions is not necessarily also a good solution. For this reason an alternative approach was implemented where the value of the solution is used to estimate the value of the state. A combination of AI techniques was used to create a value function and derive actions from it. The resulting controller is capable of successfully commanding the manipulator to the target from any starting point.

The problem of contradicting training data is not unique to this system. Any ill-posed problem will be faced with similar conditions and could make use of the approach followed for this system. This approach offers the creating of a value function through single-pass training as opposed to temporal difference learning.

9.2 Suggestions and recommendations for future work

Recommendations, proposals of alternative methods to follow and improvements that could be made to the different design aspects of this project are listed next.

9.2.1 Evaluation platform

The collision detection calculation of the manipulator in the simulated environment is computational intensive. By implementing a graphics engine like Java2D™, speed as well as aesthetic improvements might be achieved. The effectiveness of this engine compared to the grid-world collision detection algorithm currently implemented should however be considered.

9.2.2 Software design standard

Java™ is highly recommended for engineering software development for its ease of use, platform independence and seamless integration with MATLAB®.

The setting of interface standards prior to development holds great benefits to the reusability of the code and is also recommended.

9.2.3 System architecture

The system architecture developed in Chapter 5 has not yet been tested with the backpropagation reinforcement learning of the adaptive critic design. Neither has it yet been tested as an autonomous intelligent agent, exploring the real world in real time.

A simpler control problem should be chosen to test all the attributes and data flow patterns of the system thoroughly before attempting more ambitious problems such as this project.

9.2.4 Environment exploration

It is expected that the inconsistency problem of the genetic algorithm can be solved to a great extent through the use of memory keeping evolutionary algorithms such as particle swarm optimisation or especially artificial immune systems.

9.2.5 Controller implementation

Self-organising maps might be more tolerant to conflicting training data. Evolving self-organising maps seem to be the most promising member of this family of clustering systems and are proposed for implementation, training and testing for global control of the robotic manipulator.

Appendix I

Software Modules

Table I *The Java™ classes developer for this project is listed and briefly described here*

Class	Description	Parent
Function	Input/ Output template	-
Master	Main application	-
AntNet	Ant-Q network	ClusterNet
BasisNet	Radial basis network	ConnectNet
ClusterNet	Nearest neighbourhood clustering function	ConnectNet
FuzzyNet	Fuzzy feedforward network with backpropagation	ConnectNet
NeuralNet	Multi-layer perceptron	ConnectNet
ConnectNet	Connectionist network template	Function
GausRt	Gauss root function	Function
Matrix	Matrix utilities	Function
Mocca	Modular controller architecture	Function
Sigmoid	Sigmoid function	Function
Solver	Optimiser template	Matrixc
Map	Wireframe for grid	Spirit
GFrame	Custon window	JFrame
GMenu	Custom menu	JMenu
Grid	Grid-world	GFrame
Layer	Neural network layer	Matrix
Spirit	Wireframe prototype	Matrix
Controller	Controller template	Mocca
GeneticAlg	Genetic algorithm	Solver
Arm	Robotic manipulator	Spirit
Transform	Local controller transformation	Spirit

Appendix II

Symposium Paper

A paper titled "Action Evolution for Intelligent Agents" was presented at the 2005 IEEE International Symposium on Intelligent Control (ISIC), held in Limassol, Cyprus from the 27th until the 29th of June 2005.

This paper addresses the notion of acquiring training data for supervised learning through guided exploration of the environment. A draft version of this paper is included in this section.

Action Evolution for Intelligent Agents

Morné Neser, Rodney E. Tessendorf and George van Schoor

Abstract— **This paper introduces an alternative method of reinforcement learning for intelligent agents. The aim with this method is to closer emulate the natural thought process of problem solving. This paper only provides a conceptual description of this method. However, it puts forward realistic implementations with exciting new implications, using proven techniques. After a brief discussion of reinforcement learning, the newly suggested method will be described. Reinforcement learning methods can be divided according to two main strategies. The first strategy uses evolutionary methods while the second makes use of incremental back-propagation methods. An overview is given of the methods used for implementing the popular actor-critic architecture in these strategies. This leads to a generic architecture for intelligent controllers. The alternative method of action reinforcement is then demonstrated at the hand of this architecture. This representation of the controller emphasizes some cognitive attributes and encourages the development of advanced intelligent controllers.**

I. Introduction

INTELLIGENT agents (IA) are the ultimate form of artificial intelligence applied to system control. IAs are situated inside their target problem domains and has the ability and freedom to control their own destiny. Embodiment of the controllers is an important feature of IAs which allows them to interact with and learn from their environments. With the ability to learn online and in real time, IAs can be designed to sustain a life long process of adaptation to ever changing environments. Therefore it is required that IAs can assimilate large amounts of data from its environment and make feasible deductions from such data in a relative short time. They should be able to discover new optimal solutions to problems as the environment changes and incorporate these solutions into their control strategies. Furthermore IAs should be capable of global generalization, discard temporary exceptions and reason with uncertainties [1].

Intelligence is only one of the cognitive attributes a system can have. The grounds on which this attribute is assigned are not clear. For a system to be labeled as cognitive poses an even greater problem. This is not problem for engineering but rather quite a philosophical question. An approach has been followed in which a relative level of intelligence is assigned rather than a discrete label. The same approach should be followed for all the cognitive attributes in a controller and for the consideration of whether a system is cognitive.

Control systems are generally designed from an engineering point of view with performance as their primary goal. Therefore when cognitive attributes do emerge, they usually do so only as byproducts of advanced design. Techniques such as supervised and reinforcement learning (RL), modeling, planning and exploration of the environment have been adopted in the effort to build optimal control systems in a nonlinear world. Attributes such as curiosity, contemplation and anticipation are the byproducts of these techniques. As the level of intelligence in control systems increases, so does the expectation of presence of such other cognitive attributes. In advanced intelligent agents a multitude of these attributes should become noticeable.

In this paper a generic architecture for controllers, in particular intelligent controllers is proposed. This high level representation of the controller is primarily based on the actor-critic architecture of RL [2], [3] which has become very popular in the continuous state and action space domain [4]. The architecture incorporates controllers from the classic design [5] to the Adaptive Critic Design (ACD) family of controllers [5], [6]. This design is then further extended towards the idea of cognitive control. The design is modular, which allows for smooth interchangeability of different control, modeling and training techniques. Although modules may also be excluded from an implementation to simplify the design, the benefit of keeping to this general design is that modules may be added later to improve system performance without effecting the original design. Different primitive and advanced attributes associated with intelligence such as contemplation, planning and self-confidence are explicitly identifiable in this architecture. These attributes can be recognized by distinguished data flow patterns between the different modules. Control systems based on this architecture will inadvertently inherit the ability to enable these patterns to utilize these attributes.

The abilities of this architecture are further extended with a notion from the field of psychology. The novice idea of "evolution of ideas" will be presented in the final part of this paper. This mechanism will be tailored as a learning scheme for the actor-critic and will be called "action evolution". Alternative learning techniques like incremental gradient descent in the ACD family suffer from the local minima problem while evolutionary methods suffer from high dimensionality and long evaluation periods [8]. Action evolution uses evolutionary methods to discover feasible actions. Actions normally have much lower dimensionality and can be assessed instantaneously. These actions are used as training data for supervised learning of the controller.

In the following two sections RL and genetic algorithms (GA) are discussed in the light of controller

development. In Section IV a generic architecture is presented for intelligent and cognitive control. Section V introduces a novel idea for acquiring controller training data through offline exploration. In the final section conclusions are drawn, outstanding work is discussed and future work is proposed.

II. Reinforcement Learning

When learning occurs as a result of rewards received through interaction with the environment it is called reinforcement learning [2]. RL differs from supervised learning in that there is no expert knowledge in the form of training data (desired input/output pairs) available [8]. Instead rewards are received from the environment and used as learning feedback. Control systems generate action outputs according to internal control strategies or policies. The aim of learning in adaptive controllers is to derive an optimal control policy – the policy that maximizes the overall sum of rewards over time. It seems that the search methods followed can be divided according to two main strategies [9]. The first strategy is to explore the space of all possible policies and promote the best policy found. This strategy will be called policy reinforcement. Guided random search methods [10] are the primary means to this strategy [11]. These methods include evolutionary strategies and GAs which will be discussed in the Section III. The second strategy is to estimate the contribution of particular actions in particular states, and reinforce these actions accordingly. This will be called action reinforcement. Methods used to accomplish this include statistical techniques and incremental methods like dynamic programming (DP). These methods will be discussed in here. Refer to Fig.1 for a hierarchical classification of methods.

The policies are adapted as knowledge about the environment arises. An IA explores its environment autonomously to gather information for RL. After taking an action, the agent receives feedback from the environment which includes the new state and the associated immediate reward. The reward indicates the success of the state. Often the system needs to pass through a sequence of neutral states before the desired goal state can be reached. Such delayed rewards are the cause of the "accountability" or "credit assignment" problem. The agent has no knowledge of which of the actions taken are responsible for the rewards received [2]. This information is crucial for reinforcement of the appropriate actions. To overcome this problem the RL agent implements a scheme for estimating future rewards. The agent records experience about system states, actions, transitions and rewards received. It then uses this data to construct an internal value function that estimates the future rewards on the grounds of intermediate states and actions. This gives the controller the ability to anticipate rewards and reinforce intermediate actions.

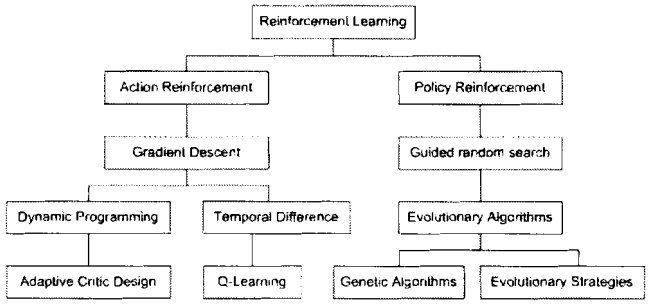

Fig. 1. Reinforcement Learning Methods

The three primary methods for action reinforcement are DP, Monte Carlo, and temporal difference (TD) learning [2]. All these methods have their shortcomings. DP requires accurate models of environmental dynamics. Monte Carlo, on the other hand is not suited for step-wise incremental computation. It only adapts the policy after a reward is reached. Although TD learning is the most complex method it requires no model and is fully incremental. These methods all have benefits in efficiency and speed of convergence but because of its convenience TD learning has gained much popularity. This method can be implemented with an adaptive or fixed model or with no model at all. It makes use of a Q-value which is an action-dependent variation of the value function. It estimates the future reward for the case when a specific next action is taken. Variations of this method include SARSA, Q-learning and Fuzzy-Q. One other popular variation is the TD(λ) method introduced by R. Sutton, which seamlessly integrates the three methods [2].

In a simple system with small discrete state and action space, the policy and estimated value function are represented in tabular form with states, actions and internal rewards. The action with the highest reward for the given state signifies the policy output. When the state space becomes very large or continuous, tabulating the results for all states experienced becomes impractical. A nonlinear function approximation scheme like a neural network can be used to create a continuous function J(t) to approximate the value of the internal reward [2]. These functions typically implement TD learning for incremental updating. Similarly, if the action space is continuous, the policy can also be represented by a multi-variable activation function. The policy and value functions become completely

separated which give rise to the popular actor-critic architecture as shown in Fig.2. The actor is responsible for generating the actions and the critic for estimating the reward.

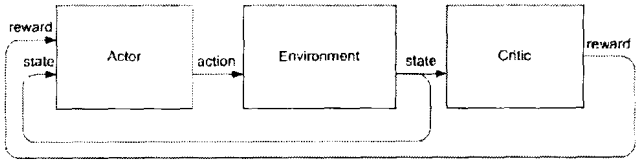

Fig. 2. Actor-critic architecture

By taking the action proposed by the policy the controller is exploiting the policy. The agent has to explore the environment in order to discover the optimal solutions and adapt its policy accordingly; therefore it should not always follow policy, but sometimes take random actions. The ε-greedy function provides a simple method to manage the balance between exploration and exploitation.

An Adaptive Critic Design (ACD) family of controllers was introduced by P.J. Werbos [5]. ACD is based on the actor-critic design and uses adaptive networks for both the actor and the critic. This property makes it very attractive for use in agents operating in continuous state and action space. Training of the actor involves back-propagation of the derivative of the cost function with respect to the actions $\partial J(t) / \partial A(t)$. This in turn might require a system model [12]. The model requirements for the different variations of the ACD are listed here:

HDP (Heuristic Dynamic Programming) – A constant value is back-propagated through the critic and then through the system model to obtain $\partial J(t) / \partial A(t)$ [6]. Therefore only actor training requires a model.

ADHDP (Action Dependent Heuristic Dynamic Programming) – The critic receives actions as input, allowing $\partial J(t) / \partial A(t)$ to be obtained by back-propagation only through the critic. Neither actor training not critic training requires a model.

DHP (Dual Heuristic Programming) – The critic estimates the derivatives of the cost function with respect to the system states. It uses the first order Bellman recursion that requires a system model. $\partial J(t) / \partial A(t)$ is acquired by back-propagating the critic's outputs through the system model. Both actor and critic training requires models.

ADDHP (Action Dependent Dual Heuristic Programming) – Critic training is the same as for DHP. The critic estimates $\partial J(t) / \partial A(t)$ directly. Therefore a model is required for critic training but not for actor training.

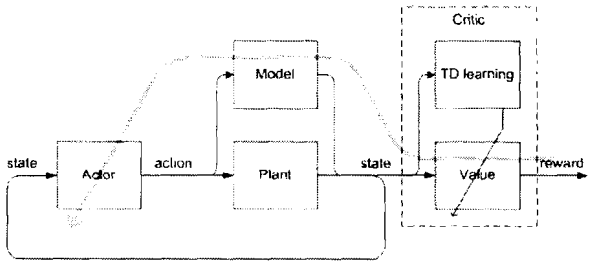

Fig. 3. DHP learning in the ACD

In Fig.3 DHP actor training is illustrated as back-propagation through the value function and model in the general ACD architecture. All but the ADHDP model requires a model of the system. Although deriving accurate system models can be very time consuming, it might be worth the effort since model-based schemes converge to a solution much quicker. Many techniques have been developed to rapidly derive feasible system models [13]. In addition non-stationary systems require adaptive models. Differentiable model can be used here, but since control errors occurs mainly as a result of inaccurate system modeling, model dependence is still a problem.

III. Genetic Algorithms

Evolutionary computing encapsulate all the methods of soft computing that mimic the natural process of evolution and genetics. It includes concepts like population, chromosomes, recombination, mutation and selection by survival of the fittest. Unlike the methods discussed in the previous section that use gradient back-propagation to calculate parameter adaptations, evolutionary methods make random changes and then evaluates its effect.

There have originally been two methods uses by evolutionary algorithms: GAs and evolutionary strategies [10]. Both these methods are used for searching and optimizing parametric solutions to general problems in a predefined solution space. The parameters are represented as chromosomes and the algorithm follows the same basic steps of evolution namely selection, mutation and replacement. The GA is initialized with random chromosomes. In every generation or iteration of the process the chromosomes are evaluated in respect of a fitness function. The fittest

chromosomes are retained for use in the next generation while the rest are replaced by new chromosomes. The goal of the problem is defined as finding the parameters of the chromosome that maximize the fitness function. This process is repeated until a certain generation or fitness threshold is reached, or the maximum fitness level has stagnated. The evolutionary cycle is depicted in Fig.4.

In the classic GA design the parameters are encoded as binary strings in the chromosomes. Multiple chromosomes are randomly created to form an initial population. New chromosomes are created by randomly selecting two of the fittest parent chromosomes and a crossover point. The chromosomes are split at the crossover point and exchanged between the parents to create new offspring chromosomes. Mutation is the flipping of a "bit" in the binary string between 0 and 1. This occurs with very low probability. In every generation a percentage of the population is replaced by new chromosomes. Selection of the chromosomes to be replaced is done on the grounds of their fitness.

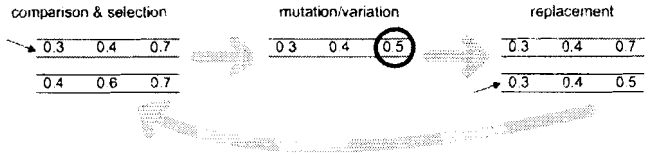

Fig. 4. Evolutionary cycle

Evolutionary strategies on the other hand, preserve only one chromosome. The chromosome consists of a real value list of the parameters. In every generation the chromosome is copied and mutated. Small random value changes are made to the parameters with some probability. The fitness of the new chromosome is compared with that of its parent and the stronger of the two is preserved.

Many hybrids and new variations of these methods have been implemented, including self-adaptation, variable-length chromosomes and multi-objective GAs. One of the most popular changes is the inclusion of real numbers in GAs. This leads to more systematic mutations and it also eliminates the back-and-forth conversion of chromosomes between binary and real numbers for crossover and evaluation. GAs have been applied with great success in many complex problems. They can solve combinatorial problems without the need for a continuous, convex solution space since they do not require gradient information. They do not suffer from the local minima problem commonly seen in gradient descent approaches and their parallel nature allows global search.

GAs have been proposed repeatedly for policy reinforcement [7], [11], [14]. In these schemes the actor-critic RL architecture is assumed and the parameters of the chromosomes represent the parameters of the actor. This approach seems especially attractive for the design of fuzzy controllers since their solution space could be infinite, non-differentiable and complex. Rules can be added arbitrarily and can have discontinuous effects on system performance. However, GAs are generally not suited for online learning [11]. GAs require the calculation of fitness of every chromosome, a process which can become extremely time-consuming. Note that the fitness function of evolutionary algorithms and the value function for RL are both used for evaluation and in both cases the goal is to find parameters to maximize these functions. While the value function for RL estimates the reward for a single state, the fitness of the policy is the summation of rewards over all states [2]. This value can at best be estimated by the summation of the value function over a finite duration of time [7]. In dynamically changing environments this process becomes to time consuming for from real-time application.

Schemes that have been developed that use GAs for policy reinforcement include evolutionary algorithm reinforcement learning (EARL), temporal difference and genetic algorithm based reinforcement (TDGAR) learning [7] and genetic algorithm fuzzy reinforcement learning (GAFRL) [11]. These schemes all extend the actor-critic design. TDGAR implements a neural network with TD learning for the critic and proposes a neural network or adaptive fuzzy network for the actor. This architecture can be seen in Fig.5. Each actor parameter set proposed by the GA is applied to the plant and evaluated by the critic. Results are fed back to the GA. GAFRL proposes a 5-layer neural fuzzy network for the actor. This allows for the complete fuzzy controller to be defined by a set of weights and for linguistic fuzzy rules to be extracted from and inserted into the network.

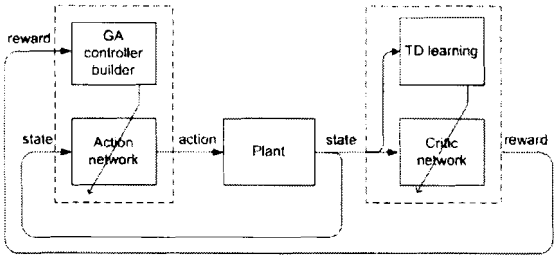

Fig. 5. TDGAR learning system

IV. Cognitive Control System

Controllers induce actions on dynamic systems in order to steer the system state towards some preset goal values. The actions generated by feedback controllers are based on the system state variables and the goal values. Classic controllers calculate control actions based on the error of a single parameter. This type of controllers is effective for linear systems, but since their designs require mathematical analysis of the system dynamics, they are not well suited for control of nonlinear multi-variable systems. Such systems require adaptive nonlinear controllers to perform the mapping between system states and control actions. Artificial intelligence techniques such as neural and fuzzy controllers are implemented to performing this task. These controllers are capable of learning the control policy from expert training data by means of supervised learning.

Intelligent agents are the ultimate form of intelligent control. Through RL they can autonomously learn to control complex systems. Schemes like TDGAR, GAFRL and the ACD family described in the previous sections have been implemented successfully in various applications.

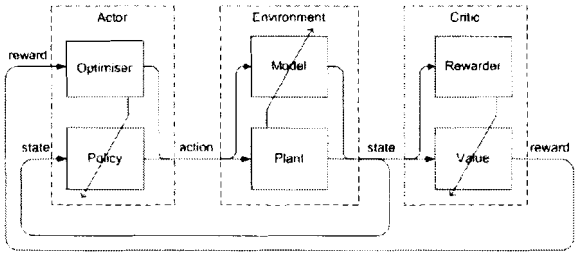

Fig. 6. Modular cognitive controller architecture

The architectures of these schemes are combined in Fig.6 into a universal modular cognitive controller architecture (MOCCA). Different techniques can be employed and tested in these modules without affecting the system design. Modules may be omitted in simpler applications. The names and functions of the modules are as follows:

1) Policy: This can be any adaptive control system like tunable PIDs, look-up tables, neural and fuzzy controllers or evolutionary systems. It produces actions based on the system state and goal. It is trained or tuned by the solver.

2) Plant: This is the online real-time system or process that is being controlled. In the case of a robotic system or agent it is the real-world environment the system is operating in. Its internal state changes as a result of action commands received from the actor. It might also have some uncontrolled inputs or disturbances.

3) Evaluator: The system satisfaction is symbolized by this module. It implements a value function, cost-function, fitness function, Q-value table or some other means of representing an estimated reward.

4) Rewarder: The role of generating the external reward is assumed by this module. This moves the responsibility of goal and failure detection from the environment to the critic. It uses any methods such as DP, Monte Carlo, TD or Q-learning to propagate the reward to the evaluator.

5) Model: This is an offline representation of the plant. It supplies the system with the expected next state. This model can be a pre-developed, pre-trained or learn online.

6) Optimizer: The control problem is finally solved by this module. If training data is available it could be use to adapt the policy through supervised learning. If reward data is available it could be used by RL method for action reinforcement or policy reinforcement.

A cognitive controller should be familiar with the consequences of its actions and its relationship with the environment. It should be able to plan ahead, anticipate, evaluate and optimize future results based on an internal representation of its environment. It should also be able to explore strategies, validate representations and evaluations and adapt accordingly. In the proposed architecture the modules can interact in many different ways to produce various cognitive attributes. These attributes are characterized by distinct patterns of data flow between the modules. Although some attributes are still quite primitive, they bare the potential to evolve into strong cognitive processes. Cognitive attributes that can be identified in this architecture is shown in Fig.7.

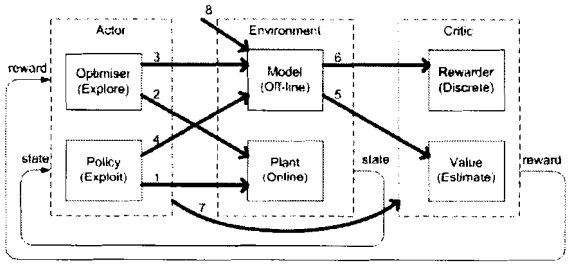

Fig. 7. Data flow patterns

1) Exploitation (policy to plant): The actor follows policy to control the plant. It is a direct reflex or instinct which can gradually change over time.

2) Exploration (optimizer to plant): The actor takes actions to get to know the plant. It could be random playing or guided experimentation in search of certain goals.

3) Contemplation (optimizer to model): The actor considers actions offline in search of expected results before taking the action. Creativity is expected to emerge here. This is discussed further in the next section.

4) Validation (policy to model): The actor tests its policy offline against its internal model and evaluator to verify that policy actions taken will produce desired results. This surety creates to self-confidence.

5) Planning (model to evaluator): The controller can go through many offline iterations of action evaluation to plan a sequence of actions in advance.

6) Perception (model to rewarder): Perceptions can be built by updating the evaluator based on the model instead of on the plant. This can significantly speed up adaptation, but false perceptions can seriously hamper performance.

7) Emotion (actor to critic): When the evaluator output is based on actions only, without regard of the consequences (model-free control), it often results in unjustifiable actions. This can be considered as emotional, irrational response. This pattern is implemented in action-dependent ACD.

8) Dreaming (preset model): The model can be set up to arbitrary states and the controller can operate on the model as if it was online. Unusual states can be explored and random goals can be pursued. This "dreaming" can lead to innovative new ideas and insights.

As the policy improves, actions taken should gradually shift from exploration to exploitation. This is generally the case with RL systems. Contemplation, dreaming and validation can continue during idle time. Discrepancies actual and expected results should prompt directed exploration of the plant. Planning should be considered whenever ample time is available for taking actions.

Perception improves the critic's training time, but should be phased out as the system matures. Emotion would normally not be implemented although it might be used to hardwire apparent irrational actions. By implementing neuro-fuzzy networks the actor and the model can also be initialized with prior knowledge about the system and a preliminary controller, which could drastically reduce training time.

V. Action Evolution

The two main strategies in RL with their respective methods have been described in Sections II & III. Evolutionary methods used for policy reinforcement require extended time for fitness evaluation and further suffers from high dimensionality. This renders this approach impractical for real-time applications. Gradient descent methods like the ACD family have been used successfully for action reinforcement [12], but they also have their shortcomings. They do not perform a global search and can easily get stuck in local minima. The iterative back-propagation through three networks (critic, model and actor) can also slow convergence down. The method of action evolution proposed here makes use of evolutionary methods for action reinforcement. This is achieved by introducing an intermediate stage of finding feasible actions for a given state. A GA explores the critic to find such actions. Finding this action is usually a problem of much less dimension than finding the full set of policy parameters. It further offers the parallel global search benefits of GAs with immediate fitness evaluation. These actions can be assessed instantaneously and do not require any trial period as is the case with policy reinforcement [7]. The results are recorded as state-action data pairs for supervised learning of the actor. The actor can be implemented by any mechanism that allows supervised learning.

In real-time applications, actions from the policy can be used for exploitation, and actions from the GA for random exploration. The critic network will be trained through RL in the normal fashion with a method such as TD learning.

In model-free systems the evaluator uses the proposed action to create a fitness function for the GA. In model-based systems it uses the expected states instead.

A one-pass supervised learning mechanism like dynamic evolving connectionist systems (ECOS), evolving fuzzy neural networks (EFuNN) or dynamic evolving neural-fuzzy inference systems (DENFIS) [15] are favorable for actor training. It doesn't require storage of the training data and has the additional advantages of adaptive rule based controllers: it can be initialized with available knowledge and a preliminary controller. However, dynamic systems should also be capable of gradual forgetting. Iteratively learning controllers such as neural networks can accomplish this but they are required to store the training data.

VI. Conclusion

This paper discussed different approaches followed in RL for building intelligent agents for continuous systems. It then proposed a generic control architecture which encapsulates previous architectures. The aim with this architecture is to elucidate the emerging cognitive attributes and assist in the progress towards a unified model for

cognitive control. The notion of "evolution of ideas" has been incorporated in the proposed architecture to provide an alternative method of RL. This scheme provides a method that adds the benefits of evolutionary methods to online learning real-time applications. It also promotes the prospect of advanced cognitive processes in agents.

A full evaluation of action evolution and a comparison with the alternative schemes like ADHDP and GAFRL should be conducted. Finally the potential of the advanced cognitive attributes needs to be researched in future work.

REFERENCES

[1] N.K. Kasabov, "Introduction: hybrid intelligent adaptive systems", Int. Journal of Intelligent Systems, 1998, vol. 6, pp. 453-454.

[2] R.S. Sutton, A.G. Barto, "Reinforcement Learning: An Introduction", MIT Press, Cambridge, MA, 1998

[3] J.A. Dominguez-Lopez, R.I. Damper, R.M. Crowder, C.J. Harris, "Adaptive neurofuzzy control of a robotic gripper with online learning", Robotic and Autonomous Systems, 2004, vol. 48, pp.93-110.

[4] P. Wawrzynski, A. Pacut, "A simple actor-critic algorithm for continuous environments", Proceedings of the 10th MMAR Int. Conf. Miedzyzdroje, Poland, 2004, pp. 1143-1149.

[5] P.J Werbos, P.J., "Approximate Dynamic Programming for Real-Time Control and Neural Modeling", in D.A. White & D.A. Sofge eds., Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches, Van Nostrand Reinhold, New York, 1992, pp. 493-525.

[6] D.V. Prokhorov, D.C. Wunsch II, Adaptive Critic Designs", IEEE Transactions on Neural Networks, 1997, vol. 8, no. 5, pp.997-1007.

[7] C.T. Lin, C.P. Jou, "Controlling Chaos by GA-Based Reinforcement Learning Neural Network".

[8] H. Hagras, T. Sobh, "Intelligent Learning and Control of Autonomous Robotic Agents Operating in Unstructured Environments".

[9] L.P. Kaelbling, M.L. Littman, A.W. Moore, "Reinforcement Learning: A Survey", Journal of Artificial Intelligence Research, 1996, vol. 4, pp. 237-285.

[10] M. Dianati, I Song, M. Treiber, "An Introduction to Genetic Algorithms and Evolutionary Strategies".

[11] C. Zhou, "Robot learning with GA-based fuzzy reinforcement learning agents" Information Science, 2002, vol. 145, pp.45-68.

[12] G.K. Venayagamoorthy, R.G. Harley, D.C. Wunsch, "Comparison of Heuristic Dynamic Programming and Dual Heuristic Programming Adaptive Critics for Neurocontrol of a Turbogenerator" IEEE Transactions on Neural Networks, 2002, vol. 13, no.2, pp. 764-773.

[13] G.G. Lendaris, T.T. Shonnon, "Designing (approximate) optimal controllers via DHP adaptive critic & neural networks".

[14] H. Hagras, V. Callaghan, M. Colley, "Learning and adaptation of an intelligent mobile robot navigator operating in unstructured environment based on a novel online Fuzzy-Genetic system", Fuzzy Sets and Systems, 2004, vol. 141, pp.107-160.

[15] H.K. Kasabov, Q. Song, "DENFIS: Dynamic Evolving Neuro-Fuzzy Inference System and Its Application for Time-Series Prediction", IEEE Transactios on Fuzzy Systems, 2001.

Appendix III

Source code and Documentation

An accompanying compact disk contains a soft copy of this document, all the Java™ source code developed for this project as well as an archive of public domain papers referenced in this dissertation.

References

[1] A.E. Ruano ed., *Intelligent Control Systems using Computational Intelligence Techniques.* IEE Control Series 70, 2005.

[2] G.F. Luger and W.A. Stubblefield, *Artificial Intelligence, Structures and Strategies for Complex Problem Solving, 2nd Ed.* The Benjamin/Cummings Publishing Company, Inc., 1993.

[3] R. Penrose, *The Emperor's New Mind.* Oxford University Press, 1989.

[4] S. Hameroff, "Consciousness, the Brain, and Spacetime Geometry", *The Annals of the New York Academy of Sciences, special issue Cajal and consciousness*, 2001.

[5] D.R. Hofstadter, *Gödel. Esher, Bach: an Eternal Golden Braid.* Penguin Books, 1979.

[6] V. Kecman, *Learning and Soft Computing, Support Vector Machines, Neural Networks, and Fuzzy Logic Models.* The MIT Press, 2001.

[7] S.Franchi and G. Güzelder eds., *Mechanical Bodies, Computational Mi*nds: Artificial Intelligence from Automata to Cyborgs. The MIT Press, 2005.

[8] R.M. Harnish, *Minds, Brains and Computers: In Historical Introduction to the Foundations of Cognitive Science.* Blackwell Publishers, 2002.

[9] C. Emmeche, *The Garden in the Machine: The Emerging Science of Artificial Life.* Princeton University Press, 1991.

[10] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach 2nd Ed.* Prentice Hall, 2002.

[11] C.M. Bishop, *Neural Networks for Pattern Recognition.* Oxford University Press, 1996.

[12] R.V. Florian, "Autonomous artificial intelligent agents," *Technical Report Coneural*, February 2003.

[13] R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 1998

[14] D.R. Hush and B.G. Horne, "Progress in supervised neural networks," IEEE Signal Processing Magazine, January 1993.

[15] Lecture notes, *Computational Intelligence*, Umeå University, Sweden, 2002.

[16] J.A. Freeman and D.M. Skapura, *Neural Networks: Algorithms, Applications, and Programming Techniques.* Addison-Wesley Publishers, 1991.

[17] E. Pampalk, G. Widmer and A. Chan, A new approach to hierarchical clustering and structuring of data with self-organising maps," *Intelligent data Analysis Journal.*

[18] W. Liu, L.Yang and L. Hanzo, "Recurrent neural network based narrowband channel prediction"

[19] H. Jaeger and H. Haas, "Harnessing non-linearity: Predicting chaotic systems and saving energy in wireless communication," *Science*, vol. 304, pp. 78-80, April, 2004.

[20] S. Kak, "A class of instantaneously trained neural networks," May 2002.

[21] T. Ojala, *Neuro-fuzzy systems in control*, Master of science thesis Tapere University of Technology, 1994.

[22] L.X. Wang, *Adaptive Fuzzy Systems and Control, Design and Stability Analysis*. Prentice Hall, 1994.

[23] J.R. Jang, C. Sun, E. Mizutani, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Pearson Education, 1996.

[24] J.H. Holland, *Hidden Order: How Adaptation Builds Complexity*. Perseus Books, 1995.

[25] A. Chipperfield, P. Flemming, H. Pohlheim and C. Fondeca, *Geneic Algorithm Toolbox For Use with MATLAB®*, Automatic Control and Systems Engineering, University of Sheffield, version 1.2.

[26] A.P. Engelbrecht, *Fundamentals of Computational Swarm Intelligence*. Wiley, 2005.

[27] M. Darigo, E. Bonabeau and G. Theraulaz, "Ant algorithms and stigmergy," *Future Generation Computer Systems*, vol. 16, pp. 851-871, 2000.

[28] L.N. de Castro and J. Timmis, *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, 2002.

[29] L.N. de Castro and J. Timmis, "Artificial Immune Systems as a novel soft computing paradigm," *Soft Computing Journal*, vol. 7, Issue 7, July 2003.

[30] K.A.J Doherty, R.G. Adams and N. Davey. *Hierarchical Growing Neural Gas.*

[31] H.K. Kasabov and Q. Song, "DENFIS: Dynamic Evolving Neuro-Fuzzy Inference System and Its Application for Time-Series Prediction," *IEEE Transactios on Fuzzy Systems*, 2001.

[32] O. Cordón, F. Herrera, F. Hoffman and L. Magdalena, *Genetic Fuzzy Systems: Evolutionary Tunning and Learning of Fuzzy Knowledge Bases*. World Scientific, 2001.

[33] N.K. Kasabov, "Introduction: hybrid intelligent adaptive systems", *Int. Journal of Intelligent Systems*, 1998, vol. 6, pp. 453-454.

[34] Z. Zhang, C. Zhang. *Agent-Based Hybrid Systems, An Agent-Based Framework for Complex Problem Solving*. Springer, 2004.

[35] P.J Werbos, "Approximate Dynamic Programming for Real-Time Control and Neural Modeling", in D.A. White & D.A. Sofge eds., *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, Van Nostrand Reinhold, New York, 1992, pp. 493-525.

[36] D.V. Prokhorov, D.C. Wunsch II, Adaptive Critic Designs", *IEEE Transactions on Neural Networks*, 1997, vol. 8, no. 5, pp.997-1007.

[37] C.T. Lin, C.P. Jou, "Controlling Chaos by GA-Based Reinforcement Learning Neural Network" *IEEE Transactions on Neural Networks*, vol. 10, pp. 846-859, 1999.

[38] P. Baerlocher, *Inverse Kinematics Techniques for the Interactive Posture Control of Articulated Figures*, Bachelors thesis, Department of Information, Ecole Polytechnique Federale de Laussanne.

[39] E. Burdet, R. Osu, D. Franklin, T.Yoshioka, T.E. Milner, M. Kawato, "A method for measuring endpoint stiffness during multi-joint arm movements," *Journal of Biomechanics*, 2000.

[40] F. Nori, *Symbolic Control with Biologically Inspired Motion Primatives*. Ph.D. Thesis, Department of Information Engineering, University of Padova.

[41] R.V. Florian, "Thyrix: A simulator for articulated agents capable of manipulating objects," *Technical Report Coneural*, August, 2003.

[42] S.R. Lindemann and S.M. LaValle, "Smoothly blending vector fields for global robot navigation."

[43] S.M. LaValle, J.H. Yakey and L.E. Kavraki, "A probabilistic roadmap approach for systems with closed kinematic chains," *Proceedings of the 1999 IEEE International Conference on Robotics & Automation*, May 1999.

[44] L.M. Gambardella and M. Dorigo, "Ant-Q: A reinforcement learning approach to the travelling salesman problem," *Proceedings of the Twelfth Intern. Conf. on Machine Learning*, pp. 252-260, 1995.

[45] H.M. Deitel and P.J. Deitel, *Java: How to Program, 3rd Ed.* Prentice Hall, 1999.

[46] T.Ziemke, "When is a cognitive system embodied?" Cognitive Systems Research, vol. 3, pp. 339-348, 2002.

[47] M. Neser, R.E. Tessendorf and G. van Schoor, "Action evolution for intelligent agents," *Proceedings of the 2005 IEEE International Symposium on Intelligent Control*, pp. 610-615, June 2005.