Computation Approaches for Continuous Reinforcement Learning Problems

Effraimidis, D.

Computation Approaches for Continuous Reinforcement Learning Problems

Dimitrios Effraimidis

A thesis submitted in partial fulfilment of the requirements of the University of Westminster for the degree of Doctor of Philosophy

Department of Computer Science
University of Westminster
September 2016

Abstract

Optimisation theory is at the heart of any control process, where we seek to control the behaviour of a system through a set of actions. Linear control problems have been extensively studied, and optimal control laws have been identified. But the world around us is highly non-linear and unpredictable. For these dynamic systems, which don't possess the nice mathematical properties of the linear counterpart, the classic control theory breaks and other methods have to be employed. But nature thrives by optimising non-linear and over-complicated systems. Evolutionary Computing (EC) methods exploit nature's way by imitating the evolution process and avoid to solve the control problem analytically.

Reinforcement Learning (RL) from the other side regards the optimal control problem as a sequential one. In every discrete time step an action is applied. The transition of the system to a new state is accompanied by a sole numerical value, the "reward" that designate the quality of the control action. Even though the amount of feedback information is limited into a sole real number, the introduction of the Temporal Difference method made possible to have accurate predictions of the value-functions. This paved the way to optimise complex structures, like the Neural Networks, which are used to approximate the value functions.

In this thesis we investigate the solution of continuous Reinforcement Learning control problems by EC methodologies. The accumulated reward of such problems throughout an episode suffices as information to formulate the required measure, *fitness*, in order to optimise a population of candidate solutions. Especially, we explore the limits of applicability of a specific branch of EC, that of Genetic Programming (GP). The evolving population in the GP case is comprised from individuals, which are immediately translated to mathematical functions, which can serve as a control law.

The major contribution of this thesis is the proposed unification of these disparate Artificial Intelligence paradigms. The provided information from the systems are exploited by a step by step basis from the RL part of the proposed scheme and by an episodic basis from GP. This makes possible to augment the function set of the GP scheme with adaptable Neural Networks. In the quest to achieve stable behaviour of the RL part of the system a modification of the Actor-Critic algorithm has been implemented.

Finally we successfully apply the GP method in multi-action control problems extending the spectrum of the problems that this method has been proved to solve. Also we investigated the capability of GP in relation to problems from the food industry. These type of problems exhibit also non-linearity and there is no definite model describing its behaviour.

Contents

List of Figures

List of Tables

Declaration

I declare that all the material contained in this thesis is my own work.

Signed: Dimitrios Effraimidis 30 September 2016

Chapter 1

Introduction

1.1 Problem Description

Evolutionary Computing (EC) is a powerful problem solving techniques where by a population of candidate solutions is evolved according to the merits of each one. The individual solution that comprise the population are all tested against one particular task then transformed, as a result, according to specific genetic operations; this process eventually create a new population. In contrast to other methods, where only one candidate solution is iteratively progressed towards a local minimiser, EC methods enforce a parallel search across a large number of possible solutions. As such, EC is a global optimisation technique, which has the advantage of managing to avoid local optima.

Reinforcement Learning (RL) is the theory relating to the learning of a specific task, through the interaction of an agent with an environment. The sole information transmitted back to the agent is a signal, the reward, which quantifies how "good" or "bad" the agent's action was. The spectrum of problems which can be tackled by RL range from those involving discrete states and actions, where the environment has only a discrete number of states and the agent can take actions only from a discrete set of actions, to continuous state-action problems. An important class of RL problems is that of "control problems", where a process, identified from a dynamic system, has to be optimally controlled by the agent.

EC techniques have mostly been applied to control problems through the parameter adjustment associated with classical control theory approaches. Direct use has been made of EC techniques involving Genetic Programming (GP) with encouraging results; however, such use has been applied only in limited circumstances. The two theories (Evolutionary

Computation (EC) and Reinforcement Learning (RL)) currently stand apart and are progressing in quite different directions. A major element contributing to this difference is that EC operates at an episode level, where an episode is a complete interaction with the task; in contrast, RL optimises its agent in a step by step manner.

1.2 Aims and Objectives

Evolutionary Computing methods and especially Genetic Programming have been applied in the past to simple Control problems extracted from the Reinforcement Learning literature. However, these method's abilities and capabilities, in this regard, have yet to be fully identified. Problems from RL studies like the cart-pole problem have been shown to be easily solved via EC; however, such problems are simple in structure. Here, in contrast, we investigate the application of GP to extremely challenging control problems such as the Helicopter Hovering problem. This problem is continuous in terms of states and actions and has four controls. It exhibits a nonlinear behaviour and also there exists coupling between the actuators involved.

Subsequently, we expand the area of applicability of GP by attempting to describe the inactivation patterns of a micro-organism which lives in whole milk and so try to improve on models in predictive microbiology. A comparative study is conducted, comparing results obtained, in relation to this problem, from Partial Least Squares (PLS) and Neural Networks (NN) techniques. This is in order to investigate the strengths of GP as compared to established Artificial Intelligence methodologies.

We finally attempt to bridge the theories of GP, RL and Neural Networks (NN) by unifying their structures in a coherent formalism. In order to achieve this, we enrich the function set utilised by GP methods with adaptable NNs. In particular, we propose a two level optimisation. One level operates on a per episode basis implemented via GP techniques and the other one operates on a per step basis and is implemented using NNs augmented with RL techniques.

1.3 Contributions

The main contributions to knowledge achieved by this thesis are:

- Introduction of GP into the methods used by the food industry to develop models describing complex patterns and the augmentation of the area of application of

Artificial Intelligence's tools.

- Expansion of the applicability of GP to control problems with multiple action spaces and the establishment of the methodology as a versatile problem solver for nonlinear, highly noisy environments.
- Novel modification of the Actor-Critic architecture with Levenberg-Marquardt learning on the Critic network - which strengthens its performance in terms of success rate and speed.
- Proposal of a hybrid architecture unifying the theories of GP, RL and NN. The solution power of the proposed architecture encompasses the capability to react to complex and noisy environments. The augmentation of the function set of the GP technique with NN gives the potential for the, usually fixed, tree structure to adapt to previously unforeseen circumstances.
- Development of software of the full Genetic Programming paradigm with or without the addition of Neural Networks in the function set.

1.4 Thesis Outline

The rest of this thesis is presented as follows. Firstly, the necessary background of the general theory of the Evolutionary Computation (EC) is described and the general structure of any evolutionary algorithm is outlined (Section 2.2). The representation, the role of fitness, and variation operators among others are discussed. This paves the way for a specific branch of EC the Genetic Programming (GP) (Section 2.3). The unique representational power of the tree structure, which the GP utilises, is analysed together with the variation operators used. The focus then is given to the description of the application of the GP scheme on the prediction of the survival curve of a pathogen into the milk (Chapter 3). Subsequently, successful examples from the area of intelligent control, especially those that employ the EC paradigm, are presented (Chapter 4). The present work concentrates on control problems, and the analysis of the performance of a controller derived from the GP scheme on the helicopter hovering control problem is discussed in detail (Chapter 5).

The control problems especially those from the Reinforcement Learning (RL) literature consist the main subject of this research. The necessary background of the RL theory and a special RL architecture (the Actor-Critic) follows the discussion. The former is

being described, also, in relation with the theory of the Neural Networks (NN), which are ideal for function approximation (Chapter 6). A novel modification of the Actor-Critic architecture is then introduced (Chapter 7) and its successful applicability in the triple inverted pendulum problem is demonstrated.

The next part of this work proposes a new scheme, which integrates the otherwise disparate theories of RL, GP and NN (Chapter 8). Subsequently the detail analysis of the proposed scheme, and its strengths and weaknesses, are discussed. Following this, the current state-of-the-art methodologies to regulate learning in an Actor-Critic scheme is described.

The final chapter (Chapter 9) contains the summary of the thesis, an evaluation of the results of this research and some future directions that this research can follow.

Chapter 2

Theory of Evolutionary Computation and Genetic Programming

2.1 Introduction

One of the most powerful mechanisms of nature is evolution. Every species has the ability, via reproduction, to perpetuate itself through time. A key component of this process has to do with the genes of each species - which convey the information about the characteristics of the offspring. The ability of the individual to survive and reproduce, in often adverse environmental conditions, provides a measure of the quality of the genes that the individual possesses. As nature is an ever changing environment, the species has to evolve and adapt in order to address the external challenges. The survival of the fittest, as described by Charles Darwin, exhibits a very strong mechanism which nature utilises to adapt to the environmental conditions.

Computer science has adopted this process in order to address problems that were intractable, analytically. Using this paradigm, the individuals of the population can be defined to be strings representing variables relative to the problem or, perhaps, complete computer programs whose purpose is to control a process. The genes represent the structure of the individuals themself and reproduction rules are enforced in order to transform the initial structure to a novel one. The driving force of this process is a measure (fitness) that designates how good or bad each member of a population is. The process usually continues to a pre-specified number of generations or until a satisfactory solution is reached.

There are a number of methodologies which exist under the umbrella of Evolutionary Computing (EC), namely: Genetic Algorithms (GA), Genetic Programming (GP), Evolu-

tionary Strategies (ES) and others. All of these have successfully been applied to a number of difficult and challenging problems.

2.2 Evolutionary Computation

There is no consensus among the scientific community of what definitively constitutes an Evolutionary Computing system. An abstract definition of EC is that it is a system that changes incrementally over time or, adopting the biology view point, a system that follows Darwinian evolutionary rules. In either case the characteristics of an EC system can be described in terms of four fundamental components [15]:

- one or more populations of individuals representing the problem structure and competing for scarce available resources;
- the dynamic evolution of the populations through the birth (creation) and death (deletion) of individuals;
- a well-defined measure (fitness) that represent the competence of the individual in terms of solving the problem, which plays a pivotal role in the reproduction and selection process; and
- a variation process which creates offspring that inherit characteristics from parent individuals but which are also distinct in relation to the seantecedent structures.

The first issue in relation to constructing an EC algorithm is the representation of the problem solutions that will constitute the evolving population. The individuals have to incorporate the important traits essential to solve the problem at hand. They may take the form of a vector of real numbers or more abstractly a continuously string of binary numbers (genotype). An individual will record, within its structure, a set of traits that are sufficient to face the particular problem; in this way, individuals simulate the actions of chromosomes. An entirely abstract formulation of an EC algorithm would have the steps outlined in the pseudo-code shown as Algorithm 1.

The initial population is comprised of randomly generated individuals. For each one of them we calculate a measure of their competence - their *fitness* to solve the problem. Then repeatedly we choose individuals, which we evolve under a set of transformations. The offspring subsequently compete for survival with the rest of the population and some

Algorithm 1 Generic EC algorithm

```
Generate an initial population
Calculate the fitness of each individual
for ever do
    Select an individual from the current population to be a parent
    Produce an offspring of the selected parent
    Calculate offspring's fitness
    Select an individual from the population to die
end for
```

individual eventually die. The whole process continues for ever or until the termination criteria has been fulfilled.

2.2.1 Representation

The first step in compiling an Evolutionary Algorithms (EA) - this term will be used interchangeably with that of EC - is the representation. Under this process, a connection is established between the physical system in the real world and the problem-solving procedure, where the evolution occurs. Objects representing the original problem are called phenotype and their encoding in the artificial evolutionary world are called genotype [19]. At representation phase, a mapping has to be established between the genotype to the phenotype.

One of the earliest representations, especially used with Genetic Algorithms (GA) is the binary [46]. The constituting variables of the problem are transformed into a continuous string of binary numbers in a platform independent way. The string is divided into smaller strings, the individual genes, which take a specific value, called the *allele*. The genes do not have to be equal in size and the encoding can take different forms - numerical binary or perhaps Gray encoding, for instance.

The representation is a mapping from the real world to the evolutionary search space, and sometimes it comes naturally to use language which is derived from the problem. In the case of discrete variables which take the form of distinct values from a particular set, infinite or finite, it is convenient to use an integer representation. If the possible instances of a variable representing navigation are in the set {*North, East, South, West*} then the set 0, 1, 2, 3 is sufficient and well defined for this purpose.

From a mathematical point of view, the most straightforward way to describe a problem is often through real-value variables. This is usually what happens in the case Evolutionary

Strategies [7]. The genotype is constructed as a set of genes represented as a vector of real-values. Obviously this occurs when the values comes from a continuous, rather discrete, distribution. Of course in a computer system real numbers have limited precision (even in floating point format) which means that, theoretically the set of real numbers is mapped onto a very large but discrete and finite set of values.

As noted, representation is problem specific. In cases where the solution is the order of a sequence of distinct actions, binary encoding fails. The alleles of the genes have to be unique and binary encoding cannot guarantee this. The best way, in this case, to bridge the natural world to the artificial, is to use a permutation of a string of integer values. Another possibility is to build the solutions in the form of a well structure tree, as is the case with Genetic Programming (GP). This type of EC will be covered independently in the next section.

2.2.2 Population

The population is the set containing all the possible solutions (individuals). It is a collection of genotypes which are all participating in the evolutionary process. At every instance in time the individuals are fixed; the population is changing on a generational level and progressively adapts to the problem requirements. The population facilitates the parallel search of solutions as it constitutes the basis for generating new search points in the solution space. The most important characteristic of the population is its size, which usually is constant throughout the process but can vary from generation to generation. The selection mechanism is applied to the whole population and in most cases is a probability distribution relative to the quality of the solution represented by each individual. In some cases, a similarity measure is also enforced on a population through the application of a distance or neighbourhood measure. This serves as a means to diversify the different solutions and avoid the homogeneity of the individuals of the population.

2.2.3 Evaluation Function (Fitness)

In the natural world, the ability of an individual to survive, determines its potential for reproducing and having a chance for its genes to perpetuate. In the artificial world, the survival of the fittest is replaced by a measure of how well the particular phenotype solves a problem. The evaluation function serves the role of quantifying this ability. The terms used in the literature for this measure is *fitness*. Fitness is problem specific and the selection

mechanism relies on it. In a control problem, fitness takes the form of how far the solution is from an equilibrium point. In a navigation problem, the evaluation function measures the steps required to reach a goal. Defining the fitness evaluation function is one of the most important decisions a practitioner has to make as the quality of the derived solutions is heavily reliant on the correct quantification of the members (solutions) of the population.

2.2.4 Selection Mechanism

In each phases of the evolutionary process, one or more individual are chosen to participate in either variation processes or simply to reproduce (survive) to the next generation. The selection processes plays the role of the environmental conditions that favour for the fittest. Parent selection, the first step in the variation process, will distinguish between individuals based on their quality [19].

There are a number of ways to implement this pressure towards optimality. The selection mechanism can be either *deterministic* or *stochastic* [15]. In the deterministic methods, each individual is selected a fixed number of times. In contrast, in the stochastic case, individuals are selected according to a probability distribution. The distribution varies from uniform, where each individual has the same probability $p(i)$, to linear, where the probability mass is transferred from the low fitness individuals to high fitness ones. The linear ranking mechanism can be farther extended into nonlinear ranking, where pressure for selection of the fittest is higher than in the linear case. In such a way, the method can be seen to become more elitist and greedy. In the extreme case, the truncation method chooses only the best or the worst individuals.

A more flexible selection method, that is widely used, is the fitness proportional method. Every member of the population is given a probability evaluated according to the formula:

$$p(i) = \frac{f(i)}{\sum_{k=1}^{M} f(k)} \tag{2.1}$$

where $f(i)$ is the fitness of each individual and $p(i)$ the awarded probability. The fitness proportional distribution is dynamic. In the early stages of the evolution, when there is a wide range in fitness-es, this method tends to be elitist but later on, when the population is becoming more homogeneous, the distribution transforms to a more uniform one.

Another method for exerting evolutionary pressure onto the population is the tournament selection method. A number, k, of individuals are selected uniformly from the pool. Then the best or the worst is selected and are designated as winners or losers, respectively.

In the case where $k = 2$, we have a binary tournament and it can be shown that this is equivalent to linear ranking. If $k = 3$, the distribution becomes quadratic and as the parameter k increases so does the level of elitism.

2.2.5 Variation Operators

The role of the variation operators is to alter the genotype of the parent/s, in order to search the solution space for better solutions. In nature the genes of an individual can change due to environmental conditions and in reproduction the offspring inherit some of the genes of their parents. Variation operators try to imitate nature's mechanism and enhance the capabilities of the individuals in the population. The major classification of the variation operators used is delineated by the number of parents involved in generating new candidate solutions - called the arity of the operator. Operators are divided into two categories, asexual and sexual. In asexual operators only one parent is selected and offspring are varied from their parent via mutation. In sexual operators two or possibly more parents are combined together to generate new individuals.

Mutation

Mutation is a unitary (arity equals one), asexual variation operator. It applies to one individual and generates one offspring. Mutation is always stochastic and involves a series of random operations on the initial structure. The amount of variation incurred is controlled by the number of genes modified and the specific processes used. Usually a natural distance measure can be associated with the genes of an individual and this measure is problem specific. An example is the Euclidean distance between two real valued vectors of genes. Mutation, in this case, will operate on a specific number of genes and to a degree controlled by the parameters of the probability distribution parameters used. A Guassian distribution is a typical example, which almost always will have a median of 0 and variance of s. The variance of the distribution regulates the extent of the adjustment of the genes and consequently the amount of exploration pressure placed on an individual. The average distance between the parent to the offspring in the Gaussian case would be s.

In GA, mutation takes the form of a bit-flip. The Hamming distance (number of different bits in two binary strings) prescribes the extend of the variation. In binary representation the level of mutation is not controlled only by the Hamming distance. The mapping of genotype to phenotype is a significant factor as a mutation with a distance of

one (one bit-flip) can potentially have a significant impact on the real number represented. If the bit that changes happens to be in the left most bits of the string of a specific gene, the value of the corresponding phenotype changes considerably.

The role of the mutation in the evolutionary process varies among the different types of EC. In genetic programming, mutation is not used frequently but in genetic algorithms it is consistently used to enrich the simulation with new solutions. In evolutionary programming it is the only variation operation employed to search the solution space.

An important property that every EC method should have is the ability to guide the search over the whole range of the solution space. Every point in the space should be accessible via the variation operations. Mutation has the theoretical (and actual) role of connecting the solution space. To satisfy this requirement, mutation has to be allowed to shift from any point in the solution space to anywhere else in the solution space. That is, any allele must be able to transform through mutation to any other allele. This property plays a pivotal role in guaranteeing that a specific method can in theory reach a global optimum.

Reproduction

Reproduction is a sexual operator. It usually involves two individuals randomly selected from the population. There exists a number of ways to combine the two genotypes, the most used one being the crossover. In crossover, the genotype of each parent is cut into two or more parts. Subsequently the offspring assembles genes from the fragmented genotype of its two parents. In general, there may be more than one cut off point. It can be considered that the odd numbered parts are derived from the first parent and the even numbered from the second. The recombination is stochastic in terms of the selection and the manner in which that the genes are re-assembled.

The analogous process in the physical world has been utilised by farmers for thousands of years; farmers and breeders consistently have chosen the best plants and livestock for their purposes - in order to pursue higher yields and animals with better characteristics. The same principal applies to the artificial world of EC. The combination of individuals which have different but desired genes generates new, hitherto unseen, solutions. In many cases the performance of the offspring is actually worse than that of their parents, but sometimes the process produces individuals that outperform their parents.

Recombination is representation specific. Different evolutionary methodologies implement reproduction is a different manner. In GAs using a binary representation, a cut-off

point is chosen on the genome strings of the participating parents and subsequently these are recombined. In this way, genes are cut into two. When the representation is a vector of real numbers the genes are taken as a whole. In some methods the average of the alleles is calculated, producing entirely new characteristics. In GP, a sub-tree is selected and exchanged between the two parents.

The variation induced in the new population depends on the number of crossover points and the similarity factor existing between the parents. As a consequence, the amount of variation introduced diminishes over time as the population becomes more homogeneous. A measure to calculate the variation of the crossover operator is the *disruptiveness*, introduced by Holland [27]. This metric evaluates the probability that a set of K genes will be inherited to a child from its parents. Increasing the crossover points increases variation, but at the same time reduces the possibility that a set of K genes will be passed over to the children.

Another important issue in reproduction is that a fixed number of crossover points raises the probability that close genes are inherited together as a whole, in contrast to more distant ones. The so called distance bias can be reduced by increasing the crossover points but with a price in terms of disruptiveness. Making the number of points stochastic tends to eliminate the distance bias. Here, for each point between two genes there is a probability that it will be a cut-off point. If that probability is 0.5, the total number of crossover points, in a genome with L genes, would be $\frac{L}{2}$ and the crossover is called *uniform crossover* [72]. The use of a uniform crossover has been proven to remove the possibility of distance bias.

Reproduction can produce either one or two children. If the new individual is a merge of the odd sections of the first parent and the even sections of the second, then if we alternate the process and use the even sections of the first parent and the odd from the second, it is possible to produce one more offspring. Some EC methods like Genetic Programming rely on a crossover which delivers two new individuals. The same is possible using Genetic Algorithms. This is predominantly a design issue, but tests have shown that if both parents participate in the evolutionary process then there is a slight improvement in the performance [15].

2.2.6 Survivor Selection Mechanism

Survivor selection or environmental selection distinguishes betweeen individuals based on their quality. This is similar to parent selection but it differs in that it is used in a different stage of the evolutionary process. Survivor selection operates in the combined population

of all parents and all offspring and is invoked after the offspring creation. From the set of μ parents and λ offspring decisions have to be made in order to compose a further set of μ individuals, which will constitute the next generation. The population size stays constant in most EC methods, and the survivor selection process is the stage whereby we renew the population and apply pressure by ensuring that the best (fittest) individuals survive.

There exist a number of options, but the ones most commonly used are fitness related or age related. In fitness related selection, the multi-set of parents and offspring is sorted, according to a fitness scale, and the top individuals are selected to survive. Another possible option is to select the λ worst individuals and replace them. In age related selection, the selection is confined to the offspring set [19]. The rule guarantees that each individual exists in the population for the same number of iterations. Finally a scheme commonly used in conjunction with the previous mechanisms is elitism. Elitism prevents the loss of the current fittest member of the population.

2.2.7 Termination Criteria

In nature, evolution never ends. It is an on-going process that responds to the state of the environment and produces species capable of surviving under (usually) adverse conditions. In the artificial world, though, there are limitations. The processing power required comes at a cost and all the algorithms are intended to produce a solution at some point in time.

In the case that, for a specific problem, we know the optimal solution, we can use this known solution as the termination condition. It is also possible to demand an optimum to within a sufficiently good precision. However, to the majority of the problems undertaken with these methods, and more importantly to the most challenging and difficult ones, there is little knowledge a-priori about their solution. This demands the use of alternative conditions that will guarantee the termination of the simulation. Popular such termination criteria are [19]:

1. The total number of generations reaches a predefined limit.
2. For a given time (number of generations) the fitness improvement is below a specific threshold.
3. The population's homogeneity increases beyond a specified level.

When any of these criteria are used, the best individual from all the generations produced is regarded as the outcome of the simulation and designated as the final solution.

2.3 Genetic Programming

Evolutionary Computation involves techniques where by, usually, a set of independent variables, whose dimension space is often high, are being searched in a guided manner to produce a solution that satisfies certain conditions. Usually the number of points or combinations to be investigated is so big that an exhaustive search is impossible. In the case of real value variables, the search space is infinite making it even harder to cover all the possible candidate cases. By borrowing processes from biology, EC is capable, through the use of a fitness function, to identify promising solutions and produce adequate results for the problem at hand. But in many cases it is not a set of variables that we have to investigate but dynamic structures that might be a function or a set of instructions. This void in the theory of EC has been addressed by Genetic Programming (GP).

Genetic Programming is a branch of EC where the objects being evolved are hierarchical computer programs. Unlike other evolutionary techniques, GP is searching through a vast set of special structures, that when compiled, represent computer programs. The size and shape of these structures is dynamically changed during the evolutionary process and the complexities of the corresponding programs can vary significantly. The generality of the paradigm and the minimal requirements of the method make it applicable to a diverse set of problems, and it has achieved some impressive success.

2.3.1 Overview

In GP the population is comprised of computer programs able to compute a certain task. The level of competence of each individual (program) is measured according to the ability that the individual possesses in solving the problem. This specific measure, namely fitness, drives the evolutionary process. At every evolutionary step the current population transforms to a new population under the influence of alterations to the initial structures. Which individual is selected to evolve depends on its fitness level. The major genetic processes taking place are reproduction and crossover. When applied to a population, they produce a new generation of individuals. The process is repeated until a stopping criterion is met. The general flowchart of the steps involved in GP is depicted in Figure 2.1 [36].

2.3.2 The Structure - Representation

In every adaptive system,by definition, at least one structure undergoes adaptation. This structure always represents a potential solution to the problem at hand. In GP, as in

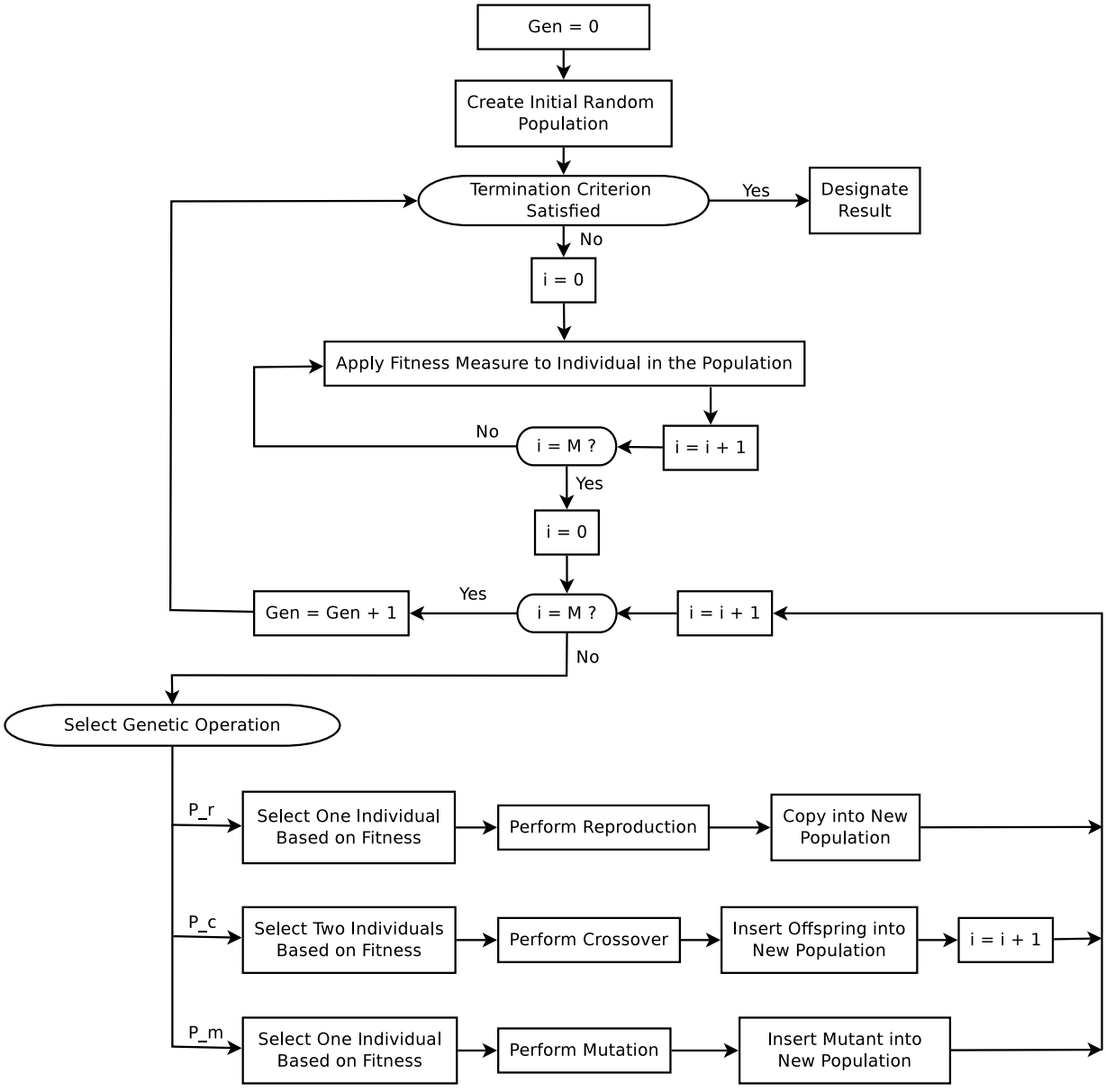


Figure 2.1: Genetic Programming Flowchart

other evolutionary methodologies, there is a population of structures, which are termed individuals, that performs, under the rules of the specific evolutionary paradigm, a parallel search in the space of the solutions.

The individual structures in GP are hierarchically structured computer programs. In contrast to genetic algorithms by which a string, usually representing the independent variables of the problem, is adapted towards an optimum, in GP systems, dynamic and complex programs are transformed through the adaptation process. This feature empowers the search method with the ability to identify complicated and unforeseen properties of the solution and the variables involved.

Each program can be represented either in LISP format or as a hierarchical tree - with each node representing a function or a specific quantity. The size, shape and contents of the tree or the LISP expression change via the adaptation process. The structure of the individuals in a GP system depends on the functions set $F = f_1, f_2, ..., f_N$ and the terminal set $T = t_1, t_2, ..., t_M$. The set of all possible structures is the set of all possible compositions of the functions and terminals. Every function f_i takes a specific number of arguments a_{f_i}. In the case of a tree representation, this dictates the number of branches that a tree node will have.

Particular functions that the function set may include are [35]:

- arithmetic operations +, -, *, /
- mathematical functions (e.g., $sin(x), cos(x), exp(x)$)
- Boolean operations (AND, OR, NOT)
- conditional operators
- functions causing iterations
- functions causing recursions
- domain-specific functions

The terminals are the leaves of the trees and usually represent the input variables to the structure. A terminal can also be a constant quantity (such as the constant number $\pi = 3.14159$). A typical example of an individual is depicted in Figure 2.2. In this case, only mathematical operations and functions were used purposely in order to illustrate that a GP system can intelligently search through a set of functions. The variables of this function, X and Y, are the terminals of the tree.

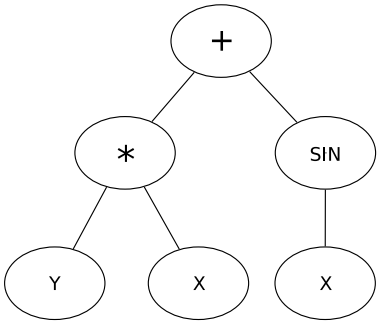


Figure 2.2: A tree representing the function $Y * X + SIN(X)$

Closure property

When using such a diverse set of functions in a tree structure to compile different computer programs, a problem that may arise is that one particular function may not be able to accept as an argument the result of some other function. In mathematics, special cases which arise out of the fact that different functions accept only a defined range of values are abundant. Typical examples of such functions are the square root, which is defined only over non-negative numbers, and division, which is undefined when the denominator is equal to zero. For this reason the terminal and function set has to be selected in a manner that they satisfy the *closure property*.

The closure property entails that each function in the function set has as a range any possible outcome that may be produced from any other function in the set and any possible values that the terminals may assume. The closure property guarantees the uninterrupted progression of the GP process. Even though this property may seem very restrictive initially, some simple modifications to the usual operations can transform the problematic functions to ones that comply with the requirements.

In the case of arithmetic division, the operation can be redefined to produce the value of 1 whenever the denominator is equal to 0. This is called protected division and the modification prevents the collapsing of the calculations due to an undefined outcome. Accordingly the square root can be prescribed to take as an argument the absolute value of quantity at hand $[sqrt(abs(x))]$. The same prescription can be applied to the logarithmic function $[log(abs(x)]$ with the provision that in the case of the input being 0 the function will return 0. Lastly to include logical function side by side with arithmetic ones, a workaround is to apply numerical-value logic; the value $True$ is defined to be equal to 1 and the value $False$ equal to -1 [35].

The closure property is desirable but not absolutely necessary. An alternative is to de-

fine that the outcome of an operation can be *:undefined* and that this is propagated through the overall expression. If any result is *:undefined* the final result becomes *:undefined*, and we can assign a big penalty to the infeasible individuals which produce this result. Thus, the chances of such an individual being selected and reproducing are minimum.

Sufficiency property

The purpose of GP is to search through the space of the function and terminal sets for a solution to the problem. The sufficiency property demands that the set of the compositions possible using the primitive functions and terminals includes the solution. Otherwise the GP system will fail to produce a solution or will converge to a sub-optimal one.

This property can be decomposed in two parts, one for the functions and one for the terminals. In the case of the terminals the problem is universal. In almost every scientific problem the researcher has to identify the variables that have sufficient explanatory power to solve the problem. These variables are problem specific and they might be obvious, as in the case where we try to optimise a multivariate function or to control a dynamic system for which the independent variables involved must be adequate to synthesise an appropriate solution. It might require considerable insight to be able to recognise the essential quantities that affect the solution.

The second part of the sufficiency property demands that the function set is an appropriate one for constructing a function that has the ability to solve the problem. In some domains this requirement has been studied and has a definitive answer. For instance, in relation to the Boolean functions, the function set $\{AND, OR, NOT\}$ is known to be sufficient to represent any Boolean function. Also if the problem involves finding a polynomial, the four fundamental arithmetic operations will suffice. But for the vast majority of problems the primitive function set needed is not clear. A good strategy is to try to include more functions than that which may be minimally required to guarantee a solution. The extraneous functions have multiple effects on the solution process as they may result in the degradation of the performance at some cases, but on the contrary they may accelerate and simplify the generation of the outcome.

The sufficiency property has a vital role in GP and in all the other Artificial Intelligence methods. A GP system searches for, and constructs, programs with the ability to solve a specific task. If all the ingredients together, in this case the function and terminal sets, lack the capability, overall, to compose a solution, then no methodology, intelligent or otherwise, will lead to a solution.

2.3.3 Initial population

The first step in GP is to supply an initial population. This population will consist of randomly generated trees. Obviously the initial population will comprise individuals that cannot solve the problem and will score poorly. To construct an individual, a function is selected to be the root of the tree. This way degraded cases, where the tree is just a single terminal, are avoided. As each function is defined to require a number of arguments $z(f)$, an element from the combined set $C = F \cup T$ of functions F and terminal T is selected (using a uniform probability distribution) to be the argument of the function. This step is repeated as many times as the number of arguments. If a terminal is chosen, the process for that branch is completed. If a function is attached to the branch then the process continues recursively for the corresponding arguments of the new function. The procedure ends when all the endpoints are terminals.

The generation of the initial population can be implemented in different ways. It is advantageous for the variety of the initial individuals to be high in terms of size and shape as this facilitates the search for competent structures later on. That necessitates the employment of methods which guarantees the creation of diverse individuals. Two such methods are the "full" and the "grow" methods. Both control the length of the trees. The length is defined as the longest path from the root to an endpoint.

The "full" method generates trees where the length of every endpoint from the root equals a maximum permitted value. This is accomplished by restricting the selection of every element only from the function set F, if the distance from the root is less than the specified valued. The "full" method creates similar structures in term of size and shape. In contrast, the "grow" methods allows a terminal to be selected, because the combined set $C = F \cup T$ of the functions F and the terminals T is used irrespective of the depth of the element. If the maximum depth is reached in a branch then the "grow" method dictates that the next element can only be from the terminal set, T. In the "grow" method, the length of a path never exceeds a pre-specified maximum value. The process creates different shapes and sizes and enriches the initial population with unique structures.

In practice both methods, "full" and "grow", are employed in parallel to generate the individuals for the initial population. In the so-called "ramped half-and-half" method the maximum depth for each individual is set to be between two values; in addition, half of the individuals will be built by the "full" method and half by the "grow" method. The resulting tree population varies considerably in terms of shapes and sizes, increasing the diversity of the initial population; this is essential for the GP to perform well.

2.3.4 Fitness

Genetic Programming, as any other evolutionary methodology does, mimics Darwinian natural selection. In nature, the merit of an individual is measured by its ability to reach reproductive age. In the artificial life of mathematical algorithms, the selection of an individual is proportional to its ability to nearly solve or completely solve the given problem. The term used, in reference to nature, for this ability is *fitness*. Each individual is tested and a numeric value representing its competence is assigned to it. In GP, fitness takes the form of four distinct measures:

- raw fitness
- standardised fitness
- adjusted fitness
- normalised fitness

Raw fitness is a measure naturally arising from the problem at hand. If the problem is the navigation of an agent from a start to an end point, the number of steps used quantifies the agent's competence. In regression, the difference between the calculated value of the tree and the correct one designates the efficiency of the individual. Usually the fitness is evaluated over a range of fitness cases covering the spectrum of all possible cases that the individual may face. The fitness cases should be representative of the domain space of the problem. For control problem where the objective is to reach a point in the state space and remain there, the fitness cases may be defined in terms of sets of initial conditions.

The most common raw fitness used is error. The type of the result of the individual tree expression can be anything from Boolean, integer, floating point, vector to symbolic. The raw fitness is measured by the sum of the distances over all the fitness cases. When the result of the expressions are integer or real numbers, the sum of the absolute values of the differences between the identified reference points and the outcomes of the individual, i, is used to calculated the fitness $r(i)$:

$$r(i) = \sum_{j=1}^{N_f} |S(i,j) - C(j)| \tag{2.2}$$

where $S(i,j)$ is the value returned from the individual tree, $C(j)$ is the correct value and N_f is the number of fitness cases. In the case of a vector-value problem, the measure is

generalised by summing the error in every element of the vector separately.

The standardised fitness $s(i)$ is used to transform the raw fitness to a measure whereby the lower numerical value is better. In cases where the raw fitness favours lesser values, standardised and raw fitness coincide, as in the case of equation 2.2. This transformation is problem specific and is applied to facilitate the calculation of the adjusted fitness.

The ajusted fitness range is $[0, 1]$ where greater values correspond to better individuals. The formula to compute the adjusted fitness is:

$$a(i) = \frac{1}{1 + s(i)} \tag{2.3}$$

where $s(i)$ is the standardised fitness. An important property of adjusted fitness is that as the evolutionary process progresses and the individuals score better (and when the ideal outcome is 0) then small differences in values are projected to large differences in the adjusted measure.

Finally, in the majority of cases, where the selection mechanism is fitness proportionate, normalised fitness is used. Normalised fitness is computed according to the formula:

$$n(i) = \frac{a(i)}{\sum_{k=1}^{M} a(k)} \tag{2.4}$$

where M is the number of individuals in the population and $a(i)$ is the adjusted fitness. From this formula, it is apparent that the normalised fitness is between 0 and 1 and the sum of the normalised fitnesses of the population equals 1. These properties make this fitness measure appropriate to assign probabilities that are fitness proportional.

2.3.5 Primary Evolving Processes

The two primary operations that are constituent parts of all GP methods, and are employed to evolve a population of trees to a new one, are:

a. Reproduction

b. Crossover

Reproduction can be found in almost every evolutionary computation methodology. Crossover is specific to the special representation of the individuals as trees.

Reproduction

Reproduction in GP imitates one of nature's processes that ensure the survival of the fittest. Reproduction is asexual as it operates in regard to one parental individual - and it produces one offspring. The operation is conducted in two steps. Initially an individual is selected from the parent population in accordance with some selection mechanism which relies on fitness. At the next step, the selected individual is copied from the parent population to the offspring one, unaltered.

The most prominent selection mechanism is fitness-proportionate selection. It is based on Holland's seminal work *Adaptation on Natural and Artificial systems* [27], and the method designates that the probability that a member of the population is selected is proportional to that individual's fitness. This means that if the fitness of an individual i is $f(i)$ and the population has M members, then the probability (of selection) is calculated as:

$$P(i) = \frac{f(i)}{\sum_{k=1}^{M} f(k)} \tag{2.5}$$

The fitness measure used is usually the normalised fitness $n(i)$, which fulfils the axioms of probabilities. So an individual is selected to reproduce to the new population in correspondence with its normalised fitness.

Fitness proportionate selection is not the only selection method used. Alternative methods are tournament selection and rank selection [21]. In tournament selection, two or more individuals are selected randomly (under a uniform probability distribution) from the population and the one with the highest fitness is appointed the winner. In the rank selection method, a number of individuals are selected according to their rank in respect to fitness. This counterbalances the dominance of a few good individuals, which happens usually at the early stages of the evolutionary process. Another effect of rank selection is that at the later stages, when the population converges to individuals with a similar good fitness, the rank selection favours the very best even though the difference in fitness is small. This effect is not possible to achieve under fitness proportionate selection.

Under any of these selection methods, the individual remains in the current population and in general can be chosen multiple times. Reproduction with alongside crossover helps the evolutionary process to converge towards a set of individuals with desired properties.

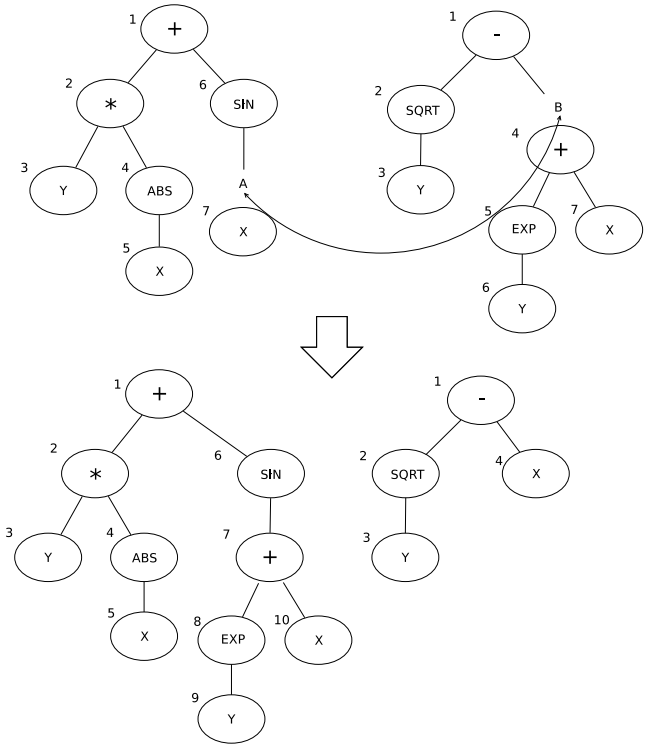


Figure 2.3: Crossover operation

Crossover

Crossover is the primary search method employed by GP systems in order to investigate new points in the solution space. With crossover, offspring are produced from fragments of the trees that participate in the operation. Two parents are selected from the current population and two offspring are copied after the operation to the new population.

The selection of the two parental individuals may be undertaken via any fitness related method. Once the selections have been made, a crossover point in each of the two trees is selected, using a uniform probability distribution. A crossover point defines a sub-tree consisting of the crossover point itself and whatever lies below it. Finally the two fragments of the parental trees are exchanged. This means that the first parent's sub-tree is attached to the crossover point of the second parent and vice versa. The operation can be seen in Figure 2.3. Because of the closure property and the fact that entire sub-trees are exchanged, the crossover operation produces syntactically correct trees.

A number of interesting cases arise from the positioning of the crossover points. If the two crossover points are at the root of the trees then the crossover degenerates into a straightforward reproduction. In the case that two terminals are selected as crossover

points, then the structure of the trees remain the same and a point mutation is performed instead. Also when the root is selected in one tree, but not the other, the entire tree will be add as a sub-tree to the second parent, increasing considerably the depth and the complexity of the resultant offspring. Usually the trees produced from crossover have a maximum permissible size to avoid the construction of extremely large individuals. This doesn't have a negative effect to the quality of the solutions as if, for instance, the maximum depth is defined to be 17, the maximal program specified as such a binary tree will consist of 2^{17}, or 131.072 elements.

2.3.6 Secondary Evolving Processes

In addition to the two primary genetic operations of reproduction and crossover, there are a number of secondary operations that can be useful, and facilitate the process in moving towards the optimum solution.

Mutation

Mutation is an operation predominately in used Genetic Algorithms principally to reintroduce diversity into the population. In the case of premature convergence whereby a gene in the chromosome has been lost, mutation may reintroduce into the string a particular section that had vanished during earlier stages. However, mutation does not play the same role in GP. As the constituent elements of a GP structure are a small number of functions and terminals, it is quite improbable that one of these will be eliminated.

Mutation is asexual and is implemented differently in GP from the way it is implemented in GA. One tree is chosen, proportionally to each fitness, by the same selection method as reproduction or crossover. Then a point in the tree is selected at random (as in crossover) and the sub-tree at and from that point is discarded. Finally a new randomly generated sub-tree, produced using a process similar to that used in the construction of the initial trees, is attached to that point. Mutation introduces new structures to the population, but in general, crossover provides the same functionality. As an example, consider the case where a terminal is selected as the mutation point and another terminal replaces it. Crossover could, potentially, have the same effect on the structures when two terminal points are selected.

Permutation

Permutation is asexual. It starts by selecting an individual in the same way as in reproduction. Subsequently a function (an internal point) of the tree is chosen. If the number of arguments of the specific function is k, then there exist $k!$ possible permutations of the order to the arguments. One of these permutations is randomly selected and the subtrees/leaves of the functions are permuted accordingly. Of course if the function happens to be commutative then the operation has no effect in terms of the final result - as is the case with addition.

2.3.7 Termination Criteria - Result Designation

In nature, evolution is a never ending process. But in the case of artificial systems, a result has to be designated as the desired solution at some point in the process. The termination criteria used depend on the problem at hand. If there exists a correct, identifiable, solution then this can serve as the criterion causing the termination of the evolutionary process. In general, though, this is not feasible. In problems such as those that involve optimisation, we do not know from the start what the optimum solution might be, and we will be unable to identify it subsequently. In such cases, a predefined number of generations, G, are specified and when we reach this number of generations, the individuals, from all generations, with the best fitness are designated to be the final result.

2.4 Next Chapter

Having covered the theoretical background around evolutionary methods, we continue with specific applications in order to evaluate the merits and limits of the theory on the related problems of this thesis. System Identification tries to find the relationship between an input and the dependent output. The area becomes essential to Reinforcement Learning problems when the model of an environment is required. GP could search through the set of functions composed by a set of elementary mathematical functions, as System Identification asks for a functional relationship. The coming chapter investigates the applicability of GP in a food processing problems where the concentration of a pathogen in different environmental conditions is required.

Chapter 3

Modelling of Survival Curves of Listeria monocytogenes using Genetic Programming

3.1 Introduction

High pressure processing is a non-thermal food preservation method that utilises hydrostatic pressures in the range of 300-700 MPa. The effects of this preservation method on foods include the inactivation of microorganisms, protein denaturation, enzyme activation or inactivation, and the retention of quality and freshness. A variety of pressure-treated commercial products such as jams, fruit juices and fresh whole oysters, are already commercially available in the United States, Japan and Europe [74], whereas another potential application in the food industry is the production of novel meat, poultry, fish and dairy products.

However, as foods are, by definition, implicated as carriers of food-borne pathogens, it is important to provide information on the effects of high-pressure processing on these micro-organisms. The inactivation of micro-organisms by high pressure is well documented/explained in the literature; however it is generally accepted that high pressure also results in morphological, genetic and biochemical alterations causing cell death due to accumulated damage [67].

Listeria monocytogenes is a Gram positive bacteria which acts as the main agent of listeriosis. Listeriosis is usually a severe disease with a mean mortality rate in humans of at least 20%. It primarily causes infections in immune-compromised patients, pregnant

women, and those at extremes of age (new-borns and the elderly). The occurrence of listeriosis, caused by forborne Listeria monocytogenes, in the European Union has shown a fluctuating trend during the last five years. But there have been particularly high notification rates in some countries, including Finland and Spain where the total number of cases increased by 97% and 90%, respectively, from 2005 to 2010.

The pathogen can grow at refrigeration temperatures and survive in foods for prolonged periods of time under adverse conditions [42]. It is a very hardy micro-organism that can grow over a wide range of pH values (4.3 to 9.1) and temperatures (from 0 to 45 °C) . In addition, it is relatively resistant to desiccation and can grow at a_w values as low as 0.90 [51]. To establish a process that is sufficiently effective for the safety of a food commodity, the pressure-destruction kinetics of spoilage and the pathogenic micro-organisms related to the specific product should be established and described in detail [68]. The inactivation of micro-organisms by heat and other processing methods has been traditionally assumed to follow first-order kinetics. However, significant deviations from linearity have frequently been reported [57]. Three kinds of deviations have been observed: curves with a shoulder, curves with tailing, and sigmoid-type curves. A number of models have been proposed to describe these nonlinear survival curves, such as the Weibull [75] and the modified Gompertz [8].

Developing models from observed data is a fundamental problem in many fields, such as statistical data analysis, signal processing, control, forecasting, and computational intelligence. There are two general approaches to system identification, namely, the parametric and the non-parametric approach. When the observed data is contaminated (such that it does not follow a pre-selected parametric family of functions or distributions closely) or when there are no suitable parametric families, the non-parametric approach provides more robust results and hence, is more appropriate. Neural networks have become a popular tool in non-parametric function learning due to their ability to learn rather complicated functions. The multi-layer perceptron (MLP), along with the back-propagation (BP) training algorithm, is probably the most frequently used type of neural network in practical applications [24].

An alternative, well-known, solution for non-parametric identification is Genetic Programming (GP). GP applicability can be expanded by using the method for identification of complex nonlinear systems that describe natural phenomena. The aim of this current research study is to investigate the feasibility of utilising GP as an alternative to classical approaches in the area of food microbiology. The overall objective of this study is to design

an accurate prediction scheme which models, using a proposed GP structure, the survival of Listeria monocytogenes in ultra high-temperature (UHT) whole milk during high pressure treatment. The proposed prediction scheme is compared against the multilayer neural network perceptron (MLP). Similarly, an assessment is made against two well-known nonlinear conventional models (Weibull, Gompertz) used for food microbiology. The goodness-of-fit of all resultant models is evaluated and compared.

3.2 Experimental Case

3.2.1 Bacterium and preparation of cell suspension

The entire experimental case study was performed at the Agricultural University of Athens (AUA), Greece [6]. Stock cultures of Listeria monocytogenes were maintained in vials of treated beds in a cryoprotective fluid at -80 °C until use. The culture was revived by inoculation into 9 ml of Tryptic Soy Broth, supplemented with 0.6% yeast extract and incubated at 30 °C for 24 h. For all the experiments, a loop-full of the culture was transferred into 9 ml of the same medium and sub-cultured twice at 30 °C for 24 h. The cells were harvested by centrifugation (6,000 rpm for 30 min at 4 °C), washed twice with sterile phosphate-buffered solution, re-centrifuged, and then re-suspended in the same diluent to give a final concentration of about 9.0 log CFU ml^{-1}, as assessed by a Neubauer counting chamber. This inoculum was used in all experiments.

3.2.2 High pressure equipment

The high-pressure system consists of a high-pressure intensifier unit for the build-up of pressure in the system and an electric motor to drive a hydraulic pump. The oil in the pump is used to propel the oil-driven double-acting intensifier, which is actually a hydraulically driven reciprocating pump. In the intensifier, the pressure of the high-pressure fluid can be ranged up to 1000 MPa. The pressure is adjustable in steps of approximately 25 MPa. Moreover, the system consists of a block of six small (42 ml) high-pressure vessels measuring 2.5 cm in diameter and 10 cm in length, respectively. The vessels are closed with a unique Resato thread connection at the top of the vessel. The pressure is transmitted from the intensifier to the vessels by the pressure fluid through high-pressure stainless steel tubing. Air-operated high-pressure needle valves are used for the control of the circulation of the pressure fluid so that each vessel can operate independently. Each vessel is equipped with

a heating/cooling jacket to control experimental temperature in a range of -40 to +100 °C. Temperature transmitters are mounted in each vessel to monitor temperature. Finally, two pressure transducers are used to monitor the pressure in the system.

3.2.3 Pressurisation of samples

Aliquots of 3.0 ml of sample containing 0.3 ml of cell suspension and 2.7 ml of UHT whole milk were transferred in polyethylene pouches (20 mm width X 80 mm length) and heat-sealed, taking care to expel most of the air. Each pouch was placed in a second slightly bigger one in order to prevent accidental leakage of cell suspension and contamination of the pressurising liquid. The pouches were placed in duplicate in each of the six small vessels of the high pressure unit, and the system was pressurised at 350, 450, 550 and 600 MPa, respectively. The initial temperature of the vessel jacket was adjusted to 25 °C by means of a water-glycerol solution circulating from a water bath. The come-up rate was approximately 100 MPa per 7s and the pressure release time was less than 3s. At selected time intervals, pressure levels were isolated individually from the pressure system, and the pressure of the vessel in question was released. Pressurisation time reported in this work did not include the pressure come-up and release times. Each experiment was repeated twice and duplicate pouches were used for each pressure/time combination.

3.2.4 Enumeration of survivors

Immediately after treatment, the pressurised pouches were removed from the vessels and their contents were aseptically diluted in 1/4 sterile Ringers solution. One hundred micro-litres (100 μl) of at least three serial dilutions were spread-plated in triplicate on, the non-selective, Tryptic Soy Agar medium supplemented with 0.6% yeast extract for the enumeration of L. monocytogenes. The plates were allowed to incubate at 30 °C for 48 h to form visible colonies, and then re-incubated at the same temperature for an additional 24 h to allow injured cells to recover. The data from the plate counts were transformed to log10 values prior to further analysis. The survival curves of Listeria monocytogenes inactivated by high hydrostatic pressure, obtained in relation to six pressure levels (350, 400, 450, 500, 550 and 600 MPa) applied to UHT whole milk, are shown in Figure 3.1. The shapes of the survival curves change considerably depending on the treatment pressure levels. However, in all pressure levels assayed, a clear inactivation pattern was observed, including a lag phase (or shoulder), a log-linear and a tailing phase. As expected, the

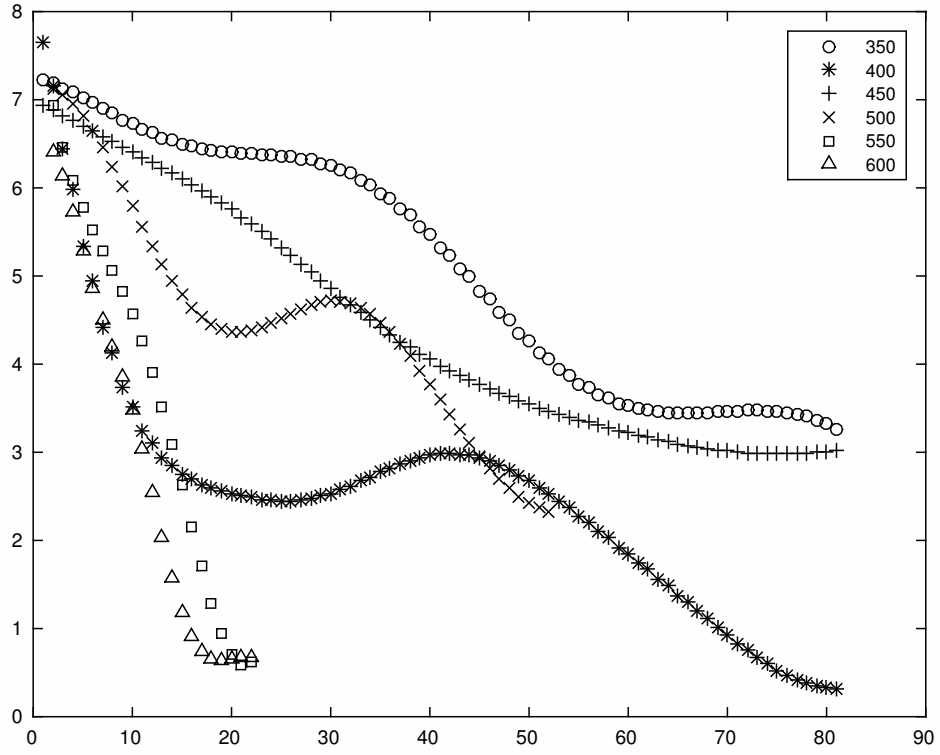


Figure 3.1: Illustration of the experimental data

duration of the shoulder was pressure dependent, so higher pressures resulted in a lower shoulder time. At different pressure levels, survival curves showed a pronounced curvature and tailing, indicating that a small population of the pathogen could resist pressurisation and eventually survive in milk.

3.3 Model development

3.3.1 Primary Modelling

The survival curves of L. monocytogenes related to high pressure inactivation were fitted to two parametric primary models in order to determine the kinetic parameters of L. monocytogenes in UHT whole milk. The first model applied was the re-parameterised Gompertz equation [83] determined by the following equation:

$$log_{10}N(t) = log_{10}N(0) + Aexp\{-exp[\frac{ke}{A}(t_s - t) + 1]\} \tag{3.1}$$

where t_s (min) is the duration of the shoulder, $k(min^{-1})$ is the maximum specific inactivation rate, $N(0)$ (log CFU ml^{-1}) is the initial population density, and A (log CFU ml^{-1}) is the difference between the initial and the residual population.

The second model was based on the modified Weibull equation [4] which can be defined as:

$$log_{10}N(t) = log_{10}[(N(0) - N_{res})10^{-(\frac{t}{\delta})^p} + N_{res}] \tag{3.2}$$

where δ (min) is a scale parameter denoting the time for the first decimal reduction, and p (dimensionless unit) is the shape factor of the curve. For $p > 1$, convex curves are obtained whereas for $p < 1$ concave curves are described. $N(0)$ and N_{res} (log CFU ml^{-1}) are the initial and residual population densities of the pathogen, respectively. Both models were fitted by the nonlinear regression procedure provided in the Statistica software.

3.3.2 Non-Parametric Modelling

Partial Least Squares (PLS) regression, a multivariate calibration technique, projects the initial input-output data down into a latent space, so extracting a number of principal factors (also known as latent variables) with an orthogonal structure, while capturing most of the variance in the original data. In brief, it can be expressed as a bi-linear decomposition of both X and Y as:

$$X = TW^T + E_X \tag{3.3}$$

and

$$Y = UQ^T + E_Y \tag{3.4}$$

such that the scores in the X-matrix and the scores of the yet unexplained part of Y have maximum covariance. Here, T and W, U and Q are the vectors of X and Y PLS scores and loadings (weights), respectively, while E_X, E_Y are the X and Y residuals. The decomposition models of X and Y and the expression relating these models through regression constitute the linear PLS regression model. In cases of one Y-variable, y, the model can be expressed as a regression equation

$$y = bX + E \tag{3.5}$$

where b is the regression coefficient. The PLS model is developed in two stages; the initial dataset is divided into training and testing subsets. The former dataset is used to build the

models and compute a set of regression coefficients (b_{PLS}), which are subsequently used to make a prediction of the dependent variable in the test subset.

The multi-layer perceptron is probably the most widely used neural network paradigm and has long proven nonlinear modelling capabilities/performance. The knowledge possessed by the network is stored in the weights connecting the artificial neurons. The massively interconnected structure of the MLP provides a large number of these weights and as such a great capacity for storing complex information. The generalised delta rule can be applied for the adjusting of the weights of the feed-forward networks in order to minimise a predetermined cost error function, using gradient descent [20]. The rule for adjusting weights is given by the following equation:

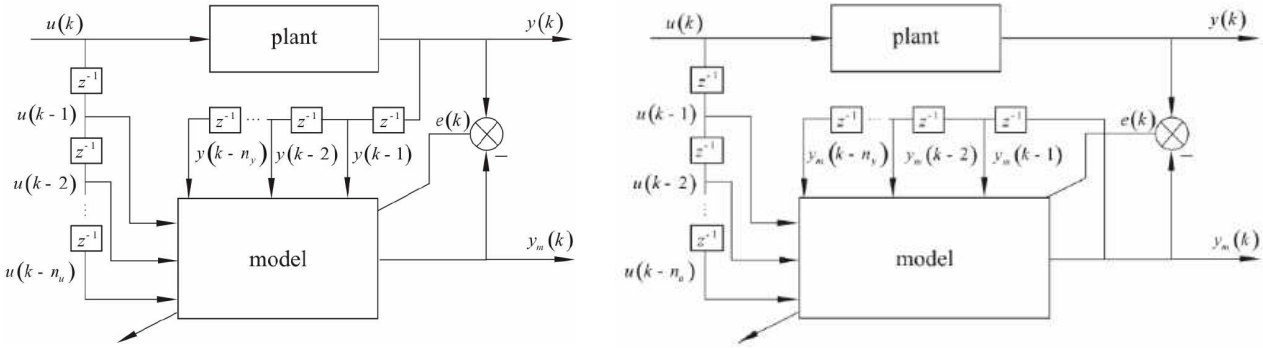
$$w_{ij}^{p}(t+1) = w_{ij}^{p}(t) + \eta\delta_{j}^{p}y_{j}^{p} + \alpha\Delta w_{ij}^{p}(t) \tag{3.6}$$

where η is the learning rate parameter, α the momentum term, and δ is the negative derivative of the total square error with respect to the neurons' outputs.

3.4 Nonlinear Dynamic System Identification

In general, dynamic systems are complex and nonlinear. An important step in nonlinear systems identification is the development of a nonlinear model. In recent years, computational-intelligence techniques, such as neural networks, fuzzy logic and combined hybrid systems algorithms have become very effective tools for the identification of nonlinear plants. The problem of identification consists of choosing an identification model and adjusting the parameters such that the response of the model approximates the response of the real system to the same input.

Since 1986, neural networks have been applied to the identification of nonlinear dynamical systems. Most of this works has been based on multi-layer feed-forward neural networks with backpropagation learning algorithms. A novel multi-layer discrete-time neural network was presented in [47] for the identification of nonlinear dynamical systems. In [12], a new scheme for estimating the on-line states and parameters of the members of a large class of nonlinear systems using RBF (Radial Basis Function) neural network was designed. An identification method for nonlinear models in the form of Fuzzy-Neural Networks was introduced [56]. Within the framework of this research study, we investigate the ability of GP systems to search for, and optimise, tree structures that will be utilised as nonlinear models.



(a) The NARX model (b) The NOE model

Figure 3.2: The NARX and NOE models

Different methods have been developed in the literature for nonlinear system identification. These methods use a parameterised model. The parameters are modified to minimise an output identification error. A wide class of nonlinear dynamic systems with an input, u, and an output, y, can be described by the model:

$$y_m(k) = f_m(\phi(k), \theta) \tag{3.7}$$

where, $y_m(k)$ is the output of the model, $\phi(k)$ is the regression vector and θ is the parameter vector. Depending on the choice of the regressors in $\phi(k)$, different models can be derived [48]:

- **NARX** (Nonlinear AutoRegressive with eXogenous inputs) or series-parallel model:

$$\phi(k) = (u(k-1), u(k-2), \ldots, u(k-n_u), y(k-1), y(k-2), \ldots, y(k-n_y)) \tag{3.8}$$

  where n_u denotes the maximum lag of the input, while n_y denotes the maximum lag of the output.

- **NOE** (Nonlinear Output Error) or parallel model:

$$\phi(k) = (u(k-1), u(k-2), \ldots, u(k-n_u), y_m(k-1), y_m(k-2), \ldots, y_m(k-n_y)) \tag{3.9}$$

The NARX and NOE models (Figures 3.2a and 3.2b) are the most important representations of nonlinear systems.

For identification, we consider the case where the plant is observable and where input-output measurements are available. Dependent on the kind of inputs used, either parallel

or series-parallel models can be utilised.

- Parallel model: the output of the model itself is used to create the time-lagged inputs. This model can be considered as a fully recurrent model. The parallel model is able to give predictions over a short period of time. The model is said to have internal dynamics.

- Series-parallel model: the outputs of the actual system are used as inputs to the model. Only a one-time ahead prediction is possible. The model is said to have external dynamics.

In both cases, the prediction error of the model, in relation to the true plant outputs, is used as a measure to optimise the model parameters. For dynamic systems, the model must have some way to implement time lags. In other words, some memory function must be present in the model. In modelling using computational intelligence schemes, such as neural networks, neuro-fuzzy systems and WNNs, this can be done in two ways: either, delayed inputs and outputs are used as extra external inputs, or some memory is included in the individual neurons.

Models with external dynamics can be seen as one-step-ahead predictors. Models with internal dynamics are best used for simulation purposes, as the model doesn't need the actual plant outputs. The latter case has a higher potential for output errors in the long term. This is certainly the case for nonlinear systems, where the internal nonlinearities can drive the system into a chaotic state. Since for nonlinear problems the complexity usually increases strongly with the input space dimensionality (curse of dimensionality) the application of lower dimensional NARX or NOE models is more widespread. One drawback of these models is that the choice of the dynamic order, n_y, is crucial for the performance and really efficient methods for its determination are not available. Often the user is left with a trial-and-error approach [48]. However the main advantage of the NARX and NOE models, as compared to those models which do not have output feedback (such as Nonlinear Finite Impulse Response - NFIR) is that they lead to a very compact description of the process. As a consequence, the regression vector $\phi(k)$ contains only a few entries and thus the input space for the approximation $f(g)$ has relatively low dimensionality. Such an advantage is even more important when dealing with nonlinear systems as opposed to linear systems.

3.5 Model Validation

The proposed GP, the established neural network approaches and the statistical models were comparatively evaluated in order to determine whether they could successfully predict the responses of the pathogen at pressure levels other than those initially selected for model development. For this reason, two different high pressure levels, within the range employed to develop the models, were selected, namely 400 and 500 MPa. Additional milk pouches were prepared, inoculated with the pathogen and then subjected to pressurisation as previously described. At predetermined time intervals, the surviving population of *L.* monocytogenes was enumerated and compared with the survival curves predicted by both the GP and the statistical models. The accuracies of the predictions were estimated via calculations of bias (B_f) and accuracy (A_f) factors [63], the standard error of prediction (SEP), the mean absolute percentage residual (MAPR) and the root mean square error (RMSE) [55].

3.6 GP implementation

The GP algorithm was applied in two consecutive cycles. The first cycle was based on 200 independent runs, and the best of each run was stored to be used for the second cycle. The initial population for the first cycle of GP optimisation comprised from 100 random generated trees, and the maximum number of generations was 100. In both cycles, the reproduction probability was 0.2 and the crossover probability was 0.80. The relatively high reproduction probability ensured that good individuals were not lost in the course of the process.

The elements of the terminal set which represented the inputs to the GP method consisted only of the pressure level, P, at time step t and the concentrations of the microorganism at time steps t-1 and t-2. The terminal set also included random generated real numbers, which were introduced in the initial population and were fixed for the rest of the run. These random real numbers increase the efficiency of the function search. The function set used can be seen in Table 3.1. For the first round no elitism was imposed at the reproduction stage. In general, elitism is used to ensure that the best individuals will reproduce to the next generation.

In each of the 200 independent runs the best individual of all the generations was stored. After the first cycle was over, the 200 best trees of the first cycle became the initial population of the second cycle. At this stage, elitism was introduced to apply

Function	**Symbol**	**Function**	**Symbol**
Addition	ADD	Square	SQ
Subtraction	DIV	Cube	CUB
Multiplication	MUL	Greater Than	GT
Division	DIV	Sign	SIG
Absolute Value	ABS	Sin	SIN
Square Root	SQRT	Cosine	COS
Exponential	EXP	Logarithm	LOG

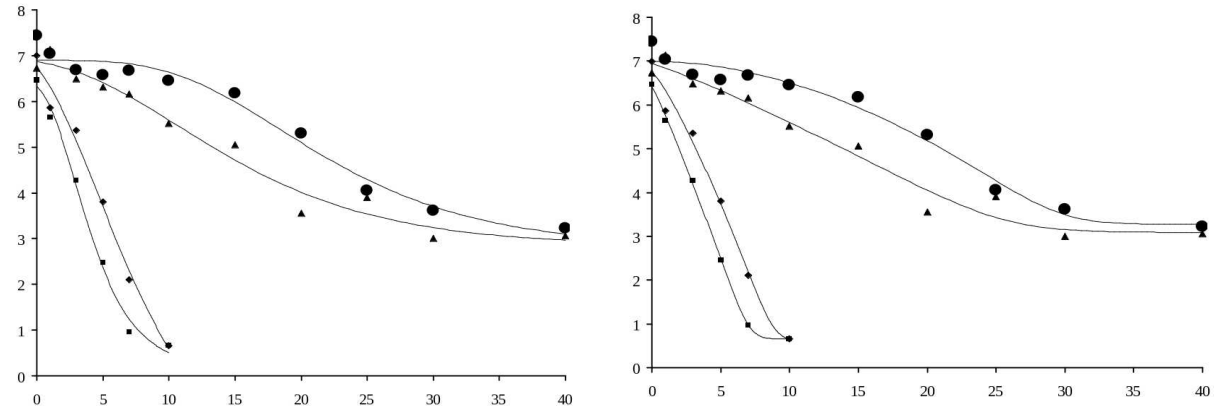
Table 3.1: The function set used by GP

pressure to the learning process. As the population for the second cycle was 200 trees and the reproduction probability was 0.2, the overall number of trees copied to the next generation at each stage was 40. The elitism introduced to the second cycle, ensured that the best 15 individuals of each generation were reproduced to the next generation. The number of generations for the second cycle was 400 and the best individual overall - from all generations - was designated as the outcome of the whole process.

3.7 Discussion of results

The objective of this study was to investigate the feasibility of using a GP scheme for the development of a series-parallel model of the survival curves of Listeria monocytogenes under high pressure in whole milk. The survival curves of Listeria monocytogenes inactivated by high hydrostatic pressure were obtained at six pressure levels (350, 400, 450, 500, 550 and 600 MPa) in UHT whole milk (Figure 3.1). Interestingly, the shapes of the survival curves that follow those experimental data change considerably depending on the treatment pressure levels. However, at all pressure levels assayed, a clear inactivation pattern was observed, including a lag phase (or shoulder), a log-linear and a tailing phase. As expected, the duration of the shoulder was pressure dependent; higher pressures resulted in lower shoulder time. At different pressure levels, survival curves showed a pronounced curvature and tailing indicating that only a very small population of the pathogen could resist such pressurisation and eventually survive in milk.

The estimated kinetic parameters of inactivation based on the re-parameterised Gompertz, modified Weibull are presented in Table 3.2. All models fitted the experimental data well as can be inferred by the high values of the regression coefficients ($R^2 > 0.965$) and



(a) Gompertz model (b) Weibull model

Figure 3.3: Survival curves of Listeria monocytogenes in UHT whole milk during high pressure processing at 350 MPa, 450 MP, 550 MPa, and 600 MPa, generated by the re-parameterised Gompertz model, the modified Weibull model

the low values of the root mean square error ($RMSE < 0.45$). Figures 3.3a, 3.3b illustrate the models performance/fitness on the training data.

Since experimental error cannot account for the nonlinearity frequently encountered in survival curves, many explanations have been proposed, such as variabilities in sensitivities to lethal agents in the bacterial population, mixed bacterial populations in samples wherein each component has a first-order inactivation kinetics, inactivation kinetics of a different order or of a different kind of kinetics, or adaptation to the stress that makes the remaining cells more resistant. However, there are still no satisfactory explanations for the phenomena of shoulders and tails [75].

The prediction capability of these models was considered by adopting a two-step standard procedure commonly applied in predictive microbiology [80]. Initially, the primary

Model type	$log_{10}N_0$	$log_{10}A^b$	$log_{10}N_{res}$	k_{max}	t_s	δ	p	$RMSE$	R^2
Gompertz									
350 MPa	6.90 ± 0.16	4.08 ± 0.59		0.41 ± 0.03	10.07 ± 2.51			0.307	0.969
450 MPa	6.99 ± 0.42	4.15 ± 0.69		0.39 ± 0.04	2.25 ± 0.16			0.313	0.971
550 MPa	7.36 ± 0.32	6.21 ± 0.18		1.68 ± 0.17	-			0.432	0.987
600 MPa	6.5 ± 0.58	6.30 ± 0.96		2.26 ± 0.16	-			0.311	0.993
Weibull									
350 MPa	7.00 ± 0.15		3.27 ± 0.25			14.53 ± 1.88	1.88 ± 0.40	0.266	0.977
450 MPa	6.94 ± 0.24		3.09 ± 0.25			7.71 ± 2.14	1.13 ± 0.28	0.318	0.97
550 MPa	6.74 ± 0.37		0.61 ± 0.48			2.05 ± 0.74	1.24 ± 0.34	0.4136	0.988
600 MPa	6.41 ± 0.10		0.65 ± 0.10			1.46 ± 0.14	1.11 ± 0.07	0.102	0.999

Table 3.2: Parameter estimation and statistical indices of the different models used for fitting the survival of L. monocytogenes in whole UHT milk during high pressure treatment

Model type	Parameter	Equation	Estimated Value	P	R^2
Gompertz	k_{max}	$k_{max} = \alpha_1 P^2 + \alpha_2 P + \alpha_3$	$\alpha_1 = 4.43E^{-5} \pm 0.71E^{-5}$ $\alpha_2 = -0.034 \pm 0.007$ $\alpha_3 = 6.908 \pm 1.674$	0.009 0.005 0.002	0.981
Wiebull	δ	$ln(\delta) = \alpha_1 T + \alpha_2$	$\alpha_1 = -0.009 \pm 0.001$ $\alpha_2 = 6.175 \pm 0.517$	0.006 0.011	0.977

Table 3.3: Parameters and statistics of secondary models for the effect of high pressure on the kinetic parameters of Listeria monocytogenes in UHT whole milk

models (i.e., Gompertz, Weibull) were fitted to high pressure inactivation data and the respective kinetic parameters were calculated (Table 3.2). Subsequently, the derived kinetic parameters were related to high pressure levels through the development of first or second order polynomial models (Table 3.3) and new estimates for their values were determined for 400 and 500 MPa, pressures which had been pre-selected for model validation. As regards the kinetic parameters which did not present a clear trend with pressure, respective values for 400 and 500 MPa were determined by interpolation.

Finally, based on the new values of the kinetic parameters at the (now) selected pressures for validation, Equations 3.8 and 3.9 were refitted and compared with the survival data of the pathogen at the same pressures, in order to determine the potential of the models for generalisation - i.e., their ability to foresee survival curves at pressures for which there had been no previous training.

The performance, against the unknown 400MPa and 500MPa pressure levels, of PLS and MLP has been studied and has been shown to have superior performance as compared to Gompertz, Weibull models in [5, 6]. The proposed GP model creation will be tested against PLS and MLP models in order to establish its validity and test its forecast behaviour.

The determination of kinetic parameters during microbial inactivation, whether through high pressure, chemical or other agents is a rather complex task. Particularly in the case of non-thermal inactivation caused by adverse environmental conditions, the shape of survival curves indicates very pronounced heterogeneity according to the intensity of the stress and may also vary with the physiological state of the cells, the phase of growth (exponential or stationary), and the conditions of adaptation before the stress [78, 79]. Consequently, concave curves may become convex or sigmoidal when the intensity of the stress is changed.

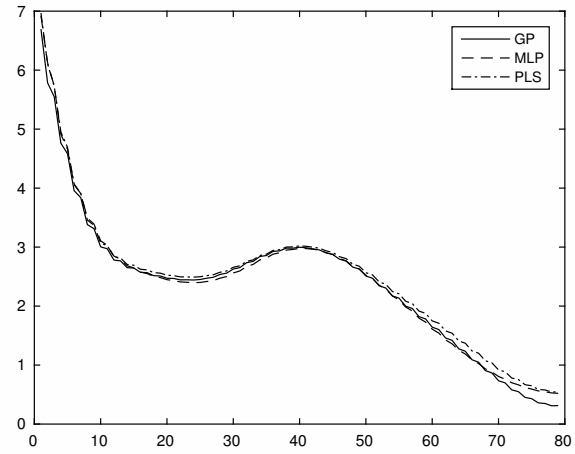
It must be pointed out that despite the plethora of proposed inactivation models in the literature none is flexible enough to account for all changes of shapes with the intensity of

stress. The selected models were able to describe the survival of the pathogen at 350, 450, 550, and 600 MPa quite accurately. However, the prediction at 400 and 500 MPa was not very accurate as the experimental values related to the pathogen showed a pattern which possibly indicated the presence of two sub-populations: one sensitive to high pressure that was inactivated within the first 10 min of the process and a second more resistant to the applied stress [5, 6]. The discrepancy observed in the prediction at these pressure levels could be attributed to the fact that neither model accounted for the presence of a mixed population of the pathogen with a variable resistance to high pressure.

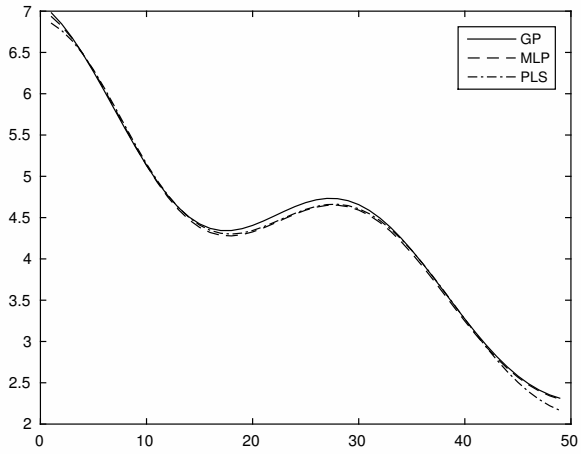
Small data set conditions exist in many fields, such as food analysis, disease diagnosis, fault diagnosis or deficiency detection in mechanics, aviation and navigation, etc. The main reason that small data sets can not provide the sufficiency of information that large ones do is that there exists gaps between the samples; even the domain of samples can not be ensured [39]. It is hard to catch the pattern of high order non-linear functions by a standard feed-forward neural network-like scheme, when only a small sample set is available, since such samples have shown weaknesses in providing the necessary information for forming population patterns. Lacking the whole picture of a function means that the network cannot precisely identify which sections of the function are ascending and which sections are descending. Hence, for learning systems that working on insufficient data, the knowledge learned is often unacceptably rough and/or unreliable.

For PLS and MLP, the inputs included the pressure level and the sampling time-step, while the output was related to the bacteria counts. For GP we excluded the sampling time-step from the terminal set and allowed the method to be free to deduce the behaviour of the survival of the pathogen in relation with time. Each “continuous survival curve” was verified against the real experimental samples. Based on these continuous datasets, the capabilities of the proposed GP model creation has been verified as resulting in a one-step-ahead prediction system. Comparative studies have been conducted in relation to PLS regression, and MLP neural networks. The data generated by pressure levels of 400 MPa and 500 MPa have been used as testing datasets, while the remaining levels were used for training.

Following the principles of nonlinear identification, NARX models have been developed for the GP, the MLP and the PLS systems. The training dataset, consisting of 204 data points from the 350, 450, 550, and 600 MPa ”continuous survival curves”, was employed, while 81 data points from the 400MPa case and 51 data points from the 500MPa curves were kept for validation.



(a) The prediction of GP, MLP, PLS on 400 MI



(b) The prediction of GP, MLP, PLS on 500 MPa

Figure 3.4: Graphs of all the methods on test data

The following structure has been adopted as a NARX model:

$$Count(t) = f(Count(t-1), Count(t-2), MPa) \tag{3.10}$$

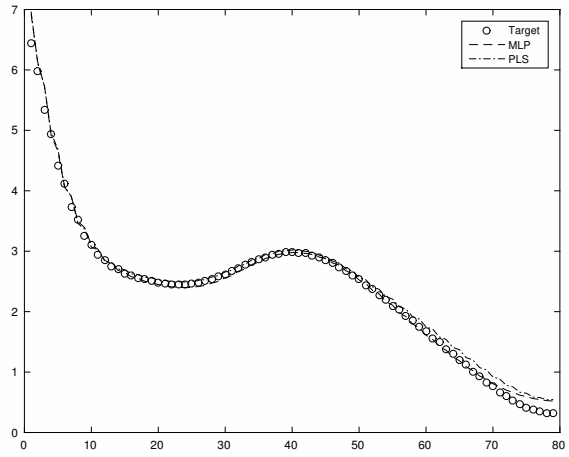
During trials, it has been found that the model is sensitive to the previous number of bacteria counts, thus proving its dynamic behaviour. In the proposed GP these were the terminals used: that is, $Count(t-1)$, $Count(t-2)$ and MPa.

The MLP network structure consisted of two hidden layers (12 and 6 nodes for each hidden layer) and a single sigmoidal output node. The learning algorithm was responsible for the MLPs slow convergence, which took approximately 30,000 epochs. The PLS model was initially constructed using the "continuous survival curve" dataset which is comprised of two inputs and one output. The XLSTAT software package was used to perform the PLS analysis. Two latent variables (LVs) were selected and the resulting equation has the following form

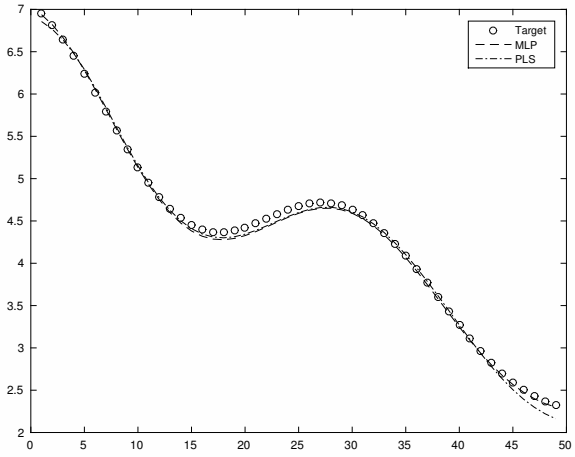
$$Y_1 = 12.8684247 - 0.0150321224X_1 - 0.1096417853X_2 \tag{3.11}$$

In Figures 3.4a and 3.4b, we can see the curves in relation to the test data for GP, MLP, PLS. We can observe that they all follow a main trend. GP differs from the other two approaches mainly at the first and last sections of the data. Also it possesses a distinct behaviour for 500 MPa in the middle part of the curve.

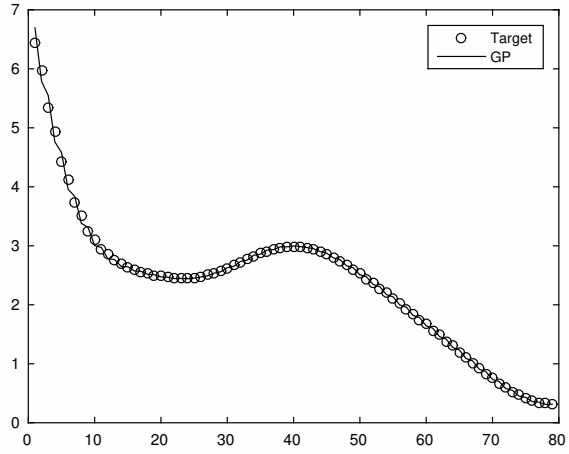
In Figure 3.5, we can observe that the GP generated prediction follows the test data



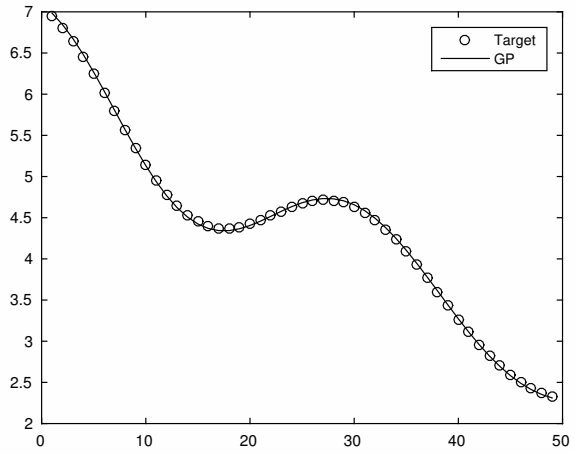
(a) PLS and MLP prediction at 400 MPa



(b) PLS and MLP prediction at 500 MPa



(c) GP prediction at 400 MPa



(d) GP prediction at 500 MPa

Figure 3.5: Comparison of the performance of GP method against the test data

Statistical index	Model	400 MPa	500 MPa
Mean squared error (MSE)	GP	0.0041	3.36 E^{-4}
	MLP	0.011	0.0026
	PLS	0.0177	0.0037
Root mean squared error (RMSE)	GP	0.0642	0.0183
	MLP	0.105	0.051
	PLS	0.133	0.061
Mean relative percentage residual (MRPR %)	GP	0.0195	-0.0181
	MLP	-4.4547	0.8653
	PLS	-9.291	1.0202
Mean absolute percentage residual (MAPR %)	GP	1.8284	0.3616
	MLP	6.2513	1.0004
	PLS	9.4085	1.3143
Bias factor (B_f)	GP	0.9995	1.0002
	MLP	1.0366	0.9913
	PLS	1.0825	0.9896
Accuracy factor (A_f)	GP	1.0185	1.0036
	MLP	1.0556	1.0101
	PLS	1.0838	1.0134
Standard error of prediction (SEP %)	GP	2.6819	0.4155
	MLP	4.3855	1.1557
	PLS	5.5526	1.3835

Table 3.4: Various statistical criteria for the different methodologies at test data

better. The MLP and PLS models deviate from the test data at the end of the curve. Also they don't follow the exact test measurements for the middle part of the curve.

With regard to the assessment of the overall quality of the model predictions, various statistical measures were calculated for all the tested validation experiments undertaken.

The RMSE values the GP system yielded were significantly better for the two "test" survival curves, i.e. 400MPa and 500MPa than were those yielded by the other methods. This index is calculated between the desired and output values and then averaged across all data and it can be used as an estimation of the goodness of fit of models. It can also provide information about how consistent the model would be in the long run [55].

The MAPR term provides information about the average deviation from the observed value. The relevant figures from Table 3.4 again indicate better performance by GP as compared to the other methods. In particular, the high-nonlinear features of the 400 MPa curve proved to be difficult to model from the perspective of the other models. The SEP index is determined as the relative deviation of the mean prediction values and it has the

advantage of being independent on the magnitude of the measurements [55]. Based on this index, the GP scheme achieved again a superior performance in relation to the other methods.

The benefits of having mathematical models to predict pathogen growth, survival and inactivation in foods include: the ability to account for changes in microbial load in food as a result of environment and handling; the use of predictive microbiology in the management of food-borne hazards; and the ability to preparate accurate Hazard Analysis Critical Control Point (HACCP) plans. The usual measures of goodness-of-fit, for model comparison, in food microbiology are the bias (B_f) and accuracy (A_f) indices. Models describing pathogen growth rate with B_f in the range $[0.9 - 1.05]$ can be considered good, those with B_f in the range $[0.7 - 0.9] \,\|[1.06 - 1.15]$ are considered acceptable, while those yielding $< 0.7 \,\|> 1.15$ are considered unacceptable. The bias factor is a multiplicative factor that compares model predictions and is used to determine whether the model over- or under-predicts the response time of bacterial growth. A B_f greater than 1.0 indicates that a growth model is fail-dangerous. Conversely, a B_f less than 1.0 generally indicates that a growth model is fail-safe, i.e. observed generation times were longer than predicted values, so that predicted values give a margin of safety. Perfect agreement between predictions and observations would lead to a B_f of 1. The accuracy factor (A_f), is a simple multiplicative factor which indicates the spread of accuracy of the prediction results. A value of 1 indicates that there is perfect agreement between all the predicted and measured values. Table 3.4 also shows the bias and accuracy factor values obtained for the two testing survival curves. The B_f parameters for GP were superior to those for MLP and PLS. The relevant figures for A_f indicate, again, better performances for the GP scheme, as compared to those of the others.

3.8 Conclusion

In conclusion, the survival curves of L. monocytogenes in UHT whole milk take different shapes, depending on the treatment pressure levels. The development of accurate mathematical models to describe and predict the pressure inactivation kinetics of microorganisms, such L. monocytogenes would be very beneficial to the food industry for the optimisation of process conditions and for the improved dependability of HACCP programs. In this research study, we have developed a model through Genetic Programming, and validated it within a NARX identification scheme for the modelling of the Listeria monocytogenes

survival/death curves. The results, and comparisons with classical statistical methods and established neural network schemes, have revealed very high accuracy accompanied by a relatively speedy training process.

3.9 Next Chapter

GP is a universal problem solver paradigm. Its applicability is not confined in the area of curve fitting. GP has produced successful results in a wide range of disparate problems. Another kind of problems are those that comes under the terms of Control problems. In Control we try to regulate the state of a dynamic system to a desire value. Evolutionary Computations techniques have been employed in a variety of ways. Next chapter presents the variety of algorithms and structures that managed to produced significant results in control problems.

Chapter 4

Application of Evolutionary Computing in Control Problems

4.1 Introduction

Evolutionary Computing (EC) methods have been applied to a diverse set of problems. The methodology is universal and does not rely on problem specific structures. EC is an optimisation procedure, which when supplied with an appropriate fitness measure, is capable of performing a parallel search for solutions on the domain of the problem.

Control problems are ubiquitous. They can be found in any sector of technology, where a quantity or a set of quantities has to be regulated. Domestic appliances, automotive industry, unmanned aerial vehicles and robot manipulation are some examples for which the solution of control problems is vital for their success.

Control theory has been proven very successful in solving linear problems. Fuzzy logic has broaden the range of such applications. However, the solution of complex non-linear, noisy problems and also those with coupling in the control variables is intractable by such methodologies. This gap has been addressed with the introduction of EC methods.

4.2 Genetic Algorithms

Genetic Algorithms have been applied to a range of control problems. Due to their versatility, they are suitable to incorporate to existing control methodologies. When there is no analytic solution, GA can perform parameter tuning of the employed models. The result is a close to optimal controller for the problem at hand.

4.2.1 Control Theory applications

Krishnakumar and Gorlberg [37] successfully applied Genetic Algorithms (GA) and a variant micro-GA to optimise a controller for the lateral autopilot and the wind shear optimisation problem. The application of GA discovered a solution with some gains different from the one calculated from the Linear Quadratic Regulator but the fitness values of the two approaches were close to each other. Hunt [30] demonstrated that for the case of Linear-Quadratic Gaussian (LQG) problem and the H_∞ mixed sensitivity problem the application of GA simplifies the process by eliminating the need for analytic calculations. Kristinsson and Dumont [38] applied GAs in system identification of both continuous and discrete time systems. Their proposed method was effective in both domains and was able to identify directly the poles and zeros of the system. The poles and zeros estimates were subsequently used to design a discrete time pole placement adaptive controller. The method made use of the knowledge about the poles and zeros to obtain a desired transfer function or desired response. In comparison to some widely known identification techniques the method performed as well or even better in terms of number of samples required to converge. Herreros et al. [26] proposed a method for adjusting the parameters of a PID controller based on multiobjective optimisation and GA. They developed a tool for solving the class of multiobjective optimisation problems that results from a PID design problem. The objective functions to be optimised had been chosen to obtain a generic method for designing single-loop PID controllers. The Pareto optimality concept was used, which allowed to find the limits of performance that could be obtained with a PID controller. Subsequently GA were used to solve the resultant multiobjective problem.

4.2.2 Optimal Control applications

The application of GA in Optimal Control has been demonstrated from Hunt [29] where the structure of the controller is given and the free parameters of the controller are optimised within the structure. The aim is to make certain close-loop transfer-functions as close as possible to a target transfer-function over a specified frequency range. The problem specification requires that the closed-loop system is stable. For this reason a stability check is performed before the evaluation of each structure and any unstable system was assigned a fitness which was arbitrarily high. Michalewicz et al. proposed a modification of a GA designed to enhance the performance in discrete-time optimal control problems [45]. The modification involved mutation and crossover operators on real value coding.

The algorithm was applied in the linear-quadratic problem, the harvest problem, and the (discretised) push-cart problem. The results demonstrated that GA produced numerical solutions very close to the analytic ones.

4.2.3 Fuzzy Logic Control applications

Herrera et al. [25] developed a genetic learning process for learning fuzzy control roles from examples. It has been developed in three stages. A genetic generating process was formulated for obtaining desirable fuzzy rules capable of including the complete knowledge from the set of examples. The process was based on an iterative rule learning approach. Subsequently a GA processed the combined rules and simplified them, thereby avoiding the possible overlearning, removing the redundant fuzzy rules. Finally, a genetic tuning process for adjusting the membership functions of the fuzzy rules fine tuned the architecture. Castillo et al. [11] considered the case of controlling the anaesthesia given to a patient, as a problem of finding an optimal fuzzy controlling system. Type-2 fuzzy logic for intelligent control was used making the task of designing the fuzzy system more difficult. They compared the performance of the fuzzy systems that are generated by the genetic algorithms, against the ideal control system given by the experts in this application. After the application of the genetic algorithm the number of fuzzy rules was reduced from 12 to 9 while achieving a similar performance from the simpler fuzzy controller.

Karr and Gentry [13] showed that Fuzzy Logic Controllers (FLC) augmented with GA offer a powerful alternative to conventional process control techniques in the nonlinear, rapidly changing pH systems commonly found in industry. To develop an adaptive FLC using GA, a computer model of the physical system was used. The objective of the control problem was to drive the pH of the solution to the desired set point in the shortest time possible, by adjusting the valves on the two control input streams. Sarimveis and Bafas [64] proposed a Fuzzy Model Predictive Control methodology, which was based on a dynamic non-linear fuzzy model of the plant belonging to the family of indirect fuzzy control methodologies. The method formulates a dynamic non-linear optimisation problem, where the objective function consists of two terms: the differences between the fuzzy model predictions and the desired output trajectory over a prediction horizon, and the control energy over a control horizon. The on-line optimisation problem was solved using a specially developed genetic algorithm, which guaranteed the feasibility of all the generated potential solutions. The method was applied to a chemical reactor with one input variable and one output variable, using a discrete Takagi-Sugeno fuzzy model for predicting the behaviour

of the process. The methodology produced very good results, even when constraints were applied on the control moves.

Chad et al. [59] developed a fuzzy controller capable of manoeuvring a helicopter effectively and aggressively. The controller subdivided the tasks necessary to fly the aircraft. Individual fuzzy logic controllers were introduced to achieve four goals associated with flying the aircraft: (1) the longitudinal velocity goal, (2) the vertical velocity goal, (3) the lateral velocity goal and (4) the heading goal. This division of tasks produced a unique fuzzy logic controller architecture that was subsequently optimised by GA for the correct set of fuzzy rules. The genetic algorithm was applied in two stages to search for the fuzzy rules. In the first stage, the genetic algorithm was employed to search the entire parameter space for a solution that was not immediately divergent. In the second stage, each section of the controller was trained independently.

4.2.4 Predictive Control applications

Martinez et al. [44] used GA to optimise Generalised Predictive Controllers (GPC). A GPC needs a derivable cost index, and if a linear controller is going to be obtained, this index must be quadratic. In the proposed GA GPC, none of these restrictions are necessary, therefore it was possible to use non-derivable and even non-quadratic indexes. Due to the flexibility in the formulation of the cost index, the GA GPC was successfully and easily applied to different problems: nonlinearities in the actuator or modified indexed which were non-quadratic. Onnen et al. [54] used GAs for optimisation in nonlinear model-based predictive control (MBPC). Advanced genetic operators were introduced to increase the efficiency of the genetic search. A coding scheme was developed to implement level and rate constraints on the controlled process inputs. A method of initialising the population was suggested, which introduced a specified number of best solutions from the previous time step to the new population. Termination conditions to abort the evolution were proposed in order to cope with the real-time requirements of MBPC, after a specified level of optimality was achieved. Simulated pressure dynamics of a batch fermenter, a highly nonlinear system, was selected to test the proposed algorithm, which outperformed other optimisation techniques.

4.2.5 Various other applications

Lewis et al. [41] evolved a neural network to generate a sequence of signals that could drive the legs of a 12 degree of freedom hexapod robot. They used GA to search the weights of a NN that control the movement of the legs. The evolution was performed into two distinct phases. In the first phase the learning concentrated on the creation of oscillator circuit. The second phase evolved the connections between the oscillators. Crawford et al. [14] applied genetic search to solve a set of aerospace guidance and control problems. These problems include missile guidance, spacecraft reorientation, and advanced aircraft control and guidance logic. The fitness for missile guidance was assigned to be the maximum of the flight time to target intercept. For the spacecraft attitude control a model was optimised through GA. The fitness was measured by the integral time-weighted absolute value of the error, which was in respect with the desired piece wise linear time trajectory. Pessin et al. [58] proposed a system that relied on two steps: (i) group formation planning and (ii) intelligent techniques to perform robots navigation for fire fighting. For planning, GA were used to evolve positioning strategies for firefighting robots performance. The sensory information of each robot was used as an input to an artificial neural network (ANN). The ANN controlled the vehicle (robot) actuators and allowed navigation with obstacle avoidance. Simulation results showed that the ANN satisfactorily controlled the mobile robots and the GA adequately configured the fire fighting strategy.

4.3 Genetic Programming

Genetic Programming (GP) automatically creates computer programs. Because the functions in the function set are constrained only by the closure property, they can take the form of the usual mathematical functions to specialised abstract instructions emanating from the problem structure. This attribute of GP in conjunction with its searching capabilities, provides a platform suitable for finding complex control laws.

Sweriduk et al. [71] have used GP to design a Pilot-Activated Recovery System (PARS) for a modern fighter aircraft. The objective was to bring the aircraft from any initial attitude to a wings-level, nose-up, recovered state. The fitness measure used was the attitude error and altitude change during the recovery. This was realised with a time-weighted integral of the error squared for the attitude variables. Busch et al. [9] proposed SIGEL, an integrated software package that combines the simulation and visualisation of robots with a GP system for the parallelised evolution of walking. The capabilities of the

system were investigated with six different robot morphologies. They demonstrated that it is possible to get control programs for arbitrary walking robots without having insight into their architecture and kinematic structure. Nordin and Banzhaf [53] developed an evolutionary approach to robotic control of a real robot based on GP. They used GP techniques that manipulated machine code to evolve control programs for robots. The usual mutation and crossover operators were transformed to accommodate the process of machine code instructions. The variant GP was tested on two tasks, obstacle avoidance and object following, with successful results. Nordin and Banzhaf [52] developed a GP system that could be used to control an existing robot in a real-time environment with noisy input. The goal of the GP-system was to evolve real-time obstacle avoiding behaviour from sensorial data. The evolved programs were used in a sense-think-act context. The individuals used six values from the sensors as input and produced two output values that were transmitted to the robot as motor speeds. The fitness had a pain and a pleasure part. The negative contribution to fitness, called pain, was simply the sum of all proximity sensor values. A positive contribution to fitness, called pleasure, was obtained when the robot was going straight and fast.

4.4 Next Chapter

From the review it is apparent that very few times GP has been applied to solve control problems directly. We don't know how is behaving in different circumstances and in the variety of forms that a dynamic systems could have. A very difficult problem from the Reinforcement Learning is the Helicopter Hovering Control problem. The number of action variables is four and this poses an extra challenge for GP. In the following chapter we investigate the application of GP on highly complex and non-linear problem as the one of the helicopter hovering.

Chapter 5

Genetic Programming on the Helicopter Hovering Control Problem

5.1 Introduction

Limitations in classical control theory has led to the use of computational intelligence techniques (neural networks, evolutionary computing and fuzzy logic) for the control of nonlinear, noisy and other dynamic systems which involve complexities.

Genetic programming (GP) aims at the automatic discovery (evolution) of computer programs for a given task given minimal or no information about the task. The computer programs involved are based on a fitness function solely - without any other information (the details of the dynamic system, certain derivatives of the system, etc.) being available. Such a computer program can serve as a control law when the underlying task for which it is evolved is the control of a dynamical system. Following this approach, the automatic generation of controllers by GP methods seems ideal, assuming that such generated controllers can be proven to be successful in practice as well as in theory. In the past, GP has been applied only to a small number of challenging control problems [18], as compared to other computational intelligence techniques which have been applied more frequently [16, 65, 76].

One of the challenging control problems found in the literature is that of helicopter hovering, in which a helicopter attempts to hover as close as possible to a fixed position. The dynamics of the helicopter are nonlinear, high dimensional, complex, asymmetric and noisy [34, 49]. The problem has been included in the past in a number of different control competitions (e.g. [62]) as a benchmark for developing new more powerful control

architectures.

The control of a helicopter in a hovering position must take into account the different forces of the main rotor and the tail rotor. While the main rotor rotates clockwise, the air which is forced downwards generates upwards thrust that keeps the helicopter in the air. Based on this, one could suggest that the balancing of the generated thrust with the force of the weight of the fuselage is sufficient to achieve hovering. However, the clockwise torque of the main rotor has, as a consequence, an anti-torque force, that tends to spin the main chassis. In order to balance the fuselage, the tail rotor has to force air rightwards in order to generate the appropriate moment to counteract the spin [49].

This chapter describes in detail how genetic programming can be applied to the problem of generalised helicopter hovering. The results are compared with those produced by the neuro-evolutionary approach which won first position in the 2008 Reinforcement Learning (RL) competition in relation to the same problem [34, 62]. The problem of generalised hovering includes noise in the form of unknown wind forces. Wind is added to each generalised version (domain) of the problem, according to some unknown probability distribution. The controller evolved has no knowledge about the amount of wind which is present in each domain. Any control algorithm attempting to guess the wind in a single domain would over-fit the model for that particular case and cause it tp perform badly in other domains which include different wind forces.

5.2 The Problem of Helicopter Hovering

The state of the dynamic system represented of a helicopter's flight is described by the following vectors:

$$q = \begin{bmatrix} P & v^p & \Theta & \omega^b \end{bmatrix}^T = \begin{bmatrix} x & y & z & v_x^p & v_y^p & v_z^p & \phi & \theta & \psi & \omega_1^b & \omega_2^b & \omega_3^b \end{bmatrix} \tag{5.1}$$

$$u = \begin{bmatrix} T_m & T_t & a_1 & a_2 \end{bmatrix}^T \tag{5.2}$$

where P is the helicopter position in inertial coordinates, $\Theta = \begin{bmatrix} \phi & \theta & \psi \end{bmatrix}^T$ are the Euler angles corresponding to the roll, pitch and yaw respectively; and v^p is the velocity vector where v_x^p is the forward, v_y^p is the lateral and v_z^p is the vertical velocity. In addition, $\omega^b \in \mathbb{R}^3$ is a vector which contains the angular velocities of the body.

The main rotor produces the forces $f^b \in \mathbb{R}^3$ and torques $\tau^b \in \mathbb{R}^3$ which are controlled

by the main rotor thrust T_m and the longitudinal a_1 and lateral a_2 tilts of the tip path plane of the main rotor with respect to the shaft. The thrust T_t controls the anti-torque of the tail rotor [22].

The equations of motion of a rigid body (based on the body coordinate frame) are described by the *Newton-Euler* equations:

$$\begin{bmatrix} mI & 0 \\ 0 & \mathcal{I} \end{bmatrix} \begin{bmatrix} \dot{v}^b \\ \dot{\omega}^b \end{bmatrix} + \begin{bmatrix} \omega^b \times mv^b \\ \omega^b \times \mathcal{I}\omega^b \end{bmatrix} = \begin{bmatrix} f^b \\ \tau^b \end{bmatrix} \tag{5.3}$$

where $m \in \mathbb{R}$ is the mass, $I \in \mathbb{R}^{3\times3}$ is the identity matrix and $\mathcal{I} \in \mathbb{R}^{3\times3}$ is the inertial matrix [33].

Equations (5.3) can be rewritten using $v^p = R(\Theta)v^b$ and $\dot{\Theta} = \Psi(\Theta)\omega^b$, where $R(\Theta)$ is the rotation matrix of the body axes relative to the inertial axes (superscript p):

$$\begin{bmatrix} \dot{P} \\ \dot{v}^b \\ \dot{\Theta} \\ \dot{\omega}^b \end{bmatrix} = \begin{bmatrix} v^p \\ \frac{1}{m}R(\Theta)f^b \\ \Psi(\Theta)\omega^b \\ \mathcal{I}^{-1}(\tau^b - \omega^b \times \mathcal{I}\omega^b) \end{bmatrix} \tag{5.4}$$

The above system is coupled, non-linear, multivariable and under-actuated - with fewer independent control actuators than degrees of freedom to be controlled [22].

Helicopter hovering belongs to the set of reinforcement learning paradigm problems [70, 81], as it is a sequential decision making problem in which there is limited feedback during learning. Unlike in supervised learning, there are no signals available which indicate the "correct" actions for sample states. The fitness of a controller is measured by the difference between the desired state and the actual one: this difference is then used to construct a reward or penalty (the reinforcement signal) to the controller at every point in time.

The results presented here have been obtained using the XCell Tempest helicopter simulator [3] which was implemented for the RL-Competition [62].

The aim in all the simulations runs was to hover the helicopter around the origin. A single simulation lasts for 600 seconds and it is undertaken in 6000 discrete steps (each step accounts for 0.1 seconds). A penalty (or reward) is applied to the controller at every time step. This reinforcement is based on the deviation of the helicopter from the origin. The penalty (or reward) is the reinforcement signal and it is the only information available

to the controller (agent) for the purposes of learning. The reward is calculated as follows:

$$R = -\sum_i (s_i - t_i)^2 \tag{5.5}$$

where s_i and t_i are the current value and the target value respectively, of the ith state feature.

A crash of the helicopter during a simulation run results in a very large penalty, which is calculated by summing the largest penalty for every remaining step. The values of the four controls of the helicopter are normalised in the range $[-1, 1]$.

5.3 The Winner of the 2008 RL Competition

The 2008 RL Competition included a helicopter hovering task [62]. First place in this competition, for this problem, was won by a neuro-evolutionary reinforcement learning approach [34] which was evolving neural network controllers using the accumulated reward. The evolved neural networks corresponded to different policies mapping observations to actions. The initial population of controllers contained 50 randomly generated networks. A steady state evolutionary optimisation was applied, i.e. generations of populations were not used. In every evolutionary step, the worst performing network was replaced by modifying its weights using crossover and mutation operations [34].

Four different types of networks were used, as based on their architecture and the initial configuration of their weights. Single layer perceptrons (SLP) and multi-layer perceptrons (MLP) were the two architectures used. The MLP topology was designed by a human specialist. Initial knowledge can be applied (in two out of the four different types of networks) to the initial population of neural networks, by setting all the weights of one of the networks, such that it (the network) becomes equivalent to a base-line controller provided by the RL competition software. Although this controller avoids crashes, it is naive and can not solve the problem satisfactorily; it performs poorly and does not approach the hovering point. The rest of the initial population (in the case of the base controller being present) is created by mutating the weights of the base-line controller with a predefined probability. The mutation operator consists of the random generation of a weight, from a Gaussian distribution with a mean of zero and a standard deviation of 0.8, which either replaces or adds itself to the initial weight.

During the evolution process, two parent networks are selected via roulette wheel selec-

tion when crossover occurs. The weights of the new networks are probabilistically assigned to be, either the average of the weights of the two parents networks, or the weights of one of the parents. After this application of crossover, the networks undergo weight mutation with a low probability, by following the same procedure as that described above during the generation of the initial population. If crossover is not chosen, then a network is selected using the roulette wheel method and its weights are subsequently mutated. The mutation creates a new neural network which will replace the worst performing network in the population [34].

5.4 The Genetic Programming Approach

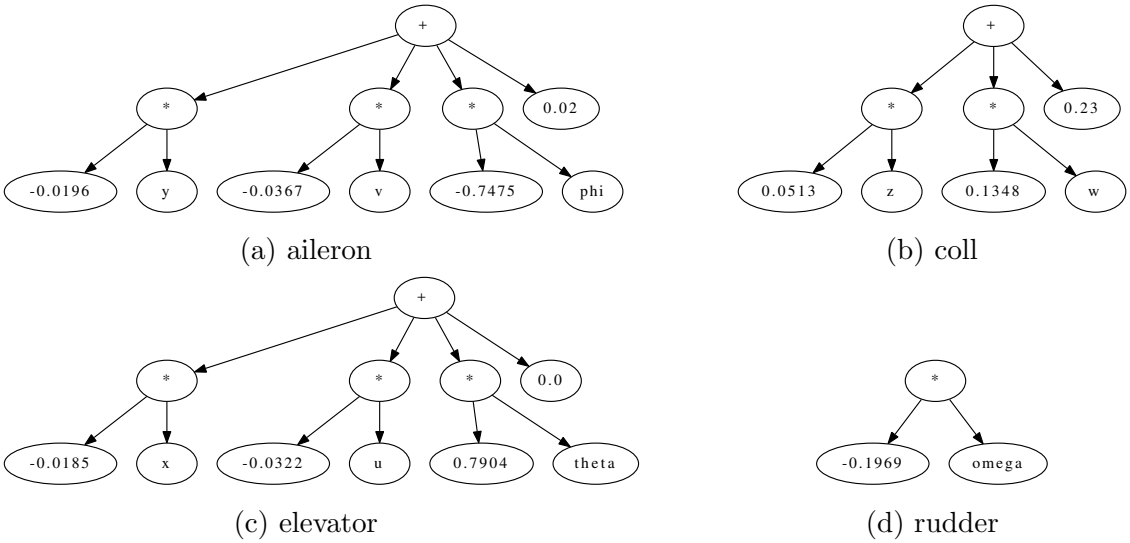
The action space for the helicopter hovering control is continuous and includes the following four actions:

α_1 : longitudinal cyclic pitch (aileron)
α_2 : latitudinal cyclic pitch (elevator)
α_3 : main rotor collective pitch (rudder)
α_4 : tail rotor collective pitch (coll)

Unlike in the usual GP approach, this application of GP for helicopter hovering uses four independent trees allocated to each individual in the population, one for each action. Thus, the population consists of 500 quad trees.

The individuals in the initial population are generated with the 'ramped half-and-half' method [35]. The 'full' method which, here, creates full trees of length 2 to 6, or the 'grow' method which sets the maximum length of a tree are chosen with equal probability. Creation of the same individual (duplication) was not allowed in the initial population. The initial population included an individual which was equivalent to the default baseline controller supplied with the RL Competition software as described in the previous section. The controller can avoid helicopter crashes but it is unable to bring the helicopter close to the hovering position. It has to be emphasised that the same baseline controller was also included in the initial population of the neuro-evolutionary approach which won the RL Competition. The baseline controller is shown in Figure 5.1.

The GP derivation of controllers for helicopter hovering used the normalised fitness to drive the evolution. First, the reinforcement signal $R(x_n) = -\sum_{i=1}^{n} x_i^2$ is utilised to calculate the raw fitness. To do so, the sum of the reinforcement signals at every time step



(a) aileron (b) coll

(c) elevator (d) rudder

Figure 5.1: The default baseline controller in the initial GP population.

Function	**Symbol**	**Function**	**Symbol**
Addition	ADD	Square	SQ
Subtraction	DIV	Cube	CUB
Multiplication	MUL	Greater Than	GT
Division	DIV	Sign	SIG
Absolute Value	ABS	Sin	SIN
Square Root	SQRT	Cosine	COS
Exponential	EXP		

Table 5.1: The function set used by the GP scheme for the helicopter hovering problem.

in an episode is calculated. The raw fitness is equal to this sum. The reinforcement signal is negative, therefore the standard fitness $s(i)$ of the i-th individual is the negation of the raw fitness. Then the adjusted fitness is calculated by the following:

$$a(i) = \frac{1}{1 + s(i)}$$

Finally, the normalised fitness was calculated:

$$n(i) = \frac{a(i)}{\sum_{k=1}^{M} a(k)}$$

The function set chosen for the application of the GP scheme is shown in Table 5.1.

The state of the dynamic system is described by 12 variables. However, only 9 of them are independent because the angular velocities can be calculated from the rest. Therefore, the terminal set used included only the independent variables, i.e. the coordinates x, y, z, the velocities u, v, w, the angles θ, ϕ, ω and the constants that were essential to construct the default baseline controller.

The generalised version of the helicopter hovering problem, which includes noise in the flight dynamics, was considered by the GP process. The details of the actual noise, i.e. its magnitude and range were unknown to the controller. A derived controller for this application should not over-fit to a specific control situation (i.e. to a specific noise regime); it should be able to hover the helicopter even when the environmental conditions (noise representing wind) change. This is what the generalised version of the problem requires (and was the case for the RL competition and its software simulator used in this work). The wind (noise) has x and y components. The wind velocities vary randomly between $-5m/sec$ and $5m/sec$ (neither the value nor the range is known to the evolving GP controllers). The helicopter hovering simulator provides 10 different wind patterns corresponding to 10 different modes of the simulation: i.e. it includes 10 different sequential decision problems (SDPs). GP has been applied to all 10 of them. For every different mode, 10 different runs were executed. The objective was to derive a controller which generalises well across different SDPs, including unseen cases which are not included in the simulator.

Each GP run evolved controllers for 119 generations (excluding the first initial population). The probability of selecting for reproduction was 10% and the crossover selection was probability 90%. The maximum depth permitted after the crossover operation was 17, and in the case that one of the off-springs had reached a depth which was longer than that which was allowed, the selected parent tree was copied without modification. In the case that both off-springs exceeded the allowed depth, both parents were copied to the new population - a process is equivalent to literal reproduction. The selection of an individual for reproduction was proportional to fitness. Permutation and mutation were not allowed. It should be noted that the mutation operator was initially tested, but it did not contribute to the production of successful results.

The default baseline controller was copied to the next population independently of the reproduction process, as this was proved to generate better results overall.

The output of an evolved GP controller is calculated based on the following:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5.6}$$

where x is the evaluation of the GP tree. The above limits the range of the control inputs to the plant in $[-1, 1]$, a requirement of the simulator.

5.5 Results

To address the generalised version of the helicopter hovering problem, a controller must be able to cope with different environmental conditions (i.e. unknown wind conditions) related to the problem. The RL Competition simulator includes only 10 different SDPs for both training and testing. In order to optimise the process of finding the best controller - which will then be capable of being applied successfully to other SDPs as well (besides the 10 available) - the 10 SDPs had to be used for training, validation and testing [16].

The results presented here for both the GP and the neuro-evolutionary approaches were produced via the same optimisation scheme. 10 different populations (runs) are created for every mode of wind pattern (10 modes in total). Every run consisted of 120 generations for the GP method and 120000 episodes for the neuro-evolutionary method.

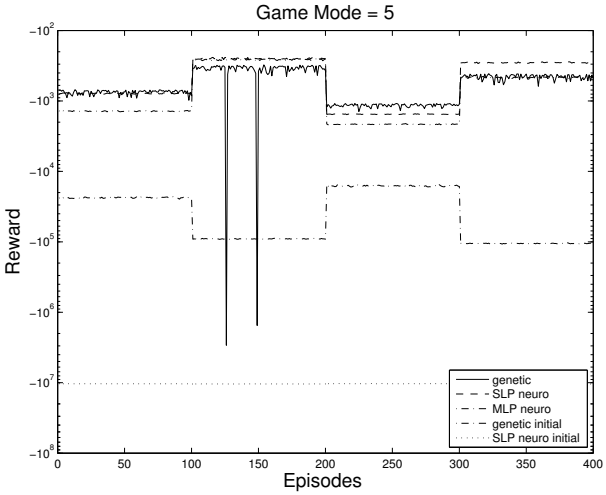
The optimum controller was derived by the following process:

1. The best controller from each generation was subsequently validated in relation to modes 0 to 4. If the controller had, in fact, been evolved on any of the modes 0 to 4, this mode was excluded from the validation. According to its performance in the validation set, the best individual for every mode was selected. Thus, the validation runs determined which individual performed better (generalised to) unseen SDPs (i.e. SDPs not used for its evolution) .

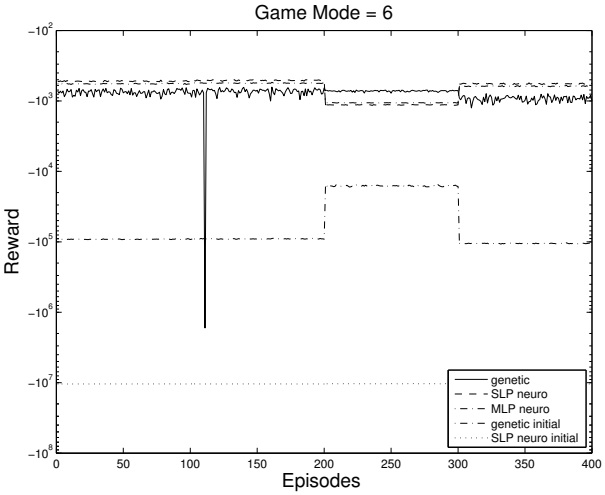
2. The 10 best controllers (one from each mode) from both approaches (GP and neuro-evolutionary) were tested on modes 5 to 9. The comparison between the GP method (implemented here) and the neuro-evolutionary method which won the 2008 RL competition was based on this. If a controller had been evolved (trained) in one of the modes 5 to 9, then this mode was excluded from the test runs. To obtain the test results, 100 episodes were run with each controller for each test mode.

   The best overall controller found by GP (based on the test modes) is shown in Appendix A.1.

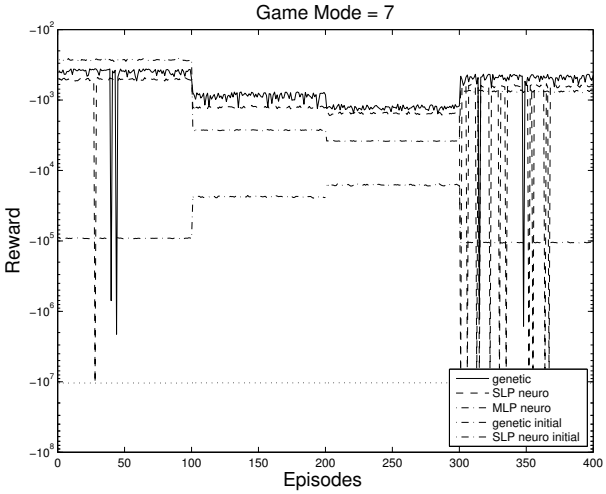
Figures 5.2a–5.3 show the performance of the two approaches in relation to the 5 test modes (SDPs). It can be seen that in most cases the performance of the GP generated



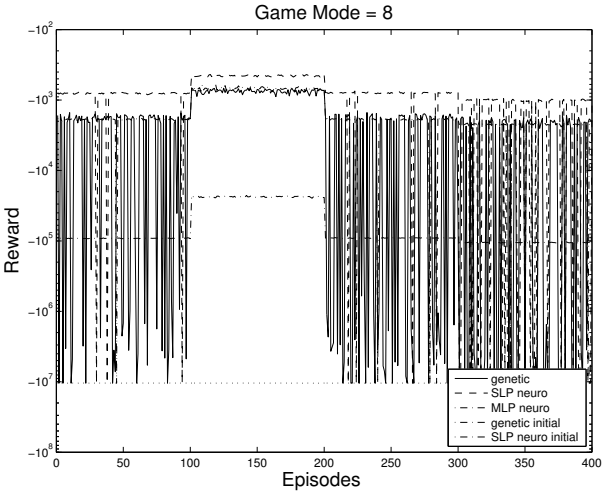
(a) Best Agents from mode 5.



(b) Best Agents from mode 6.



(c) Best Agents from mode 7.



(d) Best Agents from mode 8.

controller is similar to that of the SLP and MLP controllers evolved using the neuro-evolutionary approach. For some modes, however, the GP generated solution performs better.

The diagrams also include the performances of the controllers in the initial populations of the two approaches. The GP method manages to improve on the initial (baseline) controller by two orders of magnitude.

The performance of the best GP controller and the best neuro-evolutionary controller among all modes is illustrated in Figures 5.4a, 5.4b. From Figure 5.4a, it can be seen that the GP controller stands between the two different versions of the neuro-evolutionary approach - with a few exceptions when it crashes the helicopter. The interrupted lines are

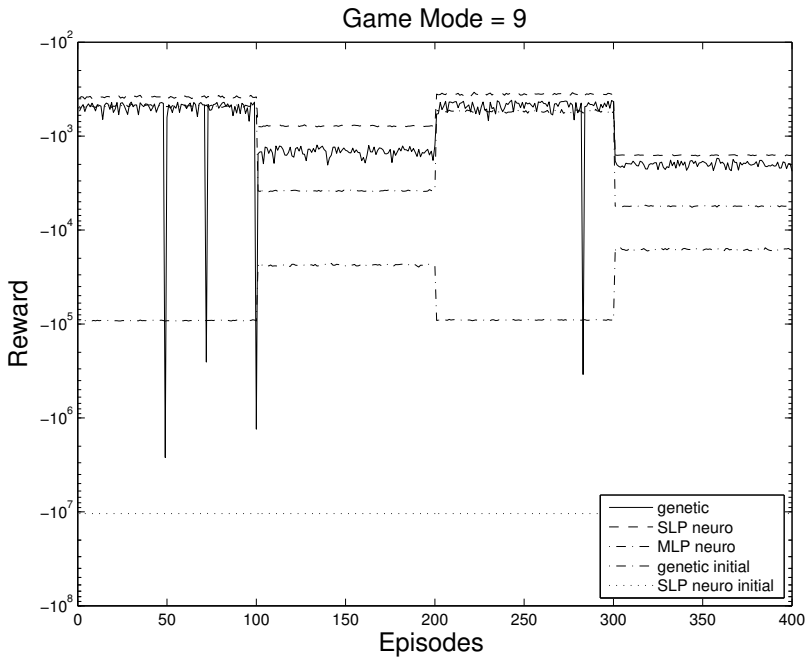
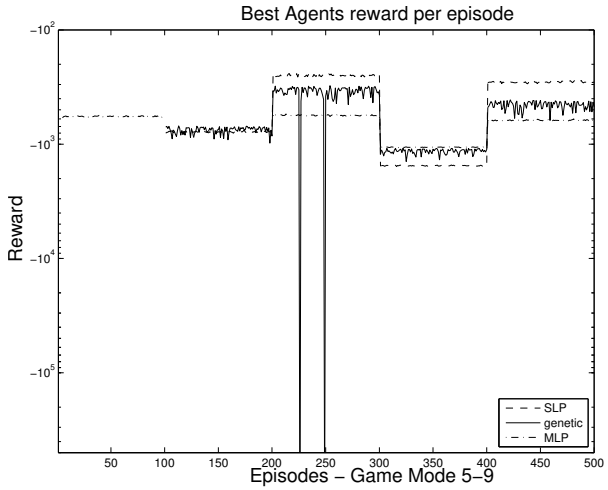


Figure 5.3: Best Agents from mode 9.

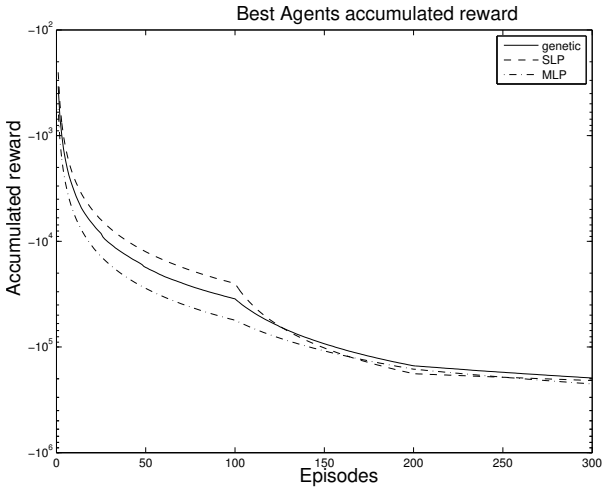
due to the fact that some agents did not participate in all test modes (the training mode was excluded from the test cases, i.e. if a controller was trained for mode 7, then the tests cases included only modes 5, 6, 8 and 9).

Figure 5.4b compares the accumulated rewards of the different controllers for modes 7, 8, 9 (modes 5 and 6 were used by some controllers for training, so these modes are excluded). Accumulated reward was the performance measure used in the 2008 RL competition to determine the winner of the competition. This diagram excludes the two crashes experienced by the GP based controller and it can be seen that the GP controller's performance is very similar to that of the controllers resulting from the neuro-evolutionary approaches (SLP and MLP).

Tables 5.2 and 5.3 show the comparison statistics between the GP and SLP, and GP the MLP respectively, for each of the 10 controllers (one for each mode), evolved by each approach, and then validation. `Crh` indicates the number of crashes, `Bet` is the number of episodes for which the approach performed better than the other and `Acc` is the accumulated reward over the test episode cases. For example, the first row of Table 5.2 compares the best controllers of GP and SLP which were derived based on training in the mode 0 SDP (the best controller for a mode was selected after running the best of each generation in each validation mode). To check the generalisation ability of the controllers, the test



(a) The performance of the best agents from the various approaches.



(b) The performance of the best agents from the various approaches.

modes (i.e. modes 5 to 9) are used. 100 episodes were run per mode for each controller. In the cases of a crash, that episode was not included in the calculation of the accumulated reward.

5.6 Conclusions

The capability of GP to automatically generate controllers for challenging, complex control problems has not been tested sufficiently in the literature. Here, GP is applied to the helicopter hovering problem.

Helicopter hovering is a nonlinear, high-dimensional control problem which has been included in the literature as a member of the set of known challenging control domains. The generalised version of helicopter hovering is considered, here, which includes unknown noise (in the form of wind), as an additional complexity. The problem was included in the set of benchmarks in recent RL competitions for deriving new controllers.

The details of how GP can derive controllers for this problem are described. The proposed approach is compared with the neuro-evolutionary approach which won first place for helicopter hovering in the 2008 RL competition. The performance of the evolved controllers is tested in variants of the dynamic system that they have not encountered during evolution (training). The results presented demonstrate that GP performs similarly to the winner of the RL competition. In some cases, the GP evolved controllers experienced a crash, but this is also happened with the winner of the competition. In certain modes (SDPs), the

		Mode 5			Mode 6			Mode 7			Mode 8			Mode 9		
		Crh	Bet	Acc	Crh	Bet	Acc	Crh	Bet	Acc	Crh	Bet	Acc	Crh	Bet	Acc
Agent	GP	1	0	-87792	13	0	-83736	0	0	-83681	28	72	-88301	4	0	-123590
Mode 0	SLP	0	100	-56242	0	100	-40383	0	100	-56332	0	28	-99329	0	100	-54965
Agent	GP	3	34	-50547	0	3	-83438	4	2	-49350	0	0	-133935	2	0	-66411
Mode 1	SLP	0	66	-47000	0	97	-74001	0	98	-41285	0	100	-60516	0	100	-46266
Agent	GP	7	35	-47512	51	0	-73633	5	11	-45030	63	29	-92901	23	0	-61243
Mode 2	SLP	0	65	-45238	0	100	-36679	0	89	-40606	0	71	-98758	0	100	-48464
Agent	GP	0	100	-51732	0	100	-65718	3	97	-48658	0	1	-143389	0	100	-67374
Mode 3	SLP	0	0	-96287	1	0	-109319	0	3	-90617	0	99	-130921	30	0	-154985
Agent	GP	1	0	-87257	2	0	-176645	1	2	-82308	0	0	-226003	3	0	-134277
Mode 4	SLP	0	100	-74344	0	100	-130160	0	98	-69355	0	100	-171885	0	100	-86932
Agent	GP	-	-	-	0	79	-75581	2	0	-34468	0	100	-115604	0	0	-45969
Mode 5	SLP	-	-	-	0	21	-78253	0	100	-25051	0	0	-154021	0	100	-28622
Agent	GP	0	0	-75617	-	-	-	1	0	-73645	0	100	-72229	0	0	-93995
Mode 6	SLP	0	100	-52513	-	-	-	0	100	-51048	0	0	-113669	0	100	-57072
Agent	GP	2	97	-39566	0	98	-88398	-	-	-	0	99	-128574	2	97	-48604
Mode 7	SLP	1	3	-50939	0	2	-127104	-	-	-	0	1	-155043	11	3	-63450
Agent	GP	31	3	-178694	0	0	-75126	45	4	-174114	-	-	-	42	25	-203970
Mode 8	SLP	3	97	-80322	0	100	-45421	4	96	-78954	-	-	-	27	75	-100123
Agent	GP	3	0	-49156	0	0	-147568	1	0	-47917	0	0	-200945	-	-	-
Mode 9	SLP	0	100	-38384	0	100	-77955	0	100	-35751	0	100	-159757	-	-	-

Table 5.2: Performance comparison for each mode between GP and SLP.

GP derived performance is better than that derived from the neuro-evolutionary approach.

Based on the results obtained for this problem, it can be stated that the automatic generation of computer programs by genetic programming could also provide an ideal candidate in the automatic generation of controllers for complex control problems. This is because GP methods, per-se, do not require any knowledge about the problem domain; the evolution is driven by an opaque (domain specific) fitness function. Problems which belong to the reinforcement learning paradigm, where a limited (and possibly delayed) single reward or penalty signal is the only input available for learning, seem suitable to the application of GP.

5.7 Next Chapter

The successful application of GP on the Helicopter Hovering Control problem has demonstrated that the GP is a powerful methodology but the agents produced are inflexible and static. Every tree structure is immutable after been created. In order to enhance the abilities of the method, we have to include parameterised functions that somehow adapt to the requirements of the problem. This second optimisation needs the incorporation of a critic module that will be able to make prediction in a step by step basis, in contrast with per episode optimisation of GP. In the subsequent chapter we introduce the theory of Reinforcement Learning and Neural Networks. These seemingly disparate theories along

		Mode 5			**Mode 6**			**Mode 7**			**Mode 8**			**Mode 9**		
		Crh	Bet	Acc	Crh	Bet	Acc	Crh	Bet	Acc	Crh	Bet	Acc	Crh	Bet	Acc
Agent	GP	1	99	-87792	13	0	-83736	0	100	-83681	28	72	-88301	4	96	-123590
Mode 0	MLP	0	1	-173088	0	100	-63677	1	0	-172145	0	28	-108277	2	4	-195310
Agent	GP	3	0	-50547	0	100	-83438	4	0	-49350	0	100	-133935	2	0	-66411
Mode 1	MLP	0	100	-33185	0	0	-259734	0	100	-33027	0	0	-397802	0	100	-46010
Agent	GP	7	93	-47512	51	31	-73633	5	95	-45030	63	37	-92901	23	77	-61243
Mode 2	MLP	0	7	-104444	0	69	-74733	0	5	-106599	0	63	-161663	0	23	-114475
Agent	GP	0	100	-51732	0	100	-65718	3	97	-48658	0	100	-143389	0	100	-67374
Mode 3	MLP	0	0	-110556	0	0	-106775	0	3	-106793	0	0	-187406	0	0	-150084
Agent	GP	1	97	-87257	2	98	-176645	1	92	-82308	0	100	-226003	3	97	-134277
Mode 4	MLP	0	3	-113862	0	2	-429170	0	8	-98078	0	0	-537801	0	3	-252467
Agent	GP	-	-	-	0	100	-75581	2	0	-34468	0	100	-115604	0	60	-45969
Mode 5	MLP	-	-	-	0	0	-139729	0	100	-25958	0	0	-214368	0	40	-46168
Agent	GP	0	0	-75617	-	-	-	1	0	-73645	0	100	-72229	0	0	-93995
Mode 6	MLP	0	100	-57107	-	-	0	0	100	-55847	0	0	-106240	0	100	-61786
Agent	GP	2	0	-39566	0	100	-88398	-	-	-	0	100	-128574	2	98	-48604
Mode 7	MLP	0	100	-26953	0	0	-266833	-	-	-	0	0	-382199	0	2	-74924
Agent	GP	31	52	-178694	0	10	-75126	45	45	-174114	-	-	-	42	48	-203970
Mode 8	MLP	1	48	-188102	0	90	-68571	2	55	-188486	-	-	-	6	52	-221276
Agent	GP	3	54	-49156	0	100	-147568	1	82	-47917	100	100	-200945	-	-	-
Mode 9	MLP	0	46	-47782	0	0	-382501	0	18	-54595	0	0	-556661	-	-	-

Table 5.3: Performance comparison between GP and MLP.

with GP can actually integrate in a hybrid system that potentially could solve difficult Reinforcement Learning problems.

Chapter 6

Reinforcement Learning and Neural Networks Theory

6.1 Introduction

In relation to the subject of Machine Learning (ML), two major directions have been studied, with promising results, from the birth of the subject in the 60's': that is the supervised and the unsupervised learning directions. From a set of data, where there is a relationship between some independent variables and one or many dependent variables, supervised learning tries to find the law governing the relationship by providing to the learning agent the correct answer for each instance of the inputs (independent variables). The agent learns from the difference between its output, s and to the correct one t, that is provided. This direction can be named "learning by a teacher". The type of the response determines whether the learning is called regression (for continuous variables) or classification (in the discrete case).

On the other side of the spectrum is unsupervised learning where the agent is oblivious of the correct output and tries to find patterns and relationships only from the data input. In between these two methodologies lies a third option. Instead of providing the correct output, the system at hand provides a signal which indicates the quality of the agent's performance. This signal is a single real number which serves as a way to characterise whether the agent has behaved "well" or not. This methodology is called Reinforcement Learning (RL).

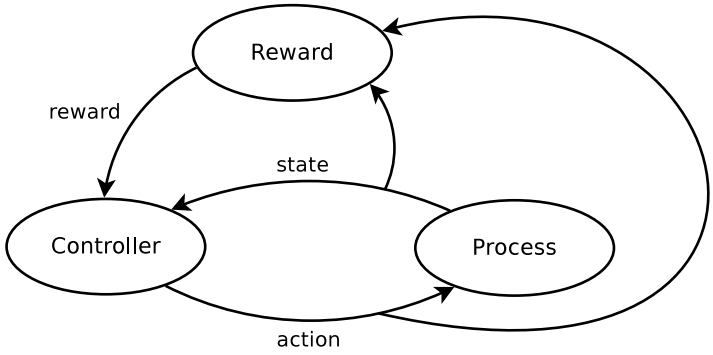


Figure 6.1: The Reinforcement Learning problem structure

6.2 Reinforcement Learning

The main constituent parts of a reinforcement learning problem is the environment (process), which is unknown, the agent (controller), which is under our influence and a special signal - the reward. The environment is described via the state signal, S, the agent's influence on the environment is denoted by U and the reward by R. The overall evolution of the system takes place in discrete time steps. The agent receives the state of the process x_i at time step i and decides to apply the action u_i. The reward function subsequently generates a signal r_i which represents feedback on the agent's immediate performance. The action applied to the process forces the environment to change to state x_{i+1}. From that state the sequence of events starts all over again. The problem is for the agent to control the environment and guide it to an ultimate goal. The goals may be as diverse as reaching an exit in a maze or stabilising a dynamic system to an equilibrium. The general properties of the RL paradigm can be seen in Figure 6.1.

The controller translates states into action. The form of this relationship, which is described by a function, dictates a policy. The process, that is the unknown environment, changes state according to its initial state and the action imposed by the controller. The dynamics can be deterministic or stochastic. In the deterministic case, the same state and action will always yield the same next stage but in the stochastic case the next state is a random variable. The rule that governs the evolution of states, has the form of a reward function $r(x_i, u_i)$. The goal of maximising its reward causes the controller to search through the policies available and converge to the optimal one; this will maximise the expected return. The type of RL problem, which has been more comprehensively studied in the literature is the infinite-horizon return. This possesses some elegant mathematical properties which guarantees a convergence to optimal policies in relation to a number of

important algorithms.

6.2.1 Markov Decision Processes

A deterministic Markov Decision Process (MDP) is defined by a state space X, an action space U, and a transition function f of a process that describes the process' dynamics, and a reward function, ρ. At a specific time step, k, the action u_k is applied to the system, which is at state x_k. As a result of the action the state changes to x_{k+1} according to the transition function $f : X \times U \to X$:

$$x_{k+1} = f(x_k, u_k) \tag{6.1}$$

The transition triggers the scalar reward signal r_{k+1}, in accordance with the reward function $\rho : X \times U \to \mathbb{R}$:

$$r_{k+1} = \rho(x_k, u_k) \tag{6.2}$$

where we require that $\|\rho\|_\infty = \sup_{x,u} |\rho(x, u)|$ is finite. The reward is a measure of the immediate effect of the application of action u_k, for the one step transition from state x_k to state x_{k+1}. The decision, from the controller's side, is governed by its policy $\pi : X \to U$:

$$u_k = \pi(x_k) \tag{6.3}$$

These two functions, the transition function, f, and the reward function, ρ, constitute the *model* of the RL problem. The f and ρ, with a knowledge of the current state x_k and the action u_k, are sufficient to determine both the next state, x_{k+1} and a reward r_{k+1}. A system that exhibits this kind of behaviour is said to have the Markov property. The Markov property states that the next state of the process is fully determined by the current state and the action applied, irrespective of the chain of previous states and actions [10].

In the stochastic case, and when the number of states is countable - as in the discrete form of MDP wherein the states are discrete - the transition function provides the probability of changing to a state x_{k+1} from a specific state x_k and the taking of a chosen action, u_k, by the controller. The transition function, $\bar{f} : X \times U \times X \to [0, 1]$, is a rule that represents the probability of reaching the next state:

$$P(x_{k+1} = x' | x_k, u_k) = \bar{f}(x_k, u_k, x') \tag{6.4}$$

As the function $\bar{f}$ is a probability measure, it accumulates to 1, when summed for all the possible next states, $\sum_{x'} \bar{f}(x, u, x') = 1$.

Reward measures

The reward function provides a measure concerning the immediate effect of an action. Optimality in relation to a MDP postulates that a controller is optimum when the accumulated rewards are maximised from any initial state x_0. There are a couple of different options in terms of calculating the accumulated rewards.

The *infinite-horizon discounted return* is given by:

$$R^{\pi} = \sum_{k=0}^{\infty} \gamma^k r_{k+1} = \sum_{k=0}^{\infty} \gamma^k \rho(x_k, \pi(x_k)) \tag{6.5}$$

where $\gamma \in [0, 1)$ is the discount factor. The discount factor guarantees that for bounded rewards, the sum will also be bounded. This property is essential, from a mathematical point of view, to ensure convergence to the optimal policy. Intuitively the discount factor γ, according to how close it is to unity, designates how far in time we account searching for optimality. When the factor is close to 0 the immediate rewards guide the agent to short-sighted optimal solutions. For the undiscounted case, $\gamma = 1$, the sum of the rewards is calculated. This case is of limited importance as the final outcome in infinite-horizon problems is usually unbounded.

An alternative option is the *infinite-horizon average return*:

$$\lim_{K \to \infty} \frac{1}{K} \sum_{K=0}^{K} \rho(x_k, \pi(x_k)) \tag{6.6}$$

One problem with this criterion is that we cannot distinguish between two policies where one gains a large quantity of reward in the initial phases and another one that doesn't. If we are interest only in a fixed and finite distance to horizon, K, the following formula can be used:

$$\sum_{K=0}^{K} \gamma^k \rho(x_k, \pi(x_k)) \tag{6.7}$$

In this case the undiscounted return can be used ($\gamma = 1$) as the sum is bounded when the rewards are bounded [10][32].

6.2.2 Value Functions

The policies which can be applied to RL problem also have two forms. The one already mentioned is that of *deterministic stationary policies* where there is a mapping π from states to actions. Following π requires that at every time step, $k \geq 0$, the action u_k is selected according to the rule $u_k = \pi(x_k)$. A policy is said to be stationary when the policy does not change through out the course of the episodes. In contrast, *stochastic stationary policies* map states to probabilities distributions over the action space. At every state, x_k, an action, u_k, is selected from a set of possible actions, $U(x_k)$, according to a probability, $u_k \sim \pi(*|x_k)$ [73].

Optimality depends on the reward function and there are two special functions in the RL methodology that summarise a policy h. These are the *state value function* (V-function) and the *state-action value function* (Q-function). The Q-function, $Q_\pi : X \times U \to \mathbb{R}$, of a policy, π, is the return when starting from a given state taking an action and subsequently following the policy π:

$$Q^\pi(x, u) = \sum_{k=0}^{\infty} \gamma^k \rho(x_k, \pi(x_k)) \tag{6.8}$$

where x_0 and u_0 are the initial states and actions, $x_{k+1} = f(x_k, u_k)$ and the policy followed is $u_k = \pi(x_k)$. A better formula for the Q-function which shows its importance is derived as follows:

$$\begin{aligned} Q^\pi(x, u) &= \rho(x, u) + \sum_{k=1}^{\infty} \gamma^k \rho(x_k, \pi(x_k)) \\ &= \rho(x, u) + \gamma \sum_{k=1}^{\infty} \gamma^{k-1} \rho(x_k, \pi(x_k)) \\ &= \rho(x, u) + \gamma R^\pi(f(x, u)) \end{aligned} \tag{6.9}$$

From this formula it is apparent that the Q-functions calculate the immediate reward plus the expected return from the next time step.

The V-function is defined similarly but with the difference that it creates a relationship of states to real numbers. It designates the importance and the merit of being at a particular state under a specific policy:

$$V^\pi = R^\pi = Q^\pi(x, \pi(x)) \tag{6.10}$$

For the stochastic case, where we are taking the expected value of an infinite sequence

of states, the two functions take the following form:

$$Q^{\pi}(x,u) = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^{k}R_{k+1}|x,u\right] \quad V^{\pi}(x) = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^{k}R_{k+1}|x\right] \tag{6.11}$$

6.2.3 Bellman Equation - Optimallity

The Bellman equation for a Q^{π} states that the action taken under a policy, π, when in a state, x, equals the sum of the immediate reward and the next step's discounted value as it will be achieved by the same policy, π, at the next state:

$$Q^{\pi}(x,u) = \rho(x,u) + \gamma Q^{\pi}(f(x,u),\pi(f(x,u)) \tag{6.12}$$

The equations for every possible state, in the discrete case, are actually creating a system of $|X|$ equations and can be solved analytically.

The optimal Q-function maximises the return amongst all the possible functions of all the possible policies:

$$Q^{*}(x,u) = \max_{\pi} Q^{\pi}(x,u) \tag{6.13}$$

Any policy, π^{*}, that employs this rule, that at each state we select the action with the largest Q-value, is considered optimal. The actions selected at all steps are obtained by the formula:

$$\pi^{*}(x) \in \arg\max_{u} Q^{*}(x,u) \tag{6.14}$$

The specific policy constitutes a "greedy policy". In RL, the greedy policies exploit the full knowledge of the value function, accumulating as much reward as they can. The drawback is that during the convergence process, the knowledge that we have is inconclusive and partial. This is a core question in RL, to balance the exploratory power of an algorithm with the exploitative one. In an exploratory stage, new states are visited or updated, providing better approximations of their true value. But the aim of the agent is to accumulate the maximum of rewards, which translates to the use of greedy policies. This balance between the two opposing forces is termed the *exploration-exploitation trade-off.*

The Bellman optimality equation determines Q^{*} and states that the optimal value of the action u at state x equals the sum of the immediate reward and the discounted optimal value of the optimum action at the next state:

$$Q^{*}(x,u) = \rho(x,u) + \gamma \max_{u'} Q^{*}(f(x,u),u') \tag{6.15}$$

In the stochastic case, the Bellman optimality equation takes into account the stochastic expectation of the quantities, and has the form of:

$$Q^*(x,u) = \mathbb{E}_{x'}\left\{\rho(x,u,x') + \gamma \max_{u'} Q^*(x',u')\right\} \tag{6.16}$$

6.3 Temporal Difference

Throughout an RL episode, information about the policy is being provided incrementally. In every time step, an action is chosen and the reward value is then provided. Even though this scalar provides no interpretation regarding the optimal V-function, it is an indication of a sequence of similar types of events. There are a number of ways to exploit this scalar value. The first would be to try to accumulate the rewards from each state and average them over the episodes. But as the environment is stochastic, the accumulated sequence of rewards is just a sample. As in any other statistics-oriented method, this averaging over the episode requires a large number of sequences to produce safe results. This approach is known to the literature as the Monte Carlo technique. In contrast, Temporal Difference (TD) needs only one time step in order to make a prediction. In every time step, it performs an update using the observed reward, r_{k+1}, and one more estimation of the value of $V(x_{k+1})$. The simplest form of this method is TD(0) and the update rule is the following:

$$V(x_k) \leftarrow V(x_k) + \alpha(r_{k+1} + \gamma V(x_{k+1}) - V(x_k)) \tag{6.17}$$

When the state changes from x_k to x_{k+1} and the reward is provided we can make a correction to the value of $V(x_k)$. The concrete information is the reward received, r_{k+1}, and it can be discounted and added to the value of the next state $V(x_{k+1})$. The new information provides a better approximation of the previous state $V(x_k)$. The constant step-size parameter α controls the process and has to be decreased during learning. The conditions that the parameter have to comply with are:

$$\sum_{k=0}^{\infty} \alpha_k^2 < \infty \text{ and } \sum_{k=0}^{\infty} \alpha_k = \infty \tag{6.18}$$

TD methods learn their estimates in part on the basis of other estimates, which in statistics is called a bootstrap. They (TD methods) have the ability to be implemented on-line and they do not need a model of the environment. Learning is achieved thoughout

the process of the episode and not only at the end of it [69] [70].

TD(λ)

The TD(0) algorithm is actually an instance of a more general class of algorithms called TD(λ). Instead of looking just one step ahead, TD(λ) updates all the visited past states also, in accordance with the TD measure. The update rule in this case is:

$$V(x_k) \leftarrow V(x_k) + \alpha(r_{k+1} + \gamma V(x_{k+1}) - V(x_k))e(x_k) \tag{6.19}$$

and is applied not only to the immediately previous state, x_k, but to every state according to the eligibility $e(x_k)$. There are a number of ways to calculate the eligibilities and one of these is:

$$e(x) = \sum_{k=1}^{t} (\lambda\gamma)^{t-k}\delta_{x,x_k}, \text{ where } \delta_{x,x_k} = \begin{cases} 1 & \text{if } x = x_k \\ 0 & \text{otherwise} \end{cases} \tag{6.20}$$

The eligibility of a state, x, is the degree of the state's visits in the past. States further back in the chain (i.e. in the past) will be updated proportionally to the time distance between the current state and the past state. When $\lambda = 0$ the algorithm becomes $TD(0)$. For larger λ the algorithm has been observed to converge faster [32].

TD(λ) takes a more general form if we approximate the value-functions with a parameterised structure. If we employ a linear function, P, of x_k and the parameter vector w, that is if $P_k = \sum_i w^T(i)x_k(i)$ where $w(i)$ and $x_k(i)$ are the i^{th} component of w and x_k and we use in respectively to approximate a value-function, then the update rule takes the form:

$$\Delta w_k = \alpha(P_{t+1} - P_k)\nabla_w P_k \tag{6.21}$$

Incorporating the λ formalism, the resulting rule is [69]:

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t} \lambda^{t-k}\nabla_w P_k \tag{6.22}$$

6.3.1 SARSA Algorithm

The SARSA algorithm is one of the most prominent RL algorithms, and a great many other such algorithms are derived from its core assumptions. The SARSA algorithm makes use of the state-action function $Q(x, u)$ and of TD error. In order to converge, it demands

exploration of the state-action space in a way that guarantees that every pair of state and action will be visited infinite times. As the Q-function converges to its optimal value, a greedy policy has to be employed to exploit the knowledge gathered so far.

A classic way to balance exploration and exploitation during learning is the ϵ-greedy method, which selects actions according to:

$$u_k = \begin{cases} u \in \arg\max_{u'} Q_k(x_k, u') & Prob = 1 - \epsilon_k \\ \text{a uniformly random action in U} & Prob = \epsilon_k \end{cases} \tag{6.23}$$

where $\epsilon_k \in (0, 1)$ is the exploration probability at time step k. Another option is to use Boltzmann exploration according to the formula:

$$P(u|x_k) = \frac{e^{Q_k(x_k,u)/\tau_k}}{\sum_{u'} e^{Q_k(x_k,u')/\tau_k}} \tag{6.24}$$

where the τ_k is called the temperature and controls the randomness of the exploration. Values close to 0 are equivalent to greedy action selection. On the other spectrum, as the parameter increases, a more uniform strategy of exploration is applied [10] [70].

SARSA receives its name from the acronym state-action-reward-state-action. Initially the agent perceives its state and selects an action, receiving a reward. Subsequently the agent observes the new state and selects a new action according to the policy employed. Then the algorithm makes use of the TD error to update the Q-function. The full process can be seen in Algorithm 2.

Algorithm 2 SARSA with ϵ-greedy exploration

exploration schedule $\{e_k\}_{k=0}^{\infty}$, learning rate schedule $\{\alpha_k\}_{k=0}^{\infty}$
initialise Q-function
measure initial state x_0

$$u_0 = \begin{cases} u \in \arg\max_{u'} Q_0(x_0, u') & Prob = 1 - \epsilon_0 \\ \text{a uniformly random action in U} & Prob = \epsilon_0 \end{cases}$$

for every time step $k = 1, 2, \ldots$ **do**
apply u_k, measure next state x_{k+1} and reward r_{k+1}

$$u_{k+1} = \begin{cases} u \in \arg\max_{u'} Q_k(x_{k+1}, u') & Prob = 1 - \epsilon_{k+1} \\ \text{a uniformly random action in U} & Prob = \epsilon_{k+1} \end{cases}$$

$Q_{k+1}(x_k, u_k) \leftarrow Q_k(x_k, u_k) + \alpha_k[r_{k+1} + \gamma Q_k(x_{k+1}, u_{k+1}) - Q_k(x_k, u_k)]$
end for

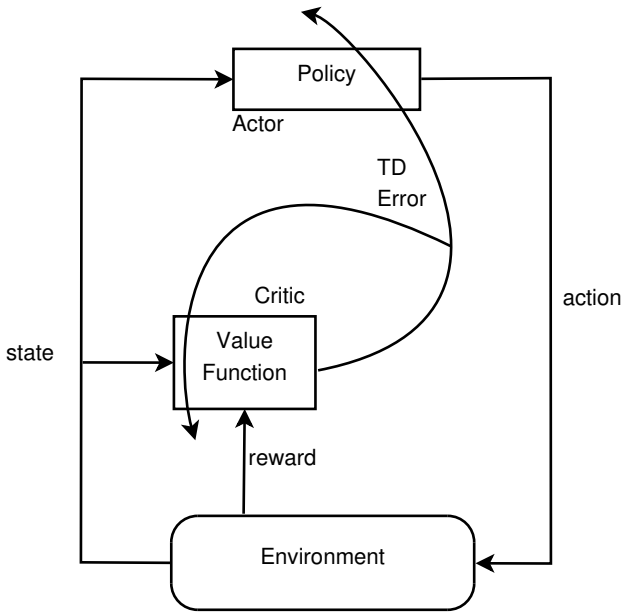


Figure 6.2: Actor-Critic architecture

Algorithms like SARSA, which evaluate the policy they are currently employing to control the environment, are called "on-policy" algorithms in the RL literature.

6.4 Actor-Critic Design

A very powerful architecture with a simple design, and the ability to solve a great many RL problems, is the Actor-Critic architecture. The agent is divided between the Actor, whose role is to choose actions, and the Critic, which criticise the Actor's decisions. Learning is always on-policy; the Critic has to learn and then criticise whatever policy is being employed by the Actor. The quantification of the reasoning of the Critic is implemented via the TD error at each time step. A general Actor-Critic architecture can be seen in Figure 6.2 [70]. The TD signal emanates from the Critic and shapes the learning both in the Actor and in the Critic.

Typically the critic is a value-function, either $V(x_t)$ or $Q(x_t, u_t)$. Any algorithm designed to optimise the value-function is applicable to control the critic part of the Actor-Critic design. In the case of the V-function, the action u_t can be evaluated in relation to the incomplete information we have from the so-far optimised $V(x)$-function. For each

action selected the TD error is calculated by:

$$\delta_t^{TD} = r_{t+1} + \gamma V(x_{t+1}) - V(x_t) \tag{6.25}$$

The immediate use of this quantity can be to optimise the V-function. But the information acquired can also be used to evaluate the impact of the current action. A positive δ_t^{TD} signifies that the action taken positively improved the state of the system and a negative sign shows degradation in comparison with the previous state.

The Actor part can be constituted by any kind of mapping from states to action. At any given time, this mapping represents the policy, π, being employed. The sequence of policies generated by the Actor-Critic architecture become greedier over time. If the actor follows a so-called *Gibbs softmax* method

$$\pi_t(x,u) = Pr\{x_t = x, u_t = u\} = \frac{e^{p(x,u)}}{\sum_{u'} e^{p(x,u')}} \tag{6.26}$$

then the update rule of the parameters that control the probabilities for selecting actions, makes use of the TD error:

$$p(x_t, u_t) \leftarrow p(x_t, u_t) + \beta\delta_t \tag{6.27}$$

where β is another positive step size parameter. If the action-space is large or continuous, the Actor itself may be represented by some function-approximation.

Algorithms that update the policy before it is completely evaluated are said to follow the *generalised policy iteration*. The policy that generates the samples x_t in general could be different from the policy that the Actor-Critic targets. This has the advantage that the Critic can learn about actions not preferable to the current policy, in order to improve the target one [73].

Function-Approximation and Gradient Descend

In very large discrete or continuous spaces, function approximation of the original value-functions is needed. The same is true for the representation of the Actor if the action space falls into the same category. Function approximation techniques project the initial space, with dimensions d_i, to a space of fewer dimensions d_f ($d_i > d_f$). During the process we lose some information but we get a simpler and more manageable function.

In linear models we can represent the target function as:

$$V(x_t) = \theta^T \phi_\theta(x_t) \tag{6.28}$$

where the space of parameters is the set of all vectors $[\theta(1), \theta(2), \ldots, \theta(n)]^T$. The functions $\phi(x_t)$ are called features and they actually perform the transform to reduce the dimensionality. They may take the form of a set of polynomial functions of the original dimensions as in:

$$\phi(x) = \alpha_0 x^0 + \alpha_1 x^1 + \cdots + \alpha_n x^n \tag{6.29}$$

or of a net of multi-Gaussians functions.

We assume that in a sequence of steps, the state x_t is completely known and the same applies to the correct value function $V^\pi(x_t)$. Because function approximators employ limited resources and as a consequence possess limited resolution, the problem of optimising the approximator even with complete knowledge of the environment is a difficult one.

A fundamental and very successful method to optimise the behaviour of such structures is gradient descent. What is demanded, is to minimise the difference between the actual sample and the one that is being approximated:

$$E = (V^\pi(x_t) - V_t(x_t))^2 \tag{6.30}$$

A solution to this problem is to update the parameters vectors towards the direction that reduces the error faster. The steepest descent is the negative of the direction of gradient of the error $\nabla_\theta E$ and the complete update rule is:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \frac{1}{2}\alpha \nabla_\theta (V^\pi(x_t) - V_t(x_t))^2 \\ &= \theta_t + \alpha (V^\pi(x_t) - V_t(x_t)) \nabla_\theta V_t(x_t) \end{aligned} \tag{6.31}$$

where α is a positive step-size parameter. We do not try to eliminate the error in all states completely but to find a balance in the errors of the different states. The difference between the actual function and the approximated one, for the whole range of the inputs, reduces while following the negative of the gradient direction. This methodology can be employed in both the Critic part and the Action part of the Actor-Critic design.

6.5 Neural Network Theory

A Neural Network (NN) is a machine that is designed to model the way in which the brain performs a particular task or function of interest. The structure of a NN varies from a simple linear one to a highly adaptive non-linear one. The general architecture comprises a massive interconnection of simple computing cells referred to as "neurons". Formally a definition of NN is [24]:

Definition 6.5.1. A Neural Network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

1. Knowledge is acquired by the network from its environment through a learning process.

2. Inter-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

6.5.1 The Structure

The building block of NNs are the *neurons*. These perform a functional transformation of the linear combination of the input variables as:

$$y(x) = \phi(\sum_{i=1}^{K} w_i x_i + w_0) \tag{6.32}$$

The w_i are adjustable parameters, termed synaptic weights. The first weight, w_0, is referred to as the bias. The function $\phi(x)$ is called the activation function and can take the form of:

- linear as $\phi(x) = x$
- non-linear and non-differentiable as $\phi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$
- a non-linear and differentiable $\phi(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

The neurons actually divide the input space, by a hyper-surface, into two parts: those x that yield $y(x) < 0$ and those for which $y(x) \geq 0$. If, for the problem at hand, a linear

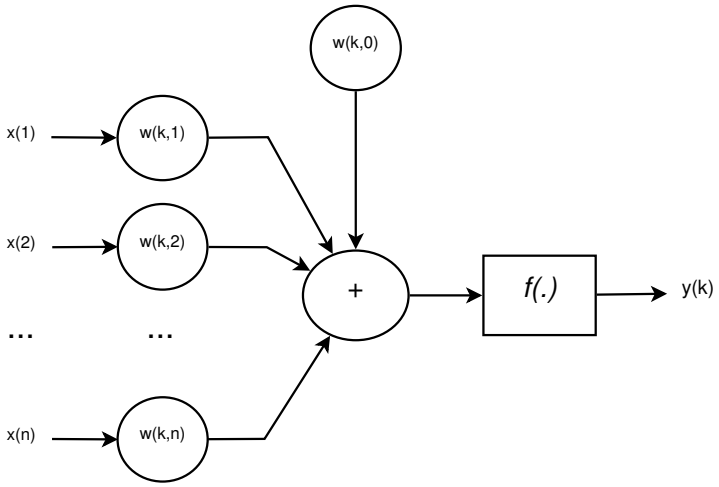


Figure 6.3: The neuron

hyper-surface is sufficient, as in the case of linear separable classes, then one perceptron has all the capability that is required. A graphical depiction of a single neuron is given in Figure 6.3. But it is the possible combination of a great many linear hyper-planes and non-linear transformation functions that gives rise to the potential of a NN to be a universal approximator.

A NN acquires significant representation abilities if we create layers of neurons, where each layer is the input of the next. The so-called multi-layer network's architectural graph is given in Figure 6.4. The general equation governing the multi-layer NN, with two layers and one output, is:

$$y = \phi_o(\sum_{j=0}^{N_{hidden}} w_j \phi_h(\sum_{i=0}^{N_{input}} w_{ji} x_i)) \tag{6.33}$$

where in each neuron the bias, w_0, is implicitly multiplied with an input regarded as 1.0.

Multi-layer NN with sigmoid activation functions have been proved to be *universal approximators* [28]. According to the theorem, a neural network with hidden layers and an appropriate number of neurons in each layer can approximate any function arbitrarily well. This ability of NNs has found application in a diverse arena of subjects such as pattern recognition, regression, control and anywhere else we need a highly adjustable function.

6.5.2 Backpropagation

The multi-layer NN is usually a non-linear mapping of the input variables to the corresponding dependent variables. The non-linearity poses a challenge in terms of describing

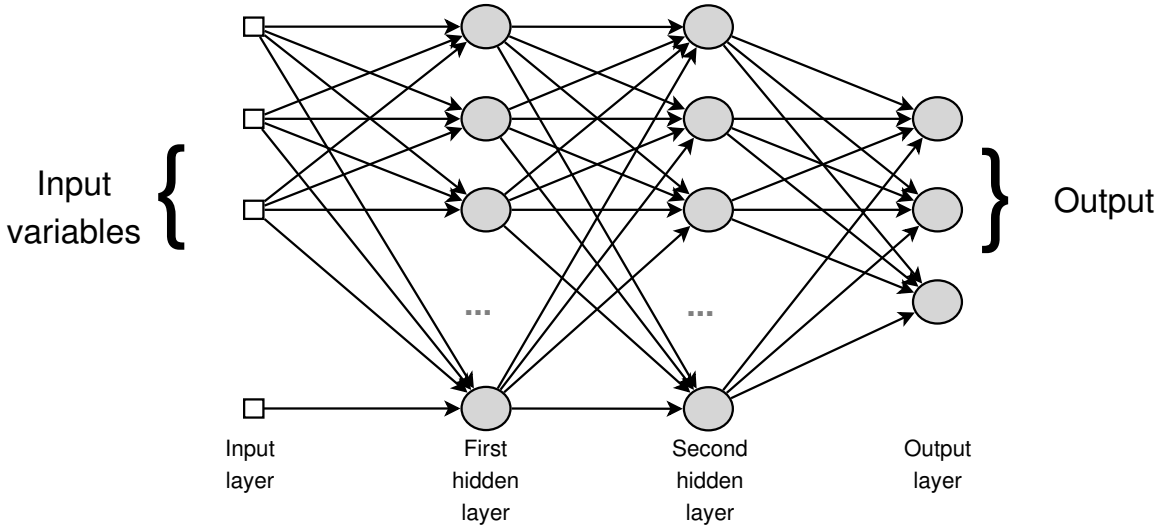


Figure 6.4: Multi-layer Neural Network

and manipulating the mathematical expressions that govern the NN, but this is compensated by the highly adaptable capabilities of the network. The problem of finding a method to optimise a neural network's weights in all its layers, was overcome with the introduction of the backpropagation learning scheme [77].

NN learning demands that by exposing to the network a series of examples of correct behaviour, the NN learns not only to imitate the response to the provided data but also to generalise to unforeseen cases. The example data takes the form of duples (x_i, t_i), where the input is x_i and the desired response is t_i. The error measure $\mathscr{E}$ is the sum of the square distances of the NN output to the desired values:

$$\mathscr{E} = \frac{1}{2}\sum_{i=1}^{N}(t_i - y_i(x_i))^T(t_i - y_i(x_i)) = \frac{1}{2}\sum_{i=1}^{N} e_i^T e_i \tag{6.34}$$

where $e_i = t_i - y_i(x_i)$ is the error vector of a single example for a multi-output NN.

The backpropagation algorithm adjusts the synaptic weights, w_{ji}, in each layer by a correction, Δw_{ji}, which is proportional to the partial derivative of the error in respect of the weights $\frac{\partial \mathscr{E}}{\partial w_{ji}}$. Applying the chain rule of calculus, we can express this derivative for every synaptic weight in every layer. In the case of the weights of the output layer the gradient is:

$$\frac{\partial \mathscr{E}}{\partial w_{ji}} = \frac{\partial \mathscr{E}}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} \tag{6.35}$$

where $v_j = \sum_{i=1}^{N_k} w_{ji}x_i + w_{j0}$. Each partial derivative of the expression can easily be computed:

$$\frac{\partial \mathscr{E}}{\partial e_j} = e_j, \qquad \frac{\partial e_j}{\partial y_j} = -1, \qquad \frac{\partial y_j}{\partial v_j} = \phi'(v_j), \qquad \frac{\partial v_j}{\partial w_{ji}} = x_i \tag{6.36}$$

where the prime in $\phi'(v_j)$ signifies differentiation with respect to the argument. Finally the update takes the form of the *delta rule*:

$$\Delta w_{ji} = -\eta \frac{\partial \mathscr{E}}{\partial w_{ji}} \tag{6.37}$$

where η is the learning rate parameter of the algorithm. As the gradient points to the direction in which a quantity increases, the minus sign signifies gradient descent in the weight space [24].

If we define the net input as the quantity before the activation function

$$n^k(x) = \sum_{i=0}^{N} w_{ji}x_i \tag{6.38}$$

where the index k denotes the layer, and subsequently define the sensitivity of the error to changes in the net input of unit i in layer k as:

$$\delta^k(i) \equiv \frac{\partial \mathscr{E}}{n^k(i)} \tag{6.39}$$

then the partial derivative with respect to the synaptic weights can be written:

$$\frac{\partial \mathscr{E}}{\partial w_{ji}} = \frac{\partial \mathscr{E}}{\partial n^k(i)} \frac{\partial n^k(i)}{\partial w_{ji}} = \delta^k(i) y^{k-1} \tag{6.40}$$

where y^{k-1} is the output of layer $k-1$. It can thus be shown that the sensitivities satisfy the following recurrence relation [23]:

$$\delta^k = \dot{\Phi}^k(n^k) W^{k+1^T} \delta^{k+1} \tag{6.41}$$

where $\dot{\Phi}$ is the following matrix:

$$\dot{\Phi}^k(n^k) = \begin{bmatrix} \dot{\phi}^k(n^k(1)) & 0 & \dots & 0 \\ 0 & \dot{\phi}^k(n^k(2)) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \dot{\phi}^k(n^k(N_k)) \end{bmatrix} \tag{6.42}$$

and W^{k+1} is the matrix of the synaptic weights of the $k+1$ layer. From the recurrence relation we see that we can calculate the sensitivities of one layer by the sensitivities of the next layer. The starting point for the recurrence is the output layer and the error of the NN:

$$\delta^M = -\dot{\Phi}^M(n^M)(t_i - y_i) \tag{6.43}$$

The learning process under backpropagation consists of two passes. The forward pass whereby, from the problem input, we calculate the outcome of the NN, and a backward pass whereby the error propagates back layer by layer to adjust the synaptic weights of the network.

6.6 Actor-Critic Revisit

Actor-Critic design demands a good approximator for the two elements, the Actor and the Critic. The approximator has to be flexible, with high representational power, and have high resolution in relation to the output - in order to always produce a valid solution. Neural Networks cover all these requirements with the non-linearity introduced in the activation functions and the large number of free parameters, providing a highly adjustable function approximator.

Two NNs can be created, one representing the Actor and one representing the Critic. In practice, a network with two layers is enough but the number of neurons is dictated by the nature of the problem. The critic network is optimised with backpropagation in reference to the TD error. We define the prediction error as:

$$\begin{aligned} e_c(t) &= r(t) + \alpha J(t) - J(t-1) \\ &= \alpha J(t) - (J(t-1) - r(t)) \end{aligned} \tag{6.44}$$

where $J(t)$ is the output of the Critic. Also we have to shift error calculation one time step

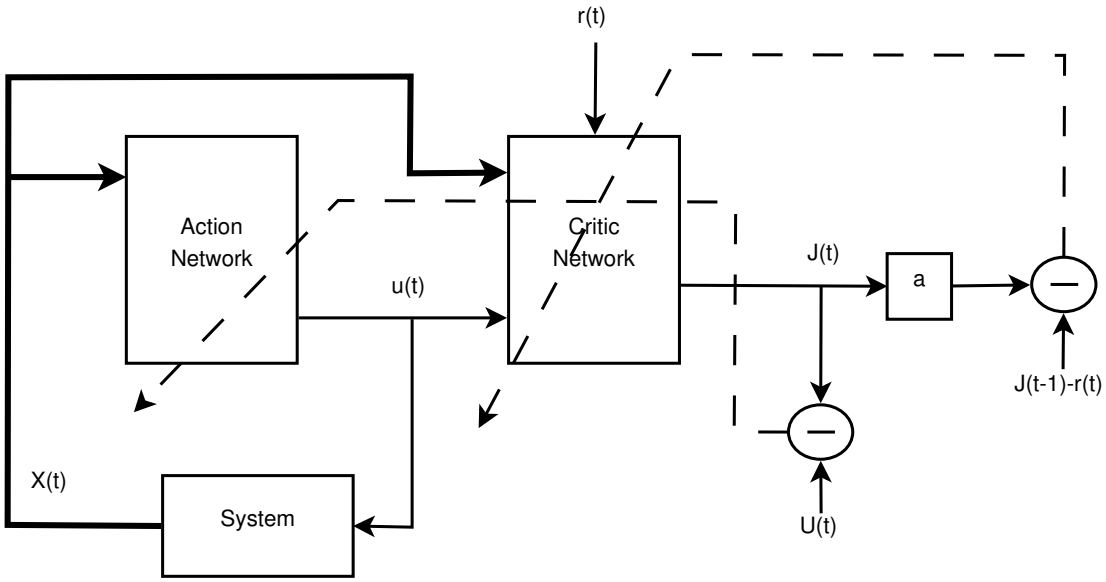


Figure 6.5: Neural Networks implementation of Actor-Critic. The solid lines represent signal flow, while the dashed lines are paths of parameter tuning

back. If the objective function to minimise is

$$E_c(t) = \frac{1}{2} e_c^2(t) \tag{6.45}$$

then the weight update rule for the critic network is:

$$\Delta w_c(t) = l_c(t) \left[-\frac{\partial E_c(t)}{\partial w_c(t)} \right] \tag{6.46}$$

$$\frac{\partial E_c(t)}{\partial w_c(t)} = -\frac{\partial E_c(t)}{\partial J(t)} \frac{\partial J(t)}{\partial w_c(t)} \tag{6.47}$$

On the Action side, we backpropagate the error between the desired objective U_c and the J function of the critic:

$$e_a(t) = J(t) - U_c(t) \tag{6.48}$$

For most of the cases $U_c = 0$, which signifies success. The performance error measure that is minimised on the action side is:

$$E_a(t) = \frac{1}{2} e_a^2(t) \tag{6.49}$$

and finally the gradient descent takes the form:

$$\Delta w_a(t) = l_a(t) \left[-\frac{\partial E_a(t)}{\partial w_a(t)} \right] \tag{6.50}$$

$$\frac{\partial E_a(t)}{\partial w_a(t)} = \frac{\partial E_a(t)}{\partial J(t)} \frac{\partial J(t)}{\partial u(t)} \frac{\partial u(t)}{w_a(t)} \tag{6.51}$$

where u(t) represents the output of the action network [66].

The overall diagram of the learning process that takes place can be seen in Figure 6.5. With the solid lines: are the state of the system, the action applied, and the function $J(t)$. The dashed lines represent the partial differentiation of the two errors propagated back to the two NNs representing the Actor and the Critic.

6.7 Next Chapter

RL is a powerful paradigm and NN serves as an universal approximator, required to overcome the curse of dimensionality when we have to work with continuous state or action space. The prediction of the behaviour of the learning system in a per step basis offers the ability to exploit the information regarding the agent's performance in two ways. As a whole at the end of each episode the accumulated reward designates the agent overall performance. But with a critic module we could make decisions regarding the agent in a time step basis. The Actor-Critic architecture use Gradient Descent to optimise the networks. This requires a learning rate that has to be regulated during the learning process. Gradient Descent make use only of the Gradient information. Another option would be to use the Hessian too and this is the subject of the next chapter.

Chapter 7

Actor-Critic with modified Levenberg-Marquardt

7.1 Introduction

The Actor-Critic topology has been very successful in a wide range of RL problems. The power of the method rests in the modular structure of the two entities, the Actor and the Critic, represented by two different Neural Networks. The objective of RL, to optimise the long term reward, which is represented by the Critic network, is achieved by updating the weights of the Action network at every step in relation to the latest knowledge of the state-action function. This is accomplished by taking the negative of the derivative of the $Q(s, a)$ in relation to the action and propagating it backwards in the Action network [66].

A major limitation of this approach is the learning rates of the backpropagation method [77]. Backpropagation is a steepest descent method and in order to avoid losing the local minimal, we have to decrease the learning rate as the process evolves. This applies both to the Actor and to the Critic network. Another drawback is the slow rate of convergence.

A methodology that overcomes these limitations is Levenberg-Marquardt (LM) [40, 43]. This method is a mix of gradient descent and Gauss-Newton optimisation. It is stable and eliminates the concept of the learning rate. The LM provides a solution to the Nonlinear Least Squares Minimisation problem. As NNs are nonlinear functions, the method has been proven to be very successful in training of NNs under the supervised learning paradigm.

7.2 Levenberg-Marquardt algorithm in Neural Networks

The Levengberg - Marquardt algorithm is an approximation to Newton's method [60]. According to Newton's method for a function $V(x)$ the update rule would be:

$$\Delta x = -[\nabla^2 V(x)]^{-1} \nabla V(x) \tag{7.1}$$

where $\nabla^2 V(x)$ is the Hessian matrix and $\nabla V(x)$ is the gradient. The performance index that is being optimised is the sum of the squares of the error between the desired value t_i and actual output of the network s_i:

$$V = \frac{1}{2} \sum_{k=1}^{N} (t_k - s_k)^T (t_k - s_k) = \frac{1}{2} \sum_{k=1}^{N} e_k^2 \tag{7.2}$$

It can be shown that when the function optimised is a sum of squares then [23]:

$$\nabla V(x) = J^T(x) e(x) \tag{7.3}$$

$$\nabla^2 V(x) = J^T(x) J(x) + S(x) \tag{7.4}$$

where $J(x)$ is the Jacobian matrix:

$$J(x) = \begin{bmatrix} \frac{\partial e_1(x)}{\partial x_1} & \frac{\partial e_1(x)}{\partial x_2} & \dots & \frac{\partial e_1(x)}{\partial x_n} \\ \frac{\partial e_2(x)}{\partial x_1} & \frac{\partial e_2(x)}{\partial x_2} & \dots & \frac{\partial e_2(x)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_N(x)}{\partial x_1} & \frac{\partial e_N(x)}{\partial x_2} & \dots & \frac{\partial e_N(x)}{\partial x_n} \end{bmatrix} \tag{7.5}$$

and

$$S(x) = \sum_{i=1}^{N} e_i(x) \nabla^2 e_i(x). \tag{7.6}$$

The Gauss-Newton method assumes that $S(x) \approx 0$ and subsequently the learning rule becomes:

$$\Delta x = [J^T(x) J(x)]^{-1} J^T(x) e(x). \tag{7.7}$$

The Levenberg-Marquardt modification to the Gauss-Newton method is:

$$\Delta x = [J^T(x)J(x) + \mu I]^{-1} J^T(x)e(x). \tag{7.8}$$

The parameter μ controls the algorithm and when it is large the method tends to perform like steepest-descent (with step size $1/\mu$); when this parameter is small the method acts more like Gauss-Newton. The adaptation of μ is done through multiplication by some factor (β) in case the error increases. In the opposite case the parameter is divided by the same factor.

To compute the Jacobian in the case of a neural network, we first note that x represents the adaptable weights $w^k(i,j)$ of the NN. So we formulate a vector x such as:

$$x = [w^h(1,1), w^h(1,2), \cdots, w^h(n,m), w^o(1,1), w^o(1,2), \cdots, w^o(k,l)]. \tag{7.9}$$

The elements of the Jacobian are calculated from terms such as:

$$\frac{\partial e_l(m)}{\partial w^k(i,j)}. \tag{7.10}$$

These terms can be calculated using a parallel process to backpropation with one modification at the final layer:

$$\Delta^M = -\dot{\Phi}^M(n^M) \tag{7.11}$$

where $\dot{\Phi}^M(n^M)$ have been defined in Equation 6.42.

7.3 Modified Actor-Critic topology with Levenberg-Marquardt

Ni et al. [50] have implemented an improvement on the classic Actor-Critic algorithm by using Levenberg-Marquardt as learning procedure for the Critic network. The Critic is represented by a NN with a sigmoid transfer function in the hidden layer and a linear function in the output layer. The equations that govern the NN, for ease of reference, are

described by:

$$q_i(k) = \sum_{j=1}^{N_{cin}} w_{ij}^h(k) x_j(k), \qquad i = 1, \cdots, N_{ch} \tag{7.12}$$

$$p_i(k) = \frac{1 - exp^{-q_i(k)}}{1 + exp^{-q_i(k)}}, \qquad i = 1, \cdots, N_{ch} \tag{7.13}$$

$$J(k) = \sum_{i=1}^{N_{ch}} w_i^o(k) p_i(k) \tag{7.14}$$

where N_{ch} is the number of hidden nodes and $N_{cin} = n + 1$ is the number of inputs. Let's assume that the control system is described by n state variables and one control variable. The objective function to be minimised in the critic network is:

$$e_c(k) = \alpha J(k) - [J(k-1) - r(k-1)] \tag{7.15}$$

$$E_c(k) = \frac{1}{2} e_c^2(k) \tag{7.16}$$

Based on Equations 7.15, we can apply the Leverberg-Marquardt as a solving technique for the least square optimisation problem. Therefore we can achieve adaptation of the Critic NN by integrating the LM algorithm into its chain backpropagation. We represent the weights modification from the input to the hidden layer as δw_c^h and with δw_c^o the modification from the hidden to the output neuron. In order to form the Jacobian matrix we need to calculate the derivative of e_c with respect to w_c. Therefore, we get [50]:

$$J_{f_c} = [\delta w_{ij}^h \quad \delta w_i^o] \tag{7.17}$$

$$\delta w_i^o = \frac{\partial e_c}{\partial J} \frac{\partial J}{\partial w_i^o} = \alpha p_i(k) \tag{7.18}$$

$$\delta w_{ij}^h = \frac{\partial e_c}{\partial J} \frac{\partial J}{\partial p_i} \frac{\partial p_i}{\partial q_i} \frac{\partial q_i}{\partial w_{ij}^h} = \alpha w_i [\frac{1}{2}(1 - p_i^2)] x_j \tag{7.19}$$

The LM Actor-Critic has been tested at the Cart-Pole balance with respect to different noise patterns and has been shown to outperform the classic Actor-Critic and to speed up the convergence rate [50].

7.3.1 Proposed modification

For the LM supervised learning algorithm the Jacobian matrix is composed of the derivative of all the errors $e_i \quad i = 1..N$ with respect to the parameters. Every row represents the error for a different pair of NN output and actual value. In LM used for Actor-Critic we have only the current error between the $\alpha J(k)$ and $J(k-1) - r(k)$.

In our proposed modification we accumulate the errors from a "depth" of 100 past values. We also keep track of the trajectory of the input variables of the system, the control applied to it and the reward acquired. When we calculate the Jacobian matrix we reevaluate for every row the quantity $J(k-1)$ from the input vector at the step $k-1$ by obtaining the responce of the NN at the corresponding state. Then we formulate the error by subtracting the corresponding reward for that time step.

$$e = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{100} \end{bmatrix} = \begin{bmatrix} \alpha J(t-99) - (J(t-100) - r(t-99)) \\ \alpha J(t-98) - (J(t-99) - r(t-98)) \\ \vdots \\ \alpha J(t) - (J(t-1) - r(t)) \end{bmatrix} \tag{7.20}$$

Subsequently we calculate the δw_c of the NN in regard to the error, e, as stated by the LM algorithm.

This modification has three substantial advantages. First, the method, instead of learning single instances of the unknown function, learns the trajectories of the behaviour of the dynamic system. This way learning becomes more efficient since, usually, the controlling of a dynamic system involves reacting to similar circumstances, which occur as patterns, when we try to guide the states of the system to a specific equilibrium. The second advantage has to do with the large reward value for the last step when the controller crashes the system. Taking it in isolation the derivatives have a very large value which is used to prevent the controller from reaching this specific state. This large value has a side effect of destabilising the learning process. With the proposed modification, this last large reward is being applied in addition to the previous states that guide the system to crashing. As a result the Critic learns to avoid states that in the long run may lead to crashes. Finally, at every step we recalculate the quantity $J(t-1)$ according to our current knowledge, represented by the Critic network. This has, as a consequence, the effect of adjusting the error, e, at every time step, according to the latest information that we have acquired through interaction with the problem. For a time window of n steps, the optimisation of

the Critic network for the state x, will be executed n-times. In every application of the LM algorithm the error will be readjusted and whole optimisation becomes more accurate.

7.4 Results

The performance of the two methodologies, the LM and the proposed modification, has been tested against the triple inverted pendulum problem. The system comprises a cart able to move in one direction and on its top there is a modular pendulum with three poles attached together at each end. The link of each pole can only rotate in the direction of the whole cart. The problem is regarded as difficult and for the case study we required the agent to solve it by starting from different random initial conditions, which amount to -3% to $+3\%$ of the range of the state variables from the equilibrium point.

The equation governing the triple inverted pendulum system is:

$$F(q)\frac{d^2q}{dt^2} = -G(q, \frac{dq}{dt})\frac{dq}{dt} - H(q) + L(q, u) \tag{7.21}$$

The sampling time interval is chosen to be $5\mu s$. From the nonlinear dynamical equation in 7.21, the state-space model can be described as follows [66]:

$$\dot{Q}(t) = f(Q(t), u(t)) \tag{7.22}$$

with

$$f(Q(t), u(t)) = \begin{bmatrix} 0_{4\times4} & I_{4\times4} \\ 0_{4\times4} & -F^{-1}(Q(t))G(Q(t)) \end{bmatrix} Q(t) + \tag{7.23}$$

$$\begin{bmatrix} 0_{4\times4} \\ -F^{-1}(Q(t))[H(Q(t)) - L(Q(t), u(t)) \end{bmatrix} \tag{7.24}$$

and

$$Q(t) = [x(t)\theta_1(t)\theta_2(t)\theta_3(t)\dot{x}(t)\dot{\theta}_1(t)\dot{\theta}_2(t)\dot{\theta}_3(t)]^T \tag{7.25}$$

The details of the rest of the quantities mentioned can be found in [66].

For the test, we performed 100 different runs and calculated the success rate of each of the two methods. Each run had a maximum of 1000 episodes. If no solution was found after 1000 episodes, the run was regarded as a fail. The modified version of the method (i.e., the Actor-Critic Levenberg-Marquardt) managed to solve the problem 82% of the

time and the average number of episodes required for the agent to reach a solution was 135. This outperforms the simple LM version of the Actor-Critic algorithm which had a success rate of 8% and an average success time of 248 episodes. So the modified version (i.e., Actor-Critic Levenberg-Marquardt version) increased the success rate ten-fold and also shortens the period needed to reach the solution by almost one half.

7.5 Next Chapter

With the review of the theory of the last two chapters we are now ready to propose a new methodology to enhance the GP paradigm. A hybrid architecture composed of GP, RL and NN is described that exploit to the maximum the information provided by the problem, and unifies the otherwise disparate theories. Next chapter describes and introduces a system that manages to optimise an agents in every time step and in every episode.

Chapter 8

Proposed Augmentation of the Function-Set of Genetic Programming with Adaptable Neural Networks

8.1 Introduction

Genetic Programming is a very powerful optimisation technique that can be applied to a range of problems - from the control of dynamic systems, to program creation for specific purposes. The function-set used, during search, has few restrictions on its definition. The members of the function-set have to obey the closure property, whereby the arguments of the function must result in a legitimate value that can be an argument to any of the other functions. That is the only requirement. NNs are non-linear functions that fullfil the closure property. The benefit of embedding NNs into to the function-set is their high adaptability, as it is known that they are universal approximators [28].

The main obstacle to incorporating NNs is that optimisation under GP is achieved through epochs. This means that information from the problem at hand comes on an episodic basis. The proposed algorithm distances itself from a pure GP paradigm and regards the control problem at hand from a Reinforcement Learning perspective also. An independent NN is added to the structure, representing a Critic that through the Temporal-Difference measure processes information on a step by step basis. Then the derivative of the Critic function $J(t)$ is propagated through the Criric network into the tree and up to

the NN that we want to optimise. The only precondition is that the function-set of the GP has to be comprised from differentiable functions.

The proposed architecture and algorithm achieves, in this way, two separate optimisations. One performing on a step by step basis for the NN modules and one applied on an episodic basis collectively in the overall population.

8.2 Insertion of Neural Networks into tree structure

Neural Networks are universal approximators and as such, they represent a specific input-output mapping. The high number of parameters that they possess makes them adaptable and able to represent any possible function. If we confine the number of inputs to the range of 2 to 4, with one hidden layer and one output then, with a moderate number of neurons in the hidden layer, we reduce the number of free parameters. These reductions in terms of the size of the NNs mean that these require less effort to configure and manage; however, not a great deal of their representational power is lost.

The NNs can then be placed anywhere in the tree structure, like any other function from the function-set. The resulting topology may see a number of NNs cascading from the root to nodes directly leading to terminals. This creates a hierarchical structure of NNs where between two NNs an intermediate function composed from the GP function-set intervenes and serves as a functional connection. Another possible topology may be one, where the root of the tree is a NN. A typical tree can be seen in Figure 8.1.

8.3 Optimisation of the Neural Networks

The only (additional) prerequisite for the function-set is that the constituent functions have to be differentiable. The reason for this restriction is the learning process employed to optimise the NNs. The whole structure is a generalised version of the Actor-Critic topology, as can be seen in Figure 8.1. The Critic module remains the same in role and in structure. The Actor, though, is a hierarchical tree of functions and NNs. In order to apply the chain rule of differentiation used by a typical NN we still have to convey the $\frac{\partial J(x)}{\partial w_{ij}^k}$ through the tree functions, up to the relevant NN.

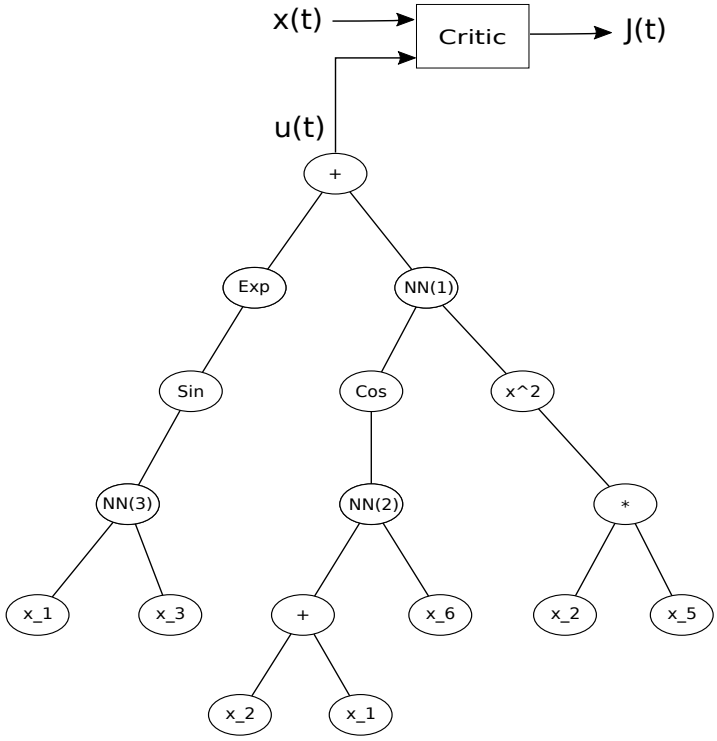


Figure 8.1: Topology of modules and the hierachical positioning of NNs in a tree

8.3.1 Critic optimisation

The Critic NN can be optimised to represent the reward to go, $J(t)$, by applying backpropagation of the error $e(t) = \alpha J(t) - (J(t-1) - r(t))$. In this part of the architecture the proposed algorithm does not depart from the known Actor-Critic structure. A subtle issue exists regarding the learning rate - that has to decrease as the process evolves. In the early generations of the GP optimisation, most of the individuals will be too simplistic and all the episodes will result in failures at very early stages. As a consequence, the Critic learns on a lot of cases that then substantially fail. On top of this, the learning rate of the Critic has to decrease through time for the Critic network to converge to the optimal function. But the naive controllers in the early stages of the GP process dominate the optimisation process. As the learning rate of the Critic has to decrease, its influence becomes insignificant when the controllers become more competent. One possible solution is to update the learning rate on a generation basis, but still this does not safeguard against the situation where the learning process does not overshoots the local minimal.

A possible solution of the above defects of the classic Critic with backpropagation is to use the Levenberg-Marquardt for the optimisation of the Critic network [50]. Applying

this learning algorithm eliminates the use of a learning rate, as such, and stabilises the optimisation process. For this reason, it has been employed for the proposed architecture.

8.3.2 Embedded Neural Network optimisation

The embedded Neural Networks can be found in any node in the tree structure. The topologies encountered may be NNs cascading from the root, and down to the leaves. The main idea is to calculate the derivative $\frac{J(x)}{\partial w_{ij}^k}$ of the $J(x)$ in relation to the weights of the NN that is being optimised.

In the general case, we are interested only in the functions created by visiting all the parent nodes of the NN, that concerns us, up to the root. The derivative is calculated in relation to the weights, w_{ij}^k, of the NN. The branches that perform the main mathematical operations (addition, multiplication, subtraction and division) and deviate from the route of the NN up to the root, are not dependant on the weights of the optimised NN. As a consequence, they create terms which are numeric constants. This is convenient for the calculation of the derivatives at the output of the NN. We need to calculate the derivatives through the Critic up to the root, then the possible function comprised from functions that are part of the function-set up to the next NN. Then we have to calculate the derivative of the NN output with relation to one of its inputs and repeat that process up to the NN that we need to optimise. At the last stage we calculate the derivatives with respect to each weights of the optimised NN.

The process, in a general case, can be seen in Figure 8.2. The chain rule of calculus for the calculation of the derivative $\frac{J(x)}{\partial w_{ij}^k}$ dictates for the NN_3 case that:

$$\frac{J(x)}{\partial w_{ij}^3} = \frac{J(x)}{\partial u(t)} \frac{u(x)}{\partial NN_3(x)} \frac{NN_3(x)}{\partial w_{ij}^3} \tag{8.1}$$

where $u(x)$ is the output of the tree and the derivative $\frac{u(x)}{\partial NN_3(x)}$ in Figure 8.2 is depicted as $\frac{\partial f_1(x)}{\partial x}$. $NN_3(x)$ in the expression signifies the output of the third NN. Also $f_1(x) = sin(exp(x))$.

As can be seen from the figure, when we calculate the derivatives with relation to NN(3) weights, the right part of tree at the root (which is the addition operator) is always a constant expression as it does not vary in relation with the weights w_{ij}^3. This observation simplifies the calculation of the derivatives. As can easily be seen now the optimisation of

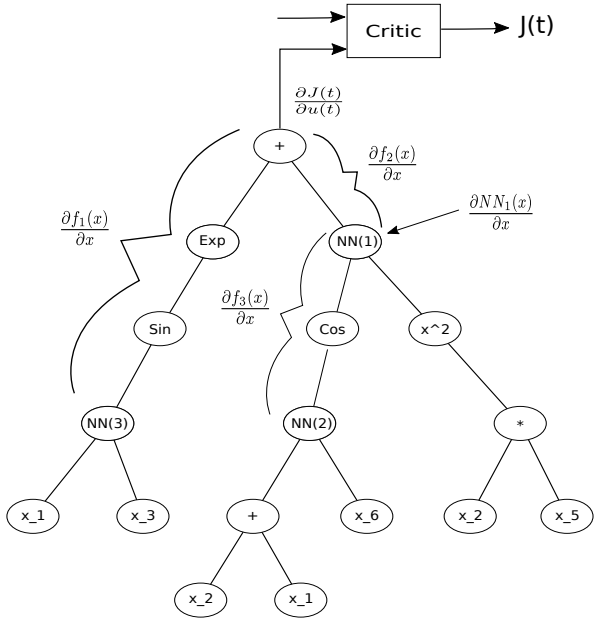


Figure 8.2: Calculating the derivatives $\frac{J(x)}{\partial w_{ij}^k}$ for a general case

the NN(2) will rely in:

$$\frac{J(x)}{\partial w_{ij}^2} = \frac{J(x)}{\partial u(t)} \frac{u(x)}{\partial NN_1(x)} \frac{\partial NN_1(x)}{\partial f_2(x)} \frac{\partial f_2(x)}{\partial NN_2(x)} \frac{NN_2(x)}{\partial w_{ij}^2} \tag{8.2}$$

where $\frac{\partial NN_1(x)}{\partial f_2(x)}$ is the derivative through the NN(1).

An issue that arises when we have multiple NNs in a tree is the order in which we update the weights of the different NNs. One possible way is to update the weights in each NN in isolation of the changes to the rest. This means that an optimised network is oblivious to changes in other NNs. This process enforces updates that are inconsistent with the state of the NNs in the resulting tree.

An alternative option is to go through a few cycles of learning in one network before moving to another one. Even though the procedure is not 100% fair for all the NNs, it guarantees that at some level the NNs are aware of the changes undergone by the rest of the NNs.

8.4 The role of the learning rates

The gradient descent method, for the optimisation of the Neural Networks, is reliant on the derivatives of an error. An important part of this method is the magnitude of the learning rate. This signifies how far we proceed along the error surface following the negative direction of the derivative. If it is too large and we might miss the local minimum and if it is too small the improvement will be negligible.

In the proposed architecture, the learning rates take a more complex form. As the number and the position of the NNs inside the tree may vary, the individual learning rate of each NN will have to take into consideration a number of factors. What is its depth in the tree, what kind of functions intervene up to the next NN or the root, how many NNs are in its path to the output of the tree? The overall influence of the learning rate of a NN in a specific topology is undetermined and a general rule has to be employed to stabilise the learning process.

8.4.1 Adaptive learning rates from Neural Network Theory

In Neural Network theory, a number of adaptive learning rate algorithms exist. All of these have been developed for the supervised learning paradigm - whereby a NN has to learn an input-output pattern. This kind of learning differs from the Actor-Critic architecture in regard to the Actor module optimisation. At the Actor NN, the weights are updated towards maximising the future reward to go function, $J(x)$, which is represented by the Critic NN. The derivative of $J(x)$ is propagated through the Critic to the weights of the Actor.

The problem of adapting the learning rates of the NNs in our proposed algorithm is equivalent to that of adapting the learning rate of a simple Actor-Critic algorithm. The only difference would be the intermediate functions (either a function from the function-set of the GP, or a mix of such functions and NNs) that connect the Critic's input and the relevant NN's output.

A number of adapting learning rates algorithms have been tested - in an effort to alleviate the negative influence of potentially large learning rates or the effects of very small ones. A prominent such algorithm is the *Delta-Bar-Delta*rule [31]. This rule requires a learning rate for each of the weights of the NN and it regulates each magnitude according to the sign of the product of the derivatives of the current and previous step. Let $w(t)$ denote the value of a single weight at time t and $\epsilon(t)$ the learning rate corresponding to

that weight at that time. The learning rate update rule is defined then as follows:

$$\Delta\epsilon(t) = \begin{cases} \kappa & if \quad \bar{\delta}(t-1)\delta(t) > 0 \\ -\phi\epsilon(t) & if \quad \bar{\delta}(t-1)\delta(t) < 0 \\ 0 & otherwise \end{cases} \tag{8.3}$$

where

$$\delta(t) = \frac{\partial J(t)}{\partial w(t)}, \qquad \bar{\delta}(t) = (1-\theta)\delta(t) + \theta\bar{\delta}(t-1). \tag{8.4}$$

The Delta-Bar-Delta has been tested on the proposed algorithm and on the classic Actor-Critic with no success. Its failure can be explained by the nature of the measure, which regulates the magnitude of the learning rates. In a typical supervised problem, the derivative of the error oscillates between positive and negative values, mainly following the sign of the error between the actual and the desired output of the NN. But in an Actor-Critic, the Critic represents the reward to go and its value depends on the type of the per step reward. That means that during the adaptation process of the Actor, the Critic will steadily produce positives or negatives values. The sign of the derivatives does not change in the same manner as in supervised learning.

The *RPROP* rule [61] also suffers the same pathology. This adaptation works also on individual weights and evolves based on the local sign of the error function E. The learning rule has:

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+\Delta_{ij}^{(t-1)} & if \quad \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \eta^-\Delta_{ij}^{(t-1)} & if \quad \frac{\partial E}{\partial w_{ij}}^{(t-1)} \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & otherwise \end{cases} \tag{8.5}$$

where

$$0 < \eta^- < 1 < \eta^+. \tag{8.6}$$

Once the magnitude of the learning rate has been updated, then a simple rule is applied. If the derivatives sign is positive $\frac{\partial E}{\partial w_{ij}}^{(t)} > 0$, which signifies an increase in error, the quantity $\Delta_{ij}^{(t)}$ is subtracted from the initial weight; the opposite process is performed in the case that the quantity is negative. The RPROP rule has also been tested, but it did not succeed in regulating the learning process in the Actor-Critic.

Another promising adaptive learning rate methodology tested upon the Actor-Critic was *ADADELTA* [82]. The main idea, here, is to regulate the value of the learning rate according to the sum of past gradients, accumulated in a window of a fixed size, w. In

order for the method to avoid storing w values, the accumulation is implemented as an exponentially decaying average of squared gradients:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g_t^2 \tag{8.7}$$

where ρ is the decay constant. The update rule of ADADELTA has the following form:

$$\Delta x_t = -\frac{RMS[\Delta x]_{t-1}}{RMS[\Delta x]_t} g_t \tag{8.8}$$

where RMS denote the quantity:

$$RMS[\Delta x]_t = \sqrt{E[g^2]_t + \epsilon} \tag{8.9}$$

where the constant ϵ is added to avoid degenerated cases where the denominator is zero. Although the adaptation of the learning rate is not reliant on the sign of the gradient, the method did not deliver useful results when tested on the Actor-Critic architecture.

The adaptation of the learning rate for the Actor-Critic architecture is an open question. The fact that the second round of optimisation of the Action network involves the maximisation of the quantity $J(t)$ with respect to the action input, renders most of the adaptive learning rate methods emanating from supervised learning inapplicable.

8.4.2 The ratio of the derivatives

Instead of controlling the learning rates according to the sign of the derivatives and per individual weights, another approach has been tested to tackle the problem of regulating the learning rate of each NN in the tree. As the structure of the trees varies and so does the position of the NNs inside the trees, the derivative of the Critic $\frac{\partial J(t)}{\partial u(t)}$ changes in magnitude at the different nodes. This means that sometimes it increases, depending on the intermediate functions, or the opposite.

In order to stabilise the learning process for each NN, the learning rate follows the ratio of the derivative at the root of the whole tree to the derivative at the output of the NN that is being optimised:

$$\epsilon(t) \propto \frac{\frac{\partial J(t)}{\partial u(t)}}{\frac{\partial J(t)}{\partial NN(t)}} \tag{8.10}$$

This ratio is able to follow the structural changes in the trees and be flexible. When it was

Function	**Symbol**	**Function**	**Symbol**
Addition	ADD	Square	SQ
Subtraction	DIV	Cube	CUB
Multiplication	MUL	Exponential	EXP
Division	DIV	Sign	SIG
Absolute Value	ABS	Sin	SIN
Square Root	SQRT	Cosine	COS

Table 8.1: The function set used by GP with embeded NN

applied it exhibited a strong stabilising force and the fitness of the trees embedded with NNs was not flactuating. The disadvantage was that it was not improving either. When the learning rates were following the ratio, the effect of the error derivative was cancelled out.

8.5 Results

The proposed algorithm possesses extreme processing power because of its two levels of optimisation. The addition of the NNs into the function-set, enriches the original function-set, giving the opportunity to the GP processes to search in an enlarged function-set. The adaptation of the individual's NNs on a step by step basis, trasnforms the initial overall function towards the requirements of the control problem. The conjecture that a fit individual is comprised from parts, that are also useful as sub-modules, applies here too [35]. The advantage of the algorithm lies in the optimisation of the NNs, which adapts the introduced sub-tree to the rest of the tree structure. The NNs' adaptation reconfigures vital sub-functions in the tree structure, resulting in more robust individuals.

The proposed algorithm has been tested on two different problems. The triple inverted pendulum and the helicopter hovering control problems. The tests were executed once with optimising the NNs and once where the NNs where fixed. In the initial population, it was enforced that every individual had a least one NN.

The function set used in both problems can be seen in the Table 8.1; and they are all differentiable functions.

8.5.1 Triple Inverted Pendulum Problem

The population size for the experiments was 500. The classic GP with the same function set but without NNs could not solve the problem after 100 generations. In both the testing cases (optimising NNs or not), the algorithm converged to a solution which would balance the pendulum for 6000 steps. The fitness for the triple inverted pendulum was calculated by adding at each step the square of the distance of the current state variables from the equilibrium point.

$$reward = \sum_{i=1}^{8}(x_i - s_i)^2 = \sum_{i=1}^{8} x_i^2 \tag{8.11}$$

For a failure, a large negative reward was given that amounted to the maximum possible error (which was half of the range of the state variables) multiplied by the remaining steps of a 6000 step episode.

The solutions acquired from the runs managed to score fitnesses below 100 in half of the runs, with a maximum (failure from the first step) of 48000, and only once was a run unable to solve the problem. The results were almost the same for both types of testing, with and without optimising the NNs. No difference in performance has been identified.

The problem was too easy for the proposed architecture and in order to increase the difficulty a population of less than 100 individuals has been tested. In this test, in most of the runs, the algorithm, either by optimising the NNs or not, was unable to reach a solution.

8.5.2 Helicopter Hovering Control Problem

For the helicopter hovering control problem, the same XCell Tempest helicopter simulator, which was implemented for the RL-competition [62], was used. The output of the trees was not supplied to the $tanh(x)$ function in order to comply with the control variables range. Also a default naive controller was not part of the population as in [17], making the problem harder and more challenging.

A population size of 800 individuals was applied and two different methodologies of crossover were used. Crossover in multi-action control problems can be implemented in two different ways. The first consists of choosing two individuals, through fitness proportional selection, and performing crossover for the first control variable. Subsequently another two individuals are selected for the second control variable and so on. The other method chooses once the individuals and applies crossover in all the corresponding trees of the

control variables. In our experiments, we used a the crossover probability of 90% and the two methodologies of crossover were used equally. Also elitism was applied. The best 20 of each generation were reproduced to the next population.

The proposed architecture managed to optimise the controllers and achieve fitnesses in the range 300 to 8500. The default controller, which merely avoids crashing the helicopter, has a fitness of around $1E^6$ - when the maximum penalty amounts to $1E^7$. The algorithm managed to converge to solutions in 400 generations, which is much more than the number of generations required by the simple GP 120 generations [17]. But again it was not possible to distinguish the effect of optimising the NNs or not.

Finally, tests were done on individuals extracted from the population in order to justify the optimisation process of the NN. Trial and error tests were performed and different learning rates on the individual NNs of the trees were applied. The tests showed that substantial improvements in the performance can take place if appropriate learning rates for each of NNs of the trees were provided. But the heuristic search used in investigating ideal learning rates, was not able to converge to a rule that could be universally applied.

8.6 Implementation

The main difficulty of the implementation was associated with the differentiation of the intermediate functions inside the tree structure that could connect two Neural Network nodes. Manipulation of specific nodes was required which with the inclusion of the Neural Network in the function set made inapplicable the various libraries that implement GP. The software solution was written in Java and architecturally was divided in the GP part of the optimisation and in the NN part. For the symbolic differentiation of any possible function an excellent solution of the problem at the CodeProject website [2] was used and coded in Java in order to incorporate it with the rest of the modules of the application.

To run the experiments Cloud computing services were used. Microsoft Azure [1] was the specific Cloud and virtual machines of one core were employed. The most demanding in terms of computing resources as in time needed was the Helicopter Hovering experiments. For a single type of wind pattern a full optimisation that created a successful agent would need around 3 days.

Chapter 9

Conclusions

9.1 Thesis Summary

This thesis has investigated EC methods in RL continuous state and action control problems. The EC methodology that this research has concentrated upon was GP. The GP methodology produce computer programs, which are immediately translated to control laws. A hard and chanllenging problem, that of helicopter hovering control problem, has been solved. This problem has four action controls, and the successful application of the GP expanded the type and difficulty of the problems that the methodology has been proven to produce solutions. Especially, two types of the crossover operator have been identified, one that operates on an action type level and one on individual level.

The GP is a generic process and as such, it is not confined on control problems only. It manages to produce impressive results in a totally differenent area, that of prediction of the survival curve for a specific pathogen. The derived solution has been compared with the already established methods of MLP and PLS. The versatility of the GP scheme was proven to outperform the mentioned techniques.

The apex of this research was the proposal of a modular scheme, which makes use of the theories of GP, RL and NN. The proposed scheme showed that the capabilities of the structure enhance the performance of these methods when we apply them in isolation. Even though the proposed methodology utilises proven mathematical procedures, it suffers from a mild but crucial weakness. The learning rates of the embedded NNs have to be coordinated in order for the whole scheme to achieve its full potential. The application of the LM method to optimise the Critic part of the scheme, eliminates the use of the learning rate for the Critic, and a modification of the initial algorigthm transformed the

Critic network to a solid and accurate predictor of the algorithm's performance. But the literature regarding adaptable learning rates in the optimisation of NNs, failed to control and to stabilise the performance of the Action part, wich was represented as a tree structure with embedded NNs.

The essential background material in Evolutionary Computing has been outlined in Chapter 2 followed by the application of Genetic Programming in a food processing problem in Chapter 3. An in depth presentation of several EC algorithms in control problems follows in Chapter 4. The next chapter (Chapter 5) describes the solution of the helicopter hovering control problem from the Genetic Programming methodology. Subsequently this thesis presents the elements of Reinforcement Learning and Neural Networks theory (Chapter 6) required for the proposed scheme that unifies Genetic Programming with Reinforcement Learning in Chapter 8. Previously, the modification of the Levenberg-Marquardt for the Critic part of the Actor-Critic architecture is analysed is Chapter 7.

9.2 Discussion

One of the major contributions to knowledge, attributed to this research, is the proposed unification of the theories of Genetic Programming, Reinforcement Learning and Neural Networks. The architecture translates and utilises the information available to the learner first in a step by step basis and later on an episodic basis. The addition of a Critic NN in the GP scheme supplies predictions of the value function, which makes possible to adapt NNs that are embedded in the trees. The only prerequisite is that all of the functions from the function set to be differentiable. As each NN represents a different function, the inclusion of NNs to the function set of the GP scheme automatically broadens the functional horizon. This has an immediate positive effect in terms of the difficulty level of the control problems that can be solved. A subtle point though is that during the GP optimisation process different NNs can possibly be lost, if the trees that they belong to are not reproduced or are not selected for crossover. So number of functions in the function set is actually smaller than the total number of NNs, if immutable NNs are used.

The sound mathematical properties of the architecture, that underpin the optimisation of the NNs, though has an Achille's heels. The resulting hierarchical structure of NNs demands a unique learning rate in relation to the position of the NN in the tree and the surrounding functions. The situation is not different from adapting the Action's learning rate in an Actor-Critic architecture and it still remains an open question. Even though the

results are not conclusive in favour of the proposed algorithm they are not against either. Individual tests on various trees showed that the structure can be optimised further when the NNs are tuned.

In the research process a novel modification of an established algorithm, Levenberg-Marquardt, has been identified. The LM method is employed to optimise the Critic NN without the need of a learning rate, and the proposed modification exhibited superior results than the original algorithm. Among the major advantages of the modification was that it made possible for the Critic to "learn" the importance of trajectories of states rather than the value of a single state.

The capabilities of the GP methodology to multi-action control problems have been thoroughly examined. The dedication of one tree per action resulted in two distinct ways to apply the crossover. One that selects new individuals for each action and the one that selects only once individuals. The specific problem, which the GP was tested upon, was the helicopter hovering control problem, which is regarded as one of the hardest in the RL literature. The results were comparable to the winner of the RL-competition and in some cases (different wind patterns) better.

Finally, it has been demonstrated that the GP methodology can serve as a prediction system for highly non-linear processes, as those encountered in the food industry. The resulting predictor outperformed the MLP and the PLS method and described better the survival curve of the specific pathogen. The food industry offers a new set of problems in which Artificial Intelligence techniques have never been tested before (with the exception of MLP and some variations of it)

9.3 Future work

Although the results of GP on the Helicopter Hovering control problem were good in relation to the difficulty of the problem, it has been identified that two different possible crossover operations can be applied for multi-action problems. The one, which was adopted in the current research, assumes a new individual for every control and then enforces the crossover. The other possible approach is to select two individuals and apply crossover to all their respective controls. Extensive tests of the two kinds of crossover on different multi-action control problems would be needed in order to evaluate their merits and differences, if any.

The modified Actor-Critic with Levenberg-Marquardt showed an impressive improve-

ment in terms of the success rate for the triple inverted pendulum problem. But further tests, particularly those with multiple actions, should be carried out to establish the full potential of the algorithm. Especially it remains to be tested how the method reacts to noisy environments and its generalisation capabilities. Also an effort has to be made to fully replace gradient descent with Levenberg-Marquardt for the Action network. This would have substantial implications as it will possibly speed up the process and would also eliminate the need for a learning rate for the Action network as well.

In the same direction, but from a different approach, the open problem of adaptive learning rates for the classic Actor-Critic architecture has to be addressed. The implications are considerable as this is very likely to speed up the process of the usual gradient descent. Also it will safeguard against the process losing the local minimal due to overshoot caused from large learning rates.

The above work, proved successful, will automatically fill up the gap in the proposed GP by the augmentation of the function-set with NNs. It will thus be possible to eliminate the dysfunction of the optimisation process of the NNs, caused by the diversity of topologies which influences the optimal value of the learning rates. In these cases, extensive tests should be performed to evaluate the solving power of the algorithm, which already has provided indications that this is much higher than for the GP or NN alone.

Bibliography

[1] Microsoft azure. `https://azure.microsoft.com`.

[2] Symbolic differentiation. `https://www.codeproject.com/articles/13731/symbolic-differentiation#Psedo`.

[3] P. Abbeel, V. Ganapathi, and A. Ng. Learning vehicular dynamics, with application to modeling helicopters. *Advances in Neural Information Processing Systems*, 18:1, 2006.

[4] I Albert and P Mafart. A modified weibull model for bacterial inactivation. *International journal of food microbiology*, 100(1):197–211, 2005.

[5] Mahdi Amina, Vassilis S Kodogiannis, IP Petrounias, John N Lygouras, and G-JE Nychas. Identification of the listeria monocytogenes survival curves in uht whole milk utilising local linear wavelet neural networks. *Expert Systems with Applications*, 39(1):1435–1450, 2012.

[6] Mahdi Amina, EZ Panagou, VS Kodogiannis, and G-JE Nychas. Wavelet neural networks for modelling high pressure inactivation kinetics of listeria monocytogenes in uht whole milk. *Chemometrics and intelligent laboratory systems*, 103(2):170–183, 2010.

[7] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies-a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[8] Saumya Bhaduri, Phillip W Smith, Samuel A Palumbo, Carolyn O Turner-Jones, James L Smith, Benne S Marmer, Robert L Buchanan, Laura L Zaika, and Aaron C Williams. Thermal destruction oflisteria monocytogenes in liver sausage slurry. *Food Microbiology*, 8(1):75–78, 1991.

[9] Jens Busch, Jens Ziegler, Christian Aue, Andree Ross, Daniel Sawitzki, and Wolfgang Banzhaf. Automatic generation of control programs for walking robots using genetic programming. In *European Conference on Genetic Programming*, pages 258–267. Springer, 2002.

[10] Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement learning and dynamic programming using function approximators*, volume 39. CRC press, 2010.

[11] Oscar Castillo, Gabriel Huesca, and Fevrier Valdez. Evolutionary computing for optimizing type-2 fuzzy systems in intelligent control of non-linear dynamic plants. In *NAFIPS 2005-2005 Annual Meeting of the North American Fuzzy Information Processing Society*, pages 247–251. IEEE, 2005.

[12] Sheng Chen, ES Chng, and K Alkadhimi. Regularized orthogonal least squares algorithm for constructing radial basis function networks. *International Journal of Control*, 64(5):829–837, 1996.

[13] E.J. Gentry C.L. Karr. Fuzzy control of ph using genetic algorithms. In *Transactions on Fuzzy Systems*, volume 1, pages 46–54. IEEE, 1993.

[14] LS Crawford, VHL Cheng, and PK Menon. Synthesis of flight vehicle guidance and control laws using genetic search methods. *SYNTHESIS*, 1000:99453, 1999.

[15] Kenneth A De Jong. *Evolutionary computation: a unified approach*. MIT press, 2006.

[16] Dimitris C. Dracopoulos. *Evolutionary Learning Algorithms for Neural Adaptive Control*. Springer Verlag, August 1997.

[17] Dimitris C Dracopoulos and Dimitrios Effraimidis. Genetic programming for generalised helicopter hovering control. In *European Conference on Genetic Programming*, pages 25–36. Springer, 2012.

[18] Dimitris C. Dracopoulos and Riccardo Piccoli. Bioreactor control by genetic programming. In R. Schaefer et al., editor, *Parallel Problem Solving from Nature (PPSN) XI*, volume II of *LNCS 6239*, pages 181–188. Springer-Verlag Berlin Heidelberg, 2010.

[19] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[20] Ivica Kostanic Fredic Mh. Principles of neurocomputing for science & engineering, 2003.

[21] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[22] A. Gonzalez, R. Mahtani, M. Bejar, and A. Ollero. Control and stability analysis of an autonomous helicopter. In *Proceedings of World Automation Congress*, volume 15, pages 399–404, Jun-Jul 2004.

[23] Martin T Hagan and Mohammad B Menhaj. Training feedforward networks with the marquardt algorithm. *IEEE transactions on Neural Networks*, 5(6):989–993, 1994.

[24] Simon Haykin. *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA:, 2009.

[25] Francisco Herrera, Manuel Lozano, and Jose Luis Verdegay. A learning process for fuzzy control rules using genetic algorithms. *Fuzzy sets and systems*, 100(1):143–158, 1998.

[26] Alberto Herreros, Enrique Baeyens, and José R Perán. Design of pid-type controllers using multiobjective genetic algorithms. *ISA transactions*, 41(4):457–472, 2002.

[27] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* University of Michigan Press, 1975.

[28] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[29] KJ Hunt. Optimal control system synthesis with genetic algorithms. In *PPSN*, page 383, 1992.

[30] KJ Hunt. Polynomial LQG and H_∞ controller synthesis: A genetic algorithm solution. In *Decision and Control, 1992., Proceedings of the 31st IEEE Conference on*, pages 3604–3609. IEEE, 1992.

[31] Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

[32] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, pages 237–285, 1996.

[33] T.John Koo, Yi Ma, and S. Sastry. Nonlinear control of a helicopter based unmanned aerial vehicle. *IEEE Transactions on Control Systems Technology*, Jan 2001.

[34] R. Koppejan and S. Whiteson. Neuroevolutionary reinforcement learning for generalized helicopter control. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 145–152. ACM, 2009.

[35] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[36] John R Koza and Riccardo Poli. A genetic programming tutorial. *Introductory tutorials in optimization, search and decision support*, 8, 2003.

[37] K Krishnakumar and Davide Goldberg. Genetic algorithms in control system optimization. In *AIAA Guidance, Navigation and Control Conference, Portland, OR*, pages 1568–1577, 1990.

[38] Kristinn Kristinsson and Guy A Dumont. System identification and control using genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(5):1033–1046, 1992.

[39] Yvan Le Marc, Carmen Pin, and József Baranyi. Methods to determine the growth domain in a multidimensional environmental space. *International journal of food microbiology*, 100(1):3–12, 2005.

[40] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 2(2):164–168, 1944.

[41] M Anthony Lewis, Andrew H Fagg, and Alan Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2618–2623. IEEE, 1992.

[42] CL Little, FC Taylor, SK Sagoo, IA Gillespie, K Grant, and J McLauchlin. Prevalence and level of listeria monocytogenes and other listeria species in retail pre-packaged mixed vegetable salads in the uk. *Food microbiology*, 24(7):711–717, 2007.

[43] Donald W Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[44] Miguel Martınez, Juan S Senent, and Xavier Blasco. Generalized predictive control using genetic algorithms (gagpc). *Engineering applications of artificial intelligence*, 11(3):355–367, 1998.

[45] Zbigniew Michalewicz, Cezary Z Janikow, and Jacek B Krawczyk. A modified genetic algorithm for optimal control problems. *Computers & Mathematics with Applications*, 23(12):83–94, 1992.

[46] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[47] Kumpati S Narendra and Kannan Parthasarathy. Identification and control of dynamical systems using neural networks. *IEEE Transactions on neural networks*, 1(1):4–27, 1990.

[48] Oliver Nelles. Nonlinear system identification, 2002.

[49] A.Y. Ng, H.J. Kim, M.I. Jordan, S. Sastry, and S. Ballianda. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems*, 2004.

[50] Zhen Ni, Haibo He, Danil V Prokhorov, and Jian Fu. An online actor-critic learning approach with levenberg-marquardt algorithm. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 2333–2340. IEEE, 2011.

[51] DA Nolan, DC Chamblin, and JA Troller. Minimal water activity levels for growth and survival of listeria monocytogenes and listeria innocua. *International journal of food microbiology*, 16(4):323–335, 1992.

[52] Peter Nordin and Wolfgang Banzhaf. Genetic programming controlling a miniature robot. In *Working Notes for the AAAI Symposium on Genetic Programming*, volume 61, page 67. MIT, Cambridge, MA, USA, AAAI, 1995.

[53] Peter Nordin and Wolfgang Banzhaf. An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behavior*, 5(2):107–140, 1997.

[54] C Onnen, R Babuška, U Kaymak, JM Sousa, HB Verbruggen, and R Isermann. Genetic algorithms for optimization in predictive control. *Control Engineering Practice*, 5(10):1363–1372, 1997.

[55] EZ Panagou and VS Kodogiannis. Application of neural networks as a non-linear modelling technique in food mycology. *Expert Systems with Applications*, 36(1):121–131, 2009.

[56] PC Panchariya, AK Palit, D Popovic, and AL Sharrna. Nonlinear system identification using takagi-sugeno type neuro-fuzzy model. In *Intelligent Systems, 2004. Proceedings. 2004 2nd International IEEE Conference*, volume 1, pages 76–81. IEEE, 2004.

[57] Micha Peleg and Martin B Cole. Reinterpretation of microbial survival curves. *Critical Reviews in Food Science*, 38(5):353–380, 1998.

[58] Gustavo Pessin, Fernando Osório, Alberto Y Hata, and Denis F Wolf. Intelligent control and evolutionary strategies applied to multirobotic systems. In *Industrial Technology (ICIT), 2010 IEEE International Conference on*, pages 1427–1432. IEEE, 2010.

[59] Chad Phillips, Charles L Karr, and Greg Walker. Helicopter flight control with fuzzy logic and genetic algorithms. *Engineering Applications of Artificial Intelligence*, 9(2):175–184, 1996.

[60] Ananth Ranganathan. The levenberg-marquardt algorithm. *Tutoral on LM algorithm*, 11(1):101–110, 2004.

[61] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference On*, pages 586–591. IEEE, 1993.

[62] Reinforcement learning competition. http://www.rl-competition.org, 2009.

[63] T Ross. Indices for performance evaluation of predictive models in food microbiology. *Journal of Applied Bacteriology*, 81(5):501–508, 1996.

[64] Haralambos Sarimveis and George Bafas. Fuzzy model predictive control of non-linear processes using genetic algorithms. *Fuzzy sets and systems*, 139(1):59–80, 2003.

[65] Jennie Si, Andrew G. Barto, Warren B. Powell, and Donald Wunch II, editors. *Handbook of Learning and Approximate Dynamic Programming*. Wiley, 2004.

[66] Jennie Si and Yu-Tsung Wang. Online learning control by association and reinforcement. *IEEE Transactions on Neural Networks*, 12(2):264–276, 2001.

[67] RK Simpson and A Gilmour. The effect of high hydrostatic pressure on listeria monocytogenes in phosphate-buffered saline and model food systems. *Journal of Applied Microbiology*, 83(2):181–188, 1997.

[68] NG Stoforos and PS Taoukis. Pressure process evaluation through kinetic modelling. In *Proceedings of the eight international congress on engineering and food*, pages 1437–1441, 2001.

[69] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[70] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[71] GD Sweriduk, PK Menon, and ML Steinberg. Design of a pilot-activated recovery system using genetic search methods. Technical report, DTIC Document, 1999.

[72] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann Publishers, Inc., 1989.

[73] Csaba Szepesvári. Algorithms for reinforcement learning. *Morgan and Claypool*, 2009.

[74] J Antonio Torres and Gonzalo Velazquez. Commercial opportunities and research challenges in the high pressure processing of foods. *Journal of Food Engineering*, 67(1):95–112, 2005.

[75] Martinus AJS van Boekel. On the use of the weibull model to describe thermal inactivation of microbial vegetative cells. *International journal of food microbiology*, 74(1):139–159, 2002.

[76] P. J. Werbos. Foreword - ADP: The key direction for future research in intelligent control and understanding brain intelligence. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 38(4):898 –900, aug. 2008.

[77] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974.

[78] RC Whiting, S Sackitey, S Calderone, K Morely, and JG Phillips. Model for the survival of staphylococcus aureus in nongrowth environments. *International Journal of Food Microbiology*, 31(1):231–243, 1996.

[79] Richard C Whiting. Modeling bacterial survival in unfavorable environments. *Journal of Industrial Microbiology*, 12(3-5):240–246, 1993.

[80] Richard C Whiting. Microbial modeling in foods. *Critical Reviews in Food Science & Nutrition*, 35(6):467–494, 1995.

[81] Marco Wiering and Martijn van Otterlo, editors. *Reinforcement Learning: State-of-the-Art*. Springer Verlag, 2012.

[82] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[83] MH Zwietering, Il Jongenburger, FM Rombouts, and K Van't Riet. Modeling of the bacterial growth curve. *Applied and environmental microbiology*, 56(6):1875–1881, 1990.

Appendix A

Agents

A.1 Helicopter Hovering Control Problem

The best individual derived through GP for which all results are presented in the Chapter **??** is shown (unedited) below:

α_1 - `aileron`:

(+ 0.02 (* -0.0367 v)(+ (+ (* (* -0.0196 y)(+ (* -0.0196 y)(+ (* -0.7475 -0.0196)(* -0.0367 v)(+ (* -0.0196 v)(+ 0.02 (* -0.0196 v)(+ (* -0.0196 y)(* -0.0367 v)(+ (* -0.0196 (* phi v))(+ -0.0196 (* -0.0196 y)(* -0.0367 v)0.02)(* -0.7475 phi)0.02)0.02)0.02)(* -0.7475 phi)0.02)0.02)(+ -0.0367 (+ (* 0.02 -0.0196)(* -0.7475 v)(* (* -0.0367 v)(* -0.0196 -0.0367))0.02)(* -0.7475 phi)0.02)0.02))(* -0.0367 (* -0.0196 y))(* -0.7475 -0.0196)0.02)(+ -0.0196 (* -0.0196 v)(+ (* -0.7475 -0.0196)(+ (* (+ (* -0.0196 y)(* -0.0367 v)(* -0.7475 0.02)0.02)y)(* -0.0367 v)(* -0.7475 (* -0.0196 -0.0367))0.02)-0.0196 0.02)(+ (* -0.0196 y)(* -0.0367 v)(* -0.7475 (+ (* -0.0196 (* -0.7475 phi))(* (* -0.0367 v)v)-0.0196 0.02))0.02))(+ (* (* -0.0367 -0.0196)(+ (* (* -0.0367 0.02)y)(* -0.0367 v)(* -0.7475 (* -0.0196 -0.0367))0.02))(+ (* 0.02 y)(* -0.0367 v)(* -0.7475 (* phi v))0.02)(* -0.7475 phi)(* -0.0196 y))(+ 0.02 (* -0.0367 -0.0196)(+ (+ (* -0.0196 -0.0196)(* 0.02 v)(* -0.7475 -0.0367)(* -0.0196 y))(* 0.02 -0.0196)(+ (* (+ (* -0.0196 y)(* (* 0.02 -0.0196)v)(* -0.7475 -0.0196)0.02)y)(+ (* -0.7475 y)(* -0.0367 v)(* -0.7475 (* -0.0196 -0.0367))-0.0367)(* (* (+ (* -0.0196 y)(+ 0.02 (* -0.0367 v)(+ (* -0.0196 v)v (* -0.7475 phi)(* 0.02 y))0.02)(+ (* (+ (* 0.02 -0.7475)(* -0.0367 v)(+ (* -0.0196 y)(+ (* -0.0196 y)(* -0.0196 0.02)(* phi (* (* -0.0367 v)v))0.02)(* (* -0.7475 (* (* -0.0367 v)v))phi)0.02)0.02)y)(+ (* -0.0196 y)(* (* -0.0196 -0.0367)v)(* (* -0.0367 v)(* -0.0196 -0.0367))0.02)(* -0.7475 phi)0.02)0.02)(* -0.0196 -0.0367))phi)0.02)0.02)(+ (* -0.0196 v)(+ 0.02 (* -0.0367 v)(+ (* -0.0196 v)(+ 0.02 (* -0.0196 v)(+ (* -0.0196 y)(* -0.0367 v)(+ (* -0.0196 (* (* -0.0367 v)(* -0.0367 v)))(+ (* -0.0196 y)(* -0.0367 v)(* -0.0367 v)0.02)(* -0.7475 phi)0.02)0.02)0.02)(* -0.7475 phi)0.02)0.02)(+ (* 0.02 y)(+ (* -0.0196 y)(* 0.02 v)(* (* -0.0367 (* -0.0196 y))(* -0.0196 -0.0367))0.02)(* -0.7475 phi)(* -0.0367 y))0.02)))(+ 0.02 (* -0.0367 y)(+ (* (* -0.7475 v)0.02)(+ 0.02 (* -0.0367 v)(* -0.7475 (* (* -0.0367 v)-0.0367))0.02)(* -0.7475 phi)0.02)(* -0.0196 -0.0367)))

α_2 - `elevator`:

(+ (* -0.0185 x)(* -0.0322 u)(+ (* -0.0185 (* -0.0185 (+ (* -0.0185 x)(* -0.0322 u)(* 0.7904 theta)0.0)))(+ (* -0.0322 x)(* -0.0185 x)(+ (* 0.7904 (* -0.0322 theta))(* -0.0322 -0.0185)(+ (* -0.0185 x)(* -0.0322 u)(+ (* -0.0185 x)(+ (+ (* -0.0185 x)(* -0.0185 x)(+ (* x x)(+ (* -0.0322 x)(* (* 0.7904 (* -0.0322 u))(* u x))(+ (* (* -0.0322 u)(* x theta))(* -0.0322 u)(+ (* -0.0185 x)(* -0.0322 u)theta (* -0.0185 -0.0185))0.0)(+ (* -0.0185 (+ (* 0.7904 x)(* (* -0.0185 u)u)(* 0.7904 theta)(+ (* -0.0185 x)-0.0185 (* -0.0322 u)-0.0322)))(* -0.0322 u)(* 0.7904 theta)(* -0.0185 u)))(+ (* -0.0185 (+ (* u x)(* -0.0185 u)(* 0.7904 theta)0.0))(* -0.0322 u)(* x theta)(* -0.0185 x))(* -0.0185 x))(* x -0.0322))(* (* 0.7904 -0.0322)u)(* -0.0322 u)(* (* 0.7904 (+ (* -0.0185 x)(* -0.0185 x)(* (+ (* (* -0.0322 (* -0.0185 u))(* -0.0322 theta))(* u u)(+ (* -0.0185 x)(* -0.0322 u)theta (* -0.0185 x))0.0)(* (* -0.0322 u)u))(+ theta (* -0.0322 u)(* 0.7904 theta)(* (* -0.0185 x)x))))x))(+ (* (* -0.0185 x)(+ (* -0.0185 x)(* (* -0.0185 (* -0.0185 x))-0.0185)(* -0.0185 theta)0.0))(* -0.0322 u)(+ (* -0.0185 x)(* -0.0322 u)(* 0.7904 theta)(* -0.0185 x))(* -0.0185 x))(* (* 0.7904 (+ (* -0.0185 x)(* -0.0185 x)(* 0.7904 x)(+ (* -0.0185 x)(* -0.0322 u)(* 0.7904 theta)(* (* -0.0185 x)x))))x))theta)(* -0.0185 x))(+ (* -0.0185 (+ (* -0.0185 x)(* -0.0322 u)(* 0.7904 theta)(+ (* -0.0185 x)(* (* -0.0322 u)(* 0.7904 u))(* x -0.0322)-0.0322)))(* -0.0322 u)(+ (* -0.0185 x)(+ (+ (* -0.0185 x)(* -0.0185 x)(+ (* -0.0185 x)(+ (* -0.0322 x)(* (* 0.7904 (* u u))u)(+ (* (* -0.0185 x)(* -0.0322 theta))(* -0.0322 u)(+ (* -0.0185 x)(* -0.0322 u)theta (* -0.0185 x))0.0)(+ (* -0.0185 (+ (* -0.0185 x)(* (+ (* -0.0185 x)(* -0.0322 u)(* 0.7904 theta)(* -0.0185 x))(* -0.0185 x))(* 0.7904 theta)(+ (* -0.0185 x)(* (* 0.7904 (* -0.0322 x))-0.0185)(* -0.0322 u)-0.0322)))(* -0.0322 -0.0185)(* 0.7904 theta)(* (* -0.0322 u)-0.0322)))(+ (* -0.0185 (+ (* u x)(* -0.0185 u)(* 0.7904 theta)0.0))(* -0.0185 x)(* x theta)(* -0.0185 x))(* x x))(* (* -0.0185 -0.0185)theta))(* u u)(* x u)(+ (* -0.0185 (* u x))(* (* x theta)u)(* 0.7904 (* (* -0.0185 theta)u))(* -0.0185 u)))(+ (* -0.0185 theta)(* -0.0322 u)(* x x)(* 0.0 x))(* -0.0185 x))(* -0.0185 (* -0.0322 u))))(+ (* (* -0.0185 x)(+ (* -0.0185 x)(* -0.0322 u)-0.0185 0.0))(* -0.0322 u)(* x theta)(* -0.0185 x))(* -0.0185 x))(* -0.0185 x))

α_3 - `rudder`:

(* (/ (* (CUB (ABS omega))u u u)(* (ABS x)(ABS x)theta omega))(* (CUB (ABS w))(* x theta (CUB u))(* (ABS (/ (CUB w)(ABS x)))(ABS (CUB (ABS x)))(* u u w w))theta)u)

α_4 - `coll`:

(+ z (* (+ (* 0.0513 (+ (+ (* 0.1348 w)(+ (+ 0.1348 (+ z (* z z)w)(+ (* (* z 0.1348)(* (+ (* 0.0513 z)(* 0.1348 w)0.1348)w))(* 0.1348 w)w))(+ (* 0.1348 z)z (+ z (* 0.1348 z)(+ (+ z (* z w)(* w w))0.1348 (+ 0.0513 (* 0.1348 w)w))))(+ (* 0.0513 z)(* (* 0.1348 0.1348)z)(* z (+ z 0.1348 (+ z (* 0.0513 (* 0.1348 z))z)))))(+ 0.1348 0.0513 0.0513))(* z z)w))(* 0.1348 (+ (* 0.1348 z)(* 0.23 w)(* z 0.1348)))(+ 0.1348 0.1348 0.0513))w)0.23)

A.2 Implementation

The most essential Java class that stores and manipulates the topology of the NN in a tree is the following:

```
package tree;

import functions.*;
import genetic.GP;
import genetic.GP.Union;
import java.io.Serializable;
import java.util.ArrayList;

public class MapNeuNets implements Serializable {

  private Tree t;

  public ArrayList<Integer> NeuNetsId = new ArrayList<Integer>();
  public ArrayList<ConnectInfo> connections = new ArrayList<ConnectInfo>();

  public MapNeuNets(Tree t) {
    this.t = t;
  }

  public void constructMap() {

    for (Integer i : NeuNetsId) {
      ConnectInfo ci = new ConnectInfo();

      ci.startNNid = i;
      TreeNode startNode = t.findTreeNode(i);
      findNextNN(startNode, ci);

      int[] init = new int[1];
      constructFunction(startNode, ci, "", init);

      connections.add(ci);
    }

  }

  private void findNextNN(TreeNode node, ConnectInfo info) {
    TreeNode parent = node.getParent();

    if (parent == null) {
      info.nextNNid = -1;
      return;
    }

    if (parent.element == GP.Union.NeuNet) {
      info.nextNNid = parent.id;
    } else {
```

```
      findNextNN(parent, info);
    }
  }

  private void constructFunction(TreeNode node, ConnectInfo info, String rtn, int[] cf) {

    if (info.nextNNid == node.id || node.getParent() == null) {
      info.function = rtn;
      return;
    }

    if (node.getParent().element == Union.ADD || node.getParent().element == Union.MUL) {

      if (info.startNNid == node.id) {
        rtn = "(x";

        for (TreeNode srch : node.getParent().getClildren()) {
          if (srch == node) {
            continue;
          } else {
            rtn = rtn + funcs(node.getParent().element);

            rtn = rtn + "coef_" + cf[0]++ + "_";
            info.coef.add(srch);
          }
        }

      } else {
        rtn = "(" + rtn;

        for (TreeNode srch : node.getParent().getClildren()) {
          if (srch == node) {
            continue;
          } else {
            rtn = rtn + funcs(node.getParent().element);

            rtn = rtn + "coef_" + cf[0]++ + "_";
            info.coef.add(srch);
          }
        }

      }
      rtn = rtn + ")";

    } else if (node.getParent().element == Union.DIV || node.getParent().element == Union.SUB) {

      if (info.startNNid == node.id) {
        if (node.getParent().getClildren().get(0) == node) {
          rtn = "(x" + funcs(node.getParent().element) + "coef_" + cf[0]++ + "_" + ")";
          info.coef.add(node.getParent().getClildren().get(1));
        } else if (node.getParent().getClildren().get(1) == node) {
          rtn = "(" + "coef_" + cf[0]++ + "_" + funcs(node.getParent().element) + "x)";
```

```
          info.coef.add(node.getParent().getClildren().get(0));
        }
      } else {
        if (node.getParent().getClildren().get(0) == node) {
          rtn = "(" + rtn + funcs(node.getParent().element) + "coef_" + cf[0]++ + "_" + ")";
          info.coef.add(node.getParent().getClildren().get(1));
        } else if (node.getParent().getClildren().get(1) == node) {
          rtn = "(" + "coef_" + cf[0]++ + "_" + funcs(node.getParent().element) + rtn + ")";
          info.coef.add(node.getParent().getClildren().get(0));
        }
      }

    } else if (node.getParent().element == Union.CUB || node.getParent().element == Union.SQ) {

      if (info.startNNid == node.id) {
        rtn = "(x)" + funcs(node.getParent().element);
      } else {
        rtn = "(" + rtn + ")" + funcs(node.getParent().element);
      }
    } else if (node.getParent().element == Union.EXP) {
      if (info.startNNid == node.id) {
        rtn = funcs(node.getParent().element) + "(x)";
      } else {
        rtn = funcs(node.getParent().element) + "(" + rtn + ")";
      }
    } else {

      if (info.startNNid == node.id) {
        rtn = funcs(node.getParent().element) + "(x)";
      } else {
        rtn = funcs(node.getParent().element) + "(" + rtn + ")";
      }

    }
    constructFunction(node.getParent(), info, rtn, cf);
  }

  public void findNeuNets() {
    findNNs(t.getRoot());
  }

  private void findNNs(TreeNode node) {
    if (node.element == GP.Union.NeuNet) {
      NeuNetsId.add(node.id);
    }

    for (int k = 0; k < node.getNoArgs(); k++) {
      findNNs(node.getTreeNode(k));
    }
  }

  public String funcs(GP.Union f) {
```

```
    String rtn = "";

    switch (f) {
    case ADD:
      rtn = "+";
      break;
    case SUB:
      rtn = "-";
      break;
    case MUL:
      rtn = "*";
      break;
    case DIV:
      rtn = "/";
      break;
    case ABS:
      rtn = "abs";
      break;
    case SQRT:
      rtn = "sqrt";
      break;
    case SQ:
      rtn = "^2";
      break;
    case CUB:
      rtn = "^3";
      break;
    case SIG:
      rtn = "sign";
      break;
    case SIN:
      rtn = "sin";
      break;
    case COS:
      rtn = "cos";
      break;
    case EXP:
      rtn = "e^";
      break;
    }

    return rtn;
  }

  public String replaceCoef(ConnectInfo ci) {
    String rtn = ci.function;

    for (int i = 0; i < ci.coef.size(); i++) {
      int k = rtn.indexOf("coef_" + i + "_");

      rtn = rtn.substring(0, k) + String.format("%-1.20f", ((Element) ci.coef.get(i)).eval())
          + rtn.substring(k + ("coef_" + i + "_").length());
```

```
    }

    return rtn;
  }

  public String replaceCoef(ConnectInfo ci, boolean isolated, int isolatedNeuNet) {
    String rtn = ci.function;

    for (int i = 0; i < ci.coef.size(); i++) {
      int k = rtn.indexOf("coef_" + i + "_");

      rtn = rtn.substring(0, k)
          + String.format("%-1.20f", ((Element) ci.coef.get(i)).eval(isolated, isolatedNeuNet))
          + rtn.substring(k + ("coef_" + i + "_").length());
    }

    return rtn;
  }
}

class ConnectInfo implements Serializable {
  int startNNid;
  int nextNNid;
  String function;
  ArrayList<TreeNode> coef = new ArrayList<TreeNode>();
}
```