Optimization, Imitation and Innovation: Computational Intelligence and Games

Julian Togelius

A Thesis submitted for the degree of Doctor of Philosophy

Department of Computer Science

University of Essex

September 2007

Contents

Acknowledgements

First of all I want to thank those persons that had a direct influence on the contents of this thesis. These include first and foremost my supportive supervisor, Simon Lucas, but also the people whom I have collaborated with on various papers. In the order of number of co-authored papers, which partially coincides with their influence on my development as a researcher, these are Renzo De Nardi, Hugo Marques, Alberto Moraglio, Alexandros Agapitos, Richard Newcombe, Peter Burrow, Magdalena Kogutowska and Edgar Galvan Lopez. I have also benefited from many stimulating discussions on the nature of research and academia with Owen Holland and Marie Gustafsson.

As for the people who have supported and influenced me as a person during the last three years, these are too many to enumerate. A good way to sum it up is friends, family and Georgia. Thanks for the love, you'll have more of it back now that I'll be less preoccupied with finishing my PhD!

Of course, I am grateful for the scholarship from the University of Essex and EPSRC for making all this possible in the first place.

Abstract

This thesis concerns the application of computational intelligence techniques, mainly neural networks and evolutionary computation, to computer games. This research has three parallel and non-exclusive goals: to develop ways of testing machine learning algorithms, to augment the entertainment value of computer games, and to study the conditions under which complex general intelligence can evolve. Each of these goals is discussed at some length, and the research described is also discussed in the light of current open questions in computational intelligence in general and evolutionary robotics in particular.

A number of experiments are presented, divided into three chapters: optimization, imitation and innovation. The experiments in the optimization chapter deals with optimizing certain aspects of computer games using unambiguous fitness measures and evolutionary algorithms or other reinforcement learning algorithms. In the imitation chapter, supervised learning techniques are used to imitate aspects of behaviour or dynamics. Finally, the innovation chapter provides examples of using evolutionary algorithms not as pure optimizers, but rather as innovating new behaviour or structures using complex, nontrivial fitness measures.

Most of the experiments in this thesis are performed in one of two games based on a simple car racing simulator, and one of the experiments extends this simulator to the control of a real-world radio-controlled model car. The other games that are used as experimental environments are a helicopter simulation game and the multi-agent foraging game Cellz.

Among the main achievements of the thesis are a method for personalised content creation based on modelling player behaviour and evolving new game content (such as racing tracks), a method for evolving control for non-recoverable robots (such as racing cars) using multiple models, and a method for multi-population competitive co-evolution.

Chapter 1

Introduction

Once upon a time, it was widely believed that we could build artificial intelligence ourselves. In other words, that we could understand the mechanisms of intelligence in sufficient depth and detail to be able to write a program (or construct a machine) that was intelligent. Some people still believe this; it's a perfectly respectable academic opinion.

I don't believe we can do this, at least not within the foreseeable future. There is just too much we don't know about intelligence. But I think we will have more success with building systems that learn how to be intelligent. That is what this thesis about: how to construct systems that learn how to be intelligent. More specifically, how to write software that learns to be intelligent within computer games. My two main objectives are to show how computer games can be used to further our understanding of how to create systems that learn to be intelligent, and how systems that learn to be intelligent can be useful in the construction of computer games.

1.1 Main scientific contributions

The background chapters of this thesis can come across as somewhat eclectic, and the experimental chapters describe more than a dozen groups of experiments (though the more central experiments are described in more detail than the others; this will be discussed in the next section). Therefore, it is important that I describe what I consider to be the main scientific contributions of the thesis. They can be divided into a general theoretic framework and a few specific experiments.

The general theoretic framework is the taxonomy of three different approaches to computational intelligence and games, and the discussion about the potential for CI in games and games in CI inherent in each approach, in chapter 3. Most current research concerning CI and games fail to appreciate the nature of computer games, and focus on using them as testbeds for

reinforcement learning algorithms in a rather simplistic way. I point out several other ways in which CI can be used for games, and games for CI, within a structured framework. In doing this I draw not only on the CI literature but also on research findings in e.g. paleobiology and game development.

While I consider all the experiments described in this thesis to be scientifically sound and of at least some interest, the four sets of experiments which I claim to be the most important contributions in this thesis are the following:

- The incremental evolution of general and specific driving skills in section 5.2. There, we show that it there is such a thing as general driving skills in our car racing task, that this is something different from being able to drive a single track, and that general driving skills can be evolved through incremental evolution. We also show that general controllers can be specialized through further evolution, and thereby obtain higher fitness on particular tracks than controllers evolved directly for those tracks; for some tracks the difference is that between being able to drive the track at all or not. This technique can both have applications in computer games and represent some small progress towards a method for evolving truly complex general behaviour.

- The evolutionary personalised content creation experiments described in sections 6.1 and 7.3. The main invention here lies in the combination of modelling of human behaviour, evolution of controllers to match the modelled behaviour, and evolution of game content (in this case racing tracks) with entertainment metrics as fitness functions. These experiments combine several computational intelligence techniques in a novel way, and opens up for new applications of computational intelligence in games.

- The modelling of the dynamics of a radio-controlled car, and subsequent evolution of controllers in simulation which can then be used to control the real car, described in section 6.2. Like the content generation experiments, this experiment combines supervised learning with evolutionary computation in an at least partly novel way. A crucial invention needed to make this work was the use of several different models, acquired using different learning algorithms, during controller evolution to cancel out exploitable weaknesses of the models. With this approach, we believe to be the first to have used automated modelling techniques to evolve transferable controllers for a fast, dynamic, "non-recoverable" robot.

- The use of "diffuse" competitive co-evolution, where more than two populations are used, to compare different controller architectures, and to evolve more complex general behaviour than would be possible with single-population approaches, described in section 7.2. While diffuse competitive co-evolution has been used by a few researchers before (unbeknownst

to us at the time of conducting the experiments), it seems never to have been used with different controller architectures in the different populations. Some of the experiment results were in line with predictions, such as the superior performance of the multi-population-evolved controllers, while some were unexpected and intriguing, especially the effect where faster-learning representations reach lower eventual fitness.

1.2 Organization of the thesis

The first four chapters of this thesis, including this chapter, are background chapters discussing issues techniques, problems, issues and theories that will be used and/or investigated in the experiment chapters. Chapters 5, 6 and 7 are experimental chapters, describing the motivations for each experiment, the experimental methods particular to those experiments, and the results of the experiment together with some discussion. The experimental chapters are topically organised, depending on which approach to computational intelligence and games is taken by the experiments in the chapter. A final chapter puts the results in context and discusses future directions for the research.

1.3 List of papers

This thesis is based on theoretical and experimental research which has previously been written up in 12 scientific papers, which have all have passed the peer-review process and been accepted for publication in the proceedings of various international conferences, symposia, and workshops with high academic standards. Much of the same research is also currently being written up or under review for archival publication in journals and books. This section lists the papers on which the thesis is based, along with in what section(s) of the experimental chapters the experiments in each paper is discussed. The theory, related work and methods of each paper is instead discussed mostly in the background chapters of the thesis.

One problem with basing a doctoral thesis on 12 papers is that the amount of research to describe threatens to make the thesis too long and lose focus. I have therefore selected five papers, both because they are some of the better papers out of the 12 and because they are central to the arguments of the thesis, to be discussed extensively. This means that the experiments in those papers will be described in rather more detail in the thesis than in the papers. The experiments in the other 9 papers will be discussed briefly, in about the level of detail of an extended abstract. For those papers, the published paper rather than the discussion in the thesis is to be considered the authoritative version. (All papers are available for downloading from my web site, http://julian.togelius.com.) However, note that most of the papers share at

least some theory, related work, and methods, and these will be discussed in the background chapters. These are the papers:

- Julian Togelius and Simon M. Lucas (2005): *Forcing neurocontrollers to exploit sensory symmetry through hard-wired modularity in the game of Cellz.* Proceedings of the IEEE Symposium on Computational Intelligence and Games, 37-43. Briefly discussed in section 5.4.2. [143]

- Julian Togelius and Simon M. Lucas (2005): Evolving Controllers for Simulated Car Racing. Proceedings of IEEE Congress on Evolutionary Computation, 1906-1913. Extensively discussed in section 5.1. [142]

- Renzo De Nardi, Julian Togelius, Owen Holland and Simon M. Lucas (2006): *Evolution of Neural Networks for Helicopter Control: Why Modularity Matters.* Proceedings of the IEEE Congress on Evolutionary Computation. Briefly discussed in section 5.4.1. [36]

- Julian Togelius and Simon M. Lucas (2006): *Evolving robust and specialized car racing skills.* Proceedings of the IEEE Congress on Evolutionary Computation. Extensively discussed in section 5.2. [145]

- Julian Togelius and Simon M. Lucas (2006): *Arms races and car races. Proceedings of Parallel Problem Solving from Nature.* Briefly discussed in section 7.1. [144]

- Julian Togelius, Renzo De Nardi and Simon M. Lucas (2006): *Making racing fun through player modeling and track evolution.* Proceedings of the SAB Workshop on Adaptive Approaches to Optimizing Player Satisfaction. Briefly discussed in sections 6.1 and 7.3. [139]

- Hugo Marques, Julian Togelius, Magdalena Kogutowska, Owen Holland and Simon M. Lucas (2007): *Sensorless but not Senseless: Prediction in Evolutionary Car Racing.* Proceedings of IEEE ALife. Briefly discussed in section 6.4.1. [83]

- Simon M. Lucas and Julian Togelius (2007): *Point-to-Point Car Racing: an Initial Study of Evolution Versus Temporal Difference Learning.* Proceedings of IEEE CIG. Briefly discussed in section 5.3.2. [80]

- Julian Togelius, Renzo De Nardi and Simon M. Lucas (2007): *Towards automatic personalised content creation for racing games.* Proceedings of IEEE CIG. Extensively discussed in sections 6.1 and 7.3. [140]

- Alexandros Agapitos, Julian Togelius and Simon M. Lucas (2007): *Evolving controllers for simulated car racing with object-oriented genetic programming.* To be presented at Gecco. Briefly discussed in section 5.3.1. [3]

- Julian Togelius, Renzo De Nardi, Hugo Marques, Richard Newcombe, Simon M. Lucas and Owen Holland (2007): *Nonlinear dynamics modelling for controller evolution.* To be presented at Gecco. Extensively discussed in section 6.2. [141]

- Julian Togelius, Peter Burrow and Simon M. Lucas (2007): Multi-population competitive co-evolution of car racing controllers. To be presented IEEE CEC. Extensively discussed in section 7.2. [138]

In addition, the following papers were published during my PhD, but are not included in this thesis in order to keep its length manageable:

- Alberto Moraglio, Julian Togelius and Simon M. Lucas (2006): *Product Geometric Crossover for the Sudoku Puzzle.* Proceedings of the IEEE Congress on Evolutionary Computation. [94]

- Alberto Moraglio and Julian Togelius (2007): *Geometric Particle Swarm Optimization for the Sudoku Puzzle.* To be presented at Gecco. [93]

- Alexandros Agapitos, Julian Togelius and Simon M. Lucas (2007): Multiobjective techniques for the use of state in genetic programming applied to simulated car racing. To be presented at the IEEE Congress on Evolutionary Computation. [4]

1.4 Notes on style

In this chapter, I use the word I to refer to myself and separate my views from those of others. In the subsequent chapters, I will use the word we. The main reasons for this is that much of the experimental work is done in collaboration with other researchers, as well as most of my background reasoning have been discussed with others, and that I don't want to switch between the two forms of personal pronoun throughout the thesis. At the very least, all of the research has been done under Simon's supervision. So I think it is justified to use the first person plural throughout the thesis, even though it should sometimes be construed as referring to the reader and the author (me and you), or perhaps as *pluralis majestatis*; it is in any case not to be taken as evidence of multiple personalities of the author.

Further, I consider it important that this thesis is reasonably easy to read, if possible it should even be pleasant to read. This has a number of minor consequences, such as the limited use of acronyms, and the occasional use of interchangeable terms. A more important consequence is that several procedures and lines of reasoning that other authors would have presented in the form of equations are here presented in plain English. I consider excessive use of mathematics

to be disastrous for readability, and I therefore present some of the material in mathematical form only when I am more or less forced to.

Chapter 2

Computational intelligence

This chapter surveys a range of computational intelligence methods that will be used in the experimental parts of this thesis, and the particular issues within those methods that will be explored in the experiments. The first section is on evolutionary algorithms, a broad class of algorithms mimicking natural evolution, that are used in at least some part of all the experiments in this thesis. The next section is on neural networks, a class of function representations inspired by biological nervous systems, that is again used (in some form) in almost all the experiments in this thesis. Evolutionary robotics, which is a field that uses evolutionary computation and (most often) neural networks for robot control, and which this thesis can be seen either as a contribution to or as closely related to, is the subject of the third section.

The three first sections are by necessity only shallow overviews, as the subject of each of them is a full-blown research field in its own right. For the final section, however, we go into some more depth on a number of issues that arise when using evolutionary computation to create intelligent behaviour. These issues, which are being actively researched within both the evolutionary computation and evolutionary robotics communities, are the role and representation of sensors and actuators, learning curves and incremental evolution, competitive co-evolution, modularity, different controller representations and stateful versus reactive control. Each of these issues is addressed and explored in at least one of the experiments in the thesis, most of them in several experiments. Hopefully, the research described in this thesis will also have contributed at least something to the understanding of each of the issues.

2.1 Evolutionary computation

Evolutionary computation is a label stuck to a number of algorithms inspired by Darwin's theory of biological evolution by natural selection, and also to the scientific field that studies these

algorithms. To what extent these algorithms actually reproduce the mechanisms of biological evolution varies wildly, but most computer scientists tend to disregard the biological interpretation of evolutionary algorithms and treat them simply as global optimizers. Seen from this perspective, evolutionary algorithms are enormously versatile, as they can optimize any problem as long as the solution can be expressed as a sequence of parameters, and the *fitness* (quality) of a candidate solution can be measured quickly and reliably. Compared to other optimization techniques, little or no domain knowledge is required, though better results can usually be achieved when infusing some domain knowledge[40].

To get a bit more concrete, a typical simple evolutionary algorithm works like this: first, a number of candidate solutions (alternatively called *chromosomes*, *genomes* or *individuals*) to the problem are generated randomly; the set of all of them is called the *population*. The evolutionary process then proceeds in generations. In each generation, all indivuals in the population are evaluated for how well they solve the problem at hand, and given a numerical fitness by a *fitness function*. Then, some sort of selection takes place, whereby the more fit individuals are kept and the less fit indivduals disposed of and replaced with new individuals. The new individuals are created either from copies of the existing good (more fit) individuals, or from *recombination* (crossover, blending together) of two or more good individuals. After that, *mutation* (small random changes) are applied to all or some individuals in the population, and the next generation starts. This process is repeated for a fixed number of generations, or until a certain predetermined fitness level is reached by some member of the population.

Why this process works is easy to see on a common-sense level, if we visualize the algorithm as moving the individuals through a multi-dimensional search space where each individual constitutes a point in space, and changing some value of an individual moves that point in the corresponding direction. (If the y-dimension of the space is taken to be the fitness of the individual, we talk about a *fitness landscape*) Bad individuals are continuously removed, good individuals are kept, and new individuals are generated based on making random changes to copies of the good individuals. Thus, the fitness of the best individual in the population can never decrease (except in the case of noisy fitness, when two consecutive evaluations of the same individual yield different fitness values, or when *elitism* is not used), and from the mutations better individuals are likely to emerge, if they can be found in the vicinity of the best individuals. As a result, the best individual of the population moves closer to a *local optimum* or *global optimum*, a point from which there are no fitter individuals nearby; this can be visualized as a peak in the fitness landscape. However, mathematically proving that this process works has turned out to be extremely difficult, and currently no general convergence proof of evolutionary algorithms exists.

The description of an evolutionary algorithm above is only intended to represent a typical algorithm, and in the vast and diverse field of evolutionary computation every single part of the mechanism described has been changed or replaced at some point. In their introduction to evolutionary computation, Eiben and Smith introduce a taxonomy of evolutionary algorithms based on how they vary in six important dimensions [40]: representation, fitness function, population, parent selection mechanism, varation operators (mutation and recombination), and survivor selection (replacement). However, such an extensive overview is outside the scope of this chapter. Instead, we will here discuss the main evolutionary algorithms that will be used throghout the dissertation.

2.1.1 Evolution strategies

Evolution strategies (ESs) are particularly well-suited to optimizing problems represented as vectors of real numbers. The basic evolution strategy was introduced in the early 1960's by Ingo Rechenberg and Hans-Paul Schwefel, and was originally used for optimizing aircraft wings in wind tunnels[9]. Here we present the simple ES we use in many of the experiments in this thesis, and which forms the basis for the algorithms used in some other experiments.

1. At the start of a run, a population is created, containing $\mu + \lambda$ individuals. The first μ individuals are also called the *elite*, while the other λ indviduals are called the *tail*. Usually, we set $\mu = \lambda =$ a number between 15 and 50. Each individual is a fixed-length vector of real numbers, with all numbers initially set to small random values.

2. The population is sorted, in order from higher-scoring to lower scoring individuals.

3. The λ worst-scoring individuals (the tail) is removed from the population.

4. The removed individuals are replaced with copies of the μ first individuals (the elite). If $\mu = \lambda$ each individual in the elite is copied once.

5. Mutation is applied the λ new individuals in the tail. This means adding random numbers to all the real numbers in the vector constituting the mutated individual. These random numbers are drawn from a Gaussian distribution with mean 0 and some low standard deviation, typically 0.1. This standard deviation is also referred to as the *mutation magnitude*.

6. All individuals in the population are evaluated by the fitness function. If the fitness evaluations are noisy (i.e. the fitness for the same individual changes between evaluations due to random effects) each individual is evaluated several times (usually between 3 and 10) and the average of these values is used.

7. If the requisite number of generations have passed, or the highest fitness value in the population exceeds the desired fitness, the algorithm terminates and the most fit individual is returned as its results. Otherwise, the number of generations passed is increased by one, and the algorithm returns to step 2.

Note that no recombination is used, and that the mutation magnitude is constant throughout the run; self-adaptation is not used.

2.1.2 Genetic algorithms

Genetic algorithms (GAs) were introduced by John Holland in 1975 as a tool for biological modelling. The original version of genetic algorithms worked with individuals represented as strings of bits, and the variation operators worked on the level of flipping and exchanging bits; nowadays, however, genetic algorithms have been developed to work with a wide variety of representations. A defining feature of GAs is that they all use some sort of recombination, though the emphasis on recombination varies.

A typical simple GA works very much like the ESs discussed above, except for the inclusion of recombination:

1. At the start of a run, a population is created, containing n individuals. The first part of the population is referred to as the elite, and the second part as the tail; the proportion between these parts depend on the experimental setup, with the elite usually considerably smaller than the tail. The representation of the individuals is dependent on the problem at hand.

2. The population is sorted, in order from higher-scoring to lower scoring individuals.

3. The tail is removed from the population.

4. The removed individuals are replaced with new individuals. These new individuals are created by recombining two randomly chosen individuals in the population, and mutating the resulting individual. Exactly how the recombination and mutation works is dependent on the problem at hand.

5. All individuals in the population are evaluated by the fitness function. If the fitness evaluations are noisy each individual is evaluated several times and the average of these values is used.

6. If the requisite number of generations have passed, or the highest fitness value in the population exceeds the desired fitness, the algorithm terminates and the most fit individual

is returned as its results. Otherwise, the number of generations passed is increased by one, and the algorithm returns to step 2.

2.1.3 Cascading Elitism

In many cases, there is more than one fitness measures to maximize; a problem where this is the case is called a *multiobjective problem*. Evolutionary multiobjective optimisation is a rich and active research field, and it would take us too far from the topics of this theses to discuss that field here. However, in the experiments in sections 6.1 and 7.3 we have several different objectives, and needed a way of handling this. We invented a very simple solution to this, namely an evolution strategy with multiple elites. In the case of three fitness measures, it works as follows: out of a population of 100, the best 50 genomes are selected according to fitness measure f_1. From these 50, the 30 best according to fitness measure f_2 are selected, and finally the best 20 according to fitness measure f_3 are selected. Then these 20 individuals are copied four times each to replenish the 80 genomes that were selected against, and finally the newly copied genomes are mutated.

This algorithm, which we call Cascading Elitism, is inspired by an experiment by Jirenhed et al. [68]. We have not analyzed the effects of this algorithm in detail, nor have we compared it to "proper" multiobjective optimization algorithms, we have merely established that it works for the particular problem instances we developed it for.

In both of the cases where multiple fitness functions are used in this thesis, the component fitnesses are listed as f_1, f_2 and f_3. In all other cases, only a single fitness function is used. In most cases this fitness function is so simply defined that it is simply described as text. The exception is the helicopter evolution experiments, where an equation is given.

2.2 Neural networks

Neural networks are mathematical structures that are, to varying degrees, inspired by the biological nervous systems. There is probably more published academic papers concerning neural networks than anyone could read in a lifetime, and the variety of different types of networks and learning algorithms is dizzying. For this reason we will here limit the discussion to the particular class of neural networks that is used in the experiments in this thesis, namely the multi-layer perceptron (MLP) and some variations and derivations thereof. We will also limit our perspective on these networks to seeing them as function approximators, and not as models of biological nervous systems. These are arguably uncontroversial decisions, as the MLP is almost certainly the most common type of neural network used in applied research, and is only

distantly related to the networks used for modelling in neuroscience.

Modern introductions to neural networks from pattern recognition and machine learning perspectives can be found in [10] and [91].

2.2.1 Multi-layer perceptrons

The MLP, together with the backpropagation learning algorithm, was popularized in 1986 by Rumelhart and McClelland [114]. It can be seen as a generalization of the perceptron, a simple learning structure first described by McCulloch and Pitts in 1940 [85]. Several theoreticians have proved that a three-layer MLP can be made to approximate any numerical function, as far as the requisite number of hidden neurons are present (see [10] for details). While these results are certainly interesting and cause for cautious optimism, the practical feasibility of training an MLP to approximate a particular function is another matter. Some of the algorithms used and issues faced in this generally non-trivial task will be discussed in this chapter.

The basic abstractions in a MLP are the *neuron* and the *connection*. The neurons are organized in layers; usually three layers, called the *input layer*, the *hidden layer* and the *output layer* respectively. Connections are directional, and in a standard fully-connected MLP (in this thesis we assume all networks are fully connected unless another structure (*topology*) is explicitly given) all the neurons in the input layer are connected one-way to all the neurons in the hidden layer, and all the neurons in the hidden layer are connected one-way to all the neurons in the output layer. This means that a MLP has $i * h + h * o$ connections, where i, h, and o are the number of neurons in the input, hidden and output layer respectively.

Neurons and connections contain one numerical value each, but the connections act as constants and the neurons as variables. This means that when a neural network is used for approximating a function (propagating a vector of values), the values in/of the neurons change but not those of the connections. When training the network, however, the connection values (also called *weights*) change. It also means that the functioning of a (fully connected) MLP is completely specified by its connection weights.

Propagation

So, how does a multi-layer perceptron actually work - what does it do? It works like this. First the input to the network is placed in the input layer, one value in each neuron (the input has to be a vector with the same dimensionality as the length of the input layer). Then, the value of each neuron in the hidden layer is calculated as the sum of all the contributions to that neuron from the input neurons. The contribution from each input neuron is defined as the value of that neuron multiplied by the weight of the connection from that particular input neuron to that

particular hidden neuron. After the summing up, a *transfer function* or *squashing function* is applied to the value of each neuron, introducing the nonlinearity that is crucial for the universal computational power of the MLP. In the networks in this thesis, the hyperbolic tangent (*tanh*) transfer function is used. Differently expressed, the activation of the hidden neuron becomes:

$$h_j = tanh\left(\sum_{i=1}^{n}(i_i * w_{ij})\right) \tag{2.1}$$

The propagation from hidden layer to output layer works in exactly the same way. It is worth noting that the layers don't have to be of the same sizes at all; 5 inputs, 3 hidden neurons and 176 outputs is a perfectly possible (though improbable) MLP topology. (In most descriptions of the MLP, it is stipulated that there is an extra *bias* connection to every neuron in every layer from a constant input with the value 1, in order to achieve full computational power. (Otherwise, an input of all zeroes could not result in a non-zero output.) In our implementation, we are instead fixing one of the inputs to 1 in all experiments.)

Backpropagation

That is how a MLP works once its connection weights are set. To train MLPs, a number of different algorithms can be used. The next section describes how evolutionary computation is used to train networks; here, we describe the backpropagation algorithm for *supervised learning* with multi-layer perceptrons. Supervised learning refers to approximating the function underlying a set of training data, which can be expressed a list of pairs of inputs and outputs. The training data could consist in various demographic figures (inputs) and average house prices (outputs) for a number of UK towns (data points), or, to take an example from the experiments in this thesis, prior state and control signal (inputs) and next state (outputs) over a few thoused samples taken from a motion capture system at 20 hz (data points). If there is enough data, the data is representative enough of the underlying function (not too much noise), the underlying function is not too complicated, and the network has the right size and topology, the backpropagation algorithm can usually train the network to reproduce the underlying function with good accuracy.

The basic idea in the backpropagation algorithm is to go through all the data points several times, and for each data point feed the input to the network, and compare the output of the network (the *prediction*) with the "real thing", the output that is part of the data point (the *target*). When the prediction differs from the target, the weights of the network are changed so as to better match the target. This is done by "descending the *error gradient*"; moving the value of each connection in the direction that minimises the error that that connection contributed to in the last forward propagation. To do this, we must calculate how much each connection

weight has contributed to the error and in what direction. More formally, the equation for the weight updates is the *delta rule*:

$$\Delta w_{ij} = \eta * \delta_j * y_i \tag{2.2}$$

Here, Δw_{ij} is the change in the weight of the connection from i to j, η is the learning rate (a small positive constant value), y_i is the input signal to node j (the signal that is being propagated form i to j). η is the local gradient, which if j is an output node is defined as:

$$\delta_j = a_j * (1 - a_j)(t_j - a_j) \tag{2.3}$$

Where a_j is the activation of output node j, and t_j is the target value for that output. If we are looking for the local gradient for a hidden node, it is:

$$\delta_j = a_j * (1 - a_j) \sum_{k \epsilon outputs} w_j k \delta_k \tag{2.4}$$

Where $w_j k$ is the weight of the connection from hidden neuron j to output k, and δ_k is the local gradient of that output.

What is outlined here are the central equations behind plain vanilla backpropagation. Even disregarding the myriad variations of and improvements on basic backpropagation, there are several technical considerations to be aware of in the the application of the algorithm which we will not discuss here; interested readers are referred to Bishop's textbook [10]. In this thesis, only the "orthodox" backpropagation algorithm will be used.

2.2.2 Evolutionary neural networks

Evolutionary neural networks is the result of the happy marriage of two excellent ideas. The name refers to the research field (and application practice) of using evolutionary algorithms to set the weights (and sometimes devise the topology) for neural networks, but it can also refer to the the very networks that are developed using this method. The practice of evolving neural networks is also called *neuroevolution*. A good but somewhat dated introduction to the field with a substantial bibliography can be found in [155].

Like backpropagation, neuroevolution can be used to train a network to approximate a function given a list of data points representing the inputs to and outputs from the target function. All that is needed for supervised learning is a fitness function that returns a fitness proportional to how close the predictions of the network is to the targets. (The relative merits of backpropagation and neuroevolution for supervised learning is a topic which we will not discuss

here.) However, evolutionary algorithms are not limited to supervised learning, but can also be used for *optimization*, *reinforcement learning* and *unsupervised learning*. Everything that is needed is a fitness function that somehow judges the performance of the network.

Many of the experiments in this thesis uses what could be considered one of the simplest form of evolutionary neural network, namely an MLP whose weights are tuned by an evolution strategy. It works like this:

1. At the start of a run, a population of $\mu + \lambda$ individuals is created, each individual being a vector of real numbers of the same dimensionality as the number of connections in the neural network. Each dimension in the vector will be interpreted as a weight in the network. (Remember that a fully-connected MLP is completely specified by its connection weights.)
2. The population is sorted.
3. The tail is removed from the population.
4. The removed individuals are replaced with copies of the the elite.
5. Gaussian mutation is applied to the new individuals.
6. Each individual in the population is translated into a neural network, which is then tested using the fitness function. The individual is assigned the fitness given to the neural network constructed from it.
7. End evolution if the time is nigh, or return to step 2.

Neural networks can of course be evolved using other types of evolutionary algorithms, such as genetic algorithms or particle swarm optimization. However, while mutation, especially Gaussian mutation, usually is an efficient variation operator for neural network weights, recombination between neural networks can sometimes have disastrous effects if not done right (many types of recombination are possible). This is because of the *competing conventions* problem: given that neural networks that behave identically can have very different internal weights (e.g. different hidden neurons can be used for connecting the same input and output in the same way), recombination between two fully functioning networks can easily result in an offspring which of very low fitness[1].

Several specialized evolutionary algorithms for evolving neural networks have been developed during the last decade. These include Moriarty's SANE, and Gomez' ESP and CoSyNE, which all rely on (different) forms of cooperative co-evolution to speed up of evolution of fixed-topology

[1] Literally a "worthless bastard"

networks. The scope of these methods is not yet established, but they have managed to provide substantial speed ups for some specific control and pattern recognition tasks [95][50][51].

Evolving topologies as well as weights

While the above methods all presuppose a fixed neural network topology and just evolve the weights of the neural network, other algorithms have been developed that evolve the topology of the network as well. The argument here is that the human designer knows neither what topology is best for *representing* the solution to a particular problem, nor what topology makes it easiest to *evolve* such a solution. (We will discuss how a network might be able to represent but not learn a particular function later on in this chapter.) Many researchers would just use a fully-connected feedforward network with about as many hidden neurons as inputs, but there is no guarantee at all that this is a good topology for a particular problem.

Algorithms for evolving both topologies and connection weights can roughly be divided into *indirect* and *direct* approaches (but see [127] for a more refined taxonomy, including an in-depth discussion of different types of indirect encodings). The indirect approaches are so called because they, taking clues from biology, distinguish between *genotype* and *phenotype*. The individuals that are evolved by the evolutionary algorithm, the genotypes, are not the neural networks in themselves but rather a procedural specification for how a neural network should be built. Each time an individual is evaluated, the genotype must go through a developmental process to yield its phenotype, the actual neural network which is then evaluated by the fitness function. Early examples of neuroevolution using indirect encodings include Kitano's algorithm based on matrix rewriting [69], and Gruau's Cellular Encoding, where the genotype is an expression tree, which is "executed" in the development phase to construct a neural network [54][53]. Gruau's research inspired many researchers to develop similar systems, such as Hornby et al. [61] and Kodjabachian and Meyer [70]. Other researchers have gone back to biology for more inspiration, and returned with ideas such as artificial cell division and movement [23][66] and genetic regulatory networks [12], creating encodings that require rather complicated processes to go from genotype to phenotype.

Whereas all of the above algorithms have been proven to be able to evolve neural networks for at least some problem, such indirect encoding schemes are very rarely seen in application-oriented research, or indeed in any research papers except the ones where they are first reported. Further, there is a conspicuous absence of really impressive controllers or pattern recognizers evolved using indirect encodings. The reason for this seems to be that such encodings don't work very well in practice. This becomes clear in those cases where head-to-head comparisons have been made between direct and indirect encodings [118]. It is important to remember that

any indirect encoding favours certain topologies over others, and might even make some parts of design space completely inaccessible. The task of creating a generally efficient indirect encoding is complex, interesting and unsolved.

Direct encoding approaches, however, can be made quite efficient. In these approaches, the genotype and the phenotype are the same; the variation operators of the evolutionary algorithm act directly on the topology and weights of the network. One direct encoding topology-evolving algorithm which has attracted some attention for working well in practice is Stanley's NEAT, NeuroEvolution of Augmenting Topologies [126][123]. NEAT is a rather complex algorithm, that uses protected subpopulations to preserve structural innovations until they become viable, and innovation tracking that matches specific connections to their homologic equivalents in other individuals in order to alleviate the competing conventions problem for crossover. The central idea, though, is *complexification*. At the start of an evolutionary run, all individuals are minimal neural networks, with only one connection from one neuron in the input array to one of the output neurons. The mutation operator then works by adding, disabling (removing) and splitting connections as well as changing the weights of existing connections. A connection can be added between any two neurons; when a connection is a split, a new neuron is formed in the middle of the old connection, connections are added from the start neuron of the old connection to the new neuron, and from the new neuron to the end neuron of the old connection. In this way, networks or arbitrary topology and complexity can be created. In empirical comparisons, NEAT has been shown to be very competitive with evolving the weights of fixed-topology neural networks, sometimes performing better but sometimes not performing as well [123][51].

Evolving the topologies of neural networks together with their weights is a very interesting idea and research direction, and topology evolution might very well be a necessary part of any generally efficient neuroevolution algorithm capable of handling truly complex problems. However, topology evolution is not a focus of this thesis, and in order to be able to concentrate on other issues all the neuroevolution experiments in the experimental chapters employ fixed topologies. This being said, further investigation and understanding of the issues explored by this thesis (especially incrementality and modularity) will no doubt help in the development of the next topology-evolving algorithm.

2.3 Evolutionary robotics

In evolutionary robotics, evolutionary algorithms are used to construct the controllers (usually based on neural networks), and sometimes the body morphologies, of real or simulated robots. The research field took its beginnings in the early nineties with the work of Harvey, Husbands

and collaborators in Sussex, and Nolfi, Floreano and collaborators in Italy. The book by the two latter is a good, but again somewhat dated, introduction to the field [102]. Their work, in their turn, owes much to Rodney Brooks' pioneering of behaviour-based robotics, which focused robotics research on controlling robots by combinations of simple reactive behaviours, rather than the complicated sense-plan-act loop driven by symbolic logic that had been the only paradigm in town until the mid-eighties [14].

The most popular experimental platform for evolutionary robotics is probably the Khepera robot, a small (fist-sized) circular robot with two wheels, and eight sensors that can act either as directional light sensors or infrared range-finder sensors [92]. The robot has limited onboard processing capacity, but can be connected via a wire to be controlled by a desktop computer, and can be equipped with a gripper and a rudimentary linear (one-dimensional) camera. Importantly, the robot is semi-holonomic, meaning that it can turn on the spot, and due to it's low weight it can stop and accelerate to full speed almost instantaneously.

Using configurations of one or several Kheperas or similar robots on tabletop arenas with walls and other obstacles, controllers were evolved for many simple tasks, including variations on wall following and phototaxis [56], "garbage collection" (picking up and moving red objects) [103], and simple life-time learning capacities [44][147]. Recently, many of the original evolutionary roboticists have focused on evolving collective behaviours in swarms of more advanced but still semi-holonomic miniature robots [39], whereas other researchers have focused on e.g. evolving morphologies [61], evolving controllers for legged robots [156], evolving robots capable of modelling their own bodies [11], and using evolutionary robotics methods for investigating problems in theoretical biology.

The above is by no means intended as a complete overview of the state of the art in evolutionary robotics - the space not permitting, and this thesis not mainly being about evolutionary robotics anyway. However, from an overview of the literature in the field several observations can be made. The most important of these is that so far, no truly intelligent complex intelligence has been evolved. The most complex tasks which evolved robots have been able to solve were solved with a combination of very simple reactive behaviours, and they usually seem easily solvable by standard engineering methods. Simply stated, evolutionary robotics fails to *scale up*. Given the rather obvious and grand promise of using the same method as nature did for creating complex intelligence, this can be quite puzzling - nature obviously succeeded, so why don't we?

A lesser, but related, observation is that the vast majority of evolutionary robotics experiments are done using robots that are more or less stable, can start or stop more or less instantaneously, and if they are wheeled, turn on the spot. This is a striking difference to most real-world

vehicles and animal bodies. Neither a car nor an airplane can stop instantaneously or turn on the spot, and when moving both require constant attention so as not to crash, in contrast to a Khepera, which will usually stop and just stand there if movement commands cease. Similarly, a human body requires a complicated chain of movement to get from one position (standing up, right arm stretched out) to another (sitting down, legs crossed, hands clenching each other), and requires a constant energy expenditure and stream of compensating movement commands to not just fall over. In contrast, a Khepera's gripper takes an integer between 0 and 255 as input, promptly moves to the desired position and stays there until a new command is received. Even though other researchers have used more complex robot morphologies, they are very rarely as complex as human bodies or as dynamic as an airplane or fast car. Given the importance many philosophers of cognitive of science place on the interplay of the nervous system with a complex body for the emergence of intelligence, this is rather notable [86][26][97][107].

A third observation is that the dimensionality of the input space is generally low. A Khepera has 8 (or 16) sensors, and this is usually the length of the array of inputs to the neural network. Even when using more complex robots it is rare that more than ten input neurons are used. When visual input from a camera is used, the value of just a few of the pixels are usually input to the network; these pixels are either selected *a priori* by the human experimenter [29], or chosen by the neural network in a process similar to active vision [43]. It is hard to imagine how to respond intelligently to a complex world when presented with so low-dimensional sensory information - how would you get along with your daily tasks if you could you experience your environment only through a grey-scale screen with a resolution of 4×4 pixels? Not very well at all, probably. We will return to this later on, but first we will discuss the issue of simulation in evolutionary robotics.

2.3.1 Reality and simulation

One of the main reasons for the virtual absence of fast, non-holonomic or morphologically complex robots in evolutionary is that it is very hard to use such robots with an evolutionary algorithm. In evolution, we need to give the robot roughly equal starting positions in each trial. Therefore, any actions that breaks the robot or moves the robot to a position from which it is not trivial to move it back to the starting position (such as the robot having its legs and arms tangled, being upside down, or just being out of range of the tracking system) is not permissible. However, evolution relies on producing random strategies, and testing them to see whether they work. So the only way to make sure that we can evolve behaviour is by using a robot so simple and robust that it can withstand any sort of actions and then trivially be brought back an initial position each trial. This pretty much excludes airplanes, racing cars and human bodies.

For the purposes of the discussion here, we can call robotic systems which it is trivial to automatically return to a good starting state (member of a set of a predefined starting states, possibly just a single position and orientation) regardless of the preceding sequence of actions *recoverable*, and a robotic system where this is not possible *non-recoverable*. A good example of a recoverable system is a speed limited Khepera robot on a walled tabletop covered by an overhead webcam; good examples of non-recoverable systems includes almost any mobile robotic or vehicular system which is actually used for something.

The obvious answer to this problem is to evolve in simulation. Problem solved: just reset the simulation at the start of each trial! An added benefit would be a huge potential speedup, as an efficient simulation could be run much faster than real-time.

In practice, things are not that simple - far from. The reason that most evolutionary robotics research is still done on physical robots is that good simulations are hard to get by. If we had access to The Matrix[2], we could do fantastic evolutionary robotics experiments effortlessly, but alas, we have not. To get a good simulator of a robot, we either have to build a model of the robot, based on physical theory and manual measurements, or learn a model, based on taking actions and observing the responses from the real robot and its environment. Or a combination of both.

Complicating the problem of acquiring a model good enough to base a simulation on is the issue of *model exploitation*. Ample empirical experience has shown that evolutionary algorithms are fantastically good at "cheating", in the sense of finding strategies that use models in ways that were not intended by the designer of the model, and would not work on the modelled system. For example, if wall collision handling only works properly at low speeds, the evolutionary algorithm will probably find a strategy that involves accelerating to high speeds and driving straight through walls; if the rangefinder sensors are more accurate in simulation than reality, the evolutionary might well find a strategy that relies on the exact sensor signatures of various parts of the arena, and will not work with on the real robot with its noisy sensors. Here, the celebrated capability of evolutionary algorithms to find unorthodox solutions that transcend the conventions of human invention turns into a troublesome tendency to exploit little mistakes and errors in a model, like a malicious lawyer uses his creativity to find holes in the law.

One of the earliest approaches to acquiring models specifically for evolution is Jakobi's *radical envelope of noise hypothesis* [65]. Jakobi advocates dividing aspects of the robot and environment simulation into a *base set* and an *implementation set*. The base set contains all the aspects that are deemed (by the experimenter) to be required for a good evolved controller; these are subject to the same variability as present in their real world characteristics. The implementation

[2] "The Matrix" is a 1999 Science Fiction movie where the world turns out to be simulated on a massive computer.

set contains all other aspects, and is subject to variable amounts of noise, from none to massive, in order to discourage the evolution of controllers relying on features from that set (even on the presence of noise, which is why the noise levels are varied). For example, if a robot only needs one of its sensors for a particular part of a task, this sensor will be assigned to the base set and the other sensors to the implementation set (and their readings in simulation will only be variable amounts of noise). In other parts of the experiment, other other sensors might be part of the base set, or maybe only some aspects of the readings of some sensors.

Jakobi does not prescribe any particular way of delineating base set aspects from implementation set aspects, nor any particular way of modelling the base set. This leaves us both with the task of modelling the base set accurately unsolved, and with the additional task of deciding what information is needed when - which, even if we manage to do it right, risks restricting the creativity of the evolutionary process. Jakobi built a Khepera simulator using this method (he used lookup tables of robots placed at various position relative to walls, and returned the sensor readings in the table that were most similar to the simulated situation) that was used for several experiments where controllers were evolved in simulation and transferred to the real robot with intact functionality. However, this seems not to have been done with any significantly more complex robots than the Khepera.

A recent and very different approach, that has also been used with morphologically more complex robots, is Bongard et al.'s *continuous self-modelling* [11]. The process is best described in a cycle. The robot start by performing an arbitrary action and recording its sensory consequences. This information is passed to a set of internal models that compete for the best explanation of the behaviour. In order to improve the quality of the self-model, the robot tries to gather further information by triggering different actions. The action selection mechanism is model-driven; it selects the next action based on the maximization of information, more specifically it selects the action that causes the greatest disagreement among the competing internal models. Once an action is selected, it is overtly executed and the whole cycle starts again.

Using this highly innovative and interesting method, Bongard et al. managed to co-evolve a model and a controller for a legged robot direct in physical reality. However, it is very unclear how well this method would be applicable to an unrecoverable robot, especially one with complex dynamics or significant momentum. The crux here is that the action that maximizes information might very well be one that breaks the robot, or otherwise puts it in an unrecoverable state.

Another possibility is to use a robot for which the dynamics have already been extensively studied, and use the best available knowledge to constrain the solutions that can be learnt. Abbeel et al. [1] used this technique to acquire a model of a large single-rotor model helicopter, a model which was then used to learn a controller for the helicopter using temporal difference

learning. Their work, though technically impressive, is characterized by the frequent use of domain knowledge; whenever possible, human knowledge of the functioning of the system and the characteristics of the desired task are used to constrain the model, and in the end a least square algorithm is used to fit a handful of parameters in a mathematical model. Their approach works well in the sense that it produces the desired behaviour, and nothing but the desired behaviour, but it is certainly not in the spirit of evolutionary robotics: nothing new was learned about how to achieve a solution to the task at hand, or about the capabilities of the robotic platform, except in a strictly quantitative sense. (Strictly speaking, their approach is not evolutionary robotics at all, as they don't use evolutionary algorithms, and they are probably happy with this being so.)

2.3.2 The assumptions and philosophy of evolutionary robotics

Most "mainstream" evolutionary robotics research is driven by a set of shared assumptions and motivations, which very bluntly can be expressed as a wish to as far as possible exclude humans from the design process and leave as much as possible to evolution, because we get better results that way and/or because the results get more interesting that way. Some researchers, notably Cliff [27][28] and Nolfi [98], have attempted to spell out these assumptions and motivations and forge them into something like a coherent philosophy of evolutionary robotics.

Cliff stresses the importance of the *action-perception cycle* and modelling complete *sensory-motor pathways*. This means that the neural network controlling the robot should be connected directly to the robot's sensors in one end and to its motors in the other end, without any additional human-constructed processing of the in- or outputs. Only by studying whole organisms, whether physical robots or simulated agents in virtual worlds, in their natural environments, can we learn how and when a certain behaviour arises and how it is produced. The argument for this is that if we want to learn something about the neural mechanisms that "naturally" underlie a particular form of adaptive behaviour, we need to refrain from interfering with how that behaviour is represented. In particular, we need to avoid human interpretation and re-representation of the sensor inputs.

The roots of this argument is Harnad's critique of much of the early work in neural network modelling of cognitive processes [55]. In these studies, processes such as conjugation of verbs or recall from short-term memory were explored using mainly symbolic representations at both the input and output end of neural networks, which where usually trained with backpropagation. As a hypothetical example, the first and last letter of a word could be encoded as activations of the 52-dimensional input vector of a neural network, with each neuron representing a different letter, and the activations on the output vector representing gender of the word, language it was written

in, mood of the speaker or somesuch. Harnad (and, to be fair, several others) made the point that while such experiments might help our understanding of the learning capabilities of neural networks, they don't help our understanding of grammar processing or short term memory, as they suffer from the *symbol grounding problem*. That is, there is no guarantee information should be represented/encoded in the same way in the human brain as in the experimental setup, or that such a representation would ever occur in a naturally or artificially evolved system. Indeed, not only is there not a guarantee, there is not even a reason to believe that the input and output representation used should exist anywhere outside the experiment in question. So the symbols are not grounded, and the neural network lacks semantics. It is very possible that the actual problem of e.g. conjugating verbs in a complete system that can comprehend and produce speech is indeed completely unrelated to the experiment, and thus that an experiment with human-designed input and output representation fails to add to our understanding of the actual phenomenon.

Cliff agrees with Harnad that the symbol grounding problem can be avoided by connecting the sensors of the agents directly to the neural network controlling it, and its outputs to the agent's actuators. He also adds, following Brooks [14], that the agent should be *situated* and *embodied*, so that it takes actions in a consistent world, whether real or simulated, where the agent has a body that is affected by the interplay of the actions it chooses to take with the state of the body and world, and the state of the body and world decides what it perceives. In other words, the complete opposite of a neural network that is passively being fed a list of data and outputting some transformation of it.

Nolfi distiniguishes between *proximal* and *distal* descriptions of behaviour. The distal description is a description of some behaviour from the point of view of an external observer, that uses his own concepts to make sense of the behaviour. For example, when an ethologist describes the behaviour of a mammal from his distal perspective, he might use such terms as hunting, searching, fleeing etc. to describe different aspects of its behaviour. However, things might be very different from the point of view of the sensory-motor system that creates the behaviour, i.e. from the proximal perspective. Nolfi argues that the proximal perspective always provides the most relevant information for designing behaviour. If we try to interfere as human designers, we will probably use the divide-and-conquer approach that is the norm in most types of engineering, and divide the controller into different parts, responsible for different aspects of behaviour. However, this division might be inappropriate to the demands of the task and fail to exploit the possibilities inherent in the interplay of the robot's body, control system and the world. The best we can do, Nolfi argues, is therefore to avoid specifying the structure of the control system, and design our fitness function so that is as high-level as possible, rewarding

the overall performance on the target task rather than performance on components or aspects of that task. It is worth noting that Nolfi advocates this approach not only for studying the evolution of adaptive behaviours, but also for pure engineering purposes.

We will return to the important questions raised in the last few paragraphs in the sections below on sensing and actuating, incrementality and modularity.

2.4 Issues in evolving controllers for complex behaviour in embodied agents

Why won't evolutionary robotics scale up? There are probably many reasons. Here, we will discuss a number of open research questions and promising methodologies related to the central goal of being able to evolve complex general intelligence. These are issues that remain to be elucidated even after we have solved the problem of having a good enough robot or simulation to evolve on.

This section was originally intended to be called "Issues in Evolving Intelligence", but as we are here going to discuss the topics which the research in this thesis aims to contribute to the understanding of, and thus will have to go in to more detail, some demarcations have to be made. (And so a long sentence can be justified. For the title, that is.) The first demarcation is that we are talking about evolving controllers, of the variety that more or less conforms to the principle of the closed sensory-motor loop. We are not discussing evolving plans, trajectories or parameters for controllers where the actual control algorithm is already specified. The second demarcation is that we are talking about embodied agents. The third is that we are not primarily discussing the sort of higher-level cognitive functions that are the subject of linguistics, personality psychology or fine literature, but rather the sort of sensorimotor coordination that is also displayed by many of our fellow vertebrates.

2.4.1 Sensing

To illustrate the importance of having the right sensor data, consider the argument by palaeontologist Andrew Parker that the development of the eye was the necessary precondition and trigger of the evolution of complex behaviours and ecosystems at the time of the Cambrian explosion, some 360 million years ago [104]. Before that point in time, the fossil record indicates that complex body forms, thought to be necessary for fast and complex movement, simply did not exist. Most probably, Earth's oceans was filled with grey, slug-like beings slowly crawling the seabed, filter feeding or eating other primitive animals when they happened upon them. These creatures seem to have had simple light sensors, but not eyes capable of discriminating

shape and colour at a distance. Soon after the first apperance of proper eyes in the fossil record, we have a phenomenal diversification, with clear evidence of specialized predators of all sizes, complicated defence mechanisms in grazers and filter-feeders, parasitism, symbiosis etc.

Seeing evolutionary robotics (and more broadly, evolutionary artificial intelligence) in this light, the suspicion quickly arises that we might never move far beyond the behavioural complexity of a slug without making sure that we have the right sensor data, the right amount of it, and represented in the right way. As noted above (in section 2.3), in most evolutionary robotics experiments, less than ten inputs are used. As a comparison, the eye of a fly has about 400 directional facets, and thus at least that many dimensions of visual input data available.

It is important to note that the issue of sensing is not settled by agreeing (or disagreeing) with Cliff that we need to close to the sensory-motor loop, and feed the controller unprocessed sensor data (and the actuators unprocessed controller outputs). There are still issues about the *type*, *amount* and *representation* of sensor data.

If we use a physical robot, the type of sensor data we can use is obviously limited by what sort of sensors are available on the robot. But, budget permitting, we can choose what robot we want to use, or even develop our own. Here, there is a bewildering array of sensors to use: infrared and laser rangefinders, different type of cameras, odometry, global (GPS) or local (e.g. Vicon) positioning systems, whiskers, various exotic sensors for chemical compounds, air pressure etc. All of these sensors are available in different versions, with different precision, range, noise levels etc. If we evolve in simulation it is even harder to choose the best type of sensor, as we are free to define whatever sensors we want. Some evolutionary roboticists have taken the logical step to evolve the sensor configuration along with the controller, but these experiments have largely been confined to evolving the angles of a small number of simulated infrared rangefinder sensors [17].

As discussed above, the amount of sensor data used in evolutionary robotics experiments is typically very small, even when visual data is used. Two likely contributing reasons to this is that it is computationally expensive to simulate complex sensing, and that the hitherto used neuroevolution methods are not able to cope with large amounts of inputs to the neural networks. But it is not easy to ascertain whether these are indeed the main reasons, as people usually don't publish experiments with only negative results. Probably, there are numerous half-written up reports on experiments with massive input spaces that just failed to evolve any interesting behaviour lying around on researchers' hard drives. Alternatively, most researchers don't see scaling up the complexity of input data as a priority.

The representation of input data is another understudied question, which becomes an issues especially in studies done in simulation, where there is no "raw" format for the sensor data.

Imagine the controlled agent is at position $10, 10$ facing left; there is an adversary at $20, 30$. There are at least four obvious ways of representing this situation to the controller, and endless variations and hybrids of these, without violating the principle of the closed sensory-motor loop:

- Third person Cartesian: two inputs are given the values 10 and 10, and two other inputs are given values 20 and 30.
- First person Cartesian: two inputs are given the values 10 and 20.
- First person polar: two inputs are set to the angle and range to the adversary.
- Wraparound sensors: A number (e.g. 8) of simulated sensors are spread evenly around the agent (at angles $0, \pi/2, \pi$ etc.), each sensor is connected to one input of the controller, and all sensors return 0 except the one whose direction is closest to the direction to the adversary. Or they all return a fractional value proportional to how close its direction is to the direction to the adversary, etc.

All evolutionary robotics experiments must decide how to represent the input at some point, but this is very rarely seen as a topic of investigation in its own right. One very recent example of taking this issue seriously is Stanley's tutorial at CIG 2007 [122].

Issues of sensing and sensor data representation will be addressed in sections 5.1, 5.2, 5.3.1 and 5.4.2.

2.4.2 Incrementality

When evolving a controller to solve a complex problem, the obvious way to design the fitness function is to measure the success of each individual controller on solving the whole problem. This, however, can give rise to the *bootstrap problem*: the initial, random controllers, fail to make any progress at all towards solving the problem. All individuals receive zero fitness, and consequently no evolution takes place. It is all too easy to construct such a fitness function: imagine evolving a dogbot to fetch the slippers at the door and bring them to the armchair in the middle of the room. A natural fitness function would be how many slippers are fetched, maybe with the time taken to fetch them subtracted. However, the initial generation of controllers would be completely random, probably having the dogbot running around in circles chasing its tail or standing still and barking on the spot. Zero slippers are fetched in the first generation, and so no slippers will ever be fetched.

A solution to this problem would be to use a sequence of fitness functions: first a fitness function rewarding the controllers for getting the dogbot as close as possible to the slippers, and once the best individuals in the population have a decent fitness of this sort, the fitness

function is changed to one that assigns fitness based to how often the slippers are being picked up, and then another one could be based on how close to the armchair they are dropped. The assumption here is that the controller that performed well on the previous fitness function is likely to at least occasionally perform decently on the next fitness function, and so a fitness gradient is established in the population and evolution can proceed.

In evolutionary robotics, such a decomposition into a sequence of fitness functions, or of increasingly complex versions of the same problem, is called *incremental evoution*. Early examples of incremental evolution include Gomez and Mikkulainen's simulated grid-world predator that is first evolved to approach a static prey, then a slowly moving prey and finally a fast prey with complicated behaviour [49]. It was shown that the incrementally evolved controllers performed much better than controllers evolved using direct (non-incremental) evolution. Similarly, Harvey et al. evolved a visually guided robot first to move forward, then to move towards a large target, to move towards a small target, and finally to move towards triangles and not towards squares [57]. In this case, evolving directly for distinguishing triangles from squares did not work at all.

Incremental evolution has strong similarities to *programmed learning*, an educational technique invented by the father of behaviorism, B. F. Skinner [120]. In this paradigm, complex topics are taught to students through a "teaching machine" where simple questions are repeatedly given to the student, and if the student passes the question a new, more complex question is given; if the student fails the question, some information pertinent to the question is given, and a new question of similar difficulty is presented. (In practice, the "machine" could be a book with instructions as to which page to jump to.) This idea drew considerable attention in the 1960's, but currently holds negligible clout in educational policy, both because of dissatisfaction from students, and because of the immense amount of work required in "programming" the "machines", but mostly because of behaviorism being so completely out of fashion nowadays. However, the idea is alive and well in animal training, and it could be worth wile looking at animal training programs for new ideas about incremental evolution.

It's interesting to contrast the idea of incremental evolution with Nolfi's argument (in section 2.3.2) that we limit the potential of evolutionary robotics, both as an engineering tool and as a scientific one, if we decompose the problem ourselves, and only feed the evolutionary process bite-sized chunks. Incremental evolution is problem decomposition, if anything is. It might be that we have a trade-off here: if we tell evolution which way to take, we are more likely to get there, but we are also constraining evolution's creativity. One way to have to the cake and eat it could be to find a way to automatically decompose the problem into incremental stages, by evolution or otherwise. But to do this we need to clarify when incremental evolution works, and

when it doesn't.

Another perspective on incrementality is that in real life, real organisms only have one fitness function and it's quite a crude one (roughly, the number of grandchildren the organism has). Still, very complex behaviour manages to evolve, probably because there is not only one way to take; there are countless niches to fill, and the evolution of one capability can pave the way for (or *exapt*) another (e.g. according to one theory, language evolved to be able to gossip about other hominids in our flock, but then it turned out to be useful for lots of other things). Thus, it might be that externally imposed incrementality is only really necessary because of the simplicity of our experimental environments.

In this thesis, incrementality will be addressed in sections 5.2, 5.4.1 and 7.2.

2.4.3 Modularity

Another form of decomposition is *modularity*, where the controller rather than the task is decomposed. The majority of work done in evolutionary robotics and evolutionary neural networks is on evolving the connection weights of a single network of homogeneous structure, such as a fully connected MLP. Modular neural networks can, very broadly, be categorized as neural networks made up of structurally and/or functionally distinct parts. Theories about modularity go far back in the history of science, and there are reasons to believe that modularization of neural networks can render them more evolvable.

Modularity and the mind

The debate over whether the mind is modular goes back at least to early modern ages, with the empiricists (e.g. Locke and Hume) claiming that the mind was "a blank slate" at birth, acquiring its content through associations between sensory data, and the rationalists (e.g. Descartes), who thought that we had the mind was innately and intricately structured. Back then, however, little was known about the brain. The first to base his theory of mind explicitly on neural modularity was probably Franz Joseph Gall (1758-1828), best remembered as the father of phrenology. Gall maintained that the brain was composed of a great number of "propensities" or "faculties", such as self-esteem, imitation, tune, and cautiousness, all of which were physically localizable. The methods he proposed to localise these faculties, however, rightly became infamous.

In the early twentieth century, behaviorism was invented and gradually came to dominate the new science of psychology [119]. Like classical empiricism, behaviorism stated that no knowledge was innate, but all was formed from experience; the mechanism whereby this was accomplished was the association of stimuli with order stimuli, or of actions with consequences. At birth mind was a *tabula rasa*, blank slate. As behaviorism gave way to cognitive psychology, the

anti-modular bias persisted.

However, in the early eighties, the cognitive scientist Jerry Fodor brought modular thinking back with his influential book "The Modularity of Mind", where he argued that the mind/brain is made up of several modules at the input and the output stage (e.g. modules for perceiving various visual patterns, and executing smooth movements) but that these modules all connect to a "central processing unit" [45]. Others have since gone further, especially within the field of evolutionary psychology, which partly relies on the *massive modularity* hypothesis [7]. According to this hypothesis, the mind/brain is made of thousands of modules, which are responsible for particular behaviours, from mate-seeking to navigation to sleeping at night. One of the main arguments for this hypothesis is that there seems never to have been any evolutionary pressure for a non-modular, general-purpose brain, and that evolution usually proceeds by building new layers on top of the existing organism. Thus, the most probable path for evolution would be to incrementally add new functionality to an existing mind, rather than redesigning the mind from ground up. Some evolutionary psychologists add that it is very hard to imagine how a non-modular mind would function at all. However, this hypothesis has come under heavy criticism from more orthodox psychologists, who usually accept that the mind is modular to some degree (*a la* Fodor) but assert that there is very scant empirical support for the massive modularity hypothesis [112].

While the debate is sometimes fierce over the amount of *functional* modularity of mind, there is broad agreement among neuroscientists about there being considerable *structural* modularity in the brain [22]. The human brain (and those of most animals) is clearly divided into numerous parts, that are different to each other even to the naked eye; furthermore, there is plenty of repeating structure in some parts of the brain, e.g. in the visual areas. Some parts of the brain have also been shown to have a very distinctive functionality, in the sense that if an area is damaged you lose the corresponding functionality - for example Broca's and Wernicke's areas of the parietal cortex, involved in speech production in comprehension, respectively. The specific function of some other parts of the brain is much less understood.

Evolving modular neural networks: theory

Why would we want to evolve modular neural networks? The two broad answers, which are of course not mutually exclusive, is that we with good models can make a contribution to the ongoing interdisciplinary investigation into modularity in natural systems summarized above, and that we could conceivably increase the evolvability of neural networks by modularizing them in different ways, thus increasing the performance of technical applications of neuroevolution. While the potential for neural network modeling to contribute to natural science might be

obvious, some explanation might be needed as to why modularity would increase the evolvability of such networks (and in particular, why it could be a cure for the curse of dimensionality).

First we need to make a distinction. We can call dividing up a neural network into several modules, where none of them need to be similar to the other modules, *static modularity*. When two or more modules are identical, sharing topology and connection weights (defined by the same part of the genome) we call this *repeating modularity*. The special case of repeating modularity, where modules are repeating according to some tiling or symmetry, can be called *convolutional modularity*.

The following arguments for the usefulness of modularity are drawn from an earlier paper by Togelius [137], and another by Di Ferdinando and Calabretta [20].

- To begin with static modularity, the fact that most neurons in a module are only connected to other neurons in that module (encapsulation) reduces *neural interference*, a term coined by Calabretta for when parts of the network which have nothing to do with each other, and therefore should not be connected, are connected. The problem this causes can be understood by thinking of a fully connected network - in such a network, any connection weight change is bound to influence every neuron! This causes a lot of epistasis, and more or less rules out neutral mutations (mutations which change the network without increasing or decreasing fitness). Severing the right connections can increase evolvability by reducing neural interference. This illustrated in a lesioning experiment by Nolfi, who showed that cutting just a single connection in a robot-controlling network allowed for much higher fitness on a task [99].

- Another huge benefit of encapsulated modules is the reduction of the search space dimensionality. Most of the parameters of a neural network are connection weights; in a fully connected neural network, these increase on the order of the number of neurons squared. In a modular neural network the number of connections per neuron can be dramatically smaller. Apart from a smaller search space, fewer connections also means faster network updating.

- Given that evolved modular networks are usually structurally obscure, and their inner workings in many cases more or less impossible to understand even for the human designer of the software that evolved the network, another benefit of modularity is it might be possible to find out which evolved modules do what and reuse those modules in another network, or connect them to some other non-neural code.

- Moving on to the benefits of reusable modularity, having several modules defined by the same part of the genome further decreases the search space. This effect can be dramatic

in the case of convolutional modularity with a large number of modules.

- Finally, modularity can be combined with incremental evolution to create *layered evolution*, where modules are evolved one by one, and the weights of the old modules "frozen" (not affected by the mutation operators) while the new module are evolved. This was shown to be more effective than either incremental evolution or static modularity alone in a simulated robotics experiment [136][137].

Evolving modular neural networks: practice

Several researchers have tried evolving neural networks with fixed topologies for various tasks, and usually shown that modularization increases evolvability of the network. Starting with a non-robotic experiment, Schraudolph trained a convolutional network to evaluate board values for the board game go[117]. The network has certain simularities to the convolutional pyramids as used in optical character recognition.

Husken et al. did an interesting experiment, where they evolved a monolithic network together with which subset of its connections to activate, and showed that for a network with two completely separable inputs and outputs, more or less encapsulated modules did indeed evolve [63]. Further, Di Ferdinando and Calabretta showed that hard-coded modularity increased evolvability on a pattern-recognition task based on a simple model of the what/where visual system in the mammalian brain [42].

More interesting for our purposes is the experiments that have been done with evolving modular neural networks for robot control. An early example is Beer's work with evolving CTRNNs that act as pattern generators. Beer used six identical modules with only limited intercommunication and no central controller to successfully evolve a walking gait in a model insect [47]. The use of CTRNNs, where neurons have internal state, in this example points to the need to recognize that reusable modules are indeed separate modules with their own state, even though they are generated from the same genetic information.

A more recent example with evolving repeated modules is due to Vaughan, who evolved a segmented robot arm for grasping objects in a two-dimensional simulated environment [148]. Each segment of the arm was fitted with an identical neural network, where the output of one segment's network was the input to the next; the first segment was the only one to receive any direct sensor input.

The most well-known (or at least highly cited) experiment in evolving modular networks for robot control is probably due to Calabretta et al. [21]. Calabretta's team investigated several modular and non-modular network architectures for a rather complicated robot control task, and found that the modular structures evolved better than the non-modular ones, and of those

structures the one where human involvement was minimized (by letting the network itself choose when to use which module) performed best.

Evolving the modularity itself

Deliberately modularising neural networks can be criticised with the same argument as that used against incremental evolution in the previous section: we are constraining evolution by forcing it onto our chosen path, and might miss out on some of its most innovative solutions. One way to retain the full innovative capacity of evolution while reaping the benefits could be to evolve the modularisation itself. Several of the indirect neural network encodings mentioned in section 2.2.2 were designed to allow static or repeating modularity to emerge, and this did indeed happen in some experiments. However, the failure of these encodings to match the performance of fixed-topology neuroevolution in the general case point to the need for further research and new approaches in this area. In order to design a winning evolutionary modular neural network design algorithm, we probably need to learn more about what sorts of modularity work when through systematical experiments with hard-coded modularity.

In the experimental chapters of this thesis, different types of modular controllers will be analysed and compared to non-modular controllers in sections 5.4.2, 5.4.1 and 7.2.

2.4.4 Controller representations

In the discussion above we have implicitly assumed that the controller is based on a neural network, similar to a standard MLP in its function but possibly with a different topology. However, such neural networks are not the only evolvable representations capable of universal function approximation. The main competitors in this respect are the various forms of more complex and often more biologically faithful types of neural networks, and the various types of genetic programming.

Continuous-time recurrent neural networks is a class of more complex neural networks, whose capabilities subsume and significantly extend the capabilities of multi-layer perceptrons [8]. Like an MLP, each neuron in a CTRNN has several inputs, and produces a single output value which can be fed as input into several other neurons; unlike in a MLP, each neuron has a persistent activation, and several parameters guiding how the state evolves in time.

While CTRNNs, just like MLPs, represent the activation of a neuron as a scalar with a value defined at any instant in time, spiking neural networks borrow another key principle from the study of biological nervous systems and represent neural activations as “spikes”, short bursts of negative (simulated) electric potential interspersed with long periods of relatively low electric potential. In such networks, the activation of a neuron is defined as spike density over

a given period of time [35]. Other networks take the basic idea of a neural network in yet other dimensions, such as networks with plastic synapses [44], homeostatic networks [38], complex-valued networks etc.

Genetic programming (GP) refers to the evolution of computer programs or function approximators which are not represented as neural networks [73]. In most cases, GP programs are represented as *expression trees*. This means that they consists of a number of *nodes*, both *terminals* and *non-terminals*, the difference between these being that a non-terminal has child nodes, which are themselves either terminals or non-terminals. Evaluation of a GP tree is typically done through *lazy evaluation*: the *root node* of the tree is evaluated first; it's child nodes, and child nodes' child nodes etc. are then evaluated recursively as needed, in order to provide the values needed to evaluate the root node.

Thinking of a GP tree in terms of a neural network, the terminals can be thought of as the inputs to the tree, the root node of the tree can be thought of as the output, and the non-terminal nodes as the neurons. Two crucial differences between neural networks are that there is no multiplication of node outputs by connection weights (or, equivalently, all connections between the node have the weight 1), and that most non-terminals used in GP trees are quite different from the normal MLP neurons, which sums the inputs from its "child nodes" and then applies the *tanh* squashing function to the sum. Standard GP non-terminals include the arithmetic operators $(+, -, *, /)$ applied to two child nodes, and the conditional operator outputting the value of the second child if the value of the first child is below 0, otherwise the value of the third child. Most GP trees have a mix of different types of non-terminals, unlike MLPs and similar neural networks where all neurons function identically.

Other significant differences are in the variation operators. Almost all GP algorithms are constructive, meaning that they start small and grow bigger through mutation, recombination and selection. A common mutation operator is *one-point macro-mutation* where a node is selected at random, and exchanged for a freshly created node of a type picked at random from list of permitted node types. The new node can be either a terminal or a non-terminal, in which case its children are created recursively in the same way. A potential problem with this operator is that the tree can grow without bounds, if non-terminals are selected more often than terminals. Therefore, a maximum tree depth, at which only terminal nodes can be selected by the mutation operator, is often imposed. In most GP implementations, recombination is used as well as mutation; the simplest recombination operator simply picks a random node in one tree and swaps it with another random node in another tree (of another individual). Obviously, many more variations of mutation and recombination exist in the GP literature.

One of the main inventions in tree-based GP is automatically defined functions (ADFs)

[74], which are external trees that can be called by terminals by the main tree(s) of the same individual. The advantage of ADFs is that the same ADF can be called several times from different trees, or from different points in the same tree, hence allowing for reusable modularity.

Additionally, there are variations on GP that do not use tree representations, for example linear GP, where string of instructions similar to mainstream programming languages are evolved; we will not go into detail of any of these variations here.

With this wealth of different evolvable controller representations available, the key question is: which one is best? Put a little less crudely, there are number of interrelated questions that we currently don't know the answers to: What controller representations can learn good control fastest (*learning speed*)? Can some controller representations learn to solve more complex control tasks than others (*learning depth*)? What is the relation between learning speed and learning depth? Are some controller representations better suited for some problems and other representations for other problems? If so, is there a systematic relationship between problems and suitable representations? Which controller representations are most understandable and reusable by human designers? Which controller representations are easiest to combined with non-evolutionary learning algorithms?

There is currently no complete answer to any of these questions, and little to be learned from either theory or empirical studies. We do have partial answers to the last two questions, though. Evolved GP trees are usually much easier to understand than evolved neural networks, as the operations performed at the GP non-terminals are much closer to scientifically educated humans' ways of thinking than the operations of MLP neurons are, and as the constructive evolution of GP trees make for sparser structures than fully connected networks. Further, there are other variations of GP developed expressively to be easier to understand for humans [146]. On the other hand, MLPs and some other neural networks can easily be trained by non-evolutionary algorithms, such as the ubiquitous backpropagation algorithm, and thus it is easier to create systems that combine evolutionary learning with other forms of learning.

One take on the questions about learning depth and learning speed is how they relate to the complexity of the controller. It seems more or less obvious that a larger neural network would be able to produce a more complex behaviour, as there is simply room for more information in it. But would the larger neural network learn a behaviour of a given complexity faster or slower than the smaller neural network? E.g., the complexification approach of Stanley's NEAT algorithm relies on the assumption, borne out by some experimental results of his, that learning speed is higher for a small network [123], but see section 7.2 for results suggesting otherwise.

It is not obvious how to settle such questions other than by doing numerous independent empirical studies, where different controller architectures and different sizes of the same archi-

tecture are compared on different tasks by different experimenters. The experiments presented in this thesis therefore represent only a small contribution to answering these questions.

In this thesis, the neural networks we use are either MLPs or simple MLP-based recurrent networks, but we are also comparing these with different versions of tree-based GP programs in some cases. The more exotic version of neural networks and genetic programming have been left out in order to simplify both implementation and analysis by limiting the number of extra parameters that could affect learning speed.

Comparisons between different controller representations will be made in sections 5.1, 5.3.1 and 7.2.

2.4.5 Stateful control and internal models

In much of the discussion above, we have implicitly assumed that the controller is a function from inputs (typically sensors) to outputs (typically motor activations). Thus, at any point in time, the outputs of the controller only depends on its inputs at that very moment, and not on the state of the controller itself. The same sensor readings always result in the same motor actions. Such controllers are called *reactive* controllers. The opposite, controllers whose outputs are influenced by the internal state of the controller, are called *stateful* controllers.

Several evolvable controller representations allow for the development of stateful controllers. The most common neural networks with stateful capabilities are the above mentioned *recurrent neural networks*, which differ from feedforward networks in that they can have cycles, or connections from a neuron to itself or to another neuron which ultimately connects to itself (but note that e.g. CTRNNs can have state without being recurrent). The one we will be using mainly here is the simple *Elman network*, so called after its invention by Elman in 1990 [41]; it also commonly goes by the name of *recurrent MLP*, as it it is a straightforward extension of a multi-layer perceptron. The difference between an MLP and an Elman network is that the latter in addition to the layers of connection between the input layer and the hidden layer, and between the hidden layer and the output layer, also has a layer of connections from the hidden layer to itself, so that each hidden neuron has a connection to itself and connections to each of the other hidden neurons (in this case; variations are possible).

The propagation phase of a recurrent MLP is just like that of a normal MLP, except that every time step, a copy is made of the activations of the hidden neurons; every time step, activations are also propagated from the copy of the hidden neuronal layer *of the previous time step* to the hidden neuronal layer of the current time step, via the connections in the recurrent connection layer. The backpropagation algorithm can be adapted to work with recurrent MLPs, yielding an algorithm called *backpropagation through time*. This is an advantage of recurrent

MLPs over more complex neural networks with state capabilities, such as CTRNNs.

As discussed above, most GP systems represent solutions as simple expression trees, which only allow for reactive controllers. However, ever since the very early days of GP, researchers have been experimenting with forms of GP that allow stateful controllers to evolve. In some experiments in his first book on GP, Koza used global registers that could be manipulated with specially built storage operators [73], and later introduced the notion of *automatically defined store* (analogously to his ADFs discussed above) [75]. Similarly, Teller introduced the concept of *Indexed Memory* to allow reading and writing from an arbitrary set of memory cells [133]. Recently, Lucas introduced *Object-Oriented Genetic Programming* (OOGP) [78][2], an approach where the primitives of the evolved program are calls to methods of objects in a standard object-oriented language, such as Java. The intention is to leverage the considerable libraries of tried-and-true atomic operations available in the API's of modern programming languages, including complex data structures.

So both neural nets and GP are available in strictly stateless versions as well as versions that allow for statefulness to be evolved. Yet it's possible to use model-based control, which normally requires state, while basing a controller on a stateless function approximator such as an MLP or a GP tree. This is done by complementing the function approximator with an explicit model of the dynamics of the agent, and instead of using the function approximator to generate motor commands given sensor inputs, it is used to estimate the value of being in a particular state, as specified by the sensor inputs. The controller then takes all the possible actions in its internal simulation of the agent, and feeds the simulated sensor inputs resulting from the simulated new state to the controller, one after another, and records the value estimates. After simulating all possible actions, the highest-value action is selected as the output of the controller, and thus for being carried out in the real world (or in the "real" simulation). While there are apparent potential advantages of this approach, it relies on having a sufficiently fast and accurate model of the agent's dynamics available, something that is often not the case. It also relies on it being possible and practical to enumerate the available actions (or at least a representative and meaningful subset of them) at any point in time. It could also be argued that it violates the principle of the closed sensory-motor loop.

Despite these different ways of evolving stateful controllers, a great many evolutionary robotics experiments use strictly reactive controllers. When is statefulness an advantage, or even a requirement? This is a topic of both debate and empirical research, though there is probably more of the former than of the latter.

Several cognitive scientists and roboticists argue forcefully that the need for agents to have internal representations of their surroundings is exaggerated. (Representations are clearly a form

of internal state, but it is a point of debate whether all state can be interpreted as representational.) Brooks famously claimed that "The world is its own best model", in other words that a situated and embodied agent doesn't need to store a complicated model of its environments, as its ability to control its sensors means that it can simply examine its environment when needed, a process known as *active perception* [14]. Similar sentiments are echoed by philosophers such as Clark [26].

Some evidence for world models being far less common and extensive than commonly assumed comes from a set of experiments on human vision, the debate over the correct interpretation of which is referred to as "the Grand Illusion debate". Many fairly recent experiments in psychophysics show that the human visual system is unable to take in information at a very high bandwidth - we simply don't see nearly as much as we think we do. (These experiments are often very entertaining to read about, or to perform on yourself. For example, did you know that while we can very easily spot the slight details differing between two pictures flashed in quick succession, this becomes more or less impossible if we wait half a second after extinguishing the first picture before showing the new picture?) Philosophers such as Patricia Churchland interpret this as evidence that we just think we have a model of the world, while we really don't, at lease not a very detailed one [25].

In the same vein, several evolutionary roboticists have performed experiments aimed at showing that tasks whose solutions appear to require stateful control can actually be solved by reactive controller through active perception. To this end, Nolfi managed to evolve reactive neural network controllers managing to approach cylindrical objects and avoid walls, even though the robot (a Khepera) was equipped only with a crude rangefinder sensor array, to which walls and cylinders would look the same from most angles; evolution found a solution involving clever repositioning of the robot into a vantage point from where the objects could be ambiguated [100]. The research on active vision from Cliff's and Floreano's teams (discussed in previous sections) also serve as examples of evolutionary robotics research being used to push the boundaries of reactive controller [29][43].

Yet, there are clearly tasks where reactive controllers fall far short. For example, there is evidence that the feedback loop between your hand, your eyes and your brain is too slow for you to control the movement of your hands accurately at high speeds using only the most recent information about the position and velocities of your hand. The only way to effect the degree of control we obviously have seems to be to have an internal model of the dynamics of the arm and hand.

Similarly, many tasks require different types of memory and can thus not be solved by reactive controllers. Behaviours requiring memory are displayed by a large variety of animals;

it is common even for invertebrates to be able to find their way back to previously visited areas. This can usually not be done by a purely reactive controller, as the action selected is often dependent not only on the current state but also on previous states (i.e. what places the organism has visitied previously).

In cognitive science, there a strand of thinking emphasising the role of internal simulation of the body for higher cognitive functions, such as imagination and consciousness. For example, according to Holland's theory of *Machine Consciousness*, we need to build robots with detailed models of their own bodies as well environments, in order to build machines that can emulate, achieve or be useful for studying human-level consciousness [60].

The models we will be using in this thesis will be rather simple, and we will not be overly concerned with whether our evolved controllers will be conscious anytime soon. Rather, we will investigate the potential usefulness in having or acquiring models for learning control in certain well-defined tasks. Stateful and reactive control will be compared in sections 5.3.1 and 7.2. State-value control will be contrasted with action-value and direct control in section 5.3.2. Further, acquisition and use of dynamics models will be dealt with in section 6.2

2.4.6 Competitive co-evolution

Co-evolution is when the fitness function of an individual is made dependent on other individuals, either in the same population, or in a different population altogether. A fundamental distinction within co-evolution is between *cooperative* and *competitive* co-evolution. In the cooperative varieties of co-evolution, which will not be discussed further here, different individuals share fitness or otherwise reap benefits from co-operating with each other, whereas in the competitive variety one individuals fitness in increased from another one's fitness being decreased, either through competition for shared resources, zero-sum fitness, direct battle between agents or some other mechanism.

The promise of competitive co-evolution is the hope that linking the fitness in this way will lead to some form of global improvement, as individuals compete against each other. The idea is to encourage an evolutionary "arms race", where improvements in some individuals cause further improvements in other individuals, and vice-versa. This idea is laid out (in the context of natural, not artificial, life) in a classic paper by Dawkins and Krebs, where they provide a taxonomy of naturally occurring types of competitive co-evolution (interestingly, only a few of these appear to have been used in artificial evolution) [34].

One of the first experiments to achieve good results from artificial arms races was Hillis' co-evolution of sorting networks and sorting problems [59]. In this pioneering experiment, a population of sorting networks was scored according to how well they solved the highest-scoring

members of a population of sorting problems, whereas the sorting problems were scored according to how well they defeated the highest-scoring members of the sorting network population. The best co-evolved networks beat both networks that had been evolved without co-evolution and hand-designed sorting networks.

Inspired by this early success, several research groups attempted to co-evolve predator and prey robots, where the predators were scored according to how fast they could catch the prey, and the prey were scored according to how long they could avoid the predator [57][101]. While these studies had some success, it quickly became clear that co-evolutionary algorithms can be prone to complex dynamics, which can thwart global progress towards higher fitness. There has also been a implicit assumption that an evolutionary arms race leads directly to an increase in complexity, though this is not always the case.

An example of a potential problem with co-evolution is that of "cycling" between different strategies. If an individual develops a strategy that affords it a higher fitness relative to other individuals, it will spread through the population, which will stabilise until a strategy arises that exploits a weakness in the previous one. There is then another rapid replacement of individuals. However, there is no guarantee that the new strategy is better than the one its predecessor replaced, as the dominance relation is intransitive.

E.g. if the population is first dominated by a strategy we call "Rock", this can be replaced by the strategy "Paper", which in turn can be replaced by "Scissors" - but Scissors is actually inferior to Rock, although it's superior to Paper!

It is therefore possible for the population to cycle through the same set of possible strategies, each exploiting the weaknesses of the previous one, without any global increase in fitness (such as the development of the strategy "Reinforced Pliers", which beats Paper and is immune to Rock).

Another problem that can occur in the case of more than one population is *disengagement*, a loss of selective gradient [149]. This is where one population evolves individuals of higher fitness more quickly, and they consistently beat individuals in another population. Therefore the other population would see a uniform selection pressure, due to consistently being beaten, and thus succumb to directionless genetic drift.

Several attempts have been made to address these problems, the most prominent of which was invented by Rosin and Belew: the "hall of fame" [113]. This technique has the individuals of the current generation compete not only against other current individuals, but also against a selection of good individuals from previous generations. Exactly how this should be done has been the subject of several studies, see e.g. [129][17]; there is some evidence that this process leads to a sort of conservatisim, in that the selection pressure becomes biased towards solutions

that can successfully counter as many previously evolved opponents as possible, rather than radically new solutions.

Currently, we are far from a panacea to the problems of competitive co-evolution. Put another way, there are plenty of open research questions concerning the dynamics of algorithms of this type.

In this thesis, competitive co-evolution is explored in sections 7.1 and 7.2.

2.4.7 Evolution and other forms of reinforcement learning

Evolution is not the only way to learn control based on feedback. In the broad category of reinforcement learning we also find a class of algorithms learning from local reinforcements. These algorithms associate particular actions with rewards, rather than the global learning of evolutionary algorithms, which associate rewards with complete strategies. The most famous and widely used of these algorithms are the *temporal difference learning* (td-learning) algorithms [130].

Various forms of td-learning exist, but they all work essentially according to the following specification. Each time-step, the following steps are taken by the controller:

1. Consider a set of possible actions and score them according to the value function. This can be done either by assigning values to actions based on the current state (the *action-value* approach), or by simulating the actions and assign a value based on the simulated state, as detailed above in section 2.4.5 (the *state-value* approach). The set of possible actions can be all possible actions for the current state if this number is low, or otherwise a heuristically selected subset of that set.

2. Take the action that the value function scores highest (or, with a small probability, another action).

3. Get the reward for this action from the environment.

4. Update the value function depending on the feedback from the environment *and the previous estimate of the action's value.*

The "trick" of td-learning is in the updating of the value function in step 4. As an example, consider update equation the Sarsa(0) variety of td-learning, using the action-value approach:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2.5}$$

Where $Q(s_t, a_t)$ is the value of action a given state s at time t according to the value function, r_{t+1} is the reward observed at the next time-step, α is the *learning rate*, and γ is the *discount rate*; both of the latter constants are commonly set to small positive values, e.g 0.1. The

interesting thing about this equation is that the value function is updated with a value that is partly dependent on its own valuation of the best action of the next state. This lifting-itself-by-its-bootstrap procedure ensures that the rewards for particular actions will influence not only the value function's valuation of that action, but also, to a lesser degree, its valuation of actions that lead to the rewarded action. It can be proven that under certain conditions variations on td-learning, such as Sarsa, *converge* to the *real* value function. These conditions are rather restrictive, and for most interesting tasks there is no convergence proof.

The main lure of td-learning algorithms is the idea that because they can adapt during a trial to local reinforcements, rather than just changing their policy between trials, and that the modification of the value function is directional rather than stochastic, they can potentially learn much faster than evolutionary algorithms. On the other hand, there are certain differences with evolutionary algorithms that make them less attractive in some respects.

One is that controllers must be based on mappings from state or state and action to value, represented either as a function approximator (such as a neural network) or as a table. As discussed above, this means that they must be able to enumerate (a representative subset of) all possible actions, and in the case of state value functions, possess a fast and accurate model of the agent's dynamics. Evolutionary algorithms, on the other hand, can learn both state value functions and action value functions, but also direct control where the output of the controller is interpreted as motor commands. It also means that the value function must be able to learn from examples in a supervised fashion, something which constrains the range of useful function representations. It is hard to see td-learning creating controllers based on complexification of neural networks, or on genetic programming.

Another difference with evolution is that instead of a fitness function, a reward scheme needs to be specified. This can sometimes turn out to be non-trivial. While it is straightforward to convert a reward scheme to a fitness function (just count the rewards incurred during a trial), the opposite is not the case, and for many problems several very different reward schemes can be designed where some work well and some not at all.

Overall, despite both algorithms being able to solve reinforcement learning problems, evolutionary learning and td-learning represent two very different ways of thinking. Consequently, the research communities investigating each family of algorithms are large separate, and papers written by and for td-learning researchers are often written in a style which renders them inaccessible to researchers working in evolutionary learning. (Quite possibly the opposite is also the case.) This makes it all the more important to use empirical comparisons to investigate under what conditions one or the other type of reinforcement learning algorithm works better, and in general experimentally characterise the differences and similarities between them. The holy

grail of this research direction would be an algorithm that combined the advantages of both types of algorithm with the disadvantages of none of them.

With recognition of the importance of such comparisons, the body of work comparing temporal difference learning with evolution is growing steadily. Often, though, these efforts are disconnected from each other and use dissimilar performance measures; it is often not clear what to make of comparisons that seem to come up with partly contradictory findings. Further, many of the comparisons deal with learning evaluation functions in board games, which is quite dissimilar from the agent control domains that are the main focus of this thesis.

An influential early experiment on td-learning of board game evaluation functions is due to Tesauro, who managed achieve world-class performance when training neural network-based Backgammon evaluation functions with self-play [134]. Pollack and Blair tried training evaluation functions using the same game and function representation, but instead of td-learning they used the simplest possible form of evolution, a hill-climber where a single individual was repeatedly mutated, and the mutation was kept only if it won a number of games over the non-mutated individual [108]. The algorithm worked, but its end results were far inferior to those of Tesauro [135].

Darwen [33] did a set of similar comparisons for Backgammon, and found that population-based evolution eventually outperformed td-learning when training a linear board evaluation function, even though evolution was much slower. However, when training a nonlinear evaluation function, board evaluators trained by evolution never reached the performance of those trained by td-learning for the simple reason that evolution training took too long time; this, in turn, was because effective evolution needed to evaluate the same pair of individuals many times.

Runarsson and Lucas [115] investigated temporal difference learning versus co-evolutionary learning for small-board Go strategies. There it was found that td-learning learned faster, but that with careful tuning, evolution eventually learned better strategies. In particular, with evolution it was necessary to use parent-offspring weighted averaging in order to cope with the effects of noise. This effect was found to be even more pronounced in a follow-up paper by Lucas and Runarsson [79], comparing the two methods for learning an Othello position value function. Kotnik and Kalita [72] found that evolution outperformed TDL at learning to play the card-game rummy, which unlike the board games in the above studies is not a game of perfect information.

However, there are some comparisons using dynamic agent control domains as well. Taylor *et al* [132] compared td-learning and the NEAT neuroevolution algorithm for 'keep-away' robocup soccer, and found that evolution could learn better policies, though it took more evaluations to do so. Results also showed that Sarsa learned better policies when the task was fully observable

and NEAT learned faster when the task was deterministic.

Gomez *et al* [51] investigated an impressive range of reinforcement techniques, including several versions of neuroevolution and td-learning, on four increasingly difficult version of the benchmark pole-balancing problem, ranging from simple to very hard versions. In striking contrast to some other studies, the best evolutionary methods uiniversally outperformed the best TD-based methods, both in terms of learning speed and in terms of which methods could solve the harder versions of the problem at all. Further, there was significant differences between different neuroevolutionary and td-based methods, with the best td-based techniques sometimes outperforming some evolutionary techniques. The relative ordering of the algorithms was similar across the different versions of the problem.

Some general results are beginning to emerge, but there is still much to be learned. One tendency that can be observed in many - though not all - of the observed studies is that td-learning learns faster than evolution, but evolution eventually learns better strategies. Under what conditions this is true is an important question.

In this thesis, we won't provide the missing synthesis of all this information, but hopefully add another small piece of the puzzle, in section 5.3.2 where we compare td-learning with evolution on the car racing problem.

2.5 Summary

This chapter has provided a very high-level overview of evolutionary computation, especially as applied to neural networks and robot control. We have treated the problem of reality versus simulation in evolutionary robotics, and the philosophy of evolutionary robotics, in some more detail, and have likewise gone into some depth on a number of topics that come up when trying to evolve complex general behaviour. These topics include sensors and their representation, incrementality, modularity, controller representation, stateful versus reactive control, competitive co-evolution and the relative advantages and disadvantages of evolution when compared to other reinforcement learning algorithms. Taken together, this provides a grounding in computational intelligence methods and current research issues against which the contributions in the experimental chapters can be judged.

This chapter has also described the basic learning algorithms, in particular the simple evolution strategy, which are used in the experimental chapters of the thesis. In some of the particular experiments these algorithms are used in their naive form, whereas in other experiments slightly augmented versions are used. Those algorithms are described in the relevant experimental section, referring to the descriptions given in this chapter.

Chapter 3

Computational intelligence and games

Computational intelligence techniques can be said to have been combined with games for the first time in 1959, when Samuel applied a simple reinforcement learning algorithm to the board game Checkers (also known as Draughts) [116]. With no human instructions, through only playing itself and observing which sequences of moves won games and which sequences lost, and using the extremely limited hardware available at the time, the algorithm managed to learn strategies good enough to beat its inventor.

After Samuel's early success, all was quite quiet on the CI for a long time. But for as long as there has been artificial intelligence research, a few researchers have worked on applying classical AI techniques, essentially specially tailored search algorithms, to board games such as Chess and Checkers. This line of research eventually led to the much-publicized victory of the IBM Deep Blue Chess computer over world Chess champion Gary Kasparov in 1997 [96]. Whereas this and other results are impressive in their own right, little or no computational intelligence is used in such research. Likewise, virtually no effort was spent by the academic community on developing AI for other types of games than board games, or on developing AI whose goal was not to win the game, for a long time.

In the last few years, however, a critical mass of researchers have become interested in both computational intelligence and in games, in one way or another, for the small but rapidly growing research field of Computational Intelligence and Games (CIG) to form. Much of the research in this field is published in a small number of conferences, all annual and started in recent years, that deal mainly in CIG research: Artificial Intelligence in Interactive Digital Entertainment (AIIDE), Eurosis GAME'ON, and IEEE Symposium on Computational Intelligence and Games (IEEE-CIG). This research can also sometimes be found in mainstream CI journals and conferences,

and also in some specialised game publications, such as the AI Game Programming Wisdom series of books [111].

In the last few years, a few PhD theses have appeared whose main focus is the application of CI to games, for example the theses of Spronck [121], Yannakakis [154] and Bryant [15]. All of these also deal with rather complex games, and with the use of CI techniques for other purposes than winning games as quickly and efficiently as possible, but rather with adapting difficulty levels of opponents to that of a human player (Spronck), to optimise player satisfaction (Yannakakis) and to generate behaviour that looks intelligent to the human player (Bryant).

This chapter presents a two-dimensional taxonomy of CIG research, which will be used to sample existing research in the field and to structure the experimental chapters of the thesis. The first dimension of the taxonomy indicates how the CI algorithms are used in the game: for *optimization*, for *imitation* or for *innovation*. The second dimension indicates whether the research uses games to study or enhance computational intelligence, or computational intelligence to augment games. Obviously, many studies and experiments could be seen as falling into several of these slots, and it is quite possible that the taxonomy is incomplete in the sense that some research which is undoubtedly CIG would fall outside of it. When that happens, the taxonomy will have to be revised; currently we will use the proposed categories to discuss CIG research in general. Among the many different types of games that will be discussed, we will emphasise racing games, as such games figure prominently in the experimental chapters of this thesis. But before putting CI and games together, we will briefly discuss computer games *per se*.

3.1 Computer games

There is not one definition of what a game is, but plenty. In fact, the philosopher Wittgenstein used the concept of a game in a number of thought-experiments designed to show that it was impossible to correctly define any concept in terms of sufficient and necessary conditions; instead, concepts are implicitly defined by those things that they refer to, and which are related to each other through family likeness. Learning to use a concept is learning to play the language-game that the concept forms part of, yet another example of a game [150].

While Wittgenstein is fairly far removed from the relatively down-to-earth topics of this thesis, more pragmatically minded authors have provided other definitions. The legendary game designer Sid Meier (creator of what the author of this thesis considers the best computer game ever, the epic strategy game *Civilization*) defines a game as "a series of meaningful choices". Others, such as Zalen and Zimmermann, emphasize conflicts as central to games: a game is

"a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome". Juul provides a more academic definition: "A game is a rule-based formal system with a variable and quantifiable outcome, where different outcomes are assigned different values, the player exerts effort in order to influence the outcome, the player feels attached to the outcome, and the consequences of the activity are optional and negotiable".

In discussing these and other definitions, game designer Raph Koster remarks in his recent book that none of them contain the word "fun" [71]. As fun seems to be so central to games, he then devotes the rest of the book to understanding what makes games fun. According to Koster, a game is fun to play because we learn the game as we play; we understand and learn the patterns underlying the game, and finally "grok" how to play it. This requires that the level of challenge always is approximately right, and that new patterns are always available to learn - games that are too simple or impossible to understand are boring. (It is important to note that games can still be fun after figuring out how to win them, as long at is possible to learn how to play even better.) Thus, a game is fun because it is a good teacher, at least according to Koster.

Another theorist who has tried to nail down the essence of games and why they are fun is Thomas Malone. He claims that the factors that make games fun can be organized into three categories: *challenge*, *fantasy*, and *curiosity* [81]. The first thing to point out about challenge is that the existence of some sort of goal adds to the entertainment value. Further, this goal should not be too hard or too easy to attain, and the player should not be too certain about what level of success he will achieve.

Games that include fantasy, according to Malone, "show or evoke images of physical objects or social situations not actually present". The sensation of being somewhere else, being someone else, doing something else. As for the third factor, curiosity, Malone claims that fun games have an "optimal level of informational complexity" in that their environments are novel and surprising but not completely incomprehensible. These are games that invite exploration, and keeps the user playing just to see what happens next.

With this very general characterisation in mind, let us now take a look at what types of games there are.

3.1.1 Types of computer games

There are countless taxonomies of computer games, all of which will be ignored in this section, in favour of a homebrewn categorisation more relevant to the argument of the thesis: *computerised games*, *agent games* and *management games*. We will also take a first look at how each type of game can be used in CI research, though this will be developed further in the rest of the chapter.

Computerised games are games which you don't really need a computer to play, because the computation involved is minimal. Such games almost always have discrete state spaces and simple rule sets, and are usually atemporal. Here we find all the classic board games: Chess, Checkers, Go, Monopoly etc. We also find card games such as Poker and Bridge, and puzzles such as crosswords and Sudoku. On the far end of the complexity scale for this type of games we find tabletop role-playing games such as *Dungeons and Dragons*, which are commonly played without the help of a computer, but with thousands of pages of reference material available, a large assortment of dice, and occasionally a pocket calculator. It is worth noting that even if physical sports such as Tennis, Football or Fencing are considered to be games, they don't fall into this category, as they require physical activity from the player in a way which e.g. board games don't. Rather, simulations of such sports can be played on computers with the intense physical activity replaced by button-pressing[1], but these are rather different things that fall into either the agent game or management game category (both are categories that require a computer to play).

Due to the simplicity of implementing computerised games, they are well suited for fair comparisons of different CI algorithms with each other, and with human players. This is especially true for those games that have been extensively analysed and which have large and active communities of human players. However, due to the nature of these games, the range of cognitive skills that can be investigated is strictly limited: Chess is extremely badly suited for research into perception, movement, the codependence of brain and body etc. (Though note Stanley's innovative application of a form of active vision for learning to play Go [128].) A case in point here is how little we learned about human intelligence from Deep Blue's victory over Kasparov.

Agent games are games where the player controls an agent in a simulated world, and more or less directly decides what the agent does at any point in time. Plenty of games fall into this category, including shoot-em-ups such as *Space Invaders*, platformers such as *Super Mario Bros*, racing games such as *Need for Speed*, adventure games such as the venerable *Legend of Zelda* or the more recent *Grand Theft Auto*, first-person shooters (FPSs) such as *Halo*, simulation games such as *Microsoft Flight Simulator*, fighting games such as *Tekken*, some role-playing games such as the single-player *The Elder Scrolls: Oblivion* or the multi-player *World of Warcraft* and some aspects of sports games such as EA's *FIFA* or *NHL*. Usually, such games happen in real time, adding an element of resource-constrained decision making; they are often also games of imperfect information, as the player can not see the whole play-field or level. The complexity of the dynamics of the agent varies wildly, from the grid-world of *Snake* (of Nokia phone fame) to the intricate modelling of wing-tip turbulence of the aforementioned *Microsoft Flight Simulator*,

[1] See Nintendo's *Wii Sports* for an interesting hybrid between computer game and physical game.

but the dynamics is always so complex that the game cannot be played in real time without a computer. Similarly, the complexity of the challenges and the graphical representation vary enormously with the age, budget, target audience and quality of the game.

Agent games are especially well suited for studying questions related to cognitive science, or to testing biologically inspired ideas, especially about perception and sensory-motor coordination. This is because agents in many (not all) agent games can be said to be situated and embodied, according to the definitions discussed above. Essentially, many agent games permit much the same type of experiments to be done as can be done in evolutionary robotics, with the crucial difference that they also permit experiments to be done that could definitely not be done with real robots, lest you have an armada of spaceships to spare. Just how well suited they are will be argued at greater length later in the chapter.

Management games is a bit of a catch-all term for all the games that don't fit into one of the above two categories. In such games, the player does not control a single embodied agent, at least not most of the time. Rather, the task of the player is to devise strategies, allocate resources, schedule events, or just generally solve puzzles that don't directly involve controlling an agent, be they temporal or atemporal. Management games are, like agent games, too complex to be played practically without computers. This category includes real-time strategy games (RTSs) such as *Warcraft*, complex turn-based strategy games such as *Europa Universalis*, god games such as *Sim City* or *The Sims*, some aspects of sports games such as the above mentioned *FIFA* and *NHL*, text-based adventure games such as *Adventure* but also completely unclassifiable outings such as *Wario Ware*.

As this category of games is defined more by what it isn't than by what it is, characterising it in terms of its usefulness is a bit complicated. What can be said is that the complexity of such games makes them rather less suited than computerised games for comparisons of CI algorithms; playing a single turn in *Civilization* takes several orders of magnitude more computational effort than in Chess, the game's difficulty might make it hard for the algorithms to learn anything at all, and the game is so hard to analyse that if one algorithms performs better than another, it might be hard to draw any general conclusions. And like computerised games, management games are usually unsuited for research on motor and perception issues. On a positive note, some management games can be made to accurately reproduce aspects of real-world decision making, so that algorithms can be evaluated on the game for their suitability for solving the real-world problem. A good example of this is Miles and Louis' research on evolving path-planning algorithms and influence maps for controlling simulated forces in a naval RTS game [90]. The U.S. Navy is funding this research, hoping to eventually use the results of it for actual military decision making.

3.1.2 Commercial versus academic game AI

While the academic study of AI for games, and especially CI for games, has a rather short history, computer games have for as long as they existed needed some sort of AI to control non-player characters (NPCs). One would therefore expect a significant overlap between academic and commercial game AI. However, this is not the case: the two fields have different goals and use different types of algorithms. We will here discuss these differences and how the fields could be brought together.

If we exaggerate the differences a little bit, we could say that academic researchers use CI to try to beat games, whereas AI programmers in game companies use their techniques to try to make games more interesting. The reason why the commercial game companies are not very interested in beating their own games is that if you have full access to the game, creating an NPC that plays the game better than any human player is easy - you just have to cheat, i.e. give the NPC some information or abilities that the human doesn't have.

There are exceptions to this, as some games are so complex that the amount and nature of cheating necessary for the NPCs to win over a good human player might be such that it detracts from the game experience. For example, in *Civilization* it is a common complaint (at least on higher difficulty levels) that when waging war against computer-controlled civilisations, military units magically appear in the enemy's cities without the NPC spending the requisite time on building them. Thus, *Civilization* and similar hugely complex games would be among the minority of games where an AI that played the game *better* would have obvious industrial applications.

The motivations that drive AI programmers in the game industry is to create AI that makes the game entertaining, and keeps the players playing, ultimately driving the sales of the game. In listing desirable properties of game AI, the commercial game AI programmer Gilgenbach claims that good NPC AI should be understandable, predictable, consistent, not too fast, and "not take cheap shots" (hide and shoot you in the back etc.) [48]. This goes very well with Koster's view that games are fun when you can understand them gradually; if your opponents are too smart, the game is no fun.

Typically, game companies have a very pragmatic take on how AI is implemented, and anything goes as long as it works in the game; the techniques used to conjure the illusion of intelligence don't need to have anything to do with what academic researchers would recognize as AI or CI. Instead, the NPCs in the majority of games are controlled by finite state machines, lists of conditional statements, and the occasional $A*$ pathfinding algorithm. Typically, no learning is present in either development or execution of the AI. Everything is hard-coded by the human game developers.

So why are CI algorithms, which are undoubtedly more powerful and scalable than finite state machines and lists of conditional statements, hardly ever used in commercial games? Several answers have been given. A common argument is that there are simply not enough computational resources available; CI algorithms use many more processor cycles than the current techniques, and most of the computing power in a modern game goes to producing the pretty graphics. The argument typically goes on to claim that people will stop caring about pretty graphics sometime soon now, and together with new hardware becoming available, this will herald the new era of true AI in games. This argument has been around since the Ancient Geeks, and has not aged well. The Playstation 3 is many orders of magnitude more powerful than the Nintendo NES, but the games for the former console seem not to use much more CI than those for the latter did. And yes, people care just as much about pretty graphics now as they ever did.

A much better answer is that CI algorithms are too unpredictable for using in-game. Evolutionary algorithms, td-learning and backpropagation might deliver fantastic results, but also terrible results. The NPCs might become far too easy or far too hard to beat (or cooperate with), or just start to behave erratically and illogically, ruining the illusion of intelligence and the suspension of disbelief. Further, if NPC players change their behaviour patterns too quickly and too radically, the player might feel that he doesn't understand the game and all his learning efforts are in vain, even if the new NPC behaviour is perfectly logical and understandable in itself. If Koster and Gilgenbach are to be believed, this is very bad game AI design.

If you ask a game developer about the greatest challenge facing them right now, chances are they will not mention AI at all. Instead, they might reply that with all the processing power and storage space now available, games are expected to be bigger and prettier than ever before, and so the problem is producing enough content to fill the game. Developing a major title might require more than a hundred developer-years, and no small percentage of the people employed are artists and level designers. The huge amount of resources needed to develop a game means that fewer major games are profitable. If the CI community could come up with ways to use their techniques to cut down on development costs, especially for level design and artwork, they might get rather more attention from the game industry than they have seen in the past.

Reconciling academic and commercial game AI research

Several academic researchers have addressed this disconnect between academic and industrial game AI research, and given suggestions for how the two communities could learn to interoperate. In an influential article from a few years back, Laird and van Lent argue [76] argue that full-fledged commercial computer games are the ideal application area for human-level artificial intelligence. Laird and van Lent argue that users are driven to online games because of the

failings of current game AI. Gamers want NPCs they can interact in much richer ways than the current crop, and who behave in an overall human-like way. Both academics and game developers would have much to gain from this, as no other AI application areas demand such a broad range of human-level cognitive skills, and as better AI could make more complicated NPCs (that don't rely on cheating) possible, heightening the game experience.

Laird and van Lent are not CI researchers, and so they argue not for evolutionary algorithms and neural networks, but rather for good old-fashioned AI techniques such as planning algorithms and expert systems. As such algorithms are more similar to, and in some cases even rather straightforward extensions of, the techniques that are already in use in commercial games (an expert system can be seen as a list of conditional rules), game developers are probably less reluctant to adopt them. And Laird and van Lent make a good case that full-feldged commercial computer games are good research platforms for academic AI developers, including the essential recognition that something that hundreds of people have developed and hundreds of thousands of people play is not just like any robot simulator, and cannot be claimed to lack validity and abstract away from the challenging issues in the same way. However, they fail to explain how their proposed augmentations of NPCs with better AI would make games more *fun*, and thus translate to higher revenues for game developers.

Baekkelund expresses similar views in a more recent article which focuses on practical issues in collaboration between industry and academia [6]. While convincingly explaining some of the benefits for academic researchers of tapping into commercial game technology, he doesn't explain in detail how more advanced AI would benefit the games industry. Instead, he claims that "Research AI stands to be an important source for sparking innovation within game AI" and that "Games with particularly innovative AI will stand out among competitors". This might will be true, but doesn't help us guide our research.

Turning the issue somewhat on its head, Cowling suggests that we view the writing of AI as a sport in itself [31]. He notes that there has been competitions between AI programmers, where the goal is to write the AI that wins against the other competitors' AIs, for a long time. Fairly recently, such competitions have expanded from computerised games such as Chess into games requiring a broader take on AI, such as Robocup (robot football). Several academics have also written games specifically tailored to AI competition, such as Cellz [77], The Virus Game [32] and Terrarium [89].

Such games are unlikely to ever get a really large following, however, if they rely on the "players" being able to program and being knowledgeable about AI in order to "play" the game. But imagine that the player could teach the NPCs how to behave without actually writing code, but instead through some other means such as demonstrating the correct behaviour to them,

rewarding or punishing them, or maybe just connecting boxes in a colourful interface. Then gamers without any expert knowledge could easily train NPCs to compete with their friends' trained NPCs. This would form a new game genre, heavily dependent on modern CI techniques. (That games based on training NPCs and battling them out against each other can succeed is obvious, looking at the astonishing success (over 100 million games sold!) of the *Pokemon* series, though the creature training there is relatively simplistic and not based on CI algorithms.)

Cowling discusses some games operating at least partially according to these ideas; they will all be discussed in the next three sections. In those sections we also try to subsume the various ideas about commercial and academic game AI expressed above under our analysis of different types of CI in games.

3.2 Optimization

Most academic CIG research takes the optimisation approach. This means that some aspect of a game is seen as an optimisation problem, and an optimisation algorithm is brought to bear on it. Anything that can be expressed as an array of parameters and where success can in some form be measured can easily be cast as presenting an optimisation problem. The parameters might be values of board positions in a board game, relative amount of resource gathering and weapons construction in a real-time strategy game, personality traits of a non-player character in an adventure game, or weight values of a neural network that controls an agent in just about any kind of game. The optimisation algorithm can be a global optimiser like an evolutionary algorithm, some form of local search, or any kind of problem-specific heuristic. Even CI techniques which are technically speaking not optimisation techniques, such as td-learning, can be used. The main characteristic of the optimisation approach is that we know in advance what we want from the game (e.g. highest score, or being able to beat as many opponents as possible) and that we use the optimisation algorithm to achieve this.

3.2.1 Games for computational intelligence: testing machine learning algorithms

There is a wealth of examples of different sorts of computer games being used to test the efficiency of CI algorithms, either through comparing different algorithms against each other or, more commonly, simply through showing that a particular algorithm can be used to learn to play (or optimize the playing of) a particular game. For computerised games, the list is very long indeed; in section 2.4.7 we gave several examples of using both evolutionary algorithms and td-learning to learn evaluation functions for board games, but there are other important

studies using only evolutionary methods, such as Fogel's Blondie24, where evolution learned to play Checkers at a very high level [46].

Several researchers have also used different forms of management games, mainly strategy games, as benchmark problems for their algorithms. E.g. the above cited work of Miles and Louis on evolving path-planners and influence for a naval RTS game [90], but also Spronck's work on evolving construction priorities and other aspects of strategy in the classic RTS *Warcraft* [121].

In agent games, evolutionary computation has been used to tune parameters for hard-coded NPC controllers in the very popular FPS *Counterstrike* [30]; for the 2D space shooter Xpilot, Parker and Parker have evolved both parameters for hard-coded control algorithms [105] that turned out to play very well against standard battle bots and human competitors, and neural networks that control the spaceship on their own [106], closing the sensory-motor loop but playing less well. As an example of a different type of agent game and a different learning algorithm, Graepel et al. applied the Sarsa variety of td-learning to the behaviour learning in the modern commercial fighting game *Tao Feng* with good results [52]. Also worth mentioning is Priesterjahn et al.'s evolution of control for agents in the FPS Quake 3, based on the internal data structure in the game [110].

Several groups of researchers have taken the optimization approach towards racing games. Tanev [131] developed an anticipatory control algorithm for an R/C racing simulator, and used evolutionary computation to tune the parameters of this algorithm for optimal lap time. Chaperot and Fyfe [24] evolved neural network controllers for minimal lap time in a 3D motocross game. And there are other things than controllers that can be optimised in car racing, as is demonstrated by the work of Wloch and Bentley, who optimised the parameters for simulated Formula 1 cars in a physically sophisticated racing game [151] with the objective of lowering lap times, and by Stanley et al., who evolved neural networks for crash prediction in simple car game [125].

Our own work on optimization in three agent games (simulated car racing, Cellz and helicopter control) is discussed in chapter 5; in addition, we have performed elsewhere published experiments in evolutionary solution of Sudoku puzzles, which is a straighforward application of optimization to computerised games [94][93]. It should be noted that the work of Chaperot discussed above was done after our initial publication on simulated car racing, and the work of Tanev was done concurrently. (There are other important differences as well, for example that Chaperot's racing simulator lacks walls so that that vehicle never gets stuck, and that Tanev only evolved parameters for his controller, while we evolved the full controller.)

3.2.2 Computational intelligence for games: optimizing agents and games

None of the above optimization experiments were concerned mainly with improving the entertainment value of games. Further, the games used were games where NPCs or computer strategies can be hand-coded to beat all human players without cheating, or with only relatively non-obtrusive cheating, so the demand from the games industry for CI techniques to play the game better should be small.

The possible exceptions to this are the more complicated games, *Counterstrike* and *Warcraft*, where more experienced players might suspect that the computer players are able to "see" things that they shouldn't be able to see. Partly for that reason, those two games are played more often against other human players than against computer players.

This is not to say that it is impossible to use CI algorithms to make these games more fun. It would be very possible to e.g. evolve game parameters for a game in order to make it more fun, with the fitness function being either reports by human testers, or some sort of entertainment metric. It is also entirely possible that this is already being done in the development phase of commercial games, but that the game developers see no reason to report it in journals or conferences; its apparent absence from common publication venues only point to it not having been tried in academic game research. However, in the Innovation section below, we will discuss attempts to evolve complete control systems according to entertainment metrics.

3.3 Imitation

In the imitation approach, supervised learning is used to imitate some aspect of a game. What is learned could be either the player's behaviour, the behaviour of another game agent, or the dynamics according to an agent moves. While supervised learning is a huge and very active research topic in machine learning with many efficient algorithms developed, this does not seem to have spawned very much research in the imitation approach to CIG.

3.3.1 Games for computational intelligence: testing supervised learning algorithms

Unlike the situation for optimisation and reinforcement learning algorithms, there appears to be very little published research on using games to test supervised learning algorithms. The main reason for this is almost certainly that there are plenty of high-quality datasets for testing supervised learning algorithms freely available already. Using games to test these algorithms

would only be meaningful if what was tested was a version of an algorithm specially modified to work with the game in question, otherwise, one of the available datasets could be used.

One of the few studies in the literature that uses games to compare supervised learning algorithms is the paper by Chaperot and Fyfe discussed above, where a second experiment compared variations of the backpropagation algorithm for learning to drive based on human examples [24].

3.3.2 Computational intelligence for games: modelling behaviour and dynamics

Several commercially successful games have used the imitation approach to enhance gameplay. We have examples from both agent games and a management game in this section; it seems that no-one has explored imitation in computerised games.

An example from management games is the critically acclaimed Black and White by Lionhead Studios. In this game, the player takes the role of a god trying to influence the inhabitants of his world with the help of a powerful creature. The creature can be punished and rewarded for its actions, but will also imitate the player's action, so that if the player casts certain spells in certain circumstances, the creature will try to do the same to see whether the player approves of it.

In racing games, there are actually examples of the imitation approach being taken during both development and on-line during game-play. For the Playstation game *Colin McCrae Rally 2.0*, Hannan (an academic CI researcher turned game developer) trained neural networks to imitate his own driving in order to produce good human-like NPC driving [84]. The neural networks were standard MLPs trained with the Rprop variation of backpropagation, though much time went into choosing the size of the network and the correct set of inputs. However, the neural networks only provided the basics of the driving AI; there was a layer of rules on top, that decided what driving styles to adopt in different situations, e.g. when to overtake a competitor and recover from a crash [18].

An example of using imitation on-line, during the actual gameplay, is the XBox game Forza Motorsport from Microsoft Game Studios. In this game, the player can train a "drivatar" to play just like himself, and then use this virtual copy of himself to get ranked on tracks he doesn't want to drive himself, or test his skill against other players' drivatars. During the development of this feature, a number of approaches were tried, including neural networks, but their performance was unsatisfactory. Instead, a simpler approach was used, where the racing line and speed on each track segment was recorded, and a hard-coded algorithm kept the car as close as possible to the recorded racing line. This method necessitated certain constraints on the track design,

also that crash recovery was handled using another hard-coded algorithm [58].

Moving from racing games to real car driving, Pomerleau's work on teaching a real car to drive on highways through supervised learning based on human driving data is worth mentioning [109]. The neural networks associated human driving commands with features extracted from a forward-pointing video camera, and thus learned to drive rather well on real (American) highways. Pomerleau's reason for using imitation rather than optimisation in this case was probably not that interesting driving was preferred to optimal driving, but rather that evolution or td-learning using real cars on real roads would be costly.

Our own research presented in chapter 6 deals with imitation of both behaviour and dynamics in the context of car racing.

3.4 Innovation

The border between the optimisation and innovation approaches is not clear-cut. Basically, the difference comes down to that in the optimisation approach we know what sort of behaviour, configuration or structure we want, and use CI techniques to achieve the desired result in an optimal way. Taking the innovation approach, we don't know exactly what we are looking for. We might have a way of scoring or rewarding good results over bad, but we are hoping to create lifelike, complex or just generally interesting behaviours, configurations or structures rather than optimal ones. Typically an evolutionary algorithm is used, but it is here treated more like a tool for search-based design than as an optimiser.

3.4.1 Games for computational intelligence: evolving complex general intelligence

In section 2.3 we discussed the problems holding back progress in evolutionary robotics in some detail, especially the problem of finding suitable experimental environments and tasks. In short, evolution on real robots faces the recoverability problem, the problem of computation time, and the problem of high costs of robots and environments. Evolution in simulation faces the problems of simplicity and exploitability: most robot simulators have too simple dynamics, allow only for too simple tasks to be performed, allow only for too simple and low-dimensional inputs to the controller, and have exploitable weaknesses in the sense that evolution can produce behaviour that leads to high fitness due to a flaw in the simulator but would never work on the real robot. These could very well be the real reasons we have not seen any really complex general behaviour, or "real intelligence", emerging from evolutionary robotics simulations.

The innovation approach to games for computational intelligence means taking computer

games, especially complex agent games, seriously as a replacement for robotics in evolutionary robotics. Such computer games have a long list of advantages over real robots or robot simulations, potentially solving all the problems discussed above:

- Compared to fitness evaluation on real robots, evaluation in a computer game can happen much faster, for two reasons: one is that computer games can be run faster than real-time; even the most advanced computer games of today, that brings a top-of-the line desktop down on its knees, will have cycles to spare on next year's computer, and running games from the 1980's on a virtual machine you can get speedups of orders of magnitude. Another is that any game can be run in several replications across a cluster of computers, giving the evolutionary algorithm an ability to evaluate several individuals in parallel at moderate cost.

- Like in robot simulations, recoverability is simply not an issue - just restart the game.

- Unlike robot simulations, there are already very good tasks and associated fitness functions present. If we agree with Koster that good games are good teachers, then a good game is designed so that it is at first easy for a complete to novice to make some progress without dying, but this becomes harder as the game progresses, with more and better enemies and puzzles and more constraints in terms of time, energy etc. Really good games demand not only more of the same skills, but expands the set of demanded skills as the game goes on, providing us with a form of implicit incremental evolution. The fitness function is simple: the score of the game.

- Competitive co-evolution is also well catered for, with many games specially tailored to competitive playing over the internet. Some of these games (such as *Counterstrike* or *Halo*) have huge communities of active players, allowing evaluation against real humans and their sometimes very sophisticated tactics. Many of these games also feature ranking systems, allowing testing against humans of a particular skill level.

- The graphics of modern games, which can hardly even be compared to those of the vast majority of robot simulations, allow for very complex, high-dimensional visual input to the controllers. Further, games are developed so that they are playable using only the information given on the screen; thus there is no problem about selecting what sort of information to use, just how to make the controller understand it.

- In the same way, computer games provide well-defined outputs from the controller: just connect the outputs from the evolved controller to a simulated game controller, joystick or keyboard.

- Another important advantage over robot simulations is that all games are designed so as not to have exploitable weaknesses, and commercial games go through huge amounts of testing to make sure that there are no ways of cheating and getting top score without being able to play the game particularly well. This is a significant benefit when compared to standard robot simulations. (Turning the problem on its head, evolutionary algorithms can be used to identify such weaknesses, thereby automating game testing to some extent, as exemplified by Denzinger et al. working in collaboration with games developer Electronic Arts [37].)

Given all these advantages, it's surprising that not more CI researchers take agent games seriously. A common counter-argument is that computer games are not "real". But how can something which has taken hundreds of developer years to construct, and hundreds of thousands of players spend countless hours on playing not be real? Perhaps it's a question of age; researchers that have grown up with computer games at home are more likely to take them seriously.

One interesting example of the innovation approach to games for computational intelligence is the previously mentioned work of Floreano et al. [43] on evolving active vision in a racing game, work which was undertaken not to produce a controller which would follow optimal race-lines but to see what sort of vision system would emerge from the evolutionary process.

As for the experiments in this thesis, most of the research using the car racing game is part of a project to see how complex and general intelligence can be evolved in such apparently simple games, ranging from the rather straightforward driving behaviour in chapter 5 to the more complex interactions in chapter 7.

3.4.2 Computational intelligence for games: emergence and content creation

The innovation approach can be used to make games more entertaining as well. This can be done for all different types of games, either by improving aspects of existing games or by creating completely new types of games.

An example of improving existing games is Yannakakis and Hallam's work on evolving interesting opponents for a simple version of the classic *Pacman* game [152]. In those experiments, the ghosts (opponent NPCs) were evolved according to a fitness function constructed as a way of measuring Malone's entertainment factors; the efficacy of this method was borne out in surveys of human players.

But what is evolved does not need to be NPC behaviour, but could also be other forms of game content such as levels, environments, puzzles or artwork; in fact, it is quite possible that automatic ways of generating other content is in higher demand in the games industry than

automatic ways of generating behaviour. In section 7.3 we present a method of evolving racing tracks based on player models.

Finally, there is the possibility of creating new genres of games based on evolutionary innovation, e.g. training games as discussed above. An impressive example of this is Stanley et al.'s NERO game. Here, real-time evolution of neural networks provide the intelligence for a small army of infantry soldiers [124] The human player trains the soldiers by providing various goals, priorities and obstacles, and the evolutionary algorithm creates neural networks that solve these tasks. Exactly how the soldiers solve the tasks is up to the evolutionary algorithm, and the game is built up around this interplay between human and machine creativity.

3.5 Summary

In this chapter, we first discussed computer games *per se*, distinguishing between computerised games, agent games and management games. We then discussed various ways in which computational intelligence can be applied to games, identifying three major approaches: optimisation, imitation and innovation. Within each of these approaches we gave examples of how games can be used in CI research, and how CI research can be used to augment games.

While chapter 2 provided the necessary background in computational intelligence methods and research issues to situate the experiments in this thesis, the present chapter aims to provide the same sort of background when it comes to the application of computational intelligence to computer games. However, this chapter cannot claim to be an exhaustive overview of issues and research trends. This is mainly because the CIG research field is still young and far from well-defined. Further, important insights into the nature of games and entertainment maximisation can be found in other academic disciplines, most notable game studies, and in the often non-academic discourse of game development practice. The author willingly concedes to not be well read in these fields. However, as the field of CIG matures, insights from these fields will have to be integrated.

Chapter 4

Games in this thesis

The main experimental environments in this thesis are two simple racing games based on a car racing simulator. However, to demonstrate the applicability of the evolutionary learning methods we discuss here, and to further investigate some of the topics discussed in chapters 2 and 3, some experiments were conducted in different game environments. Both of the additional games, Cellz and helicopter control, are like the car simulation based on agent control in continuous physics-based domains. The experiments conducted in these additional environments are described in less detail in the thesis, and so these domains in themselves are described in less detail than the car racing simulation.

In this chapter, we are not discussing the important issue of which aspects of the environment is available to the controller, nor how these aspects are represented. As how to best sense an environment is one of the issues the research in this thesis aims to throw new light on, it is instead discussed in the methods subsection of the relevant experimental sections.

Also worth noting is that we are in this chapter describing all the experimental environments from a games perspective; both the helicopter control and car racing domains can very well be treated from a robotics perspective as well. In section 6.2 we treat the car racing problem from more of a robotics perspective.

4.1 Simulated car racing

Most of the experiments in this thesis use simulated car racing as their experimental environment. Several versions of a simple car racing simulator were developed specifically for these experiments, and the different versions were refined incrementally during the course of research. Two main types of car racing game was developed using this simulator, track-based racing and point-to-point racing. In this section, we first describe the car dynamics that are common to

both games, and the the rules and special dynamics of each game.

Early on in the project, we were considering using a freely available open source car simulator such as RARS *(http://rars.sourceforge.net/)*, TORCS *(http:// torcs.sourceforge.net/)*, or CarWorld *(http://carworld.sourceforge.net/)* for the experiments. This could possibly have sped up development time and allowed access to good car dynamics and 3D visualization. However, in order to have full freedom to conduct the experiments we wanted, we needed to be able to change any aspect of the simulation, and learning a complex piece of code in enough detail to change what we wanted could well be as complicated as writing it from scratch. Further, while 3D graphics output would be desirable and non-trivial to create ourselves, the dynamics and collision handling present in the available open source offering was not significantly better than what we could create ourselves. Also, most third-party simulations are not built with computational efficiency as a high priority, and computational efficiency is of paramount importance for something that acts as part of a fitness function in an evolutionary algorithm. Most important to our decision to build the simulation ourselves, however, was that we wanted the simulation implemented in pure Java, for maximum portability between platforms.

The car racing simulation is intended to qualitatively, but not quantitatively, replicate racing a small (scale 1/24) radio-controlled (R/C) car on a tabletop racing track, or on a small indoors floor. More specifically, we started with replicating a tabletop track we have set up at the University of Essex, and the feel of driving our 18 cm toy car on it. A computer-controlled R/C racing competition was organized at the 2005 IEEE Congress on Evolutionary Computation (CEC), with three entrants, but only one competitor managed to drive successfully around the track based on the overhead video feed. The difficulty of this problem contributed to our decision to concentrate on car racing in simulation; however, see section 6.2 where we return to the problem of controlling the physical car.

The specific features we wanted to model was that the car would be driving relatively fast for its size, and have poor grip on the surface, easily skidding at high speeds, due to its low mass. For the same reasons collisions would be highly elastic and easily send the car spinning in a hard-to-predict manner.

In the following subsections we will give a number of equations that specify the dynamics and collision behaviour of the cars. Here, any **bold** text signifies a two-dimensional vectorial entity; everything else is scalar. *Italic* text followed by parentheses() signifies a function.

4.1.1 Dynamics

In the simulation, a car is simulated as a $20 * 10$ pixel rectangle, operating in a rectangular arena of size 400×300 or 400×400 pixels. The car's complete state is specified by its position (p),

velocity (v), orientation (θ) and angular velocity ($\dot{\theta}$). The simulation is updated 20 times per second in simulated time, and each time step the state of the car(s) is updated according to the equations presented here. The fundamental equations for the position are:

$$\mathbf{s}_{t+1} = \mathbf{s}_t + \mathbf{v}_t \tag{4.1}$$

$$\mathbf{v}_{t+1} = \mathbf{v}_t * (1 - c_{drag}) + \mathbf{f}_{driving} + \mathbf{f}_{grip} \tag{4.2}$$

c_{drag} is a scalar constant, set to 0.1 in most versions of the simulation.

$\mathbf{f}_{driving}$ represents the vectorial contribution from the motors. The scalar component of the contribution is 4 if the forward driving command is given, 2 if the backward command is given, and 0 if the command is neutral. The vector is obtained by rotating the vector consisting of (the scalar component, 0) according to the orientation of the car.

$\mathbf{f}_{grip}$ represents the effort from the tyres to stop the car's skidding. It is defined as 0 if the angle between the direction of movement and orientation of the car is less than $\pi/16$. Otherwise, it is a vector whose direction is perpendicular to the orientation of the car: $\theta - (\pi/2)$ if the difference is > 0, otherwise $\theta + (\pi/2)$. Its magnitude is the minimum of the velocity magnitude of the car and the maximum lateral tyre traction, which is set to 2 in all versions of the simulation.

Similarly, the fundamental equations for the orientation of the car are:

$$\theta_{t+1} = \theta_t + \dot{\theta} \tag{4.3}$$

$$\dot{\theta_{t+1}} = f_{traction}(f_{steering}() - \dot{\theta}_t) \tag{4.4}$$

$f_{steering}()$ is defined as $magnitude(\mathbf{v})$ if the steering command is left, $-magnitude(\mathbf{v})$ if the steering command is right, and 0 if it is centre.

$f_{traction}$ limits the change in angular velocity to between -0.2 and 0.2.

The behaviour emerging from all this is a car that accelerates faster (and reaches higher top speeds) when driving forwards than backwards, that has a fairly small turning radius at low speeds, but an approximately twice as large turning radius at higher speeds, due to a large amount of skidding at moderate to high speeds.

4.1.2 Collisions

All versions of the car racing problem feature collisions of some sorth, either between cars and walls, between one car and another, or both. These are implemented slightly differently.

Wall collisions

In those versions of the car racing simulation which have tracks defined by walls, the walls are represented as colored pixels in a background image covering the whole track. The color of each pixel encodes the orientation of the wall segment of which it is part; these slopes are stored in a table which is indexed on the color of the pixel. (When a track is created, the walls are usually represented as lines with a 10 pixel width. Tracks are either created by a human designer using a simple graphical tool, or through an evolutionary process, as detailed in 7.3. In both cases, the track is converted to an image-based representation before being used for driving.)

Collision detection is done each time step, by checking each corner of the $20 * 10$ rectangle bounding the corner for intersection with one of the walls. The checking is done by looking at the colour of the pixel at that position in the background image; the extremely low computational overhead of this method is the reason for choosing it over more direct but also computationally more expensive methods based on computational geometry. The collision detection method can have three different outcomes: no intersection is detected, an intersection is detected with one of the corners, and intersection is detected at two corners. Due to the limited speed of the car relative to the frequency of collision checking, intersection with three or four corners is virtually impossible, and if it should occur it is treated as collision with two corners.

Obviously, if no intersection is detected, no collision handling is performed.

If one or two intersections are detected, velocity, angular velocity and position of the car is updated as follows:

$$\dot{\theta} = \dot{(}\theta) + -mag((v)) \tag{4.5}$$

$$\mathbf{v}_{new} = [cos(f_{newdir}()) * \frac{v}{2}, sin(f_{newdir}()) * \frac{v}{2}] \tag{4.6}$$

$$\mathbf{s} = \mathbf{s} + \mathbf{v}_{new} \tag{4.7}$$

In equation 4.5, the magnitude of the velocity is added to the rotational velocity if the intersection is detected at the front-right corner or the back-left corner. It the intersection is detected at the front-left or the back-right corner, the magnitude of velocity is instead **subtracted** from

the rotational velocity.

f_{newdir} is calculated as in different ways depending on where the collision was detected. First, the direction (in the global frame of reference) of v (before the collision) is calculated as $movedir$, and, and the slope of the wall at the point if collision as $wallangle$. (The average of the angles of the two walls is used if intersections with different walls are detected in two corners.) Then, $newdir = movedir + -abs(movedir - wallangle)$. The absolute of the difference is added if the collision is detected at the back left or front right corner, and subtracted otherwise. If a full frontal collision is detected, $abs(movedir - wallangle)/2$ is added, and if a collision is detected at both back corners, this term is subtracted.

The resulting wall collision behaviour has the car, for the most part, "bouncing" off walls with reduced speed and changed orientation, and some angular momentum that keeps influencing the orientation of the car over the next few time steps. Sometimes, the car bounces off the wall in such a way that it can simply continue driving, often it has to back away first, and in a few cases (mostly in corners of the track) it is simply stuck and can not easily get back on track. The actual bouncing is relatively hard to predict, which taken together with the reduction in speed makes it hard to devise a driving strategy that consistently exploits wall collisions in a way that would not work when driving a real R/C car on a tabletop track with wooden walls.

Vehicle collisions

A collision between two cars differs from a collision between one car and a wall in several respects: both cars are (usually) moving, the collision affects both cars, and it is impossible to get stuck in a corner. Vehicle collisions are therefore handled differently from wall collisions.

Each time step, all corners of both cars are checked for collision with the other car; collision checking is done for both cars before collision handling is initiated for any car. This is done simply by checking whether each corner intersects the rotated and translated rectangle defining the other car. The point of collision is calculated as the center of the points of intersection on both cars, and the collision handling method is then called on both car objects. The first thing that happens is that the velocities (both speed and direction) of the two cars is simply exchanged:

$$\mathbf{v}_{this} = \mathbf{v}_{other} \tag{4.8}$$

Then, in order to avoid the otherwise occasionally occuring phenomenon that the two cars "hook up" to each other and keep moving together as a unit, the center of each car is moved a few pixels away from the center of the other car:

$$\mathbf{s}_{this} = \mathbf{s}_{this} + [cos(\Delta_{pos}), sin(\Delta_{pos})] \tag{4.9}$$

$$\Delta_{pos} = \mathbf{p}_{this} - \mathbf{p}_{other} \tag{4.10}$$

How the angular velocity is affected depends on where the point of collision is relative to center of the car and its orientation.

$$\dot{\theta} = \dot{\theta} + -\frac{mag(\mathbf{v}_{other}) + mag(\mathbf{v}_{this})}{2} \tag{4.11}$$

If the corner closest to the point of collision is the front left or back right, the average of this and the other car's velocity magnitude is subtracted from the angular velocity, otherwise it is added. Additionally, this update of the angular velocity is only done in a particular time step if no vehicle collision was detected in the preceding time step.

The resulting collision response is somewhat unrealistical in that the collisions are rather too elastic; the cars sometime bounce away from each other like rubber balls. On the other hand, we reliably avoid the hooking up of cars to each other, and make it possible, though hard, for a skilful driver to intentionally colide with the other car in such a way that the other car is forced off course or into a wall (but also easy for the driver initiating the collision to fail this maneuver, and lose his own direction).

4.1.3 Games

Two games were based on the simulate car racing dynamics: point-to-point racing and track-based racing. Although the track-based racing game was chronologically developed (and put to use as an experimental environment) first, point-to-point racing is somewhat simpler and will therefore be described first.

Point-to-point racing

In this game, one or two cars race in an arena without walls. The objective of the game is to reach as many way points as possible within a set number of time steps, either 500 or 1000. These way points appear within a 400×400 pixel square area; the cars are not bounded by this area and can drive as far from the center of the arena as they want. At any point in time, three way points exist within this area. The first two way points can potentially be seen by the car, but only the first of them can be passed, nothing happens when a car drives through one of the other two way points. These way points are randomly positioned at the start of a trial, so that no two trials are identical. See figure 4.1 for a depiction of the game.

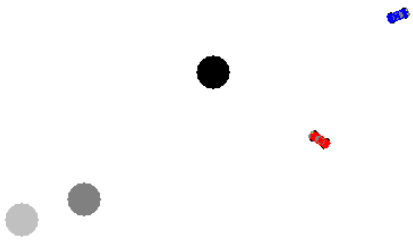


Figure 4.1: Two cars in the point-to-point racing game. The black circle represents the current way point, the dark gray circle the next way point, and the light gray circle the next way point after that.

If the centre of one of the cars comes withing 30 pixels of the first (*current*) way point, this way point is *passed*. The following things happen:

- The passed way points count (the *score*) for that car's driver increases by 1.
- The current way point disappears, the second way point becomes current, and the third way point becomes second.
- A new third way point is generated, by drawing its x and y coordinates from a uniform distribution limited by 0 and 400.

The objective of the game is simply to reach as many way points as possible within the time limits, on a single trial or averaged over several trials. With only one car this mainly comes down to maneuvering the car to the current way point as fast as possible, but some foresight is needed as the way points are not navigated to in isolation. With two cars on the same track considerably more complexity is added. The following is a probably incomplete breakdown of the tasks that need to be mastered, in order of increasing difficulty, in order to win the game in direct competition with a sophisticated opponent:

1. Reach the current way point. This requires not only driving towards the correct way point, but also not overshooting it. A driver that constantly accelerates (issues the forward command) while steering in the direction of the way point will end up "orbiting" the way point at one time or another, as the grip of the tyres is limited and decreases with speed.
2. Reach the current way point fast. While it is possible to solve the preceding task by just driving very slowly (through only accelerating when the speed of the car is below a low

threshold) and steering in the direction that the angle between the direction of the car and the way point is smallest, it would obviously take a long time to reach the way point that way. As a driver that on average reaches way points faster will win over a slower driver, it is necessary to drive at as high speed as possible without overshooting the way point.

3. Reach the current way point in such a way that the next way point can be reached quickly. For example, it is usually a bad idea to reach the current way point going at full speed away from the next way point, which will soon be the current way point. In such cases, it often pays off to start braking before reaching the current way point, and trading off a slight delay in reaching that way point with being able to reach the next way point much faster.

4. Predict which car will reach the current way point first. If a driver can predict whether his car or the opponents' car will reach the next way point first, he can take appropriate action. An obvious response is aiming for the next way point instead, and waiting for it to become the current way point; in that way he only misses one way point, instead of risking to miss two way points. However, knowing for sure which car will reach the current way point first involves understanding not only the dynamics of the car simulation, but also the behaviour of the other driver. What if he decides that he is not going to make it first to the current way point, and goes for the next way point directly himself?

5. Use collisions to ones advantage. There are situations when it is advantageous to be able to knock the opponent of course. For example in case the opponent would get to the current way point first, due to higher speed, but it is possible to get in the way of the opponent and collide with him so that he misses the way point. It could even be possible to "steal" some of his velocity this way, and use it to get oneself to the current way point faster. Conversely, if the opponent has figured out how to disrupt one's path through intentional collisions, it becomes important to learn how to avoid such collisions.

Track-based racing

The track-based racing game introduces tracks and walls. A track consists of starting points for one or two cars, impermeable walls, and a way point chain. The goal of this game is to get as many laps around the track as possible within a given amount of time, usually 700 time steps, in one trial or averaged over several trials. Progress is measured by the number of way points passed. Each passed way point awards a score equal to $1/n$ where n is the number of way points; passing all the way points constitutes one lap and awards score 1, whereas in most of the tracks used in this thesis it is this possible to reach scores of between 2 and 3 within the given time.

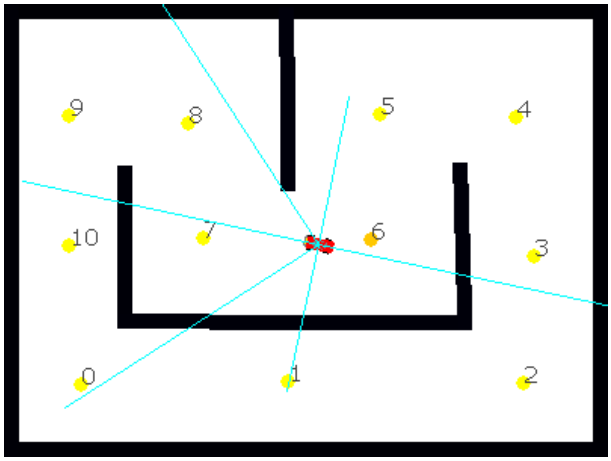


Figure 4.2: A single car on one of the easier tracks in the track-based racing game. The circles represent way points and the lines protruding from the car represent wall sensors, as detailed in sections 5.1 and 5.2.

Unlike in point-to-point game, in the track-based game more way points are used (often between 5 and 10), and their positions are (like those of the walls) fixed for the duration of the trial, and except for the experiments in section 7.3 also between trials. Like in the point-to-point game, only the current way point can be passed at any one time, and the next way point automatically becomes visible when the current way point is passed. When the last way point is passed, the first way point in the chain becomes current again. In the tracks used in this thesis, the way points are laid out so that driving around the track in the middle of the track will mark all way points as passed.

At the beginning of a trial, each car is positioned very near one of the starting points (the starting points are very close to each other) pointing in the direction specified by that starting point, which is also roughly the direction the track is intended to be driven. Very small random deviations to both heading and position are here introduced, so that no two trials start with the car(s) at exactly the same position and orientation.

The track-based racing task offers slightly different challenges than the point-to-point task. On one hand it can be seen as harder, as the presence of the walls restrict possible paths. On some tracks, it is not possible to follow a straight line between two way points, as the car would end up colliding with a wall. On the other hand, the walls provide some guidance as to which is the right way to take, and it is potentially possible to drive some (all?) tracks without relying on way points at all.

Another difference is that because the walls and way point chain are fixed, it becomes possible to learn good paths for specific tracks; in theory, a controller could perform optimally on a particular track but fail to make a single lap on other tracks. When, and to what extent,

a controller would actually specialize on a particular track or evolve general driving skills will be investigated in the experimental chapters.

As in the point-to-point game, having two cars competing against each other on the same track brings additional challenges, in that blocking, overtaking and avoiding the opponent now becomes possible (and often mandatory). The presence of walls, however, makes the collisions potentially more hazardous, as it is possible to be pushed into a wall and lose much valuable time; some collisions may mean no more progress at all on the track if the driver is not able to recognize that he is stuck, back away, and resume driving in the right direction.

4.2 Other games

The following two games are used in experiments that are described in less detail in the thesis, and as such they are described in less detail themselves.

4.2.1 Simulated Helicopter Control

One of the experiments in this thesis concerns controlling a single-rotor helicopter in simulation. Although the helicopter simulation, and the waypoint-following task to be performed in it, is here treated as a game, and the research as being about how to best evolve game agent control, it was originally undertaken as part of the UltraSwarms project. This project, initiated by Owen Holland and with Renzo De Nardi responsible for system development and scientific experimentation, aims at producing a group of autonomous micro helicopters, that move together in a swarm-like fashion and continuously exchange data and collaboratively process complex information about their environment through grid computing. The hardware platform chosen for this is a commercially available Hirobo model helicopter, with its electronics completely replaced by a custom-designed circuit board, a microcomputer running Linux, a bluetooth chip and an inertial measurement unit. As acquiring a good model of the helicopter is both dependent on all the hardware being finished and properly tested, and rather difficult in itself, the decision was taken to start investigating how to develop the controller for the helicopter in parallel with developing the hardware, using a freely available open source helicopter simulator.

The simulator in question is the Autopilot software suite, which simulates a XCell 60 model helicopter with a high degree of accuracy *(http://autopilot.sourceforge.net)*. The computation of rotor thrust and drag forces is done using blade element theory, and rotor and stabilising bar dynamics are modelled in detail, as proposed by Mettler et al. [87]. Update of the simulation is done at 100 Hz; current state of the helicopter is fed to the controller, and control signal fed to the helicopter, at 50 Hz. While the XCell 60 helicopter differs from the hardware plat-

form in development for the UltraSwarms in several ways, there are also important similarities. Most importantly the complexity of the dynamics is of the same order. Both the real Hirobo helicopter and the simulated XCell helicopter are virtually unflyable in the absence of assistive control mechanisms (most importantly yaw stabilization) or proper education in helicopter flight. Whereas almost anyone can drive a real or simulated R/C car in a straight line and stop, extensive training is required to be able to take off, fly in a straight line, and land in the Autopilot simulator or using the R/C helicopter it simulates. In this respect, the simulated helicopter control domain is more challenging than the simulated car racing domain.

Basing a challenging game on this simulation is relatively straightforward, as the simulation itself is so challenging. The task is basically an extension of the point-to-point car racing task to three dimensions. At the beginning of each trial, which lasts for 1000 time steps, a new way point chain is generated consisting of way points with a mean distance of 17.5 feet between them. As in the car racing games, only the current way point can be visited at any time, and a way point is deemed visited if the centre of the helicopter was within 1 foot of the centre of the helicopter. The basic fitness function is simply the number of way points visited per trial, although other fitness functions that took into account the mean deviation from the shortest path between the way points were used in some experiments.

The helicopter control game is included in this thesis in order to show that controllers can be developed through simulated evolution in much the same way as for the simulated car racing and Cellz games, even though the dynamics of the agent is considerably more complex, as is the dimensionality of inputs to the controller. A second reason for including it is that the experiments performed using it demonstrate and corroborate some of the earlier discussed points about modularity and incrementality in evolution of neural networks.

4.2.2 Cellz

Cellz is a simple dynamical game designed specifically as a test bed for evolutionary algorithms [77]. Like simulated car racing and helicopter control it is a game which takes place in a physics-based environment simulated in discrete time, where the goal involves reaching certain points in space within a specified time. Unlike those domains, however, the commands returned from the controller are continuous rather than discrete, there are many potential points to reach at any one time, and typically there is a whole bunch of agents active simultaneously rather than just one or two. A screen shot from a game of Cellz with movement traces of the individual agents is shown in figure 4.3.

Imagine an ooze full of single-celled organisms on the hunt for food particles. This is the central metaphor for Cellz, which was designed with simplicity of the rule-set in mind, yet

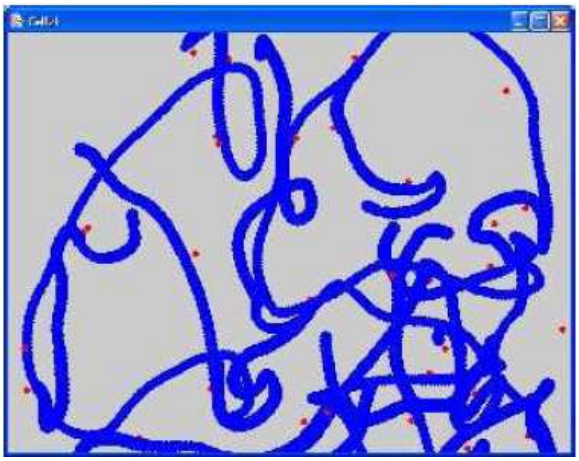


Figure 4.3: An evolved perceptron controller playing Cellz, with movement traces on.

allowing for complex dynamics to emerge. Each agent is circular, and moves about on a two-dimensional area. Within this area, ten food particles are scattered randomly. The agents each have an energy level, which can be used for accelerating in any direction; each time-step an agent's controller specifies a two-dimensional force vector. When the energy reaches zero, the agent dies, but if it doesn't move, it doesn't use any energy. When an agent reaches (comes close enough to) a food particle, two things happen: the agent's energy level increases, and the food particle disappears, only to instantly reappear at a random place in the (bounded) game area. If the energy level of the agent exceeds a certain limit, it splits into two agents, each with half the original energy level.

A game (trial) of Cellz lasts for 2000 time steps, and the score (fitness) is simply the *total energy level* at the end of the trial. In other words, high scores are reached by eating as much food as possible, and dividing into as many agents as possible, while not moving unnecessarily.

So, how does one play Cellz, and wherein lies the difficulty? If only one agent was present, the problem would be similar to a travelling salesman problem in that a number of points have to be reached while moving a minimal distance, but with the added complication that a new point appear at a random position whenever a point is reached, and that momentum and friction has to be taken into account, so continued acceleration is necessary to keep a given speed, but it's desirable to not reach a way point at too high a speed, or you will take longer time reaching the next way point. When more than one agent is present, things get more complicated still. One of the main complications is that it is now desirable to avoid going for food particles that another agent is going to get to first.

In the version of the game that is used for the experiment in this thesis, the same controller is used to control all the agents in a game - the sensors of each agent are input to the controller,

and the movement commands for that agent are output, in turn - so the controller acts as a "hive-mind". However, it is possible to envision versions of the game where each agent has its own controller, and evolution is done "in situ" as a new controller is created as an offspring of the old controller each time an agent divides. Other potential extensions involve multiple species where high-energy agents of one species can eat low-energy agents of another species, etc.

The Cellz game is included in this thesis partly to show that learning controllers through simulated evolution is feasible for multi-agent games as well, but mainly because the experiment that uses this environment nicely illustrates the usefulness of convolutional modularity.

Chapter 5

Optimization

In this chapter we present six series of experiments that take the optimization approach to CIG. The two first of these are presented in detail, while the four others are more summarily discussed. The first four of the experimental series concern simulated car racing, and concern (in order of appearance) the evolution of controllers for single cars on single tracks, scaling up to several tracks, comparisons between neural networks and genetic programming (with and without state capabilities) and comparisons between evolution and temporal difference learning. The remaining experiments concern evolution of control for Cellz and for simulated helicopter flight.

5.1 Racing single cars on single tracks

This section, based on a paper presented at CEC 2005 (where it won the Best Student Paper Award), details our first experiments with evolving control for simulated car racing [142]. The goal was to investigate whether good car racing control could be evolved at all, and if so, how the controller and its inputs should be represented in order to be evolvable.

The experiments in this section take place the track-based racing game, but using the first version of the car racing simulator, which is somewhat simpler than the one described in section 4.1. As can be seen from the figures in this section, the graphical representation is simpler, but the dynamics and the collision detection and handling are also a bit simpler. In the case of a single car on a single track, however, this should not make any qualitative difference for the results. Another difference is that trials last for 500 generations instead of 700, and that being able to complete one lap in that time yields a fitness of 5 (the number of way points) rather than 1. The results here are thus not quantitatively comparable to those in later sections.

All the experiments in this chapter used a 50+50 ES as described in section 2.1. Five different

controller architectures were tested: action sequences, open-loop neural networks, force fields, neural networks with third-person ("Newtonian") inputs, and neural networks with first-person inputs. Further, each controller was tested both with and without small random perturbations to the starting position and starting orientation.

5.1.1 No inputs and action sequences

Methods

An action sequence is a one-dimensional array of length 500, containing actions, represented as integers in the range 0-8. An action is a combination of driving command (forward, backward, or neutral) and steering commands (left, right or center). When evaluating an action sequence controller, the car simulation at each time step executes the action specified at the corresponding index in the action sequence. E.g., at time step 0 the action specified at position 0 was taken, and time step 73 the action specified at position 73, and so on. At the beginning of each evolutionary run, controllers are initialized as sequences of zeroes. The mutation operator then works by selecting a random number of positions between 0 and 100, and changing the value of so many positions in the action sequence to a new randomly selected action.

As stated in the beginning of this section, the fitness function was the same in this experiment as in all other experiments in this section, the total progress measured as a continuous approximation of the number of way points passed. Similarly, the EA used in all experiments in this section is a $50 + 50$ ES as described in section 2.1.

Results

After evolving the action sequence controllers for 100 generations, most evolutionary runs reached a fitness of about 2; after 500 generations, they often reach about 5. The resulting behaviour indeed looks more like more or less random actions that just happen take the car in the right direction than it looks like good driving. The car drives very slowly, and many evolved controllers spend considerable amounts of time standing virtually still before finally starting to move. We hypothesize that a major factor restraining fitness growth is the ubiquity of local optima in the early parts of the sequence. This comes about because each action is associated with a time step rather than a position on the track, so that a mutation early in the sequence that is in itself beneficial (e.g. accelerating the car at the start of a straight track section) will offset the actions later in the sequence in such a way that it probably lowers the fitness as a whole, and is thus selected against.

Under the randomized starting point regime, fitness is often below two and does not rise much further. Analysis of evolved controllers shows that the car often gets stuck on walls. A

plot of fitness evolution for both the fixed and random starting points is shown in Figure 5.1.

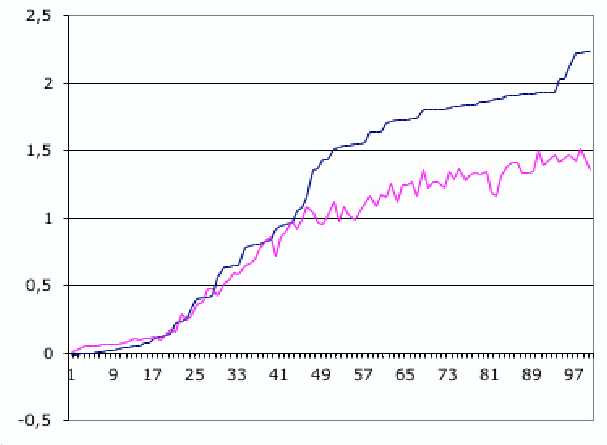


Figure 5.1: Evolving action sequences. The upper graph represents the fitness of the best individual in each generation, averaged over 10 evolutionary runs, under the fixed starting point regime. The lower graph represents the same entity when the car was evolved with randomized starting positions.

5.1.2 Open-loop neural network

Methods

A standard MLP as described in section 2.2.1, with two inputs and five hidden nodes, is fed with the number of the current time step divided by 500, yielding an input value of 0 in the first time step and 1 in the last, and a constant input with the value 1.

Results

After 100 generations of evolution, the controller typically reached fitness levels of about 2 to 3. The car behaviour looks no less random than that of the action sequence controllers, the main difference is that the car goes faster in this case. Most evolutionary runs found a way for the car to accelerate into the walls at the right angle and speed to bounce it's way around little more than half of the track, but none got further. An analysis of the actions produced by the controller reveals that the controller issues the same action for several hundred time steps in a row, and only changes action once or twice per trial. At the moment we don't know why higher-fitness controller refuse to evolve.

When evolving with randomized starting points it seems to be impossible to find a behaviour sequence that relies on bouncing off walls in the right way, and so evolved controllers tend to just run around in circles and reach very low fitness levels, barely above zero. The fitness evolution of this type of controller is shown in Figure 5.2.

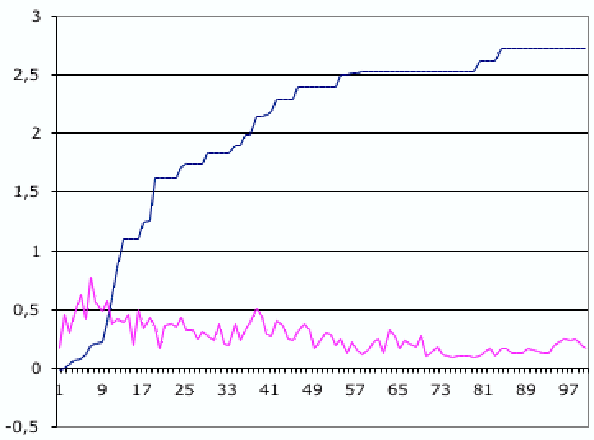


Figure 5.2: Evolving open loop neural network controllers. The upper line is for a fixed starting position, the lower line for randomised starts.

5.1.3 Newtonian inputs and force fields

Methods

A force field controller is here defined as a two-dimensional array of two-tuples, describing the preferred speed and preferred orientation of the car while it is in the field. Each field covers an area of $n*n$ pixels, and as the fields completely tile the track without overlapping, the number of fields are $(l/n)*(w/n)$, where l is length, and w is width of the track, respectively. At each time-step, the controller finds out which field the centre of the car is inside, and compares the preferred speed and orientation of that field with the cars actual speed and orientation. If the actual speed is less than the preferred, the controller issues an action containing a forward command, otherwise it issues a backward command; if the actual orientation of the car is left of the preferred orientation, the issued action contains a steer right command, otherwise it steers left. In the results reported here, we used fields with the size 20×20 pixels, evolved with gaussian mutation with magnitude 0.1, though we have tried other combinations of field size and mutation magnitude without any improvement in fitness. This is broadly similar to the kind of controllers evolved in [64], though we are controlling a car rather than a holonomic robot.

Results

The force field controllers evolved very slowly, and after 100 generations barely exceeded fitness 1; evolving for 1000 generations sometimes brought fitness up to around 4 when using fixed starting positions; when starting positions were randomised, fitness stayed at 1. The cars moved around in a peculiar fashion, sometimes following a sane path around the track for a while, only to become stuck oscillating between two force fields a moment later. Figure 5.3 shows the fitness

evolution of this type of controller, and Figure 5.4 shows a sample trace.

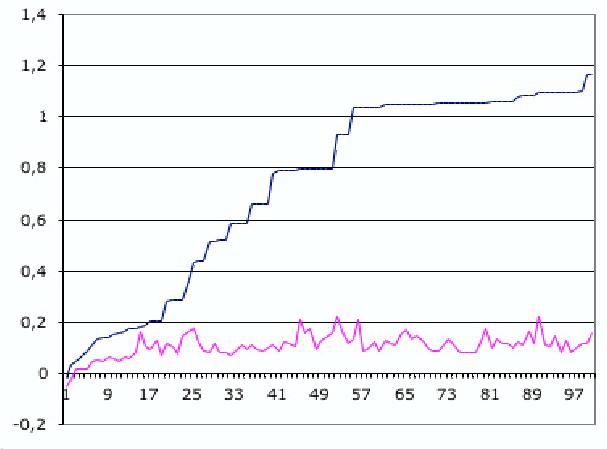


Figure 5.3: Evolving force field controllers.

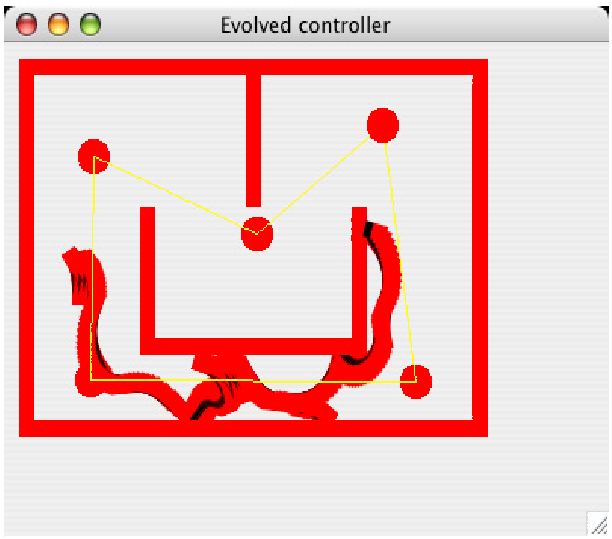


Figure 5.4: Movement trace of a car controlled by a force field controller.

5.1.4 Newtonian inputs and neural networks

Methods

The neural network is fed seven inputs: a constant input with value 1, the x and y components of the car's position, the x and y components of its velocity, its speed and its orientation. All inputs are scaled to be in the range -10 to 10. Seven hidden neurons are used in the network, and the two outputs are interpreted as described above.

Results

Evolving for 100 generations, best fitness varies considerable between evolutionary runs. While most runs produced controllers with fitness values around 3, at least one run produced a controller with fitness over 6. None of the controllers manage to the driver the car properly around the track however, the fittest controller instead drove the car into the"box" in the center of the track and exploited a glitch in the fitness function, whereby it can come close enough to the aim points on the left side of the track for the fitness function to increase without the car ever going around the left wall of the box. The cars drive fast and seem to make sensible turns, but they all eventually get stuck on a wall.

Randomising the starting position produces controllers of slightly lower fitness. Figure 5.5 shows the fitness evolution of this type of controller, while Figure 5.6 shows a sample trace of the car.

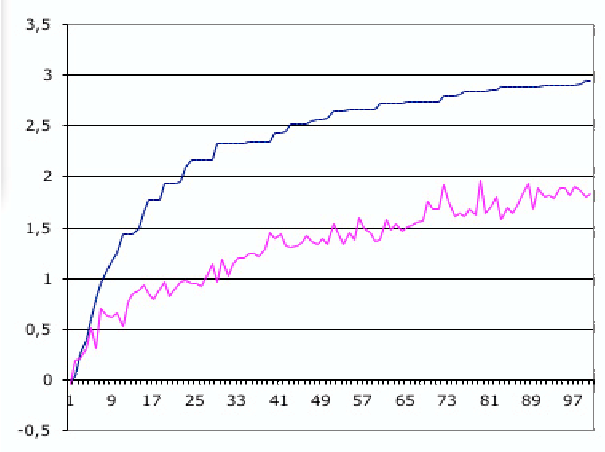


Figure 5.5: Evolving newtonian neural network controllers.

5.1.5 Simulated sensor inputs and neural networks

Methods

In this experimental setup, the six inputs to the neural network consist of one constant input with the value 1, the speed of the car, and the outputs of three wall sensors and one aim point sensor. The aim point sensor simply outputs the difference between the car's orientation and the angle from the center of the car to the next aim point, yielding a negative value if that point is to the left of the car's orientation and a positive value otherwise.

Each of the three wall sensors is allowed any forward facing angle (i.e. a range of 180 degrees), and a reach, between 0 and 100 pixels. These parameters are co-evolved with the neural network of the controller in the following manner: a genome consists both of the specification of the neural

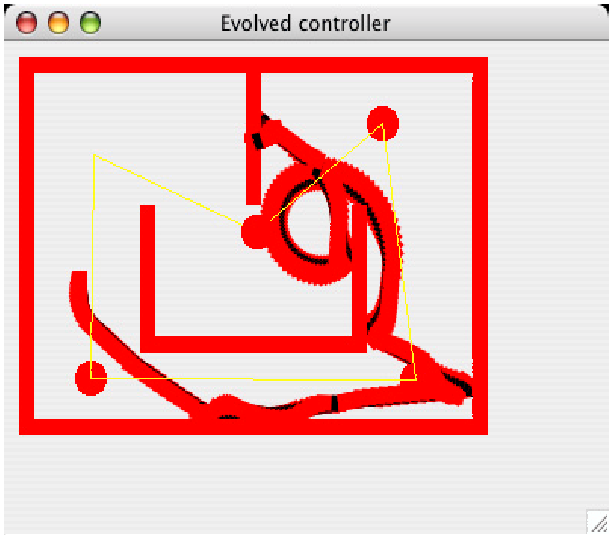


Figure 5.6: Movement trace of a car controlled by a neural controller with newtonian inputs.

network, and of the parameters of the sensors ($3 * 2 = 6$ scalar values). All sensor parameters are between 0 and 1, and are multiplied by Π and 100 to give the angle and reach, respectively. Mutation works in the same way on both neural networks and sensor parameters, adding a value drawn from a Gaussian distribution with mean 0 and standard deviation 0.1 to each value among both network weights and sensor parameters.

The sensor works by checking whether a straight line extending from the centre in the car in the angle specified by that sensor intersects with the wall at eleven points positioned evenly along the reach of the sensor, and returning a value equal to 1 divided by the position along the line which first intersects a wall. Thus, a sensor with shorter reach has higher resolution, and evolution has an incentive to optimize both reaches and angles of sensors. This type of sensor controller is related to the wrap-around vector histogram approach of [13], except that we are only using three sensors instead of full wrap-around.

Results

After 100 generations, evolution produced a controller with excellent fitness values, which equal more than three laps around the track in the allotted 500 time steps. The cars drive around the track at close to full speed, cutting corners incredibly close, crashing into walls only where they can take advantage of the rebound. The sensors vary considerably in the combination of angles and ranges, though often show a bias towards straight ahead and left. An example evolved sensor configuration is shown in Figure 5.7, which uses the short left sensor to help follow the inside wall, or take close cut corners, and the longer range sensors to help decide when to turn.

The best final fitness value found when examining ten evolved controllers was 16.04, which

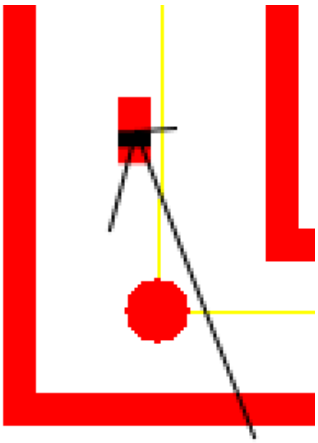


Figure 5.7: A sample evolved configuration of sensors.

narrowly beats the best human competitor so far. When evolving with randomised starting positions best fitness was slightly lower, with a similar sensor setup. The lower fitness is due to the cars slowing down in corners. Figure 5.8 shows the fitness evolution of this type of controller, while Figure 5.9 shows a sample trace of the car's movement.

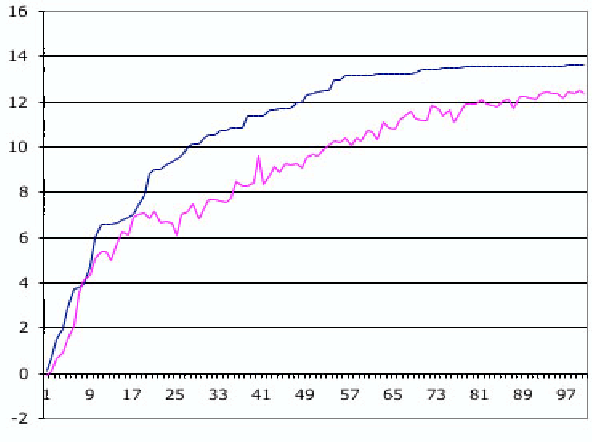


Figure 5.8: Evolving sensor-based neural controllers with full inputs.

To investigate the relative contributions of the wall and aim point sensors, we "lesioned" the controller by disabling the sensor types one at a time. First, we disabled the wall sensors, and evolved controllers making use only of the aim point sensor, speed and the constant input. Under the fixed starting point regime this resulted in cars with fitness often between 11 and 12; they drove well, but bumped into the wall protruding from the top of the track once every lap. When randomizing starting points, the aim sensor-only controller fared much worse, reaching medium fitness about 7. Evolution produced a controller that sometimes made its way around the track, but, depending on initial conditions, more often got stuck on a wall. Figure 5.10

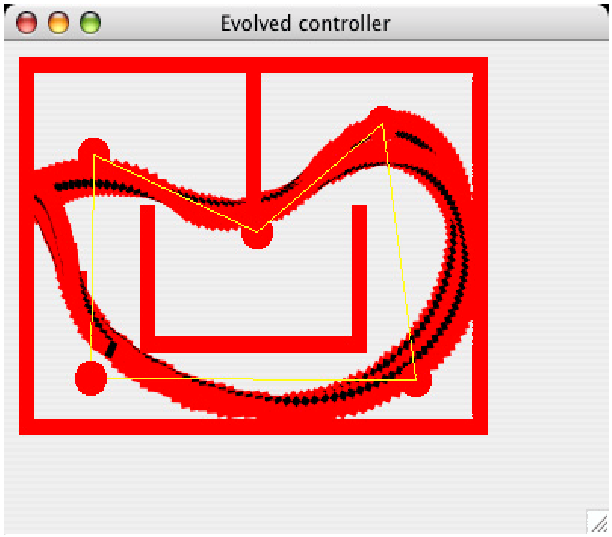


Figure 5.9: Movement trace of a car controlled by a sensor controller.

shows the fitness evolution under this restriction.

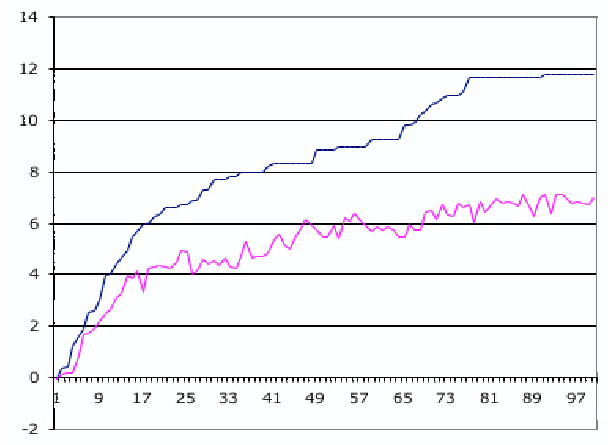


Figure 5.10: Evolving sensor-based neural controllers without wall sensors.

We then re-enabled the wall sensors and instead disabled the aim point sensor. Under the fixed starting point regime, evolution produced controllers that often had all wall sensors set long range, and pointing approximately 20 degrees left of straight ahead, and that drove at high speed, more or less following the outer wall. When randomizing starting points, they often have a long range sensor pointing straight forward, and medium range sensors pointing approximately 45 and 90 degrees to the left, and execute careful following of the outer wall without ever bumping into it.

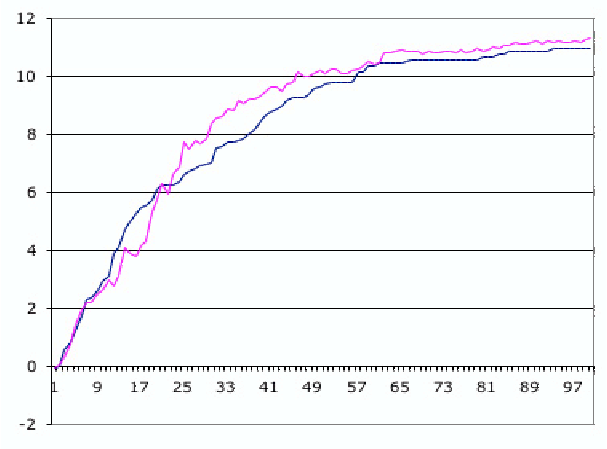


Figure 5.11: Evolving sensor-based neural controllers without aim point sensors.

Controller	Fixed	Randomized
Action sequence	2.23	1.36
Open loop neural	2.72	0.17
Force field	1.16	0.16
Newtonian neural	2.94	1.84
Sensor-based	13.59	12.4
No wall sensors	11.76	7.02
No aim point sensor	10.97	11.33

Table 5.1: Average fitness of best individuals of 10 evolutionary runs of the various controller architectures under the starting point regimes.

5.1.6 Conclusions

The most consistent effect across all the experiments reported above is that controllers (except the sensor controller with the aim point sensor disabled) have higher fitness when starting position and orientation is kept fixed. Not surprisingly, evolution is able to optimize car behaviour better in these noise-free cases, and has to develop more robust behaviour (which means longer lap times) or risk getting stuck on a wall when starting position is randomized. When racing actual physical cars, starting position will necessarily vary, and so performance under this regime is the more interesting factor when evaluating the suitability of a controller for transfer to a physical domain.

Our experiments also point to the vast superiority of first-person to third-person information for the problem at hand. Especially, the simulated range-finder sensors have turned out to be a very powerful device. This might be because of the existence of walls, which presumably makes any mapping from third-person spatial information to appropriate first-person actions extremely nonlinear. A controller using third-person information (such as visual data from an overhead web camera) could get around this problem by somehow representing the walls, and re-creating the kind of sensors used in our sensor-based simulations described above. The problems with using third-person information might also partly be due to difficulty of rotating coordinates as the car's orientation changes.

We were somewhat surprised by the poor performances of the action sequence and force field controllers, both of which should theoretically be able to represent good solutions, at least for fixed starting points. We therefore hypothesize that the poor performance is because of problems with the evolutionary algorithm rather than the representations per se; our main culprit here is the mutation methods, which seem to drive the action sequence into local optima and make for very slow progress in force field evolution. For a fixed starting position, it should be possible to achieve good lap times by seeding the EA with an action section observed by running an evolved sensor controller, but we've not yet tried this.

Regarding force field controllers, an alternative hypothesis is that it is very hard (or even impossible) to successfully drive a non-holonomic vehicle around a track using that method, as it does not take into account the state of the car when entering a particular cell (this is not a problem for holonomic vehicles, which can be treated as stateless).

5.2 Scaling up to multiple tracks

After finding out that very good car racing control could indeed be evolved, and that neural networks combined with simulated range-finder, way point and speed sensors where the way to

go, the next step was to investigate how well this approach scaled up. In this section, which is based on a paper presented at CEC 2006, we first devise a set of racing tracks of differing difficulty, evolve controllers for each track in turn, investigate different approaches to evolving controllers that can race more than one track, and finally optimize such generally proficient controllers for driving particular tracks.

The concrete questions we pose and try to answer are the following: How robust is the evolutionary algorithm, that is, how certain can we be that a given evolutionary run will produce a proficient controller for a given track? Is the layout of the racing track directly influencing the fitness landscape so that some tracks are much harder than others to evolve, while not being impossible to drive? What is the transferability of knowledge gained in evolving for one track in terms of performance on other tracks? Can we evolve controllers that can proficiently race all tracks in our training set? How? Can such generally proficient controllers be used to reliably create specialized controllers that perform well, but only on particular tracks? Finally, can this be done even for tracks for which it is not possible to evolve a good controller from scratch?

These experiments were performed in the track-based racing game using the full car racing simulator described in section 4.1, with one crucial difference: at the time of doing the experiments there was a bug in the way point sensor, so that it was impossible to tell the difference between way points behind and in front of the car. This bug was discovered well after the results were written up and published, but in hindsight it can actually be viewed as a feature. Its existence in no way invalidates the results, as it proves that the methods work even with a slightly incapacited sensor. (The main way it affects the controller is that it can lose direction and start driving backwards on the track under some circumstances.) In section 5.3.1 we repeat some of the experiments in this section with a fixed way point sensor, and the results there are qualitatively very similar though quantitatively somewhat better.

Like in the preceding section, a $50 + 50$ ES was used as the evolutionary algorithm for the experiments in this section.

5.2.1 Methods

For the experiments we have designed eight different tracks, presented in figure 5.12. The tracks are designed to vary in difficulty, from easy to hard. Three of the tracks are versions of three other tracks with all the waypoints in reverse order, and the directions of the starting positions reversed.

The neural networks controlling the cars have nine inputs: one bias input with the value 1, one speed input, one input from the waypoint sensor, and six inputs from wall sensors. All networks have two outputs, which are interpreted as driving commands for the car.

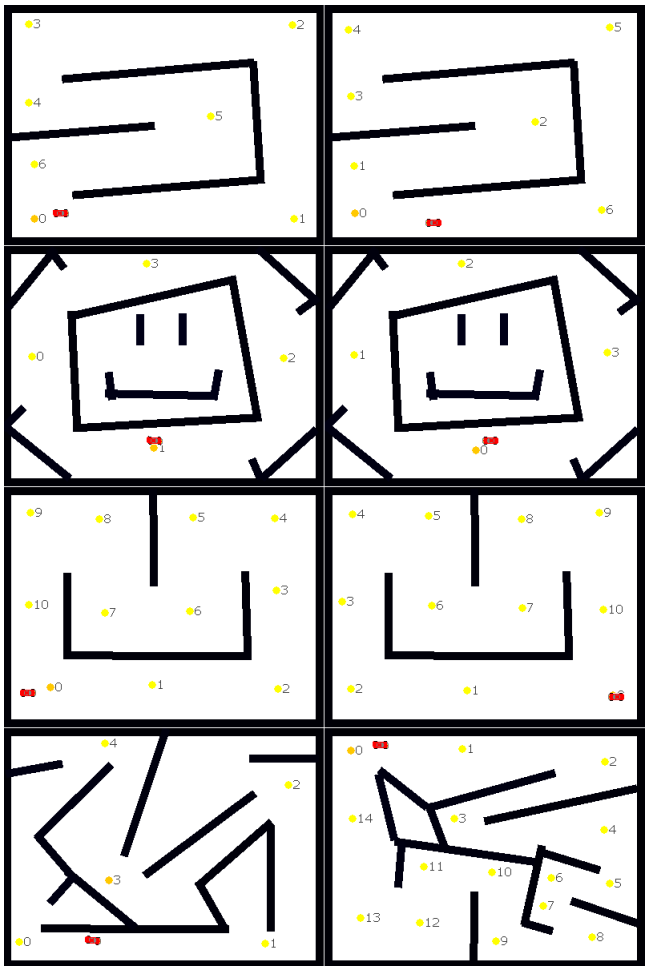


Figure 5.12: The eight tracks. Notice how tracks 1 and 2 (at the top), 3 and 4, 5 and 6 differ in the clockwise/anti-clockwise layout of waypoints and associated starting points. Tracks 7 and 8 have no relation to each other apart from both being difficult.

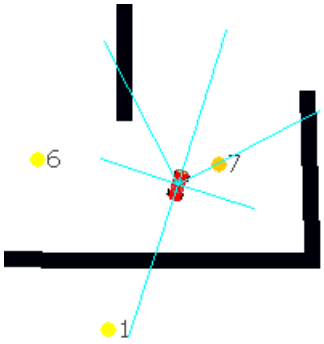


Figure 5.13: The initial sensor setup, which is kept throughout the evolutionary run for those runs where sensor parameters are not evolvable. Here, the car is seen in close-up moving upward-leftward. At this particular position, the front-right sensor returns a positive number very close to 0, as it detects a wall near the limit of its range; the front-left sensor returns a number close to 0.5, and the back sensor a slightly larger number. The front, left and right sensors do not detect any walls at all and thus return 0.

The wall sensors are slightly updated versions of the sensors used in our previous experiment. Each sensor has an angle (relative to the orientation of the car) and a range, between 0 and 200 pixels. The output of the wall sensor is zero if no wall is encountered along a line with the specified angle and range from the centre of the car, otherwise it is a fraction of one, depending on how close to the car the sensed wall is. A small amount of noise is applied to all sensor readings, as it is to starting positions and orientations.

In some of the experiments the sensor parameters are mutated by the evolutionary algorithm, but in all experiments they start from the following setup: one sensor points straight forward (0 radians) in the direction of the car and has range 200 pixels, as has three sensors pointing forward-left, forward-right and backward respectively. The two other sensors, which point left and right, have reach 100; this is illustrated in figure 5.13.

5.2.2 Evolving track-specific controllers

The first experiments consisted in evolving controllers for the eight tracks separately, in order to the test the software in general and to rank the difficulty of the tracks.

For each of the tracks, the evolutionary algorithm was run 10 times, each time starting from a population of "clean" controllers, with all connection weights set to zero and sensor parameters as explained above. Only weight mutation was allowed. The evolutionary runs were for 200 generations each.

Fixed sensor parameters: Evolving from scratch

The results are listed in table 5.2, which is read as follows: each row represents the results for one particular track. The first column gives the mean of the fitnesses of the best controller

Track	10	50	100	200	*Pr.*
1	0.32 (0.07)	0.54 (0.2)	0.7 (0.38)	0.81 (0.5)	2
2	0.38 (0.24)	0.49 (0.38)	0.56 (0.36)	0.71 (0.5)	2
3	0.32 (0.09)	0.97 (0.5)	1.47 (0.63)	1.98 (0.66)	7
4	0.53 (0.17)	1.3 (0.48)	1.5 (0.54)	2.33 (0.59)	9
5	0.45 (0.08)	0.95 (0.6)	0.95 (0.58)	1.65 (0.45)	8
6	0.4 (0.08)	0.68 (0.27)	1.02 (0.74)	1.29 (0.76)	5
7	0.3 (0.07)	0.35 (0.05)	0.39 (0.09)	0.46 (0.13)	0
8	0.16 (0.02)	0.19 (0.03)	0.2 (0.01)	0.2 (0.01)	0

Table 5.2: The fitness of the best controller of various generations on the different tracks, and number of runs producing proficient controllers. Fitness averaged over 10 separate evolutionary runs; standard deviation between parentheses.

of each of the evolutionary runs at generation 10, and the standard deviation of the fitnesses of the same controllers. The next three columns present the results of the same calculations at generations 50, 100 and 200, respectively. The "Pr" column gives the number of proficient best controllers for each track. An evolutionary run is deemed to have produced a proficient controller if its best controller at generation 200 has a fitness (averaged, as always, over three trials) of at least 1.5, meaning that it completes at least one and a half lap within the allowed time.

For the first two tracks, proficient controllers were produced by the evolutionary process within 200 generations, but only in two out of ten runs. This means that while it is possible to evolve neural networks that can be relied on to race around one of these track without getting stuck or taking excessively long time, the evolutionary process in itself is not reliable. In fact, most of the evolutionary runs are false starts. For tracks 3, 4, 5 and 6, the situation is different as at least half of all evolutionary runs produce proficient controllers. The best evolved controllers for these tracks get around the track fairly fast without colliding with walls. For tracks 7 and 8, however, we have not been able to evolve proficient controllers from scratch at all. The "best" (least bad) controllers evolved for track 7 might get halfway around the track before getting stuck on a wall, or losing orientation and starting to move back along the track.

Fixed sensor parameters: Generality of evolved controllers

Next, we examined the generality of these controllers by testing their performance of the best controller for each track on each of the ten tracks. The results are presented in figure 5.3, and clearly show that the generality is very low. No controller performed very well on any track it had not been evolved on, with the interesting exception of the controller evolved for track 1, that actually performed better on track 3 than on the track for which it had been evolved, and on which it had a rather mediocre performance. It should be noted that both track 1 and track 3 (like all odd-numbered tracks) run counter-clockwise, and there indeed seems to be a slight

Evo/Test	1	2	3	4
1	1.02 (0.14)	0.87 (0.1)	**1.45 (0.18)**	0.52 (0)
2	0.28 (0.06)	**1.13 (0.35)**	0.18 (0.1)	0.75 (0.26)
3	0.58 (0.16)	0.6 (0.22)	**2.1 (0.48)**	1.45 (0.66)
4	0.15 (0.01)	0.32 (0.02)	0.06 (0.05)	**1.77 (0.52)**
5	0.07 (0.02)	-0.02 (0)	0.05 (0)	0.2 (0.11)
6	1.33 (0.18)	0.43 (0.07)	0.4 (0.2)	0.67 (0.22)
7	**0.45 (0.11)**	0 (0.07)	0.6 (0.18)	0.03 (0.04)
8	0.16 (0.03)	0.28 (0.04)	0.09 (0.07)	**0.29 (0.18)**
Evo/Test	5	6	7	8
1	1.26 (0.17)	0.03 (0)	0.2 (0.18)	0.13 (0)
2	0.5 (0.13)	0.66 (0.19)	0.18 (0.15)	0.14 (0.02)
3	0.62 (0.13)	0.04 (0.1)	0.03 (0.09)	0.14 (0.02)
4	0.22 (0.1)	0.13 (0.13)	0.07 (0.09)	0.13 (0.02)
5	**2.37 (0.28)**	0.1 (0.04)	0.03 (0.05)	0.13 (0.01)
6	1.39 (0.42)	**2.34 (0.05)**	0.13 (0.13)	0.14 (0.11)
7	0.36 (0.08)	0.07 (0.03)	0.22 (0.15)	0.08 (0)
8	0.21 (0.03)	0.08 (0.1)	0.1 (0.09)	0.13 (0)

Table 5.3: The fitness of each controller on each track. Each row represents the performance of the best controller of one evolutionary run with fixed sensors, evolved the track with the same number as the row. Each column represents the performance of the controllers on the track with the same number as the column. Each cell contains the mean fitness of 50 trials of the controller given by the row on the track given by the column. Cells with bold text indicate the track on which a certain controller performed best.

bias for the other controllers to get higher fitness on tracks running in the same direction as the track for which they were evolved. We have not analysed this further.

Evolved sensor parameters: Evolving from scratch

Evolving controllers from scratch with sensor parameter mutations turned on resulted in somewhat lower average fitnesses and numbers of proficient controllers, as can be seen in table 5.4. The controllers that reached proficiency seemed to be roughly equally fit as those evolved with fixed sensors, but more evolutionary runs got stuck in some local optimum and never produced proficient controllers when sensor parameters were evolvable. It is not known whether this is simply because of the increase in search space dimensionality caused by the addition of sensor parameters, or if they complicate the evolutionary process in some other way.

Evolved sensor parameters: Generality of evolved controllers

Table 5.5 details the performance of one controller evolved with sensor mutation on for each track on all the tracks in the test set. Controllers evolved with evolvable sensor parameters turn out to generalize really badly, almost as badly as the controllers evolved with fixed sensors. However, there are some interesting differences, and the controllers evolved for track 1, 2 and 6 (but not the others) actually perform better on tracks for which they were not evolved. It

Track	10	50	100	200	*Pr.*
1	0.3 (0.05)	0.58 (0.17)	0.65 (0.18)	0.89 (0.4)	1
2	0.32 (0.09)	0.72 (0.4)	0.81 (0.49)	0.91 (0.6)	3
3	0.53 (0.22)	1.39 (0.51)	2.77 (0.66)	1.99 (0.7)	7
4	1.37 (0.89)	2.25 (0.34)	2.42 (0.37)	2.41 (0.36)	10
5	0.4 (0.07)	0.64 (0.35)	0.95 (0.55)	1.31 (0.66)	4
6	0.48 (0.12)	0.7 (0.29)	0.83 (0.39)	0.99 (0.65)	2
7	0.33 (0.11)	0.43 (0.08)	0.44 (0.08)	0.5 (0.15)	0
8	0.16 (0.02)	0.21 (0)	0.21 (0)	0.21 (0)	0

Table 5.4: Evolving controllers for individual tracks from scratch with sensor mutation turned on; format as in table 5.2.

Evo/Tst	1	2	3	4
1	**1.33 (0.29)**	1.04 (0.34)	0.31 (0.04)	2.54 (0)
2	0.84 (0.19)	1.81 (0.14)	1.49 (0.55)	**2.92 (0.16)**
3	0.15 (0.01)	0.07 (0.04)	**1.04 (0.22)**	0.26 (0.04)
4	0.14 (0)	0.11 (0.03)	0.51 (0.14)	**1.47 (0.49)**
5	0.04 (0.11)	0.21 (0.05)	0.28 (0.05)	0.34 (0.07)
6	0.57 (0.03)	0.31 (0)	2.51 (0.02)	**2.72 (0.26)**
7	0.19 (0.05)	0.1 (0.05)	**0.22 (0)**	0.05 (0.05)
8	0.06 (0.04)	-0.09 (0.02)	-0.04 (0.02)	-0.02 (0.01)
Evo/Tst	5	6	7	8
1	0.68 (0.11)	0.04 (0.02)	0.32 (0.16)	0.13 (0)
2	0.3 (0.03)	0.55 (0.15)	0.13 (0.14)	0.14 (0)
3	0.26 (0.06)	0.32 (0.07)	0.05 (0.08)	0.01 (0)
4	0.17 (0)	0.19 (0.11)	0.11 (0.07)	0.12 (0.02)
5	**2 (0)**	0.14 (0.05)	0.04 (0.1)	0.15 (0.02)
6	0.42 (0.03)	2.53 (0.22)	0.11 (0.09)	0.13 (0)
7	0.13 (0.09)	0.16 (0.06)	0.16 (0.11)	0 (0)
8	0.07 (0.05)	0 (0)	-0.01 (0.02)	**0.21 (0)**

Table 5.5: Fitness for individual controllers on different tracks evolved with sensor mutation turned on; format as in figure 5.3.

is quite hard to see any kind of logic in which controllers will do well on which tracks, except those they were evolved for, and more data would definitely be needed to resolve this.

5.2.3 Evolving general and robust driving skills

The next suite of experiments were on evolving robust controllers, i.e. controllers that can drive proficiently on a large set of tracks.

Simultaneous evolution

Our first attempt consisted in evolving controllers on all tracks at once. For this purpose, we ran several evolutionary runs where each controller was tested on all the first six tracks, each for three trials, and the fitness was averaged over all these trials. We ran several evolutionary runs with this setup, and with both evolvable and fixed sensor parameters, for long periods of time, but found very little progress - no controller reached an average fitness above 1.

Incremental evolution

Abandoning this method, we tried incremental evolution. The idea here was to evolve a controller on one track, and when it reached proficiency (mean fitness above 1.5) add another track to the training set - so that controllers are now evaluated on both tracks and fitness averaged - and continue evolving. This procedure is then repeated, with a new track added to the fitness function each time the best controller of the population has an average fitness of 1.5 or over, until we have a controller that races all of the first six tracks proficiently. The order of the tracks was 5, 6, 3, 4, 1 and finally 2, the rationale being that the balance between clockwise and counterclockwise should be as equal as possible in order to prevent lopsided controllers, and that easier tracks should be added to the mix before harder ones.

This approach turned out to work much better than simultaneous evolution. Several runs were performed, and while some of them failed to produce generally proficient controllers, some others fared better. A successful run usually takes a long time, on the order of several hundred generations, but it seems that once a run has come up with a controller that is proficient on the first three or four tracks, it almost always proceeds to produce a generally proficient controller. One of the successful runs is depicted in figure 5.14, and the mean fitness of the best controller of that run when tested on all eight tracks separately is shown in 5.6. As can be seen from this table, the controller does a good job on the six tracks for which it was evolved, bar that it occasionally gets stuck on a wall in track 2. It never makes its way around track 7 or 8.

The successful runs were all made with sensor mutation turned off. Some runs of incremental evolution were made with sensor mutation allowed; however, they failed to produce any proficient

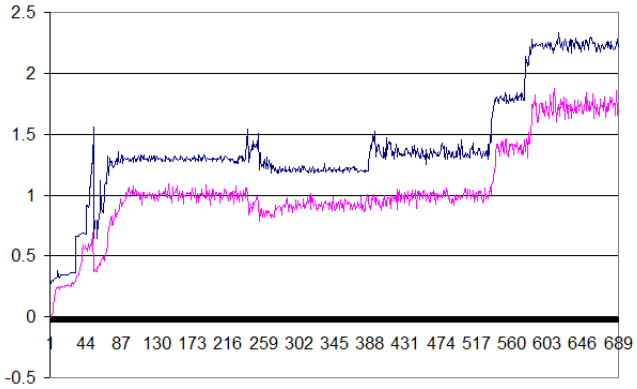


Figure 5.14: A successful incremental run, producing a generally proficient controller. New tracks were added to the fitness function when fitness of the best controller reached 1.5; this happened at generations 53, 240, 253, 394 and 536. Maximum fitness continued to increase for approximately 50 generations after that. The graph show the fitness of the best controller (dark line) and the mean fitness of the population.

Track	1	2	3	4
Fitness (sd)	1.66 (0.08)	1.48 (0.25)	2.56 (0.2)	2.49 (0.15)
Track	5	6	7	8
Fitness (sd)	2 (0.25)	2.02 (0.42)	0.4 (0.21)	0.16 (0.07)

Table 5.6: Fitness of an incrementally evolved general controller with fixed sensor parameters on the different tracks. Compound fitness over all 8 tracks is 2.01 (0.11).

controllers. We speculate that this is because these runs suffer from "premature specialization" - after evolving a good controller for the first track, the sensor setup might not be suited for good driving on the second track, and changing the parameters would diminish fitness on the first track, thus creating a local optimum. That the first two tracks are, from the point of view of the car, mirror images of each other, adds plausibility to this hypothesis.

Further evolution

Evolving sensor parameters can be beneficial, however, when this is done for a controller that has already reached general proficiency. We used one of the generally proficient controllers evolved using the incremental method as the seed for a new evolutionary run, with sensor mutation turned on and controllers tested on all six tracks simultaneously. The results was an increase in mean fitness, as can be seen in 5.7. Although the mean fitness does not increase on every single track, the best controller of the last generation races all the tracks more reliably, and is very rarely observed to crash into a wall in such a way that the car gets stuck. The evolved sensors of this controller showed little similarity to the original sensor setup, described above - see figure 5.15 for an example.

Track	1	2	3	4
Fitness (sd)	1.66 (0.12)	1.86 (0.02)	2.27 (0.45)	2.66 (0.3)
Track	5	6	7	8
Fitness (sd)	2.19 (0.23)	2.47 (0.18)	0.22 (0.15)	0.15 (0.01)

Table 5.7: Fitness of a further evolved general controller with evolvable sensor parameters on the different tracks. Compound fitness 2.22 (0.09).

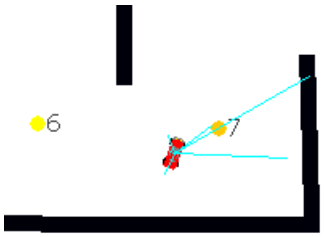


Figure 5.15: Sensor setup of the further evolved general controller analysed in table 5.7. Only three sensors seem to be long enough to be of any use, and all of those point to the right or front-right. The asymmetry and "waste" is somewhat surprising, as the controller performs well on all the first six tracks (but it does do slightly better on clockwise than on anti-clockwise tracks).

5.2.4 Evolving specialized controllers

In order to see whether we could create even better controllers, we used one of the further evolved controllers (with evolved sensor parameters) as basis for specializing controllers. For each track, 10 evolutionary runs were made, where the initial population was seeded with the general controller and evolution was allowed to continue for 200 generations. Results are shown in table 5.8. The mean fitness improved significantly on all six first tracks, and much of the fitness increase occured early in the evolutionary run, as can be seen from a comparison with table 5.7. Further, the variability in mean fitness of the specialized controllers from different evolutionary runs is very low, meaning that the reliability of the evolutionary process is very high. Perhaps most surprising, however, is that all 10 evolutionary runs produced proficient

Track	10	50	100	200	*Pr.*
1	1.9 (0.1)	1.99 (0.06)	2.02 (0.01)	2.04 (0.02)	10
2	2.06 (0.1)	2.12 (0.04)	2.14 (0)	2.15 (0.01)	10
3	3.25 (0.08)	3.4 (0.1)	3.45 (0.12)	3.57 (0.1)	10
4	3.35 (0.11)	3.58 (0.11)	3.61 (0.1)	3.67 (0.1)	10
5	2.66 (0.13)	2.84 (0.02)	2.88 (0.06)	2.88 (0.06)	10
6	2.64 (0)	2.71 (0.08)	2.72 (0.08)	2.82 (0.1)	10
7	1.53 (0.29)	1.84 (0.13)	1.88 (0.12)	1.9 (0.09)	10
8	0.59 (0.15)	0.73 (0.22)	0.85 (0.21)	0.93 (0.25)	0

Table 5.8: Fitness of best controllers, evolving controllers specialised for each track, starting from a further evolved general controller with evolved sensor parameters.

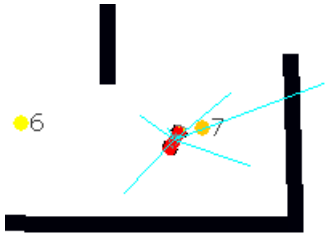


Figure 5.16: Sensor setup of controller specialized for track 5. While more or less retaining the two longest-range sensors from the further evolved general controller it is based on, it has added medium-range sensors in the front and back, and a very short-range sensor to the left.

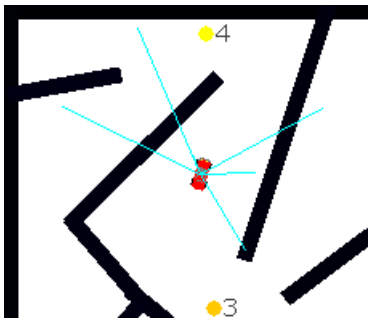


Figure 5.17: Sensor setup of a controller specialized for, and able to consistently reach good fitness on, track 7. Presumably the use of all but one sensor and their angular spread reflects the large variety of different situations the car has to handle in order to navigate this more difficult track.

controllers for track 7, on which the general controller had not been trained (and indeed had very low fitness) and for which it had previously been found to be impossible to evolve a proficient controller from scratch.

Analysis of the evolved sensor parameters of the specialized controllers show a remarkable diversity, even among controllers specialized for the same track, as evident in figures 5.16, 5.17 and 5.18. Sometimes, no similarity can be found between the evolved configuration and either the original sensor parameters or those of the further evolved general controller the specialization was based on.

There are many reasons why this could be the case. The most obvious reason is that there could well be many different ways of solving the same task. As a hypothetical example, for

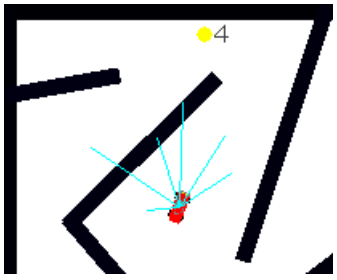


Figure 5.18: Sensor setup of another controller specialized for track 7, like the one in figure 5.17 seemingly using all its sensors, but in a quite different way.

specialized driving for a particular track, the controller might depend on either an "inside" or an "outside" wall sensor as the main trigger for turning, and either a backward or forward wall sensor for dissolving ambiguities regarding which turn to take. Another contributing reason could be that not all sensors need to be used in decision making. A sensor could have a long reach but all connection weights from it might be close to zero, and so several seemingly different sensor configurations might really be rather similar if only "active" sensors are taken into account. When trying to understand the space of sensor configurations it is also important to remember that even a sensor with a short reach can be important. If the presence of a wall in a certain direction only needs to be responded to when it is very close, it makes more sense to have a short sensor than a long one, as the shorter sensor has higher resolution given the limited number of sampling points, and will respond with a larger difference in output signal for the same difference in wall distance, than a longer sensor.

5.2.5 Observations on evolved driving behaviour

In section 5.1, we found that artificial evolution can produce controllers that outperform human drivers. To corroborate this result, the author measured his own performance on the various tracks, driving the car using keyboard inputs and a suitable delay of 50 ms between timesteps. Averaged over 10 attempts, the author's fitness on track 2 was 1.89, it was 2.65 on track 5, and 1.83 on track 7, numbers which compare rather unfavourably with those found in table 5.8. The author would like to believe that this says more about the capabilities of the evolved controllers than those of the author.

Traces of steering and driving commands from the evolved controllers show that they often use a PWM-like technique, in that they frequently - sometimes almost every timestep - change what commands they issue. For example, the general controller used as the base for the specializations above employs the tactic of constantly alternating between steering left and right when driving parallell to a wall, giving the appearance that the car is shaking. Frequently alternating between neutral and forward drive is also used a way of keeping a certain speed; an approach many engineers would use when designing a controller for a vehicle that can only be controlled with discrete inputs. Doing so is however practically impossible for a human driver, and analysis of action traces for human drivers shows much fewer changes to the commands given; as an example, the author changes drive command only four times per lap when racing on track 3.

5.2.6 Conclusions

We believe that the results presented above answer, at least in part, the several questions posed at the start of this section. Different tracks can indeed be constructed that have differing

difficulty levels, in terms of the probability that an evolutionary run starting from scratch will produce a proficient controller within a given number of generations, and the mean fitness of evolved controllers. Difficulty levels range from very easy, where the evolutionary algorithm almost always succeeds, to very hard, for which no successful controllers have been found, and agree with intuitive human difficulty ratings of the same tracks. These skills are, however, not transferable: a controller evolved from scratch to perform well on a given track usually performs very poorly on all other tracks. Evolving sensor parameters along with network weights makes for fewer proficient controllers (probably because of more local optima), a result which is not inconsistent with the good controllers that do emerge being slightly superior to fixed-sensor ones, as found in previous experiments.

As for the question on whether we can automatically create controllers with driving skills so general that they can proficiently race all tracks in our training set, this can be done by using incremental evolution, going from simpler to more complex tracks, with sensor mutation turned off. Attempts to evolve general controllers with sensor mutation turned on failed, as did attempts to evolve controllers on all tracks simultaneously. Once a general controller has been created, its fitness can be increased through continued evolution with sensor mutation turned on. Specialized controllers can be created by further evolving a general controller, using only one track in the fitness function. These specialized controllers invariably have very high fitness. Much to our surprise, this was true even for one hard track which the general controller had not been evolved on and which it had very low fitness on, and for which we have not been able to evolve proficient controllers from scratch. Apparently, the general controller is somehow closer in search space to a proficient controller for that track, even though it has no proficiency itself on that track. Exactly how this works remains to be found out.

The ease with which specialized controllers can be gotten from general controllers point to a possible application in racing games: if a user designs his own track, the game could easily evolve NPC drivers with the required driving skill for that track, starting from a generally proficient controller.

5.3 Further experiments in optimizing car driving

In this section, we briefly discuss two series of experiments aimed at further investigating how to best learn good driving behaviour for the well-defined task of single-car racing. The first series of experiments compare neural networks with genetic programming in stateful and non-stateful versions. The second series compares evolution with td-learning, and state-value control with action-value control and direct control.

5.3.1 Comparing controller architectures on the track task

This section is based on a paper presented at GECCO 2007, with Alexandros Agapitos as first author [3]. The two parallel goals of these experiments was to compare neuroevolution with genetic programming, and to compare representations that allow for stateful control and those that only allow for reactive controllers to be evolved. As a bonus, we compared the effect of the number of rangefinder sensors on the performance of the neural network-based controllers.

In this section, we will mainly discuss the results of our experiments, and the reader is referred to the paper for the background on object-oriented genetic programming and all the details on the gp implementation, including most parameters. Some analysis of the evolved gp trees are also presented in the paper.

Methods

The experiments were done in the track-based racing game using the full car racing simulation (with non-buggy way point sensor). As the order of the experiments follow those described in section 5.2 quite closely, the same tracks were used and numbered in same way. The usual $50 + 50$ evolution strategy was used for most of the experimental runs, but as crossover is often observed to work better for gp than it does for neural nets, a genetic algorithm with population 100 was used for some of the gp experiments.

The gp trees were intialised with depth 5, and were allowed to grow to a maximum depth of 8 if ADFs were used, and 10 if they were not used. Uniform crossover combined with point mutation was used in the GA based runs, and in the ES based runs subtree macro-mutation was used. In all the gp experiments, a set of non-terminals were used that included standard arithmetic functions, conditional branching, ordinal predicates (less than, more than etc.), and calls to get the value of a specific wall sensor with range and angle defined by the value of the child nodes. The terminals included a range of constants as well as the values of the speed and way point sensors. For the object-oriented gp experiments, two added non-terminals could either get or set the values in a register.

Taken together, the sets of terminals and non-terminals should give gp considerably more expressive power than neuroevolution. For example, it would be possible to evolve a gp-based controller that moved its sensors, so that it had different configurations depending on its speed.

In all, nine different configurations of controller architecture and evolutionary algorithm were compared:

- Functional gp without ADFs and crossover
- Functional gp with ADFs, no crossover

Table 5.9: Average fitness of best controller for generations 10, 50, 100, 200 (averaged over 10 independent evolutionary runs - std. deviation in parentheses)

Method	**10**	**50**	**100**	**200**
Functional/no ADFs/Macromutation	1.26 (0.65)	2.33 (0.4)	2.47 (0.4)	2.51 (0.15)
Functional/ADFs/Macromutation	1.54 (0.45)	2.54 (0.17)	2.62 (0.15)	**2.67** (0.1)
Functional/no ADFs/Recombination	1.87 (0.52)	2.38 (0.16)	2.45 (0.17)	2.46 (0.17)
Functional/ADFs/Recombination	1.62 (0.74)	2.23 (0.63)	2.39 (0.47)	2.53 (0.17)
OO/Macromutation	1.55 (0.61)	2.47 (0.7)	2.54 (0.18)	2.59 (0.18)
OO/Recombination	1.10 (0.75)	2.39 (0.3)	2.47 (0.07)	2.55 (0.07)
MLP	0.13 (0.17)	2.48 (0.67)	2.92 (0.09)	**3.08** (0.07)
Recurrent	0.19 (0.16)	1.06 (0.45)	2.43 (0.46)	2.92 (0.16)
MLP with less sensors	0.19 (0.22)	2.65 (0.08)	2.94 (0.08)	3.07 (0.02)

- Functional gp without ADFs, with crossover
- Functional gp with ADFs and crossover
- Object-oriented gp with ADFs, no crossover
- Object-oriented gp with ADFs and crossover
- Multilayer perceptrons, 12 hidden neurons, 12 wall sensors
- Elman-style recurrent networks , 12 hidden neurons, 12 wall sensors
- Multilayer perceptrons, 12 hidden neurons, 4 wall sensors

Evolving single-track Controllers

The first set of experiments concern the evolution of driving skills on a single track, namely track 5. In this set of experiments, each evolutionary run starts with freshly created controllers: neural networks with all connection weights set to 0, or GP controllers with small randomly generated trees. For each controller configuration, we ran 10 independent run of 200 generations with population 100. The results of this can be seen in table 5.9.

The first observation on the results is that all configurations of both neural network and GP controller representations managed to evolve high-performing car drivers. But the performance of GP and NN controllers are not identical. Generally, it can be seen that the GP controllers evolve much faster than the neural network controllers, but that the neural network controllers ultimately reach higher fitnesses. No significant difference can be seen between functional and object-oriented GP, and between recurrent and feedforward neural nets.

Performance on Several Tracks

Next, we looked at the generalisation capability of the controllers evolved in the preceding section. This was done by trying each of these controllers on each of the eight tracks, not only the track for which they had been evolved.

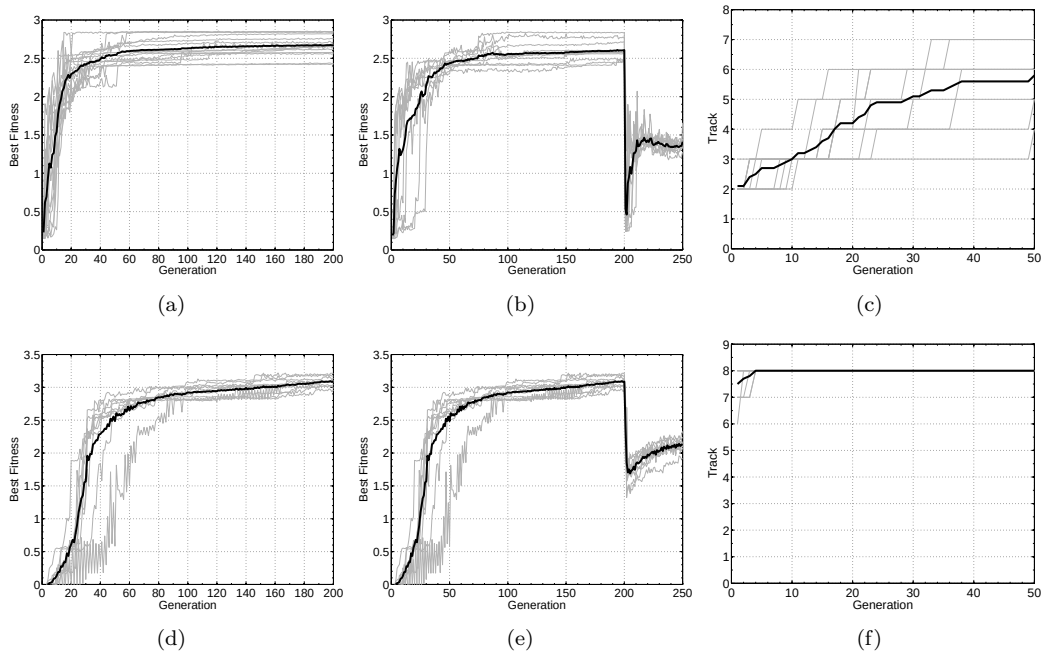


Figure 5.19: (a) Best-of-generation individuals using GP with ADFs; (b) Best-of-generation individuals using OOGP and MM (generalisation to additional tracks is shown after generation 200); (c) Tracks driven proficiently using OOGP and MM; (d) Best-of-generation-individuals using MLP; (e) Generalisation to additional tracks after generation 200 using MLP (f) Tracks driven proficiently using MLP

In general, the evolved neural networks perform significantly better than the GP controllers even for tracks for which they have not been evolved. Like in the previous section, not much of a difference was found between stateful and stateless controllers. Finally, similarly to the experiments in section 5.2, all controllers scored lower on other tracks than on track 5, for which they had been evolved, but scored slightly better on tracks which like track 5 run counter-clockwise than on those that run clockwise.

Evolving Generalisation

The final set of experiments concern the incremental evolution of general controllers. In these experiments, each evolutionary run was seeded with the results of the 200 generations of evolution on a single track described above. Then, evolution proceeded for 50 generations, with the crucial difference that the fitness function was made incremental; each controller evaluation was done as the average of the progress that controller displayed on a set of tracks. At the first generation of each evolutionary run, only track 5 was used for fitness evaluations, but every time a controller reached fitness 1.5 a track was added to the evaluation set (and kept for the duration of the evolutionary run). The sequence in which tracks were added was 5, 6, 3, 4, 1, 2, 7, 8. (Note that every second track added runs clockwise instead counterclockwise, to increase

Method	Avg. no of tracks
Functional/ADFs/Macromutation	6.0 (0.8)
OO/Macromutation	5.8 (0.91)
OO/Recombination	4.6 (1.07)
MLP	8.0 (0.0)
Recurrent	8.0 (0.0)
MLP with less sensors	8.0 (0.0)

Table 5.10: Average number of tracks, proficiently driven (averaged over 10 independent evolutionary runs - std. deviation in parentheses)

the diversity between tracks added in sequence and thus avoid overfitting to a particular driving direction.) A controller that was able to generalise completely and drive proficiently on all tracks would at the end of these 50 extra generations have all 8 tracks in its evaluation set, whereas a very poor generaliser would be stuck with only track 5 in the set.

As a consequence, the graphs depicted in figure 5.19 for these evolutionary runs include not only the fitness of the best controller in the population but also the incrementation level (number of tracks in the evaluation set) of the population.

As we can see from table 5.10, both NNs and GP are able to incrementally generalise previously evolved controllers and achieve proficiency on a majority of the eight tracks. However, there is a marked difference, in that the neural network-based controllers on average generalised significantly better, and end up being able to drive proficiently on seven of the eight tracks.

On the other hand, there seems to be no significant performance difference between stateful and stateless controllers, i.e. between MLPs and functional GP on the one hand and recurrent networks and OOGP on the other hand.

Conclusions

The main finding of our experiments is that, for the given problem and experimental setup, the various versions of GP evolve faster than NNs, but the neural networks ultimately perform better, especially on the more complicated version of the task, that takes all eight tracks into consideration.

An early suspicion was that as the GP trees evidently make use of much fewer wall sensor readings than the neural network controllers, this contributed to the fast learning but poor generalisation of GP controllers. The argument was that it is easier to learn to drive on a simple track when only caring about the speed and the angle to the next waypoint, but that this strategy breaks down when exposed to the more complicated tracks where the straight line between two waypoints might pass through a wall. As neural networks are in effect forced to consider all its sensor readings, this makes it harder to find an initial control strategy, but once found, such a control strategy will be much more robust. Much to our dismay, our hypothesis

was falsified by the inclusions of the "minimal" neural network controller that only uses four wall sensors, yet performs almost as well as those controllers that use 12 wall sensors.

Another hypothesis is that we are being unfair to GP when we are asking it to compete with neural networks on the neural networks' home arena. GP experiments typically use much larger population sizes and different selection regimes. Given such changes the GP controllers might very well outperform our neural network controllers.

An additional puzzling phenomenon is the virtual lack of difference between the performance of stateless and stateful controller representation. Our best bet as to why this is so is that we need even more complex versions of the car racing task, such as competitive multi-car racing, in order to exploit the statefulness of the controllers. Maybe we also need to introduce more complex primitive objects, e.g complete forward models of the car dynamics, as objects which could be accessed and manipulated by the OOGP system.

5.3.2 Comparing learning methods for point-to-point racing

In this section, which is based on a paper presented at CIG 2007 with Simon Lucas as first author, we compare evolution with temporal difference learning, and direct control with state-value and action-value based control [80]. The game used here is one of the simplest possible variations on car racing possible, as it is a stripped-down version of the point-to-point task, allowing only for one car (and thus no collisions at all), only five actions instead of nine, and having a rather simplified dynamics model. Fitness is calculated depending as the number of way points passed in 500 time steps. The details of the car model, and of the learning algorithms, are to be found in the paper. The paper also contains comparisons with a holonomic version of the car game, and an extremely simple one-dimensional version.

Manual control

Both of the authors of the paper attempted to solve the task themselves repeatedly in the course of experimentation. On a good day, the authors typically scored between 12 and 16.

Hand-coded control

Two hand-coded controllers were implemented: *Greedy* and *Heuristic*.

The algorithm for each of these controllers is simple: consider the state of the system after each possible action, and select the action that leads to the best score. In this case, the scores are penalty values, so this means the action with the lowest score will be selected.

The score function used for the greedy controller is simply the Euclidean distance between the car and the next waypoint. The heuristic controller improves on this by adding in a penalty

term proportional to the square of the car's speed. The square of the speed is used on the basis that stopping distance is proportional to this.

The greedy algorithm performs poorly; its failure to consider the velocity of the car leads to a tendency to significantly overshoot each waypoint. This is because given the current state of the system, the action that takes the car closest to the next waypoint usually involves accelerating toward the waypoint. The addition of the velocity penalty in the heuristic controller leads to much better performance, with a fitness of 18.8 averaged over 1000 evaluations.

Action-value control

Two different representations of the action values were used: neural networks and tables. The table-based controller had $3^3 * 5$, or 135 cells. Selection was done on the following three dimensions: speed, angle to the next waypoint and distance to the next waypoint. Cut-off values for these dimensions were set to 0.08, 0.3 and 0.31 respectively, values which were found after an exhaustive search.

Likewise, the MLP-based controller takes speed, angle and distance to the next waypoint, and the action to evaluate as input, and outputs an estimate of the value of that action.

First, we tried the Sarsa variety of temporal difference learning. Learning with the table-based representation was extremely unreliable, with only some runs learning anything at all, and frequent cases of "unlearning" were reasonably good behaviour was observed for a number of epochs, before plunging back into randomness. But at least in some cases td-learning produced controllers with performance significantly better than random. Still, the best-performing learned controller had a fitness of only 4.7. The training runs that succeeded did so within 1000 epochs. In contrast to the td-learned controllers for the one-dimensional case, which performed better when occasional random movement was turned off (ϵ set to 0) after training, the controllers learned for the car model needed a (small) non-negative ϵ in order to perform.

TDL with the MLP-based representation was unable to learn any meaningful policy at all.

We then tried evolving action-values using the same representations, and a $15 + 15$ evolution strategy. For the both the connection weights of the neural networks and the values in the cells of the tables, we used Gaussian mutation with standard deviation 0.1.

Evolution reliably produced somewhat capable action value function based controllers for the car model. The best evolved table-based controller (from several runs that produced similar controllers) scored 14.4, and basically drives well, apart from the occasional case of "orbiting". Very similar results were achieved using the MLP-based representation, with the best evolved controller scoring 15.7.

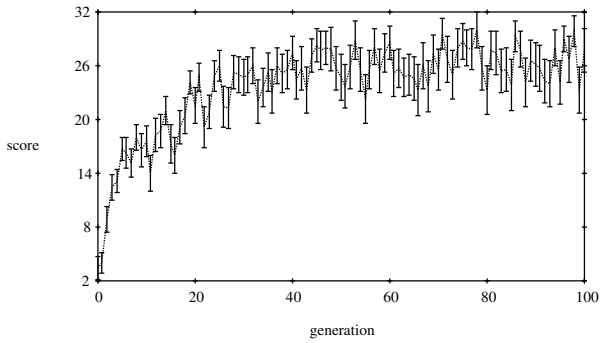


Figure 5.20: Evolving a state-value MLP for controlling a normal car. The error bars show the range of fitnesses ($pm\ \sigma$) in each generation.

State-value control

For the state-value control experiments, the controller took and retracted all the five possible actions in an internal model identical to the actual dynamics model, and used a function approximator to evaluate the value of each action based on the state resulting from that action. The highest-valued action is then selected for actual execution. What is actually trained through td-learning or evolution is the function approximator, and here we compared perceptrons with multi-layer perceptrons.

For input to the function approximator, we compared two different feature vector with two and three inputs respectively. The two-input version consisted of the Euclidean distance to the next waypoint, and the square of the car's velocity. The three input version takes these two and also adds a directional feature: the absolute value of the *sin* of the angle between the car's heading and the direction to the next waypoint. In each case the value function then applies the feature vector as input to a neural network, whose single output is taken to be the value of that state.

Using a $15 + 15$ ES running for 100 generations, very good performance was reliably evolved despite the very noisy fitness evaluations due to the way points being randomized for every trial. Figure 5.20 shows one such evolutionary run with MLP approximators (1 hidden layer with 10 units).

Temporal difference learning, on the other hand, was rather unreliable. When it did learn it often achieved high performance within the first 10 epochs, and in these cases offered much faster learning than evolution. It reached similar performance when using the linear function representation, but not when using the neural network representation.

Direct control

The above approaches learn controllers that are based on some sort of explicit representation of the values of states or actions. These are, to our knowledge, the only types of controllers that

can be learnt using temporal difference learning. But evolutionary algorithms are also capable of solving reinforcement learning through direct search in policy space, creating controllers that don't necessarily directly represent state or action values. In case of point-to-point car racing, such controllers would take (aspects of) the state of the car as inputs and output the desired action to take. We chose to represent the controllers as standard three-layer neural networks, and used the same Evolution Strategy as above.

The neural network was fed with the velocity in vertical and horizontal dimensions, the magnitude of the velocity vector, and the distance and relative angle to the next waypoint. Ten hidden neurons were used, and the two outputs were interpreted as movement commands, with one output controlling steering and the other longitudinal acceleration. (The output range sensitivity is a bit different in this experiment than in the preceding sections, as the steering output is interpreted as left if below -0.1, right if over 0.1, otherwise centre. This is a mere accident, and to our best knowledge insignificant. A number of auxiliary experiments with varying this and other parameters have shown that varying the output sensitivity makes no noticeable difference as long as the mutation magnitude is set high enough, e.g. 0.1.) Using this approach, good controllers reliably emerged from the evolutionary process. Several runs of 500 generations each produced controllers of similar fitness, the best of them scoring on average 20.1. This controller consistently keeps a high speed and only rarely misses a waypoint, and then only "orbits" briefly.

Combining evolution and temporal difference learning

As we have seen, evolution and temporal difference learning can both be used to learn functional (state, action) to value mappings, although with varying speed, reliability and ultimate success. Another interesting question is whether the two methods learn different solutions to the problems. Especially, it might be suggested that while Sarsa learns *approximately correct* action values, evolution will learn *values that work*, regardless of whether they are correct or not. To investigate this, we tried applying our evolutionary algorithm to solutions that had been learned with Sarsa, and Sarsa to solutions that had been evolved. Due to time and space constraints (for us, not for the algorithms) this was only done for the action-value approach.

Seeding the Sarsa algorithm with evolved action value based car controllers had an interesting effect, in that it had no effect at all: the further td-learning made no significant change, neither negative or positive. Or in other words, td-learning performs much better when initialised with an evolved controller. Seeding evolution with the results of Sarsa made no significant difference to seeding evolution with random controllers: good controllers evolved just as quickly as they do when evolution is initialised with empty tables or MLPs with neutral connection weights.

One way of interpreting this is that evolution actually does learn the correct action values; another is that evolution has a "lock-in effect", creating a local optimum td-learning can't break out of.

Conclusions

In table 5.11, we present the fitness of the best controller obtained through all of the different methods, tested on 1000 tracks.

Table 5.11: Best controllers found for the normal car, tested on 1000 tracks.

Method	Td-learning	Evolution
State-MLP	-	36.7
State-Perceptron	31.1	31.5
Action-Table	4.7	14.4
Action-MLP	0	15.7
Direct-MLP	-	20.1

Experiments were made on a variety of problem setups using different feature vectors, and different neural networks (multi and single layer perceptrons). In each case, state-value learning worked much better than action-value learning. Evolution worked more reliably and achieved better final fitness than reinforcement learning, but reinforcement learning, when successful, learned faster. Further, for learning action value functions with TDL, table-based representations were always superior to MLP-based representations. When evolution was used without a model, the direct approach was superior to learning action values. Our suspicion that the method learns differently and not only better or worse was supported by the "lock-in effect" observed when combining them.

The task is clearly very sensitive to the learning method, the architecture, and the chosen input features. A strong distinction can be seen between methods which work very well, such as evolved state-based MLPs, and those that work very badly, such as TDL-trained action-based MLPs. While it is likely that with more work it would be possible to improve the TDL results, a great strength of the evolutionary methods is the ease with which they can be applied.

The state-value methods can only be applied when a forward model exists, which is able to predict the next state of the car given the current state and the selected action. This is simple for the simulation, where we had access to the exact forward model. For controlling a real car we would have to infer a forward model, something we do in section 6.2. However, these models are only ever approximate, and we don't know if the state-value approach would retain its advantage when used together with such a model.

It is also an open question if these results hold up if a similar comparison is to be performed on a more complicated version of the car racing problem, involving two cars and possibly walls.

5.4 Optimization in other game domains

While most of the experiments in this thesis deal with car racing in one form or another, we have also used evolutionary optimization to learn to play several other games. In this game we present experiments with evolving neural network-based controllers for a helicopter simulation game and the Cellz game.

5.4.1 Controlling a simulated helicopter

This section is based on a paper presented at CEC 2006, with Renzo De Nardi as first author [36]. The research was performed as part of the UltraSwarms project, of which Renzo is a part, and which aims to produce swarms of miniature helicopters that perform collaborative computing over wireless networks while flying. As a first step in this project, an individual helicopter must be brought to fly autonomously, using a controller that can in turn receive higher-level commands from a swarming algorithm. As the hardware platform was still in preparation at the time of doing the research, we decided to use the open source helicopter simulator described in section 4.2.1 for these initial experiments in controller acquisition.

As the simulator is very accurate, flying the helicopter is very, very hard. None of the authors could even it get it off the ground and hovering in the same place for more than a few seconds, and trying to complete the way point following task we evolved controllers for was out of the question. The problem turned out to be too difficualt not only for humans, but for algorithms as well; a great amount of effort was spent on approaches that turned out not to work.

In the notation used below, the helicopter state is specified by the variables

$$[x, y, z, u, v, w, \phi, \theta, \psi, p, q, r]$$

The state vector follows the conventional notation used in the aircraft control community; x, y, z and u, v, w are the position in the inertial reference frame and the velocity in the helicopter's frame of reference, while ϕ, θ, ψ and p, q, r are the rotations and rotational velocity about the axis of the helicopter, respectively.

The simulator comes complete with a PID (proportional, integral, derivative) controller, which was hand-tuned by the creators of the simulator, and able perform the way point following task (though not with much grace). In our experiments, we made use of parts of this controller in various ways while we were exploring ways to automatically create controllers that would eventually outperform the PID controller by a respectable margin.

More details about e.g. the simulation, previous work and some of the experiments are to be found in the paper. The paper also has an analysis of the shortcomings of monolithic networks

and simultaneous evolution, which is largely superseded by the discussion in sections 2.4.2 and 2.4.3.

Non-working methods

In the first approach, we tried to evolve a full MLP with two layers of weights, using a $10 + 23$ evolution strategy, and a fitness function based only on progress along the waypoint chain. (The odd population size is because the simulation is very computationally expensive, so we had to distribute the evaluations to a cluster, and we had only 34 machines available.) The MLP was given all 12 state variables (position, velocity, angles and rotational velocity), plus the location of the next waypoint in 3 dimensions, and the network's four outputs were used to drive all the control surfaces of the helicopter.

Results were not encouraging. In some cases, no goal-directed behaviour at all was observed, save that of maintaining enough altitude not to crash into the ground. In some other cases, the helicopter drifted slowly towards the first waypoint, but rarely reached it. In all cases, the helicopter was continuously spinning around the z axis, though the speed with which it did this varied.

Figuring that the z spin was the problem, we tried removing the yaw control (responsible for controlling the z spin) from the neural network, and reinstating the yaw part of the PID controller, but to no avail. We then tried a wide variety of ways of modularizing the neural network, and changes to the fitness function, to be able to evolve a controller that would both follow the way points and not spin around. Nothing worked. Thinking that it would be impossible to evolve yaw stabilization together with the other aspects of the controller, we then opted for incremental evolution.

Incrementally substituting a PID with neural networks

In our first working approach, we used the PID controller delivered with the simulator to enforce functional separation in the network.

The first phase was the evolution of the yaw controller. For this purpose a very simple neural network, with four connections in all, was evolved to stabilize the yaw. This network was evolved using a fitness function that linearly penalised deviation from the target heading. Each trial lasted for only twenty timesteps, and evolution produced a fairly good solution within several tens of generations. During the second phase, the yaw network was free to evolve along with the rest of the controller.

The second phase was divided into three steps; in all steps the same fitness function was used. In step 1, a three layer MLP was substituted for the PID controller's guidance layer; it

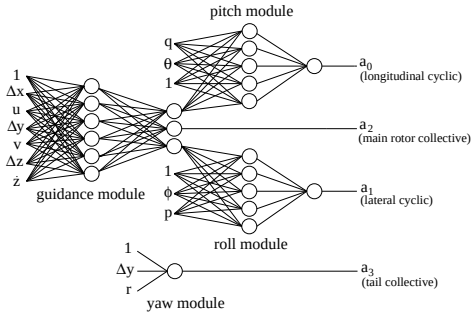


Figure 5.21: Topology of the substitution network.

took as its input the distances from the waypoint, and the velocity of the helicopter, (both in body coordinates) and had as outputs the desired pitch and roll attitude to the longitudinal and lateral PID's.

In step 2, the two inner PID loops (longitudinal and lateral) were removed and substituted by two separate MLPs. These were evolved to act on the information given by the neural outer layer, and they output helicopter motor commands (longitudinal and lateral cyclic control, and collective pitch). After this step, we had a working helicopter controller consisting solely of neural networks.

In step 3, the outer network layer, which was frozen during step 2, was allowed to evolve further, along with the inner network layer. In this way, the networks could coadapt to each other, potentially allowing them to exploit modes of cooperation not possible for the purely linear PID controllers.

The controller based on the inner network reached a good fitness level within 200 generations in all ten replications of the experiment. Evolution of the outer network showed more variability, but within 500 generations an outer network had been evolved in all replications which gave the controller a reasonable, if not good, performance. When both networks were further evolved together, however, very good performance was reached in every replication (see fig. 5.22).

Incremental/simultaneous evolution of modular networks

The next experiment aimed at evolving a controller without any involvement of the PID. The first phase, evolving a simple yaw stabilizer, was repeated exactly as in the previous working approach.

For the rest of the controller, three relatively simple custom-topology neural networks (figure 5.21) were evolved simultaneously using the standard fitness function. The three networks respectively output longitudinal cyclic control, lateral cyclic control, and the collective pitch; the topologies of the networks are depicted in figure 5.23. The longitudinal network had the

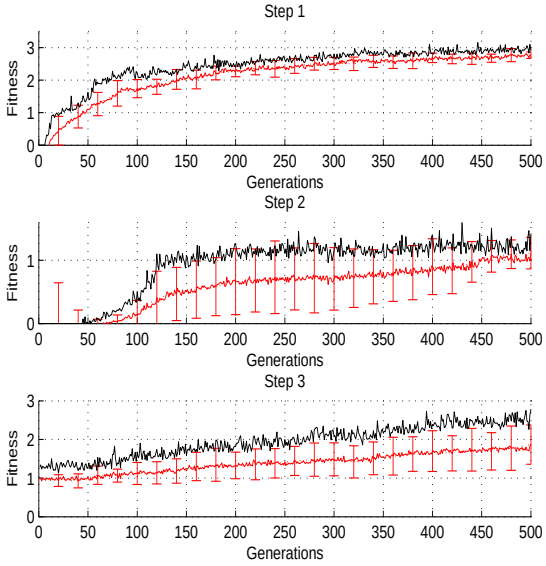


Figure 5.22: Evolving the substitution network in three steps. Best fitness (black line) and average of the best fitnesses (gray line) over 10 repetitions of the evolution are shown. Error bars show the standard deviation calculated every 20 generations.

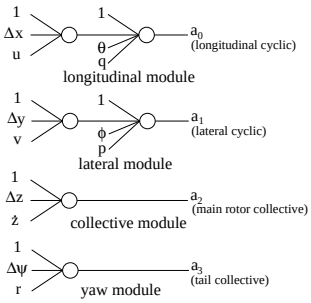


Figure 5.23: Topology of the modular network.

following inputs: longitudinal distance to waypoint (Δx), u, q and θ. Similarly, the lateral network made use of the lateral distance to the waypoint (Δy), v, p and ϕ. The collective pitch network used the difference in altitude of the waypoint (Δz) and $\dot{z}$.

The experiment was replicated ten times. The variability in fitness within the first few hundred generations was quite high, but within 500 generations very good performance was reached in all ten replications, as shown in figure 5.24.

Evolving PID gains

In order to obtain a controller to serve as a meaningful standard of comparison in our performance tests, we evolved the gains of PID controllers structurally identical to the handcrafted controller shipped with the simulator. As in the neuroevolutionary approaches above, this only

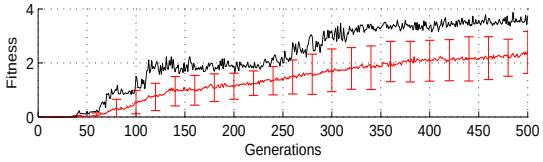


Figure 5.24: Evolving the modular network. Best fitness (black line) and average of the best fitnesses (gray line) over 10 repetitions of the evolution are shown. Error bars show the standard deviation calculated every 20 generations.

worked if yaw control was evolved first, with the rest of the PID gains "frozen" until the helicopter was stable around the z axis.

Performance comparison

However, to better understand the peculiarities of the different controllers, additional tests were performed on four tasks differing from the one used in the evolutionary process:

Task 1, Sparse waypoints

The first test was simply a repetition of the task used for the controller evolution, but over a much longer timespan. The extra time allowed for values of P_{chain} bigger than 1.0 since the helicopter was able to fly the whole waypoint chain more than once.

Task 2, Close waypoints

Closer waypoints were chosen for the second task, with the average distance between the waypoints now set to 6 feet.

Task 3, Sparse waypoints in presence of wind

The waypoints were chosen with the same criteria used for task 1, but an external disturbance in the form of wind gusts was added to the simulation in order to test the robustness of the controller. The wind was simulated as a time varying vector ($[0, 10]$ ft/s) added to the helicopter velocity.

Task 4, Sparse waypoints with varying gross weight

This task provided an understanding of the controllers' ability to handle variations in the helicopter's weight (and mass). The maximum random weight variation was set to fifty percent since the payload of a small helicopter can often reach this limit.

In addition, since the fitness function used for evolution gives only limited insight into the specific abilities of the controllers, three more specific performance indices were developed:

Table 5.12: Performance of the various controllers on different tasks

	Handcrafted PID	Evolved PID	Substitution Network	Modular Network
Close waypoints				
P_{chain}	0.96 (0.077)	1.85 (0.091)	2.27 (0.182)	**2.38** (0.106)
e_p	0.95 (0.216)	0.40 (0.048)	1.04 (0.21)	**0.34** (0.044)
e_h	**0.005** (0.003)	0.010 (0.002)	0.457 (0.002)	0.013 (0.001)
Sparse waypoints $[10, 25ft]$				
P_{chain}	0.46 (0.121)	0.74 (0.273)	1.35 (0.127)	**1.64** (0.081)
e_p	1.61 (0.265)	0.84 (0.192)	2.5 (0.748)	**0.63** (0.242)
e_h	**0.007** (0.001)	0.015 (0.004)	0.459 (0.003)	0.018 (0.002)
Sparse waypoints with wind $[0, 10ft/s]$				
P_{chain}	0.27 (0.231)	0.49 (0.330)	0.95 (0.434)	**1.07** (0.565)
e_p	1.99 (3.705)	2.58 (2.880)	3.54 (3.097)	**1.33** (1.651)
e_h	0.069 (0.083)	0.160 (0.216)	0.430 (0.146)	**0.062** (0.049)
Crash	7	2	2	4
Sparse waypoints with variable weight [±50%]				
P_{chain}	0.54 (0.128)	0.83 (0.235)	1.3 (0.170)	**1.53** (0.137)
e_p	1.68 (0.271)	0.80 (0.153)	2.28 (0.503)	**0.618** (0.112)
e_h	**0.012** (0.002)	0.017 (0.005)	0.48 (0.02)	0.031 (0.017)

- *progress along the waypoint chain P_{chain} (higher is better)*,
- *mean deviation from the shortest path $e_p = \sum_{i=0}^{N} \mid w_h \mid$ (lower is better)*,
- *mean heading error $e_h = \sum_{i=0}^{N} \mid \psi - \psi_{next} \mid$ (lower is better)*.

Table 5.12 shows the results of the tests; the best values are printed in bold. The waypoint layout was randomly generated at the start of every evaluation, and each task was repeated 20 times; the average value (and standard deviation) of the performances obtained is shown (in parentheses). The penalty for crashing into the ground was a fitness of 0, rather than a negative fitness as during evolution. On the wind task, the number of crashes in 20 trials is given; this is a measure of the ability of the controller to maintain stable flight. In all the test tasks the performance was evaluated during a predefined period of 3000 timesteps (corresponding to 60s of flight time).

Figures 5.25 and 5.26 show the trajectories of an evolved PID controller and an evolved modular neural network controller respectively, performing the same task (task 1) for 1100 timesteps. The PID exhibited a very conservative strategy, slowing down when still far away from the waypoint and almost stopping when close to it. The network-based controller instead retained a better control over the helicopter speed that produced a more linear trajectory and better progress along the waypoint chain.

Generally, the modular network (evolved without involving the PID) performed best in all four variations of the task, as it always progressed the furthest of the four controllers along the waypoint chain, and always moved in a more or less straight line to the waypoint. The

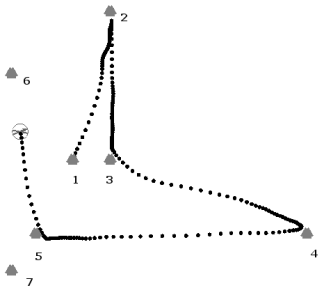


Figure 5.25: Trajectory of an evolved PID after completing 1100 timesteps of a typical sparse waypoint task. In this particular run the progress along the path (after 3000 timesteps) was 1.14, the mean trajectory error 0.39 and the mean heading error 0.019. The dots mark the position of the helicopter every 10 timesteps.

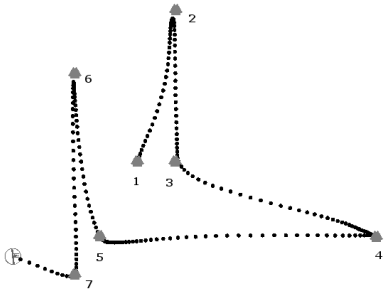


Figure 5.26: Trajectory of a modular network controller after completing 1100 timesteps of the sparse waypoint task used in Fig.6. At the and of the run (3000 timesteps) the progress along the path was 1.70, mean trajectory error 0.38 and mean heading error 0.016.

substitution network was a close second as far as progress along the chain was concerned, but had a more mixed performance when it came to deviation from the shortest path, and was the only controller that has any significant problems keeping the desired heading.

The performance of the PID controllers was much worse than the neural network controllers when it came to progress along the waypoint chain, except in the situation with the waypoints closer together, where its performance was comparable to (if slightly lower than) the neural networks. This suggests that the reason for the good performance of the neural network controllers was not only the superiority of evolutionary tuning over manual tuning, but also the lack of nonlinearity in the PID controllers. It simply does not seem possible to evolve a purely linear controller than is robust enough to perform well over the various task variations and performance measures we used here.

Conclusion

Both of the successful approaches we tried for evolving neural networks generated very capable controllers that significantly outperformed the human-designed PID controller. They also outperformed our attempts to control the vehicle manually by leaps and bounds. It is interesting to note that the evolved neural networks were quite robust when the parameters of the task, vehicle and environment were varied, something that could not be said about the PID controllers. This may be due to two factors: evolution is better at tuning weights and gains than are humans; and the nonlinearity of neural networks makes them better suited for handling such variations than linear controllers. Such robustness is obviously important when transferring controllers to real vehicles.

Without incorporating some domain knowledge in the evolutionary process, we have not been able to evolve a controller for doing anything more advanced than not crashing. Domain knowledge has been introduced either by using parts of a hand-designed PID controller, or by using knowledge of which inputs should be relevant to which outputs. Another form in which human judgement (though not really domain knowledge) was used was that the yaw control network always had to be evolved before the rest of the controller. This phenomenon resonates well with common wisdom in manual controller design: the simpler and less interconnected a controller is, the easier it is to tune.

Whether and how artificial evolution can be made to overcome this phenomenon is an open question.

5.4.2 Playing Cellz

In this short section, based on a paper presented at CIG 2005 [143], we discuss a comparison of different neural network-based controller architectures for playing the Cellz game, as described in section 4.2.2. The central hypothesis (which is borne out by the experiments) is that when a symmetry exists in the input array, higher fitness can be achieved if the structure of the neural network is modularised so that it is forced to take this symmetry into account. The paper details an additional experiment with a related function approximation task, and has some additional discussion.

In these experiments, each cell is equipped with eight cell sensors and eight food sensors spread evenly around its body (see figure 5.27); each sensor measures the distance to and concentration of other cellz or food in its 45 degree angle. The sensor arrays are used as inputs to the controllers, and their outputs are used to generate the force vectors controlling the cell.

Four different neural architectures were tested and compared, two "monolithic" and two "convoluted" . The first two architectures were standard multi-layer perceptrons (MLPs). The

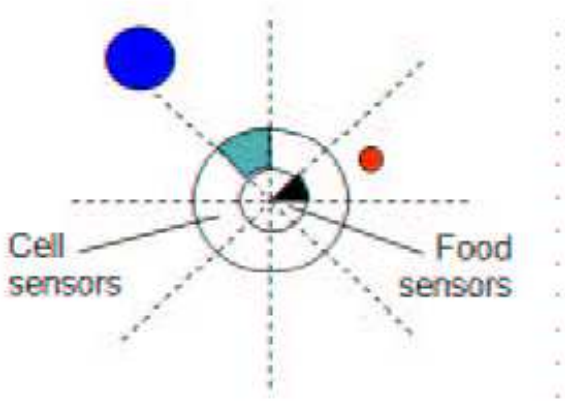


Figure 5.27: The wraparound sensors.

first MLP consisted of an input layer of 16 neurons, an 8 neuron hidden layer and an output layer of two neurons. The second MLP had two hidden neuron layers of 16 neurons each. In both architectures, positions 0-7 received inputs from the "food" input vector of the Cellz agent, positions 8-15 received inputs from the "cells" input vector, and the two outputs from the network were used to create the force vector of the cell.

The other two architectures consist of eight separate but identical neural network modules - they share the same genome. Each module can be thought of as assigned to its own pair of sensors, and thus being at the same angle r relative to the x axis as those sensors. The outputs from the each module's two output units is rotated r degrees, and then added to the summed force vector output of the controller.

In the convoluted architectures, each module gets the full range of sixteen inputs, but they are displaced according to the position of the module (e.g. module number 3 gets food inputs 3, 4, 5, 6, 7, 0, 1, 2, in that order, while the input array to module 7 starts with sensor 7; see figure 5.28). In the first convoluted architecture the modules lack hidden layer, but in the second convoluted architecture, each has a hidden layer of two neurons.

It is interesting to compare the number of synapses used in these architectures, as that number determines the network updating speed and the dimensionality of the search space. The MLP with 8 hidden neurons has 144 synapses, while the MLP with two hidden layers totals 544 synapses. The perceptron-style convoluted controller has 32 synapses per module, which sums to 256 synapses, and the hidden-layer convoluted controller has 36 synapses per module, which sums to 288 synapses. It should be noted that while the convoluted controllers have little or no advantage over the MLPs when it comes to updating speed, they present the evolutionary algorithm with a much smaller search space, as only 32 or 36 synapses are specified in the

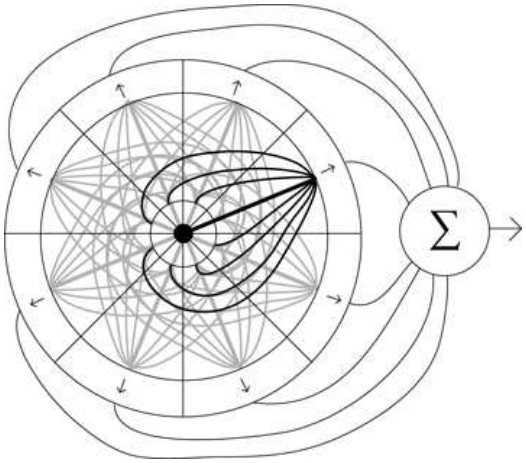


Figure 5.28: Simplified illustration of the convoluted architectures, taking only one type of sensor into account. The connections in black are the connections from all sensors to one module; this structure is repeated (grey lines) for each module.

genome.

All four architectures were evolved with a 5+25 evolution strategy; as the available computer power was limited, this was only done ten or so times for each configuration.

The two MLP architectures were evolved for 200 generations. The one-layer MLP evolved somewhat faster and reached a higher final fitness. Both architectures produced good controllers, whose agents generally head straight for the food, even though they fairly often fail to take their own momentum into consideration when approaching the food, overshoot the food particle and have to turn back.

Finally, the two convoluted controllers were evolved for 100 generations, and quickly generated very good solutions. The convoluted controller with a hidden layer narrowly outperformed the one without. Not only did good solutions evolve considerably faster than in the cases of the MLPs, but the best evolved convoluted controllers actually outperform the best evolved MLP controllers with a significant margin. As the computational capabilities of any of the convoluted controllers is a strict subset of the capabilities of the MLP with two hidden layers, this is slightly surprising, but can be explained with the extravagantly multidimensional search problems the MLPs present evolution with even if a better solution exists it is improbable that it would be found in reasonable time.

It is also interesting to note that the length of the neural path from sensor to actuator in the robots (that is, the number of hidden layers) seems to be of relatively small importance. A comparison between controllers evolved in this paper, the winner of the GECCO 2004 Cellz contest, and the hand designed controllers mentioned in the original Cellz paper (Lucas 2004) is presented in table 5.13. (The differences between the best three controllers are so small

Table 5.13: Mean fitness and standard deviations over 100 game runs for controllers mentioned in this paper in comparison to other noteworthy Cellz controllers.

Controller	Fitness	sd
JB-Smart 1.1 (Winner of GECCO 2004 Cellz contest)	1966	20
Convoluted controller with one hidden layer	1934	18
Hand-coded sensor controller	1920	26
MLP with one hidden layer	1460	13
Hand-coded greedy controller	1327	24
MLP with two hidden layers	1225	14

so as not to be statistically significant in small samples.) The convoluted controller with one hidden layer is one of the best controllers found so far (though the convolutional aspect of the controller was hand-designed). Note that the winner of the GECCO 2004 contest was also partially hand-designed, as a neural implementation of the hand-coded sensor controller. So far the purely evolved neural networks have been unable to compete with the networks that have been partially hand-crafted.

Our results clearly show that hard-coding modularity into neurocontrollers can increase evolvability significantly, at least when agents show symmetry, as they do in many computer games and robotics applications. They also show that certain types of modularity perform better than others, depending on the task at hand. Adding hidden neural layers might either increase or decrease evolvability, depending on the task. As neural encodings that intend to let modularity emerge always have certain biases, these results need to be taken into account when designing such an encoding.

Regarding the particular issue of rotational symmetry in sensors, a related set of experiments by Bryant and Miikkulainen is worth mentioning [16]. Bryant and Miikkulainen trained agents in a multi-agent turn-based strategy game, where each agent had a set of wrap-around sensors very similar to the ones used by the Cellz agents. However, instead of evolution they used backpropagation to imitate human behaviour, and instead of encoding the symmetry in the neural network the set of training data was rotated. This method significantly increased the performance of the controllers when faced with situations not encountered in the training data.

5.5 Summary

In this chapter, a number of experiments concerning the learning of control for well-defined tasks were described. The experiments can roughly be divided into foundational experiments in evolving car control, described in sections 5.1 and 5.2, auxiliary experiments in learning car control, described in section 5.3, and experiments in evolving control for other game domains, described in section 5.4. The foundational experiments in evolving car control establish that car controllers can be evolved using a particular experimental setup, where the car controller is

represented as a neural network and is fed data from a number of rangefinder wall sensors, a speed sensor and a way point sensor. They also show how general driving skills can be incrementally evolved, and how controllers can be specialized for very good performance on individual tracks. In addition, we show that a number of other combinations of controller architecture and sensor setup are apparently not possible to evolve good driving behaviour with.

The foundational experiments in evolving car control set the stage for all other car racing experiments in this thesis, as the methods developed in the foundational experiments are used and elaborated on in the later experiments. The auxiliary experiments in learning car control, on the other hand, corroborate, complement and elaborate on the results of the foundational experiments. A number of different controller architectures and learning methods were compared with complex and interesting results, but the original method of evolving neural networks proved to be at least as good as other tested methods in reliably coming up with good controllers. The experiments in evolving control for other game domains don't build directly on any of the car racing experiments, but a number of insights into what works and what doesn't work when it comes to evolving neural network controllers can be shared between these domains. Some of the key cross-domain insights concern the benefits of modularity, incrementality and ego-centric sensors, all of which are further discussed in section 2.4.

Chapter 6

Imitation

In this chapter we discuss three series of experiments where we *model* or *imitate* aspects of car racing. In the first section we model behaviour, in particular the driving styles of human players in the racing simulator. The second section looks at imitating not behaviour but dynamics, acquiring a models of a physical radio-controlled car that is subsequently used to evolve driving behaviour for the car. The third section briefly discusses another attempt at modelling dynamics, this time in the racing simulator and in a different transformation space.

6.1 Imitating driving styles

In two papers, presented at the SAB Workshop on Optimizing Player Satisfaction 2006 and CIG 2007 respectively, we developed a new method for personalised content creation in racing games; the project was done in collaboration with Renzo De Nardi [139][140]. The method consists in modelling the driving style of a human player and evolving a racing track for that particular player. The track evolution part of the project will be discussed in section 7.3. Those experiments use a controller trained to behave like a modelled human in important respects as part of the fitness function for tracks. More specifically, this controller will be driven on candidate tracks, and aspects of its performance on those tracks will be used in the fitness calculation for the tracks.

In this section we discuss our approach to player modelling, focusing on the method used in the later paper. The outcome of our modelling algorithm will be a controller, which in some respects behave like the modelled human, and can be used in the track evolution experiments. In the following, we will analyze what we actually need to model, describe the differences between direct and indirect modelling, and present the results of our modelling experiments.

6.1.1 When is a player model adequate?

The only complete model of a human player is the human player himself. This is both because human brains and sensory systems are rather more complex than anything machine learning can learn, and because of the limited amount of training data available from the few laps around a test track which is the most we can realistically expect a player to put up with. Further, it is likely that a controller that accurately reproduces the player's behaviour in some respects and circumstances works less well in others. Therefore we need to decide what features we want from the player model, and which features have higher priority than others.

As we want to use our model for evaluating fitness of tracks in an evolutionary algorithm, and evolutionary algorithms are known to exploit weaknesses in fitness function design, the highest priority for our model is robustness. This means that the controller does not act in ways that are grossly inconsistent with the modelled human, especially that it does not crash into walls when faced with a novel situation. The second criterion is that the model has the same average speed as the human on similar stretches of track, e.g. if the human drives fast on straight segments but slows down well before sharp turns, the controller should do the same. That the controller has a similar driving style to the human, e.g. drives in the middle of straight segments but close to the inner wall in smooth curves (if the human does so), is also important but has a lower priority.

6.1.2 Direct modelling

What we call direct modelling is what is arguably the most straightforward way of acquiring a player model: use supervised learning to associate the state of the car with the actions the human take given that car state. We let several human players drive test tracks, and logged the speed and the outputs of waypoint sensor and the wall sensors (as defined above) together with the action taken by the human at each timestep. Two methods of supervised learning were tried on this data set: training a multilayer perceptron for use in the controller with backpropagation, and using the unprocessed data for controlling the car with nearest neighbour classification of input data. Both methods resulted in worthless controllers that rarely completed a whole lap. While the trained neural networks were worthless in an uninteresting way, the nearest neighbour-based controllers reproduced the modelled players' driving style almost perfectly, until the slight random perturbations in the game presented the controller with a situation that differed enough from anything present in the training data, and the car crashed. None of the controllers were able to recover from crashes, as the human players had not crashed during the data collection, and thus the situation was not in the data set.

We believe this not to be a problem with the particular supervised learning algorithms we

used but rather an unavoidable problem with the direct modelling approach. As no model is perfect, controllers developed with direct modelling will tend to err, which diminish their performance to lower than the modelled human. In general, it is very unlikely that they will perform better than or as good as the modelled human (though it is theoretically possible that individual controllers could perform well), as any deviance from correct modelling will tend toward random behaviour. Such imperfect controllers will likely crash into walls, and will not know how to recover, as *the controllers can't learn from their mistakes*.

This problem was recognized by the developers of Forza Motorsport, who solved it by placing certain constraints on the types of tracks that were allowed in the game, and then recording the player's racing line over each possible track segment. Still, collisions with walls could not be entirely avoided, so a hard-coded crash-recovery behaviour was needed [58]. While this modelling method ostensibly works, it places far too many constraints on the tracks to be useful for our purposes.

6.1.3 Indirect modelling

Indirect modelling means measuring certain properties of the player's behaviour and somehow inferring a controller that displays the same properties. This approach has been taken by e.g. Yannakakis in a simplified version of the Pacman game [153]. In our case, we start from a neural network-based controller that has previously been evolved for robust but not optimal performance over a wide variety of tracks, as described in section 5.2. We then continue evolving this controller with the fitness function being how well its behaviour agrees with certain aspects of the human player's behaviour. This way we satisfy the top-priority robustness criterion, but we still need to decide on what fitness function to employ in order for the controller to satisfy the two other criteria described above, situational performance and driving style.

In our earlier paper, we measured the average driving speed of the human player on three tracks designed to represent different types of driving challenges, and then evolved controllers to match that performance as closely as possible on each of the three tracks. That method was successful, but could be argued to fail to capture much of the driving style of the player. The later paper used the following method, which is an attempt to model the driving in more detail while still using an indirect approach.

First of all, we design a test track, featuring a number of different types of racing challenges. The track, as pictured in (fig 6.1), has two long straight sections where the player can drive really fast (or choose not to), a long smooth curve, and a sequence of nasty sharp turns. Along the track are 30 waypoints, and when a human player drives the track, the way he passes each waypoint is recorded. What is recorded is the speed of the car when the waypoint is passed,

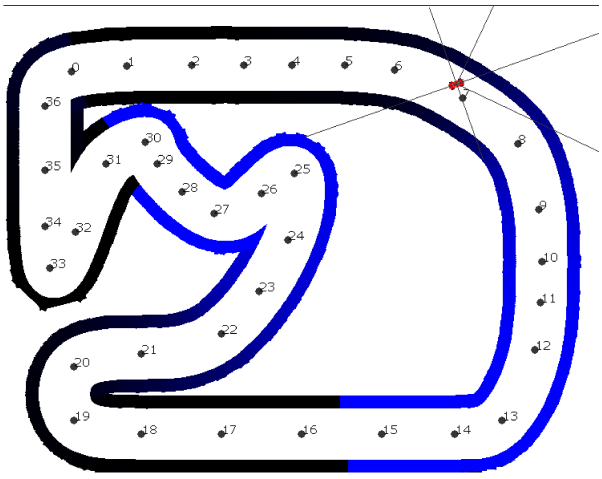


Figure 6.1: The test track and the car.

and the orthogonal deviation from the straight path between the waypoints, i.e. how far to the left or right of the waypoint the car passed. This matrix of $2 * 30$ values constitutes the raw data for the player model.

The actual player model is constructed using the Cascading Elitism algorithm (see section 2.1.3), starting from a general controller and evolving it on the test track. Three fitness functions are used, based on minimising the following differences between the real player and the controller:

- f_1: total progress (number of waypoints passed within 1500 timesteps),
- f_2: speed at which each waypoint was passed,
- f_3: orthogonal deviation where the way point was passed.

The first and most important fitness measure is thus total progress difference, followed by speed and deviation difference respectively.

6.1.4 Results

In our experiments, five different players' driving was sampled on the test track, and after 50 generations of the Cascading Elitism algorithm with a population of 100, controllers whose driving bore an acceptable degree of resemblance to the modelled humans had emerged. The total progress varied considerably between the five players - between 1.31 and 2.59 laps in 1500 timesteps - and this difference was faithfully replicated in the evolved controllers, which is to say that some controllers drove much faster than others (see the speed fitness in fig.6.2 and fig.6.3). Progress was made on the two other fitness measures as well, and though there was still some numerical difference between the real and modelled speed and orthogonal deviation at

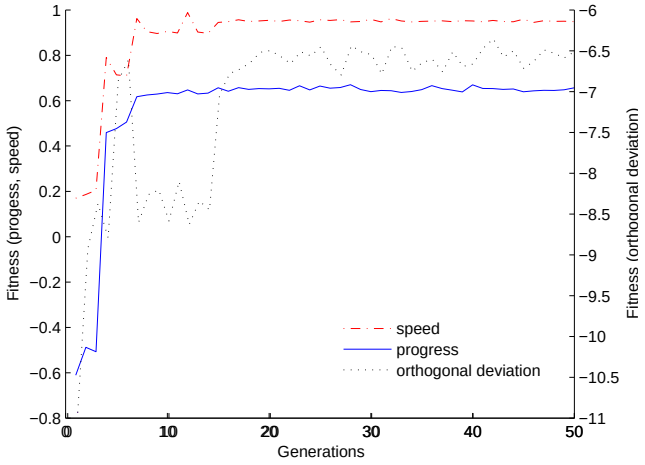


Figure 6.2: Evolving a controller to model a slow, careful driver. Since the initial general controller is quite well-performing, the evolutionary algorithm quickly adapts the driving style to obtain the required progress and speeds. At last also the ortogonal deviation fitness improves. See section 6.1.3 for the description of the fitnesses.

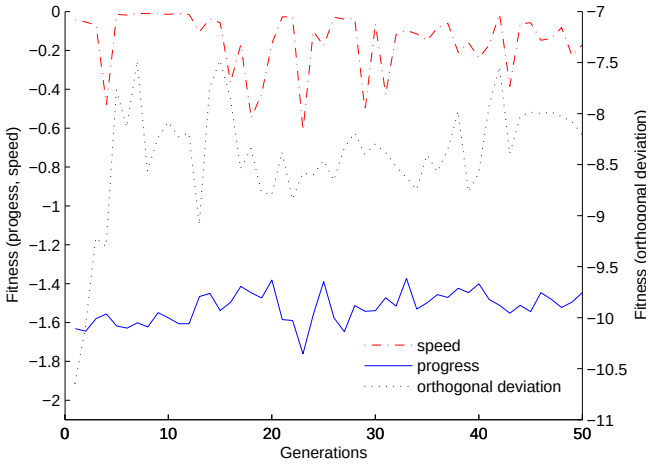


Figure 6.3: Evolving a controller to model a good driver. The lack of progress on minimising the progress difference is the result of the fact that the progress of the modelled driver is very close to that of the generic controller used to initialise the evolution. See section 6.1.3 for the description of the fitnesses.

most waypoint passings, the evolved controllers do reproduce qualitative aspects of the modelled players' driving. For example, the controller modelled on the first author drives very close to the wall in the long smooth curve, very fast on the straight paths, and smashes into the wall at the beginning of the first sharp turn. Conversely, the controller modelled on the anonymous and very careful driver who scored the lowest total progress crept along at a steady speed, always keeping to the center of the track.

Summing up, indirect modelling of behaviour produces player models which qualitatively seem to be reasonably faithful to the modelled human. We haven't attempted any quantitative validation of the accuracy of the models themselves, however in section 7.3 we demonstrate their usefulness for evolving tracks.

6.2 Imitating real car dynamics for controller evolution

The experiments detailed in this section should be seen in the context of the questions discussed in section 2.3.1: can we use a simulation to evolve controllers for a physical robot? If so, how do we acquire the models on which to base this simulation? In particular, how do we do this for a non-recoverable robot, that can break or otherwise can't easily be returned to its starting position and configuration. In this section, based on a paper presented at Gecco 2007 with Renzo De Nardi, Hugo Marques and Richard Newcome as co-authors, we investigate how to do this for the small radio-controlled car that is the qualitative basis of the car racing simulator [141].

6.2.1 Our approach to dynamics modelling and controller evolution

As discussed in section 2.3.2, one of the guiding principles of evolutionary robotics is to as far as possible exclude the human from the controller design process. The viewpoint taken in this section is that this should also be the case for the model acquisition. In other words, we are seeking to *minimise the amount of domain knowledge used to produce the model*. There are several reasons for this stance. One is that, similar to the argument for leaving controller design to evolution, we can potentially harness more of our learning algorithms' power and creativity. Another is that we want a method of model aquisition that can be used for modelling different types of robots (wheeled, winged, walking, whatever) with few or no changes. But the most important one is that for some systems we just don't have the domain knowledge available. For example, for the helicopters in the UltraSwarm project (see section 5.4.1) there is no dynamics model available.

In section 2.3.1, we discussed several approaches to acquiring models that can be used to evolve (or otherwise learn) controllers, particulary those of Bongard, Abbeel et al. and

Jakobi [11][1][65]. Here we contrast their approaches with the one we are currently taking.

While interesting, Bongard's approach seems to place certain requirements on the type of robot used, requirements which our system does not meet. Importantly, our model car is dynamical and all states are transient, whereas Bongard's robot can be kinematically modelled; it would also be possible to crash or otherwise lose control of our car during the experimentation phase performed by the algorithm.

Abbeel et al. succeed in modelling a car and a conventional single rotor helicopter, but not without putting in considerable domain knowledge, and even adapting their systems so as to be easier to model (such as voltage stabilisation on the car). This is all good if modelling a particular car or helicopter is the primary goal. We aim instead to address the more general problem of nonlinear modelling for controller evolution when the assumptions about the underlying system being modelled are relaxed.

As for Jakobi's approach, the reliance on a human to separate base set from implementation set clearly violates our goal of minimizing domain knowledge.

6.2.2 Model requirements for controller evolution

As stated above, we are interested in system identification as a means of making it possible to evolve controllers, and the requirements we have on the models we infer are thus likely to differ from those placed on models used in e.g. traditional control theory. These differences stem from evolutionary computation's well known tendency to exploit the model or fitness function at hand. As trying randomly generated strategies is essential to evolution, the candidate controllers are likely to take actions which are very different from any actions in the training data for the model, or which otherwise "break" the model so that it produces responses which are wildly different from the system it is intended to model. Such weak spots in the model can often be exploited by the evolutionary algorithm to create controllers that achieve good fitness, but are completely useless in reality. An example of such an exploit for a car racing model would be if the model moved sideways when a steering command was issued while the car was standing still. A model with such a deficiency could well be learnt if the situation (steering while not driving) was not in the training data, and the right constraints were not applied to what sort of dynamics could be learnt.

That the models behave in a way which adheres to the constraints implied by the system being modelled is paramount, however beyond learning these large scale constraints, acquisition of non-exploitable models is more important than achieving high precision with a specific training set.

However, there are ways in which the model could fail to conform to the modelled system and still be usable for controller evolution. In general, we believe that deficiencies that make it harder for the controller to perform its task (e.g. driving to a way point) is admissible, whereas deficiencies that make it easier to succeed are not. For concrete examples in our current domain, we believe that it is preferable that the model overestimates the turning radius of the car to that it underestimates it, that too long lag between command and effect is preferable to too short lag, and that it is probably acceptable if the model misjudges all accelerations by some constant, but absolutely not that the model makes it possible to turn the car on the spot.

That being said, we want to avoid encoding any of the above ideas into our model representations, in the form of explicit constraints on dynamics or otherwise, in line with our principle of minimising our use of domain knowledge.

6.3 Data collection

The car we used is a small and inexpensive radio controlled toy car with a mass of approximately $0.3Kg$ and a length and width of respectively $18cm$ and $10cm$ (Figure 6.4). The car is provided only with simple bang-bang control inputs: forward or backward at full throttle and steering right or left. The combination of the asymmetry in turning and the slack of the worn out differential drive gearbox makes controlling the car a non-trivial task. The only modification made to the car was the addition of five very light and reflective fiducial markers necessary for tracking. No electrical modifications of any kind were made, which means that the battery level had a direct impact on the car's top speed and responsiveness. The remote control of the car was modified in order to be controlled by the parallel port of our computer. A simple Java program outputs the steering commands given by a human driver via keyboard to the parallel port, and logs the commands together with the current car state.

The current car state was obtained from a Vicon infrared tracking system that can record the position of the markers placed on the car with a very high accuracy (in the order of millimeters) and a frame-rate of $200fps$. The real time data reconstruction delay offered by the system is also very low (in the order of milliseconds) and being considerably shorter than our control speed ($20Hz$) can be safely neglected. All the 50 square meters that constitute the test area were covered with white paper to reduce (but not eliminate) skidding of the car and infrared reflectivity of the floor.

The tracking system produces the absolute car position with reference to a coordinate frame fixed to the test area (a third person perspective), which is translated into the car centered frame of reference and numerically differentiated to produce the car state. As it can be expected the

Figure 6.4: The toy car and its remote, note the reflecting markers.

process of numerical differentiation introduces noise in the computed velocities. In order to limit the noise, we record the data with a time resolution of $200Hz$ and compute the velocities, which are then low-pass filtered and down-sampled to $20Hz$. To avoid distortion in the data a FIR (finite impulse response) filter was used and the delay introduced by the filtering process was compensated for. The control commands were logged together with the car state.

As it is common practice in vehicular dynamics we define the state of the car as the set constituted by its forward, lateral and angular velocities $[u, v, \dot{\psi}]$. The derivative of the state [1] $[\dot{u}, \dot{v}, \ddot{\psi}]$ will be used for the corresponding accelerations; u_1 and u_2 indicate the driving and steering input.

In order to minimise the use of domain knowledge, we decided not to collect data while a human driver was performing the point-to-point car racing task (see section 4.1.3), but instead while performing a set of various driving manoeuvres that we regarded to be representative of the space of possible driving maneuvres. In total, about 6 minutes of driving data was collected with different manoeuvres ranging from loops and figure-8s to simple starting and stopping. Visual inspection of the collected data indicated that the lag in the loop (from sending command to acquiring position information) was negligible compared to the $20Hz$ frequency of the controller, and thus did not need to be modelled.

[1] Computed as $[\dot{u}, \dot{v}]_t = R(\dot{\psi}_t)^{-1} * [u, v]_{t+1} - [u, v]_t$, $\ddot{\psi}_t = \dot{\psi}_{t+1} - \dot{\psi}_t$. Where $R(\dot{\psi}_t)^{-1}$ denotes the rotation matrix between the body frame at time $t+1$ and t. Throughout the paper $[u, v, \psi]$ and $[\dot{u}, \dot{v}, \ddot{\psi}]$ refers to the state and acceleration at time t.

6.3.1 Modelling techniques

Four different function representations, with different associated learning methods, were used to learn models of the car: a function approximator based on twenty-seven MLPs (multi-layer perceptrons) trained with back-propagation (*backprop27*), another based on three MLPs aided by an integration routine and trained with artificial evolution (*evolved3*), a simple nearest neighbour classifier, also using an integration routine, and a parametrized model of the car based on physical insight, with its parameters set by evolution.

In the nonlinear system identification literature, colour-coding is traditionally used to distinguish the level of prior knowledge that is available:

- *White Box models*: the model is perfectly known,
- *Grey Box models*: the model structure is known based on some physical insight, but its parameters remain to be estimated,
- *Black Box models*: prior knowledge is not used.

The *backprop27* is the darkest model we produced as it produces the next state of the system based on its current state and on the control inputs. The only handcrafted information pushed through the system was the choice of the neural network used depending on the active control signal (see below). The *evolved3*, the nearest neighbour and the parametrised models are grey-box models as they all assume a physical system. Given a state and a control action, they only produce the derivative of the state vector, a sound assumption for any physical system actuated by forces and torques. The *evolved3* and the nearest neighbour models are very dark grey boxes; they simply predict accelerations, leaving the computation of the new state to an external routine that takes care of the integration. In contrast, the parametrized model incorporates substantially more domain knowledge, as it explicitly tries to model coupling effects and friction that are known to be present in the real car.

Back-propagation MLPs

Our first model architecture does not assume anything about the system to be modelled, but rather about the space of inputs. Since the car has a relatively small set of action commands, and since those actions are discrete, it is possible to to take advantage of this by using a separate neural network for each possible command. This would possibly give each network a simpler function to learn, not only by reducing the output space, but also by eliminating the motor commands from the input space (as the appropriate network outputs can be selected solely on the base of the action input). In total we used 27 networks (9 possible actions $\times$ 3DoF) receiving as inputs the full state ($[u,v,\dot{\psi}]$) and the usual bias. The networks were then trained to predict

the state of the system at the next time-step. We used the standard back-propagation algorithm based on the square error between the model output and the logged car data for the training. All the almost 7000 data point logged were repeatedly used in the 1000 training epochs.

Evolved MLPs

In this second attempt we raise the level of domain knowledge in the model by assuming that the state of the system is constituted by physical quantities that can be computed by integrating acceleration, direct results of states and input commands.

Three separated MLPs are in this case evolved to produce the difference between the current and the future state of the system. Each network uses as input the current state of the system and the control command and produces the change in one of the states. The weights of the neural network were then evolved using a standard 50+50 evolution strategy, encoding the weights of the networks as real numbers, and mutating them with Gaussian mutation with variance 0.01.

The fitness function was the mean square error between the predicted state and the ground truth state logged for the car, calculated as follows. A starting point in the data was picked at random, and the state of the simulated car was initialised to be the same as the state of the real car. The same commands were then fed to the simulated car as to the real, and the state of the simulated car updated using the model under evaluation, for 10 time steps, after which the square difference between the real and the simulated state was calculated. The fitness of a model was defined as the negative mean of these errors over 100 repetitions of this process.

The *evolved3* models were generally evolved for 1000 generations, though they consistently achieved peak fitness after a few hundred generations.

Nearest neighbour

Like the *evolved3* models, but unlike the *backprop27* models, the nearest neighbour-based model maps current state and command to accelerations. The model is very simple and consists in having the real car data organized in nine tables, one for each control command, each entry mapping a recorded state $[u, v, \dot{\psi}]$ to a current acceleration $[\dot{u}, \dot{v}, \dot{\psi}]$. When predicting, given the current control command and state, the algorithm finds the state entry in the associated table with the smallest Euclidean distance and adds the corresponding acceleration entry to the current state.

Parametrised model

The parametrised model we adopted is inspired by the model presented in [1], and is mainly a fruit of the insight on the underlying physics we gained by driving the car and studying the

logged data. It consists in a set of algebraic equations, the parameters of which will be fitted to best approximate the car data. Since the lateral translations are generally very small, we deliberately neglected the lateral motion. There were three principal effects we wanted the model to be able to reproduce: 1) there can be asymmetry between forward and backward motion and right and left turning; 2) it is not possible to turn the car on the spot; 3) the motion of the car is characterised by static and dynamic friction. To achieve the first requirement we simply allowed for four different proportional constants between the control input and the respective accelerations (forward acceleration $\dot{\mathrm{u}}_{\mathrm{f}}$, backward acceleration $\dot{\mathrm{u}}_{\mathrm{b}}$, right angular acceleration $\ddot{\Psi}_{\mathrm{r}}$, left angular acceleration $\ddot{\Psi}_{\mathrm{r}}$). As suggested in the model by Abbeel, to stop the model from spinning on the spot we simply defined the rotational speed as a proportion of the forward speed (see equation 6.1). And finally to account for static and dynamic friction we defined minimum velocity factors (minimum forward velocity u_{m}, minimum angular velocity $\dot{\Psi}_{\mathrm{m}}$) and linear and viscous drag (linear drag D_{ul}, linear viscous drag D_{uv}, angular drag $\mathrm{D}_{\Psi\mathrm{l}}$, angular viscous drag $\mathrm{D}_{\Psi\mathrm{v}}$). Additional safety factors were also defined to avoid unrealistic velocities (maximum linear velocity u_{M}, maximim angular velocity $\dot{\Psi}_{\mathrm{M}}$).

$$
\begin{aligned}
\ddot{\psi}_d &= \dot{\psi}\mathrm{D}_{\psi l} + sgn(\psi)\dot{\psi}^2\mathrm{D}_{\psi v} \\
\ddot{\psi} &= \mathbf{1}(|\ddot{\psi}| > \ddot{\Psi}_{\mathrm{M}})(\mathbf{1}(u_2 = 1)u\ddot{\Psi}_{\mathrm{r}} - \mathbf{1}(u_2 = -1)\ddot{\psi}\ddot{\Psi}_{\mathrm{l}}) - \ddot{\psi}_d \\
\dot{\psi} &= \mathbf{1}(|\dot{\psi}| < \dot{\Psi}_{\mathrm{m}})u + \mathbf{1}(|\dot{\psi}| \geq \dot{\Psi}_{\mathrm{m}})sgn(\dot{\psi})\dot{\Psi}_{\mathrm{M}} \\
\dot{u}_d &= u\mathrm{D}_{ul} + sgn(u)u^2\mathrm{D}_{uv} \\
\dot{u} &= \mathbf{1}(|\dot{u}| > \dot{\mathrm{u}}_{\mathrm{M}})(\mathbf{1}(u_1 = 1)\dot{\mathrm{u}}_{\mathrm{f}} - \mathbf{1}(u_1 = -1)\dot{\mathrm{u}}_{\mathrm{b}}) - \dot{u}_d \\
u &= \mathbf{1}(|u| < \mathrm{u}_{\mathrm{m}})u + \mathbf{1}(|u| \geq \mathrm{u}_{\mathrm{m}})sgn(u)\mathrm{u}_{\mathrm{M}}
\end{aligned}
\tag{6.1}
$$

The 12 parameters that fully define the model were then evolved using the same evolution strategy and fitness function as described in section 6.3.1. In this case the parameters were encoded as arrays of real numbers and evolved with the same 50 + 50 elitist scheme adopted for the 3MLPs model. As with the MLPs, the parametrised models were evolved for 1000 generations but peaked after a few hundred.

Single- and multi-model controller evolution

The final modelling technique we introduce here is multi-model evolution: we evolved two extra controllers using more than one model, at each controller evaluation the controller was tested using two or three of the best evolved models, acquired using different techniques and representations; the fitness used was the lowest fitness of those achieved. In this way, we reasoned, any evolved strategy that relied on exploiting a weakness of a particular model would score

models	u	v	$\dot{\psi}$
backprop27	0.6528 (0.6526) [58.90]	0.0832 (0.0832) [106.58]	1.3036 (1.3033) [80.35]
parametrised	0.3213 (0.3211) [28.99]	0.0669 (0.0668) [85.63]	0.5294 (0.5291) [32.63]
evolved3	0.6668 (0.6666) [60.17]	0.1498 (0.1497) [191.80]	1.0061 (1.0055) [62.01]
n. neighbour	0.3164 (0.3157) [28.55]	0.0223 (0.0223) [28.58]	0.4122 (0.4114) [25.41]

Table 6.1: The root mean squared error $[m/s]$, (standard deviation $[m/s]$) and [root relative error %] of each model in the testing data.

badly, as this particular weakness would not be present in the other models (but rather other weaknesses). According to this hypothesis, the extent to which we can avoid any systematic weaknesses that plague *all* our models depends both on the quality of the training data and the diversity of function representations and learning algorithms used to acquire the different models. A certain kinship with Jakobi's radical envelope of noise hypothesis can be seen in that the use of multiple models can be said to implicitly separate a base set (properties that can be modelled without exploitable weaknesses) from an implementation set (those that can not).

The performance of each controller was then tested with each one of the models and finally with the best model of all: the physical car.

6.3.2 Model acquisition experiments

Using the various architectures discussed, several models were obtained and selected for controller evolution. In the case of the *n. neighbour* and *backprop27* only one model was produced for each technique; with both *parametrised* and *evolved3* the best models produced after the first evolutionary run had complete were chosen. The accuracy of the selected models were then verified using a validation dataset held back during training. In testing the models are initially given the state from the values of the real car. Each model is given the control signals (u_1 and u_2) recorded from the real car and at each time step the predicted state is propagated as the next state. In this way, we are able to see the deviation of each model from the baseline given by the testing data. In Table 6.1 are shown the root mean squared error (RMSE) of the predictions of the velocities (u, v and $\dot{\psi}$) when compared with the testing data.

A plot of the predictions made by each model is shown in Figs 6.5 and 6.6 using a subset of the testing data. Here we show only the results of the variables u and $\dot{\psi}$, since these are the

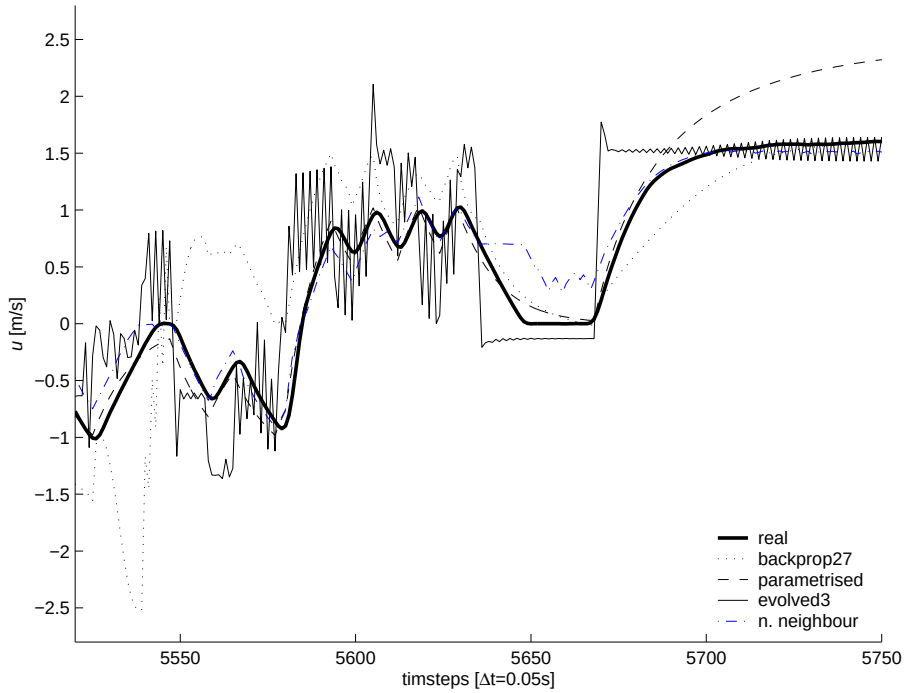


Figure 6.5: Predicted forward speed u for each of the models acquired

ones that have greater impact on the quality of the model.

The nearest neighbour model sometimes follows the baseline accurately and other times misses it completely. Without any form of interpolation, the nearest neighbour algorithm understandably fails to generalise in novel situations which are outside the envelope of the training data. In addition, the parametrised model often produces overestimates and sometimes is slow to react to the control commands. The set of 27 networks trained with back-propagation can follow the baseline relatively closely, however, as seen in the example plots large errors in the predictions can occur over short periods of time (spikes). A quite different behaviour is seen in the predictions produced by the *evolved*3 model which has high frequency oscillations of varying amplitude.

6.3.3 Controller learning experiments

Once a set of models had been derived we proceeded to investigate their usefulness as simulators with which to evolve controllers. The task we chose to evolve controllers for was point-to-point car racing, as defined in section 4.1.3 (except for the dynamics defined there, of course). The reasons for choosing this task over the more complex walled car racing task is that it does not require modelling of collisions with walls.

The fitness function of the task is defined as follows: the controller is allowed to control the car for 500 time steps, equivalent to 25 seconds of simulated time. During this time, the car has to pass as many way points as possible, and the fitness is equal to the number of passed way

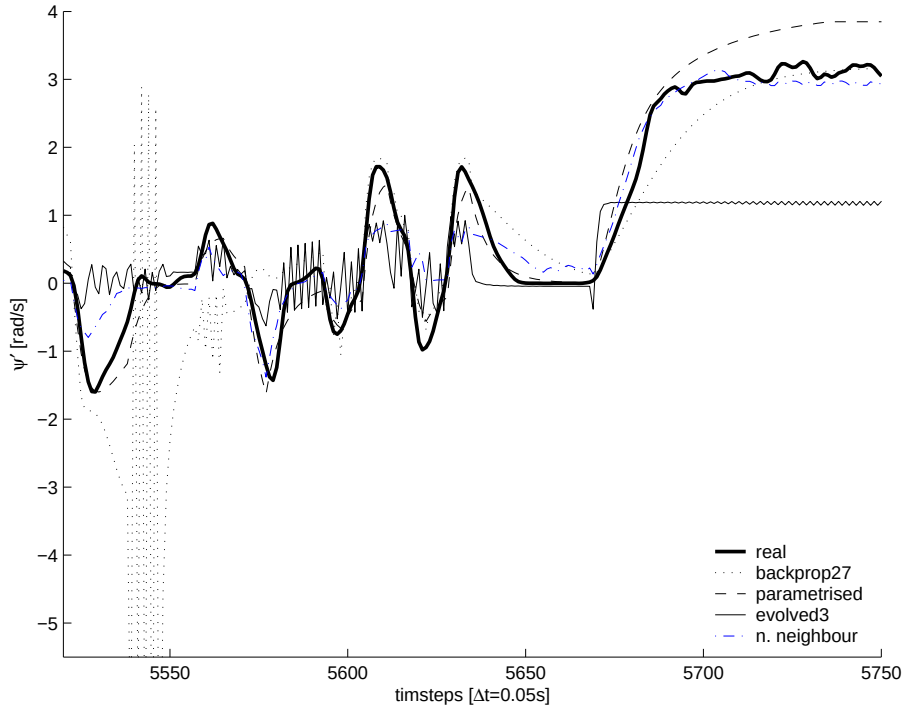


Figure 6.6: Predicted angular speed $\dot{\psi}$ for each of the models acquired

points at the end of the 500 time steps. A way point is considered passed when the center of the car is within 30 centimeters of the centre of the way point; when a way point is passed, a new way point immediately pops up at a random position within a radius of 1.5 meters from the centre of the arena. Only one next way point is available to pass at any one time.

The controllers we evolve are based on recurrent neural networks with 5 inputs, 6 hidden neurons, and 2 outputs. The inputs are as follows: a constant bias input of 1, the forward speed of the car (u), the rotational velocity of the car ($\dot{\psi}$), the angle to the next way point (relative angle between the forward direction of the car and the line that connects the center of the car and the waypoint), and the Euclidean distance to the next way point. All inputs are in SI units. The two outputs are interpreted as follows: the car is sent the command to steer left if the first output is below -0.3, to steer right if above 0.3 and straight forward otherwise. Similarly, the value of the second output means drive backward if below -0.3, forward if above 0.3 and neutral otherwise.

As for the evolutionary algorithm, the very same evolution strategy is used as is used for the evolutionary model acquisition above. All weights of the neural network are mutated in parallel by adding numbers drawn from a Gaussian distribution. Each fitness evaluation is the mean of ten trials of 500 time steps each.

contr.	bp27	param	evo3	nn	real
bp27c	18.6 (0.76)	9 (1.19)	1.3 (0.61)	14 (1.04)	9.33 -
paramc	18.1 (0.79)	15.1 (0.66)	1.3 (0.43)	15.5 (0.62)	14 -
evo3c	0.7 (0.52)	0.2 (0.19)	13.4 (0.87)	0.3 (0.29)	0.33 -
nnc	15 (1.22)	5 (1.45)	0.7 (0.43)	17.0 (0.83)	5.33 -
mm1c	11 (1.57)	4 (1.3)	15.0 (0.95)	12 (1.40)	8.33 -
mm2c	13.6 (0.72)	10 (1.04)	10.2 (0.75)	11.7 (0.89)	9.33 -
humFc	- -	- -	- -	- -	8.33 -
humBc	- -	- -	- -	- -	6.0 -

Table 6.2: Fitness and standard deviation (below) of each controller on each model: the columns refer to the models and the rows to the controllers.

Results

Controllers that successfully completed the point-to-point racing task within a given model could reliably be evolved with that model within a few hundred generations (see figure 6.7). However, this is more or less a tautology. Things start to get interesting when controllers evolved for one model are tested on another, and really interesting when they are tested on the real car. In table 6.2 we can see the results of a number of such tests. The table contains four controllers derived using the modelling techniques described: nearest neighbour, backpropagation, neuroevolution, evolutionary parameter optimisation (*nnc*, *bp27c*, *evo3c*, *paramc*), two controllers derived using versions of the multi-model controller evolution (*mm1c* being a combination of *evo3* and *nn*, and *mm2c* being a combination of *bp27*, *evo3* and *nn*), and the best efforts of one of the authors driving either forward or backward (*humFc, humBc*). The controllers tested were the best of two evolutionary runs on the same model; no attempt at measuring the fitness variance between evolutionary runs was made, but we believe it to be low.

As is clear from the table, a controller evolved using a particular model works better on that model than on any other. This effect is most pronounced for the controller produced on the evolved neural network model, but is present for the other controllers as well. Qualitatively, the evolved neural network model behaves quite differently to the other models, with abrupt accelerations and huge turning radii. The *backprop27* model is generally the one that "feels most natural", while the nearest neighbour model behaves just like the real car in many situations only to behave rather inappropriately in situations for which it has no data.

The only controller that consistently performs well on all models is one of the multi-model-

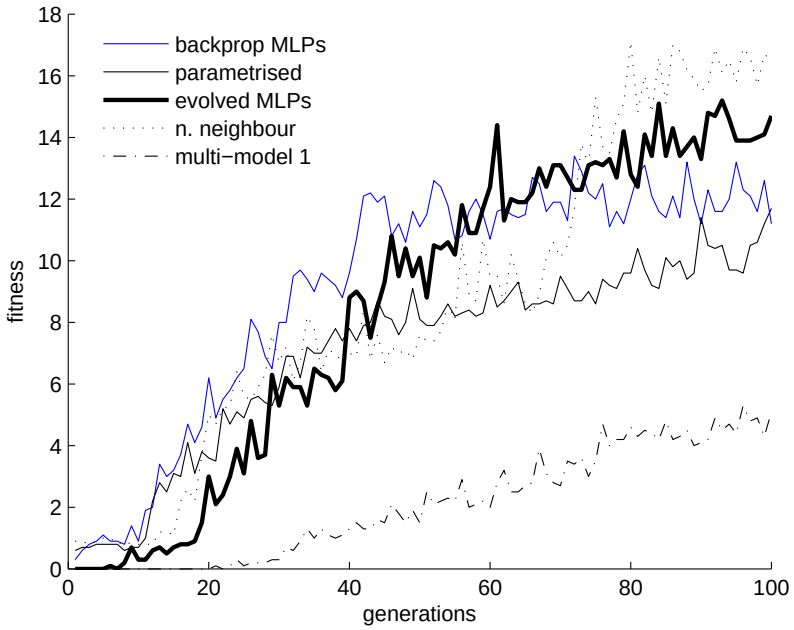


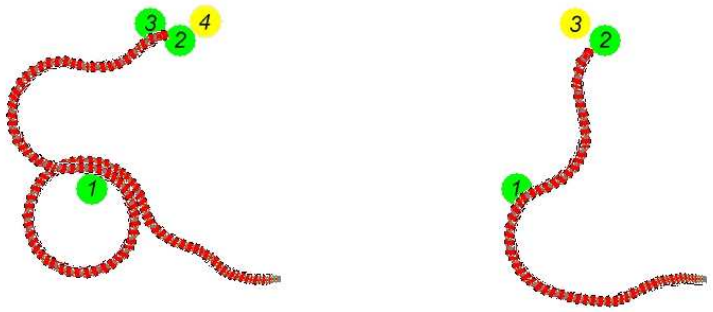
Figure 6.7: Evolution of controllers.

evolved controllers, *mm2c*. This controller also produced the most robust behaviour on the real car, even if the highest mean fitness on the real car was achieved by the controller evolved on the *paramc* model (the high fitness of that controller is only because of the limited length of the trials; on longer trials, *mm2c*-evolved controllers would have scored higher than *paramc*-evolved controller). However, as we shall see below, these controllers go about their task in rather different ways. This is illustrated by figure 6.8, that traces a few seconds of several controllers trying to reach the same sequence of way points with the real car.

Analysis of evolved control strategies

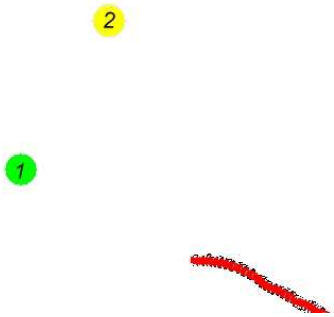
When transferring the evolved controllers to the real car it was observed that almost all of them drive the car backwards; all except the one which was evolved based on the 3-model multi-model trick (*mm2c*). The advantage of driving backwards seems to be that the car is more maneuverable; the car does go faster when driving forward, but this is apparently not very important given the limited size of the arena and the time taken to accelerate and decelerate.

Apart from mostly driving backwards, the behaviour of the controllers vary wildly. The controller evolved on the parametrised model (*paramc*) is perhaps the most straightforward (or straight-backward) as it drives directly towards the way point at full speed at all times. This works very well when the angle between the car and the way point is small or the distance to the way point is high, but if the next way point pops up right next to the car, the controller gets stuck "orbiting" around the way point, driving in endless circles without being able to reach it

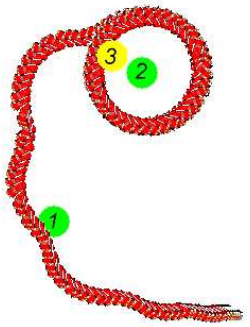


(a) *backprop27* model.

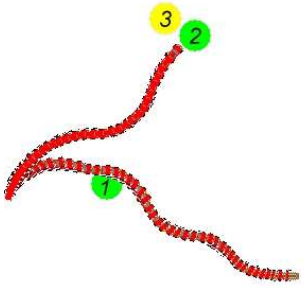
(b) *parametrised* model



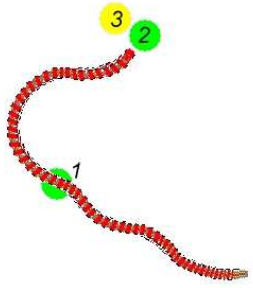
(c) *evolved3* model



(d) *nearest neighbour* model



(e) *2 model multi-model*: *nearest neighbour* and *evolved3* models



(f) *3 model multi-model*: *nearest neighbour*, *evolved3* and *parametrised* models

Figure 6.8: Traces of a number of different controllers, evolved using single models (a, b, c, d) or combinations of those models (e, f), all tested on the real physical car with the same way point configuration.

as its turning radius is too large. A variation on this behaviour is exhibited by the on average best-performing controller (*bp27c*) evolved on the *backprop27* model. This controller almost always turns right; approaching a way point it aims slightly left of the point and then does an inexplicably sharp right turn just before passing the way point. (We would have suspected an evolutionary exploit of a weakness in the model if the model was not actual physical reality.) The strategy works fabulously for most way points, but even this controller sometimes gets stuck in orbit, and some trials get very bad fitness. The reason for its high average fitness is that the trials are short, and switching between two trials usually gets the car out of orbiting.

The only controller that seems never to get stuck is the one evolved on the 2-model multi-model controller (*mm1c*) evolution. While this controller quite often misses a way point, it usually immediately brakes to a full stop just afterwards. When accelerating again, the car has a narrower turning radius than it would have at full speed, and is thus able to turn back and reach the way point. This strategy surprised us at first, as we had not thought of it ourselves, but it obviously works.

Another strategy, as displayed by the 3-model multi-model evolved controller, is to drive forward most of the time, trying to aim for the way point. When missing a way point the car backs off some distance and fully repositions so as to be able to reach the way point on a second try. The controller evolved on the 3-MLP model (*evo3c*) seems to perform some sort of peculiar dance around the arena, with little relation to the position of the way point or to the behaviour exhibited by the same controller in the model it was evolved for.

Analysis of strengths and (exploitable) weaknesses of the models

As the controller representation and evolutionary method are kept constant for all attempts at controller evolution, the characteristics of the models are bound to be the sole determinants of the differing fitnesses and behaviours of the evolved controllers. In this section, we attempt to analyse the strengths and weaknesses of these models based on observations when driving them manually, and observations on the evolved controllers driving in their native models.

The 3-MLP model is the most obviously exploitable model, as its very high rates of acceleration from standstill makes a kind of zig-zagging strategy possible, which certainly would not work on the real car. As the top speed of the model is not very high, but the turning radius is, it is all too tempting to use this exploit to quickly get to a way point. However, we note that the 3-MLP model works well in combination with another model for multi-model controller evolution, as its high turning radius precludes a turning radius exploit and the zig-zagging won't work in any other model.

The parametrised model, which is the model incorporating the most domain knowledge, is

generally admirably well-behaved. But it is still vulnerable to turning radius exploitation, as its turning becomes unrealistically sharp at very low speeds. We have seen controllers evolved for this model driving fast and straight, and suddenly slowing down and creeping around the turns. A similar exploit seems to exist for the nearest-neighbour model. A more serious exploit for that model, however, is the spinning on the spot phenomenon, whereby the car can turn around without moving forward after certain decelerations.

For the *backprop27* model, things look different: when driving fast, and suddenly braking and changing the steering at the same time, the car can suddenly accelerate in unexpected directions. This little oddity seems not to be exploitable by the controller, but we have seen its slight underestimation of the car's turning radius being exploited.

6.3.4 Discussion

The approach to dynamics modelling and controller evolution presented in this paper apparently works well enough to produce proficient (and interesting) controllers for toy car racing, using very little domain knowledge and an ill-behaved toy car. While others have been able to learn competent control of radio-controlled toy cars, we believe we are by far assuming the least about the system we are modelling. Minimising use of domain knowledge might be seen as an academic concern in the current context, but becomes all the more important in other domains, where domain knowledge is often lacking. (Also, there is nothing wrong with academic concerns.) Additionally, our concern ties in very well with the general spirit of evolutionary robotics, which is to minimise human interference in the process of controller design. We hope lessons learned and techniques developed here, in particular the use of multi-model controller evolution (which to some extent was the "trick" that finally made our experiments work) will be transferable to other domains. However, the results can currently only be considered tentative, and the multi-model evolution "trick" might not work in other domains. Especially, one might suspect that vehicles which have higher-dimensional dynamics (e.g. helicopters) will also have more complicated exploitable weaknesses, which will be harder to cancel out.

6.4 Other types of imitation

It is of course possible to model both behaviour and dynamics in other domains than car racing. For other agent games approaches similar to the ones discussed in the two sections above could easily be taken. Even for computerized games and management games, it seems very possible to at least indirectly model playing styles. In this thesis, however, we are not providing any examples of imitation outside of the car racing domain.

However, we will briefly discuss a different sort of dynamic modelling in the car racing domain, this time using only the simulated version of car racing and with transformations taking place in sensor space.

6.4.1 Imitating simulated car dynamics in sensor space

This section is based on a paper presented at IEEE-Alife 2007 with Hugo Marques as first author, and Magdalena Kogutowska as third [83]. When comparing this section, where the paper is briefly discussed, it is worth remembering that the work here is chronologically earlier than the work discussed in the above section on dynamics modelling for the physical radio-controlled car.

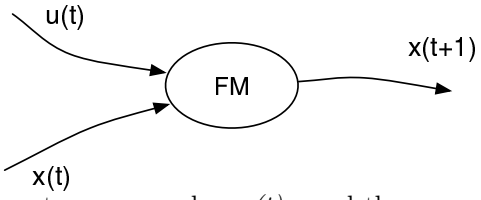


Figure 6.9: Given a set of motor commands - *u(t)* - and the current state - *x(t)* - the forward model generates a prediction of the outcome - *x(t+1)*.

The problem addressed here is how to predict sensor data based on earlier sensor data and actions taken, in a complex environment with walls (the full track-based racing game is used, with a fixed sensor array). In other words, we are learning *forward models* (see figure 6.9). In effect, this means both imitating the dynamics of the car - but in an "unnatural" transformation space - and learning the layout of the track, which is arguably a more complex task than only learning the car dynamics, as is done in the previous section. (At least if the dynamics are similar in both cases. We don't currently know whether the dynamics of the simulated or real car are more complex.)

In this section several ways of acquiring the required models are compared. The central questions we try to answer is whether good enough forward models can be learned, and whether there are qualitative as well as quantitative differences between forward models learned based on minimization of error and those learned based on maximization of performance.

To test our forward models, we developed two different tasks, which are extension to the standard task of driving around a track in track-based racing game.

Tasks

Imagine driving in an unlit tunnel, in a car whose headlights has the irritating habit of flickering on and off, sometimes staying off for several seconds at a time. Or imagine remotely controlling a vehicle based on an unreliable image feed that blacks out from time to time. These scenarios

are the motivation for the intermittent task. The delay task, on the other hand, is motivated by the delay in car state information reaching the controlling computer from an overhead webcam in some of our real-world experiments.

Technically speaking, both tasks change the sensor model of the car. In the intermittent task, there are two possible states

- The normal state, where current sensor information is presented to the controller, or
- The blackout state, with all sensors off.

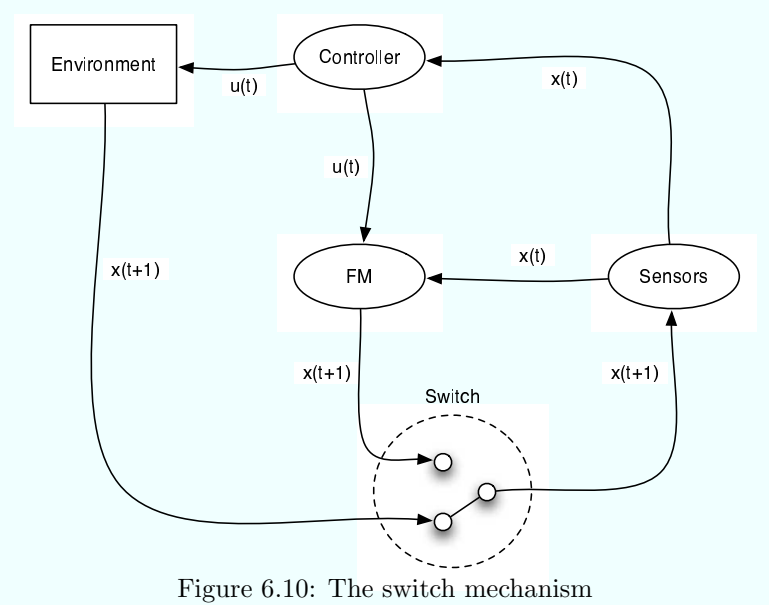


Figure 6.10: The switch mechanism

In order to decide what signal reaches the controller we used the concept of switching proposed in [82](see figure 6.10). This consists of using the signal directly from the sensors whenever that signal is available and using the output of the forward model otherwise. At each timestep, the sensor model has a probability of 0.2 to go from the normal to the blackout state, and will then stay in the blackout state for a number of time steps drawn from a uniform distribution with a maximum of 10 or 20 time steps. While in the blackout state, the information received by the controller depends on whether there is a predictor present or not, and on the experimental setup. In the case without a predictor, the controller input is a vector of all zeroes, or the last real sensor vector before the start of the blackout. When a predictor is present, the controller input is the output of that predictor, which in turn receives input from the action and sensor vector at time t, be it the real information or the output of previous prediction, and tries to predict state $t+1$.

In the delay task, the controller never receives the current sensor data. Instead, when no predictor is present, the controller input is the current sensor vector from a number of n time steps ago, n equals 3 in most of the experiments. When a predictor is present, at each timestep

the controller receives the end result of the predictor being run n times, starting with the real sensor vector and action at time t-n, and then its own predictions for time steps t-n+1 to t.

Experiments

Throughout the experiments in this section, a "general" controller is used, which has been evolved incrementally according to the procedures described in 5.2. When the sensor data is not altered in any way, this controller completes on average 2.84 laps on the chosen track, with a standard deviation of 0.007. This means that while other controllers drive somewhat faster, this is a very robust controller.

We chose to compare predictors created through three different methods: backpropagation, evolving for prediction ability, and evolving for driving ability. For the backpropagation training, a log of sensor data from a number of runs of the above controller with unaltered sensor data is used as training data. When evolving for prediction, the predictor is not affecting the control of the car, and the fitness value is the negative mean prediction error. Evolving for driving ability means measuring fitness as progress on the track, with the predictor being evaluated indirectly.

For all of these methods, we systematically varied the size of the hidden layer in the predictor networks, with all methods being tested for networks of 5, 10, 15, 20 and 25 hidden neurons. We also varied some parameters for the other predictor creation methods: training with backpropagation was done with training sets of 7, 70, 700, 7000 and 70000 data points; evolving for performance on the intermittent task was done with maximum blackout lengths of 10 and 20 time steps; and evolving for performance on the delay task was done with delay lengths of 3 and 6 time steps. We did 50 replications each of most experimental configurations.

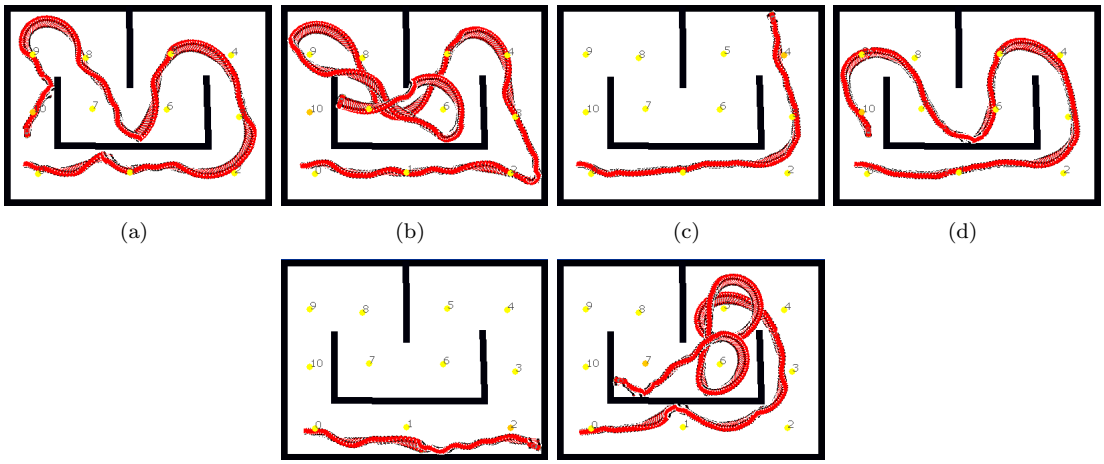
In this section we will not discuss all these results, but only some of their main features, and refer to the paper for the full story.

The Intermittent Task

Training networks to minimise the error based on logged driving the data was not very effective for producing forward models that helped to cope with the intermittent task. This was true regardless of whether the supervised learning was done using backpropagation or using evolution with minimum error as the fitness, and regardless of the amount of training data used. Further, the error on the validation set appeared to have no relevance to the actual performance of the predictor.

Evolving for higher performance, on the other hand, yielded much better performance and much higher prediction errors. Some predictors trained with backpropagation performed reasonably well, but these were not necessarily the ones that had the lowest prediction error.

Figure 6.11 shows some representative examples of the driving performed by the same predictor with predictors learned in different ways.



(a) (b) (c) (d)

(e) (f)

Figure 6.11: Figure showing the trajectories of a range of predictors during the intermittent task with blackout length set to 10: a) The best performing predictor trained with backpropagation, b) The trace of the predictor with the lowest error when trained with backpropagation, c) The trace of the predictor with the highest error when trained with backpropagation, d) The best evolved predictor, e) The best evolved controller with the lowest error, f) Predictor with no prediction

In face of such results, some observations on what behaviour these predictors give rise to could be enlightening - or so we reasoned.

Without a predictor, the car starts accelerating, turning left when sensor inputs are blocked. If the last sensation is kept at the start of a blackout the car, perhaps obviously, keeps doing exactly what it was doing before the blackout. The controller normally makes numerous small turns (a sort of wiggling motion) even on straight segments of the path, examples of this can be seen in figure 6.11, especially *a* and *c*. This often causes the car to end up driving into a wall even during straight sections of the track, when last sensations are retained. We also found that the wall bumping behaviour was popular amongst the majority of controllers trained for fitness; We were surprised to see how little influence the predictors had on the controllers - often doing nothing other than turning left and accelerating when asked to predict sensor data.

When it comes to the evolved predictors, we are not entirely sure why they are so successful. A possible solution for the controller would be to apply brakes and thereby minimising the risk of collisions until the sensors are back online. But this *does not* seem to be what happens. Instead, the controllers take different actions depending on the last non-blocked sensor vector, usually keeping the speed constant.

The Delay Task

A similar, but not identical, pattern can be seen when testing predictors on the delay task. Almost all the predictors trained with supervised learning behave very badly on the delay task as well - in fact, even worse than they do on the intermittent task. As above, the error on training and validation set have no apparent relation to performance (or lack of) when paired with the controller on the delay task. Performance of these predictors, and the baseline (using no predictor at all, feeding the controller outdated sensations), is shown in figure 6.12. Interestingly, the baseline actually outperforms most of these predictors at short delay lengths.

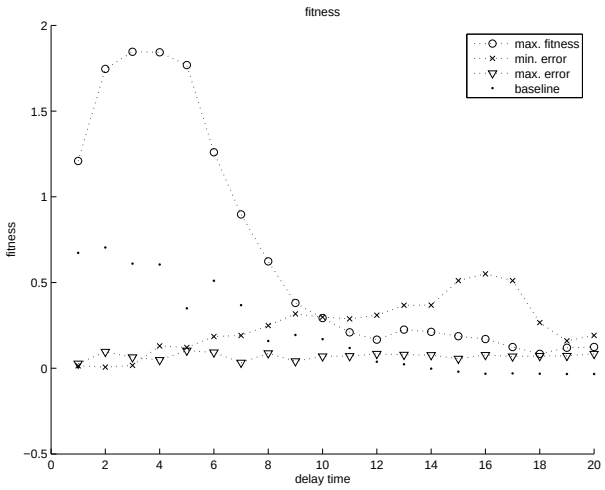


Figure 6.12: Graph shows a comparison between the best-performer (max. fitness), best-predictor (min. error), worst-predictor (max. error) using no prediction and various delay lengths, trained with backpropagation. The baseline is a controller without predictor

Controllers evolved for performance fared much better, but again had much higher prediction errors. Several evolved predictors were possible to help the controller achieve good fitness on the delay task with delay length 3. With longer delays than 5, however, *almost* all the predictors reverted to baseline fitness, little more than zero. The exception was a single predictor which showed the remarkable ability to produce suboptimal but still good fitness levels for exceptionally long delays, at least up to 20 time steps. This can all be seen in figure 6.13.

All tested predictors and non-predictors have displayed variations on two fundamental behaviours. Most of the predictors trained for prediction simply make the controller turn to the left and crash. All other predictors (including no predictor at all) make the controller swerve from side to side, as if the driver was drunk. A plausible interpretation of this behaviour is that the controllers are constantly overcompensating for being too close to one wall or another. The fitness of a predictor seems to be inversely proportional to the magnitude of these oscillations, with the best predictors rarely if ever colliding with the wall. The "remarkable" predictor that could handle long delays also display a variation of this behaviour, but seems especially good

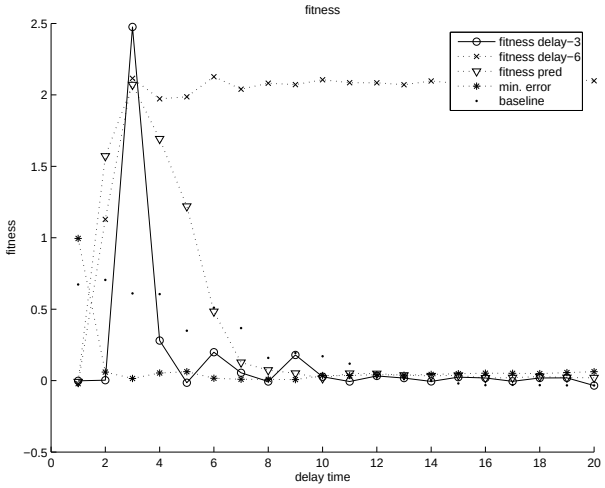


Figure 6.13: Comparison between the *evolved* best-performer, best-predictor and worst-predictor using no prediction and various delay lengths. The baseline is a controller without predictor

at hitting the walls with the back of the car first so as always being able to recover from wall collisions without getting stuck or losing its direction.

Several additional experiments (reported in the paper) test the generality of the predictors over different tracks and different controllers, and rule out that the same effect could be obtained by evolving controllers directly without predictors.

Discussion

The two main findings of these experiments are that predictors evolved for maximizing performance, as part of a controller-predictor system, vastly outperform those that were designed to minimize prediction error. This effect is consistent across different training regimes, data sets, tasks, tracks etc., and hold up over a large number of replications of the experiments. The other main effect is that there is no clear correlation between prediction error and performance as part of a controller-predictor system. Indeed, looking at the trend over all the mentioned controllers, there is a *positive* correlation - higher prediction errors means better performance. Obviously, the predictors that allow the controller to do the task to not do this by predicting the future sensor inputs of the car.

These results could be seen as bolstering an argument to always look at internal simulation of perception in the context of a complete dynamical system, and that having an accurate model might not always be necessary. It could also be interpreted in the more prosaic way that dynamics modelling is very hard if not performed in a suitable transformation space, but that evolution usually finds a way to sidestep the problem you intend for it to solve.

While a large number of experiments were done as part of this suite, many of which cannot be found in this section but only in the paper, some rather simple things were never tried. For

example, we never tried hand-coding rules for how to handle sensor blackouts or delays. It is not impossible that a simple rule of the type "set both steering and drive to neutral in every time step where all sensors output 0" would have performed better than our trained predictors, but we don't know, as we never implemented this rule. On the other hand, finding the best way of dealing with sensor outages were never the main objective of the study; we were primarily looking for greater insights into the workings of and differences between some different learning methods.

6.5 Summary

In this chapter, three series of experiments dealing with imitating various aspects of dynamics and behaviour in different car racing environments were described. The first series of experiments concerned the imitation of driving styles in the development of player models for later use in track evolution experiments in section 7.3; the next section described the acquisition of dynamics models of physical miniature R/C cars and the subsequent evolution of controllers based on those models, and the third section described several attempts at simultaneously modelling the dynamics of a simulated racing car and the layout of a track.

Though all the described experiments were done in the domain of car racing, and draw on the foundational car racing experiments in sections 5.1 and 5.2, there are considerable differences in what sort of learning was attempted and in what environment. This makes it hard to distill the results of the various studies into some sort of general wisdom about imitation or modelling, in car racing or otherwise.

One problem that recurred throughout the different experiments, however, was how to judge the quality of an acquired model. Some of the player models arrived at in the player modelling experiments (the ones obtained through direct modelling) were obviously wrong, in that the controllers didn't manage to drive a single lap on the test track. Others (obtained through indirect modelling) were at least decently good, but how faithful they really were to the modelled human player is hard to ascertain without doing extensive studies using the qualitative judgement of several humans. After any quantitative measure is satisfied, the question remains how well this measure really reflects the sort of similarity we are seeking.

Similarly, in the physical car dynamics modelling experiments, models could be acquired through satisfying some quantitative criterion (error minimisation), and controllers could be evolved for good performance on that particular model, but when tested on the real car the same controller could fail miserably. Here, we have an indirect test of imitation quality through the performance of derived controllers on the real car, but still only an indirect test, and also

quite time-consuming. Multi-model evolution was found to be a good way to achieve robust controllers, letting the shortcomings of the different models cancel out each other.

The problems with judging imitation quality are maybe most vividly illustrated by the experiments in the last section, where two different quantitative criteria (fitness maximisation and error minimisation) give rise to dramatically different models. It is hard to tell whether those experiments should be considered successful in the normal sense of the word - after all, the best evolved predictors did not do any discernible predicting - but they do offer interesting insights into the peculiar challenges of behaviour and dynamics imitation.

Chapter 7

Innovation

In this chapter we discuss a number of experiments where evolution is used, but not to achieve a well-defined goal, but rather to innovate. In the first two sections we use competitive co-evolution to evolve driving behaviour. Here, the fitness function is partly dependent on the other driver on the same track, and so it is not clear what sort of driving yields the highest fitness (in contrast to single-player races, which are about driving fast along the best racing line). In the third section, we evolve not controllers, but racing tracks. The fitness function is here based on a player model and a metric of what sort of tracks should be fun, yielding rather innovative tracks.

7.1 Co-evolving car controllers for competitive driving

This section is based on our first paper on competitive co-evolution for simulated car racing, which was presented at PPSN 2006 [144]. In these experiments, the full track-based racing game, as presented in section 4.1, is used. The difference is that the cars have eight rangefinder sensors instead of six, and that a separate array of eight Boolean values decide whether each sensor is a standard wall sensor or a *car sensor*. The car sensors work exactly like the wall sensor, except that they return a value dependent on whether the other car, rather than a wall, is within the sensors' scope and how far away it is.

Simulated competitive car racing allows some uncommon forms of competitive co-evolution to be explored. Most competitive co-evolution experiments use two populations, where one population contains prey and the other predators, or one population parasites and the other hosts, or some similar asymmetric configuration. In other words, most competitive co-evolution is *interspecific*; here, all the individuals are of the same "species" in that they are evaluted in the same way. Co-evolution can therefore be single-population and *intraspecific*.

A second way in which simulated car racing allows uncommon modes of co-evolution is through the existence of a well defined solo fitness function: any controller can be tested both for absolute solo fitness, which means the distance covered when racing without competition, absolute competitive fitness, which is the same thing when having to take the behaviour of another car into account (including the possibility of collisions), and relative fitness, which is defined as how far in front of or behind the competitor a controlled car finishes. Further, absolute competitive fitness and relative fitness can be blended seamlessly. We believe that these characteristics make our car racing games ideal for exploring co-evolution.

The first set of questions we try to answer in this section concern the extension of the car racing model and evolutionary approach to two cars: how well will controllers evolved for solo racing do with competition? Will it be possible to co-evolve controllers that do better? Is our controller architecture and sensor setup appropriate for this? Will we be able to evolve human-competitive drivers, and if not, what are the problems with our method?

The second set of questions address co-evolution. Will there be a difference in fitness, and in behaviour, if we evolve for absolute, relative or mixed absolute and relative fitness? What sort of difference will be observed? For example, will controllers evolved for relative fitness turn out to drive more aggressively? Will there be a difference in sensor setups?

7.1.1 Co-Evolutionary Algorithm

For the co-evolutionary algorithm, a modified $(\mu + \lambda)$ evolutionary strategy with $\mu = 50$ and $\lambda = 50$ was used. The difference between the co-evolutionary algorithm used here and a standard evolutionary strategy is in the fitness calculation. There are two types of primitive fitness defined: absolute and relative fitness. The absolute fitness of a controller C is calculated in the "standard" way, as defined in section 4.1 and used in several other experiments in this thesis. Relative fitness is defined as the difference in absolute fitness between C and the car it is competing against. Both the absolute and relative fitness values for a given controller was calculated as the mean of three trials of the controller on each of the tracks.

When the primitive fitnesses of all the controllers have been calculated, they are normalized, so that they are all in the range [-1..1]. The final fitness value of each controller is then calculated by blending the two primitive fitness values: $fitness = p * absfit + (1 - p) * relfit$ where p is the proportion of absolute fitness, a constant set at the beginning of the evolutionary run. It could be argued that only evolution with completely relative fitness constitutes co-evolution.

There are three mutation operators: Gaussian mutation of all neural connection weights, Gaussian mutation of all sensor parameters (angles and lengths), or sensor type mutation. Each time the mutation method of a controller is called, numbers drawn from a Gaussian

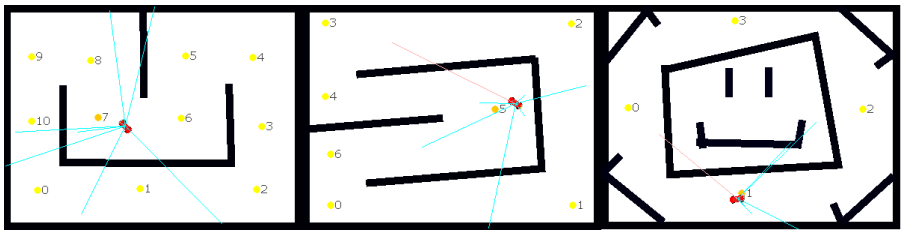


Figure 7.1: The three tracks used in the experiments, including waypoints. Each track also shows a sample car with evolved sensors.

distribution with a standard deviation of 0.1 are added to both neural connection weights and sensor parameters. With a probability of 0.4, a sensor type mutation is also performed, meaning that one of the sensors has its type changed from car to wall or wall to car.

At the start of an evolutionary run, all controllers have four wall sensors and four car sensors, pointing in random directions and with random ranges, and the neural connection weights are initialized to small random values.

Three different tracks, namely the counter-clockwise versions of the six easier tracks of the eight tracks used in section 5.2, were selected for the present experiments; these tracks are shown in figure 7.1. Each pair of controllers were tested on all three of these tracks.

7.1.2 Experiments

Giving solo-evolved controllers some competition

10 separate solo-evolutionary runs were made according to the setup described in the Methods section above. Each evolutionary run lasted for 200 generations. (The mean fitness was zero at generation 0 of every evolutionary or co-evolutionary run in this paper; fitness growth graphs have been omitted to conserve space.)

On average, the best individual of the last generation of each of the evolutionary runs had fitness 2.49 (with standard deviation $\sigma = 0.23$), and used 5.7 ($\sigma = 0.67$) wall sensors and 2.3 ($\sigma = 0.67$) car sensors. The best run resulted in a best controller with fitness 2.67, and the best controller of the worst run had fitness 1.89. Most of the evolved sensor setups consisted in a relatively even spread of medium-range wall sensors pointing forward and diagonally forward, and the few car sensors pointing backward.

One of these controllers, with fitness 2.61 (0.13), was selected for further testing. When put in a competition with another car controlled by a copy of the same controller, average fitness dropped to 1.23 (0.6). Behavioural analysis shows that the two cars collide repeatedly at the beginning of almost every trial, as they don't have any method of detecting and reacting to each other's presence. Depending on starting conditions, the outcome of the competitions vary, but usually one or both of the cars is either driven to collide with the wall, or spun around so that

Prop abs	0.0	0.25	0.5	0.75	1.0
Abs/solo	1.99 (0.31)	2.09 (0.33)	2.11 (0.35)	2.32 (0.23)	2.23 (0.23)
Abs/duo	0.99 (0.53)	0.95 (0.44)	1.56 (0.45)	1.44 (0.44)	1.59 (0.45)
Rel/duo	0 (0.75)	0 (0.57)	0 (0.53)	0 (0.55)	0 (0.47)
Wall/car	5.8 / 2.2	5.6 / 2.4	5.2 / 2.8	4.2 / 3.8	6.4 / 1.6

Table 7.1: The results of co-evolving controllers with various proportions of absolute fitness. All numbers are the mean of testing the best controller of ten evolutionary runs for 50 trials. Standard deviations in parentheses.

it starts driving the track the wrong way. A car that starts going the wrong way is usually, but not always, unable to turn around and start driving in the correct direction again; a car that crashes into the wall usually gets stuck. This is because of the controller design rather the game mechanics, as it is perfectly possible for a human player to back away from the wall and continue driving in the right direction. In many trials, however, one of the cars managed to escape the collisions in the right direction and proceeded to make its way smoothly around the track.

From this experiment, it can be seen that the problem of racing two cars concurrently is sufficiently different from the problem of solo-racing that the performance of a solo-evolved controller is catastrophically compromised when tested in competition conditions.

Co-evolving controllers: The absolute-relative fitness continuum

50 evolutionary runs were made, each consisting of 200 generations. They were divided into five groups, depending on the absolute/relative fitness mix used by the selection operator of the co-evolutionary algorithm: ten evolutionary runs were performed with absolute fitness proportions 0.0, 0.25, 0.5, 0.75 and 1.0 respectively. These were then tested in the following manner: the best individuals from the last generation of each run were first tested for 50 trials on all three tracks without competitors, and the results averaged for each group. Then, all five controllers in each group were tested for 50 trials each in competition against each controller of the group. Finally, the number of wall and car sensors were averaged in each group. See table 7.1 for results.

Analysis

It is clear that, when driving without competitors, the co-evolved controllers on average have lower absolute fitness than the solo-evolved controllers. Behavioural inspection suggests that the co-evolved controllers drive more carefully, seldom accelerating to top speeds, and take corners more conservatively. A similar but smaller difference in absolute solo-fitness seems to exist between the groups of co-evolved controllers, with controllers evolved more for absolute fitness performing better than controllers evolved more for relative fitness. The controllers within a

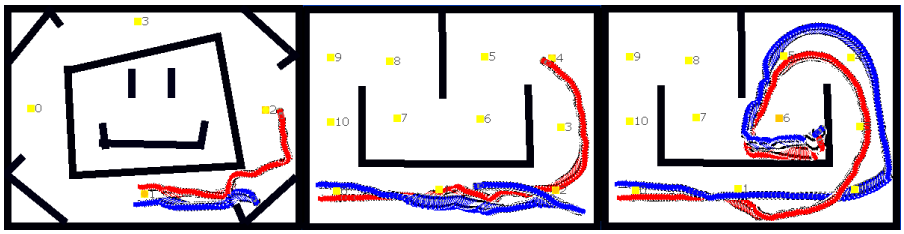


Figure 7.2: Traces of the first 100 or so time-steps of three runs that included early collisions. From left to right: red car pushes blue car to collide with a wall; red car fools blue car to turn around and drive the track backwards; red and blue car collide several times along the course of half a lap, until they force each other into a corner and both get stuck. Note that some trials see both cars completing 700 time-steps driving in the right direction without getting stuck.

group perform similarly, and the lower fitness comes from driving slower around the track rather than crashing into walls or losing direction.

The difference between controllers co-evolved with different fitness mixes becomes clearer when we measure performance in competition with other controllers from the same group, where controllers evolved mostly for absolute fitness generally get about half a lap farther than those evolved mostly for relative fitness. Behavioural analysis confirms that this is because the cars more often collide at the start of a trial, often forcing one or both of the cars to crash against the wall or spin around and lose track of which direction it is going. Often, the controllers evolved with low (0 or 0.25) proportions of absolute fitness actively look for trouble by trying to collide. (See figure 7.2).

There seems to be little consistency in evolved sensor setups, samples of which can be seen in figure 7.1 (wall sensors are blue; car sensors are pink; each car is travelling forwards in direction of the waypoints). We found one controller in the group evolved purely for relative fitness that had only wall sensors and no car sensors, and another one in the group evolved for purely absolute fitness! There is no obvious tendency towards fewer or more car sensors at either end of the fitness mix, and the data is too scarce to prove any more subtle tendency. When looking at all 50 controllers together, every controller has at least three wall sensors, and there is always at least one pointing mostly forward. On average, the cars have twice as many wall sensors as car sensors, and when car sensors are present, there seems to be at least one pointing mostly backward; overall, more car sensors point backward than forward.

In order to find out how the controllers evolved with various fitness mixtures performed against each other, we tested all the five controller groups against each other. The slightly surprising results was that the groups performed on average equally well against each other, though with considerable intra-group variation. The absolute fitnesses of the controllers in these encounters were quite low, on average 0.96, which suggest that all controllers are quite ill prepared to race against controllers from another fitness mix group.

Prop abs	0.0	0.25	0.5	0.75	1.0
Co-evo	1.41 (0.52)	0.99 (0.44)	1.32 (0.58)	1.23 (0.65)	1.41 (0.45)
Solo-evo	1.68 (0.56)	1.95 (0.62)	1.74 (0.57)	1.38 (0.65)	1.51 (0.63)

Table 7.2: Co-evolved versus solo-evolved controllers.

Co-evolved versus solo-evolved controllers

The 50 controllers co-evolved with various fitness mixes in the section above were now tested against the 10 solo-evolved controllers from section 7.1.2. For each group, the ten co-evolved controllers competed for 10 trials with each of the 10 solo-evolved controllers. See table 7.2 for results.

Note that there is a mostly small but consistent fitness advantage for the solo-evolved controllers over the co-evolved ones. (Both co-evolved and solo-evolved controllers performed significantly worse in these competitions than when tested in solo racing conditions.) The cause of this fitness difference is not completely obvious after looking at a large number of these competitions, but it appears that the solo-evolved controllers (which gain higher fitness than the co-evolved ones in solo trials) simply outrun the co-evolved controllers in many cases, and so avoid many of the collisions, and further corroborate the hypothesis that the controllers tend to be very specialized to compete against controllers similar to themselves. This could be seen either as a shortcoming of the evolutionary algorithm, or as the desired state of things; it could be argued that the co-evolved controllers should have strategies general enough to take on any opponent, or that a their more careful driving style should always make them slower than a solo-evolved controller.

Evolution with a static target

To investigate whether the tendency to specialization in co-evolved controllers could be used to create controllers that could out-compete the solo-evolved controllers from section 7.1.2, we modified a copy of the co-evolutionary algorithm to work with a static target. In this configuration, each controller is evaluated by racing three races against randomly selected controllers out of the ten solo-evolved controllers. It should be noted that this is not co-evolution at all, as the target controllers do not evolve. The car controlled by the target controller could instead be seen as an interactive feature of the environment.

The experiments we run with this configuration failed to generate any controllers with better fitness than the target controller. This was despite attempts at evolving from scratch, starting from a general controller, or starting from a clone of the target controller, and using various mixtures of absolute and relative fitness. Our interpretation of this is that the solo-evolved controller drives the tracks as fast as can be done given its sensing and processing limitations,

and that the same limitations hinder the co-evolved controllers from doing any better.

Human-competitiveness of evolved controllers

A random selection of controllers were tested by competing with a car controlled by one of the authors via the keyboard. It was found that the solo-evolved controllers were generally good contenders, driving at about the same skill level or slightly better than the author, as long as collisions were avoided. However, it was found to be quite easy to learn how to collide with the computer controlled car in such a way that it got stuck on a wall, and then continue to win the race. Most of the co-evolved controllers were pretty simple to beat by just accelerating hard at the beginning of a race and keep driving, as their slower driving wouldn't allow them to catch up.

7.1.3 Discussion

Our main positive finding concerns the effects of changing the type of fitness function. A very clear effect was that controllers evolved more for relative fitness acted more aggressively, but covered less distance both when running solo and when competing with other controllers from the same population, than controllers evolved more for absolute fitness. We could not find any systematic difference between the sensor setups evolved with the various fitness mixtures, but observed a general tendency to point car sensors backwards rather than forward, and the opposite tendency for wall sensors - it seems to be more important to watch your back than to know what your competitors are doing in front of you.

A finding that is relevant to the overarching quest to scale up evolutionary artificial intelligence using computer games is that competitive car racing is a much more complex problem than solo car racing. This can be seen both from the drastic degradation of fitness when solo-evolved controllers are put in competitive environments, and from our great difficulty in evolving controllers that can reliably outperform the solo-evolved ones. It can also be seen from the total inability of all evolved controllers to backtrack upon a frontal collision with a wall, and the relatively poor ability of most evolved controllers to find the correct direction after having been spun around. This points to the need for more complex sensors and neural networks.

However we set up the evolutionary runs, they seem to suffer from over-specialization, where the controllers in a population only learn to race each other. This result is in broad agreement with what has been found in co-evolutionary predator-prey experiments, as discussed in section 2.4.6. So even though the current experimental environments allows us to explore a larger space of variants of competitive co-evolution, it seems that at least the currently explored form of co-evolution suffers from the same basic obstacles to evolving generally good competitive

behaviour as previously attempted approaches.

7.2 Multi-population competitive co-evolution

A year later we revisited the problem of competitive co-evolution, armed with a number of new ideas, and using the point-to-point car racing game rather than the track-based racing game. In this section, based on a paper presented at CEC 2007 [138], we investigate the use of *multi-population* competitive co-evolution, where more than two populations are used. We believe that we are the first to use more than two populations for single-species co-evolution, and we are rather certain that we are the first to use it for comparing controller architectures.

Why would multi-population co-evolution be better than one- or two-population versions of the algorithm? To answer this question, we can go back to Janzen, who distinguished between *true* and or *diffuse* co-evolution [67] in the context of evolutionary biology (not computer science). The former is defined as evolutionary change in a specific trait of one population in response to the evolutionary change of one specific trait possessed by another population. In contrast, the latter is defined as *non-specific* evolutionary change in response to a *group* of traits possessed by another population, or group of populations.

Bullock argued that diffuse rather than true co-evolution would be desirable from an engineering standpoint, as diffuse co-evolution should lead to more robust solutions [19]. One way of achieving diffuse co-evolution would be to use many populations, and test individuals against other individuals in more than one other population. Following his suggestion, Hornby and Mirtich did exactly this in a predator-prey simulation [62].

Our own take on this is to use multiple populations to try to force diffuse symmetric co-evolution, but also to use it to compare controller architectures. (In Hornby and Mirtich's work, controller architectures did not vary between species.) In the process, we compare two different selection strategies: steady-state and generational selection.

Generational and steady-state selection

Standard co-evolution proceeds in generations. In each generation there is a period of evaluation, followed by population decimation and replacement.

Though this generational scheme is the standard approach to co-evolution in use today, it is not the only way that co-evolution can proceed. In nature, the process of replacement is usually less dramatic - populations usually remain stable and there is continuous replacement of individuals.

Miconi and Channon [88] introduce one method of performing steady-state co-evolution. The

N-strikes-out algorithm they propose performs both evaluation and replacement asynchronously, on an individual basis. To our knowledge, this is the first steady-state co-evolutionary algorithm.

This asynchronous updating means that the selection landscape changes gradually, in effect acting as a self maintaining archive of previous fit individuals, and avoiding the need for a hall of fame. The motivation is that this will discourage exploits of the current champion's weaknesses, as there is more likely to be other high fitness individuals which don't share that specific weakness.

Another advantage of the steady-state approach is that is not necessary to manage the generational changes in the population. The algorithm merely needs to keep track of the number of defeats for each individual, which is simpler than the generational approach.

Research questions

The main question we address in this section is what is it possible to do with multi-population co-evolution. Can we evolve controllers that perform better than those evolved with solo-evolution, or one- or two-population co-evolution? Can we use multi-population co-evolution to investigate the relative benefits of different controller architectures? When seeding a number of populations with the same architecture, can we evolve a behaviourally diverse set of controllers?

We are also interested in the relative performance of the steady-state and generational multi-population co-evolutionary algorithms. Particularly, we wonder whether the N-strikes selection mechanism manages to further alleviate the cycling problems.

And in addition to the issue of how to best compare a number of controller architectures, we are of course interested in the results of the comparison: which of the implemented controller architectures is best for the task given? For us, the underlying motivation is to be able to evolve complex general intelligence; studying the properties of particular controller architectures and evolutionary algorithms is a means to that end, rather than the other way around.

7.2.1 Controller architectures

A number of evolvable controller architectures were implemented, for purposes of comparing speed and quality of learning. All controller architectures are based on one or two evolvable function approximators (in all cases except one these are neural networks), and in an all cases the main function approximator outputs two real numbers. These two numbers are interpreted as follows: if the first output is above 0.3, the driving force is set to forward, if below -0.3 the driving is to backward, and otherwise driving set to neutral. The second output decides whether to set the steering for current timestep to left, right or centre in the same way.

MLP controllers

The MLP controller is based on a standard multi-layer perceptron with 8 inputs, 6 hidden neurons, 2 outputs and *tanh* activation function. The inputs to the network are the speed of the car, the angle to the current way point, the distance to the current way point, the angle to the next way point, the distance to the next way point, the angle to the other vehicle, and the distance to the other vehicle (both the last values are set to 0 if there is no other vehicle present). Apart from these inputs, a bias input (always set to 1) is added to the all neural networks described in this paper.

All angles are calculated as the difference between the orientation of the car and a straight line to the waypoint or competitor car in question. We have found that the input representation described above tends to produce controllers that drive backwards, and further that adding PI radians to all angles instead produce controllers that tend to drive forwards. However, the controllers evolved with angles thus reversed actually score somewhat lower on average, which is why we have decided to keep all angles as is in the experiments in this paper.

At the start of an evolutionary run, all connection weights of the neural networks are set to zero. Mutation consists of adding random numbers drawn from a Gaussian distribution with standard deviation 0.1 in all neural networks described here.

Recurrent controllers

Most of the controllers are based on simple recurrent neural networks, commonly known as Elman networks [41]. The recurrent neural networks are implemented as standard MLPs, with extra connections from the hidden layer of the last time step to the hidden layer of the current time step.

Several controller architectures based on such networks are compared. The RMLPSmall, RMLP and RMLPBig controllers are all based on recurrent networks with exactly the same inputs and outputs as the MLP controllers described above, but differ in the size of its hidden layer, being 4, 8 and 16 units respectively. Two additional controller architectures were also based on recurrent networks but with impoverished inputs: the RMLP1WPOnly has only 6 inputs to its network, as angles and distances are given only to the current way point and the competitor's car, and the SimpleRMLP does not even input angle and distance to the competitor car to its network, which only has 4 inputs.

Modular controllers

The modular controllers represent an incorporation of domain knowledge into the controller architecture. The design is based on the observation that the most important task for a good

controller, besides driving to a particular way point as quickly and reliably as possible, is to choose which way point to go for: the current or the next? Assuming that these two tasks are reasonably separable, the modular controllers are based on one MLP, that decides which way point to go for, and a SimpleRMLP controller that controls the car. The MLP receives three inputs: a bias, the distance to the current way point divided by the distance to the other car, and the speed of the car divided by the speed of the other car. If the only output of the MLP is above 0, the angle and distance to the current way point is fed to the SimpleRMLP, otherwise those of the next way point are fed.

In previous experiments with evolving neural networks in layered control architectures, we have found that it is sometimes helpful to evolve the lower layers before the higher layers [137]. Therefore, two versions of the modular architecture are tested: the ModularRMLP is initialised with all connection weights in both networks set to empty. The PrimedModular controllers, on the other hand, are initialised with a "blank" MLP but a SimpleRMLP that has already been solo-evolved to good fitness as an independent controller.

GP-based controllers

Finally, one controller architecture was based on genetic programming. Each controller consists of two function trees (evaluating to the two outputs, which are then interpreted as driving and steering) and three automatically defined functions (ADFs). The function trees are initialised randomly, and mutated with single-point macro mutation, where a randomly selected node in each tree is replaced with a randomly generated node. The trees are limited to a depth of 5 in order to make the computational expense of these controllers on par with the neural network-based controllers. When it comes to the node types, the set of terminals consists of sensor inputs (any of the eight inputs given to the MLP and recurrent controllers), constants (randomly initialised to values between 0 and 2) and ADF calls; the set of non-terminals consists of arithmetic functions (plus, minus, multiplication and protected division), trigonometric functions (sin, cos, tan and tanh) and an if-then-else function (if the first child evaluates to more than 0, return the value of the second child, otherwise the third child). To avoid loops, the ADF calls are restricted so that an ADF can only call an ADF with a higher id than itself.

7.2.2 Co-evolutionary algorithms

In this paper we compare two different co-evolutionary algorithms: generational and steady-state co-evolution.

Generational

We used a standard evolution strategy. For single population co-evolution, the steps are:

1. Evaluate each individual against another chosen at random from the best performing half of the population.
2. Pick the fittest half of the population to keep, and replace the other half with mutated versions of the fittest half.
3. Start another generation.

For multi-population co-evolution, a similar procedure is followed, only now each individual is evaluated against one of the fitter individuals from each population.

Steady-state

We use a modified version of the N-strikes-out algorithm, as detailed by Miconi and Channon. The one-population version consists of the following steps:

1. Pick two individuals A and B from the population at random.
2. Pit them against each other; determine the winner and the loser of the confrontation (if any).
3. If the loser has been defeated N times over its entire history, delete it and replace with a new individual.
4. Start another comparison.

In the two population case, Miconi and Channon found that a naive approach of comparing one random individual from each population caused disengagement, resulting in the weaker population losing any selection gradient.

They overcame this by comparing two individuals from population A against an individual from population B. The winner and loser were determined by which of the population A individuals had scored best against the individual from population B. This process was then reversed, and two individuals from B were evaluated against an individual from A.

The members of a population were therefore only competing against each other, rather than other populations. The individual from the other population was simply used to evaluate the fitness of the two individuals.

We follow a similar approach in this paper, generalised to n populations. The advantage of this evaluation scheme is that it should discourage the scenario where one member of the

population discovers an exploit which allows it to beat other members of the population, but lowers its general fitness.

One concern with the algorithm is that with a noisy fitness function, high fitness individuals can still be beaten occasionally by lower fitness individuals. Because individuals are deleted after a certain small number of defeats, this would mean losing desirable individuals. As suggested in [88], one way around this would be to introduce the concept of 'forgetting' old defeats.

The approach we took was to have a *defeat factor*, which we multiplied the number of losses by at each comparison. If less than 1, this should cause the number of losses to decay towards zero, being topped up by new losses. It means old defeats would be considered less important.

Another concern we had was that selection had no direct dependence on absolute fitness (an individual's score on the track). In theory, a controller could win many comparisons by blocking the other controller so it achieved a low score. This would mean individuals that scored lower could still win lots of contests, and spread through the population.

This should be more of a concern with the single population N-strikes, but it could still be an issue in the n population case.

To combat this, we decided to make the defeat factor dependent on absolute fitness. We want individuals with a high apparent fitness to be given more evaluations before deletion, and low fitness individuals to quickly be replaced.

We therefore made the defeat factor $\mathcal{F}$ for an individual have the form:

$$\mathcal{F} = \frac{1}{e^{x_i - \bar{x}}} \tag{7.1}$$

Where x_i is the fitness of the individual, and $\bar{x}$ is the mean fitness of its population.

This means the defeat factor will be large in low-fitness individuals, and small for high-fitness individuals, implying more rapid forgetting of old defeats.

In order to adjust this parameter in a convenient manner, we introduced a defeat factor multiplier $\mathcal{D}$, as shown below:

$$\mathcal{F} = \frac{1}{e^{\mathcal{D}(x_i - \bar{x})}} \tag{7.2}$$

When $\mathcal{D}$ is 0, we get plain N-strikes behaviour. When $\mathcal{D}$ is greater than zero, we get an increasing contribution of absolute fitness.

One advantage of this form is that it retains the predictability of the number of deletions. In plain N-strikes, the number of deletions should be approximately the number of comparisons divided by N. We found that this behaviour was preserved by using this form of the defeat factor.

Parameter tuning

One of the features of the N-strikes out algorithm is its flexibility - there are many parameters that can be varied. In initial parameter tuning we tried varying the following parameters for the single-population case:

- population size P
- number of strikes resulting in deletion N
- the number of games during an evaluation
- the value of the defeat factor multiplier $\mathcal{D}$

Preliminary results suggested that a large population meant fitness initially rose slower but reached a higher value than in a small population.

We also found small values of N performed better than larger values.

The only effect of changing the number of games during an evaluation should be to alter the level of noise in the fitness function. We found that in high noise environments (such as using only one game to compare two individuals), setting $\mathcal{D} = 0.5$ caused a quicker rise in fitness than the plain N-strikes. However, both resulted in about the same fitness eventually.

For subsequent experiments, we chose the values of:

- $P = 30$
- $N = 2$
- at least 5 games per comparison
- $\mathcal{D} = 0.5$

The other decision to make was how to perform replacement. We used only mutation and not crossover in this paper, so the most straightforward choices were replacement by a mutated version of either the winner or loser of the comparison. We chose replacement by a mutation of the winner, as this should aid the rapid spread of high fitness through the population.

The generational algorithm had less parameters to adjust. We decided on a population size of 30, and at least 5 games per comparison, to allow direct comparison with the N-strikes-out algorithm.

7.2.3 Results

Our experiments proceeded as follows: first, to establish a baseline for the experiments with the many-population co-evolutionary algorithms, we tested each of the controller architectures

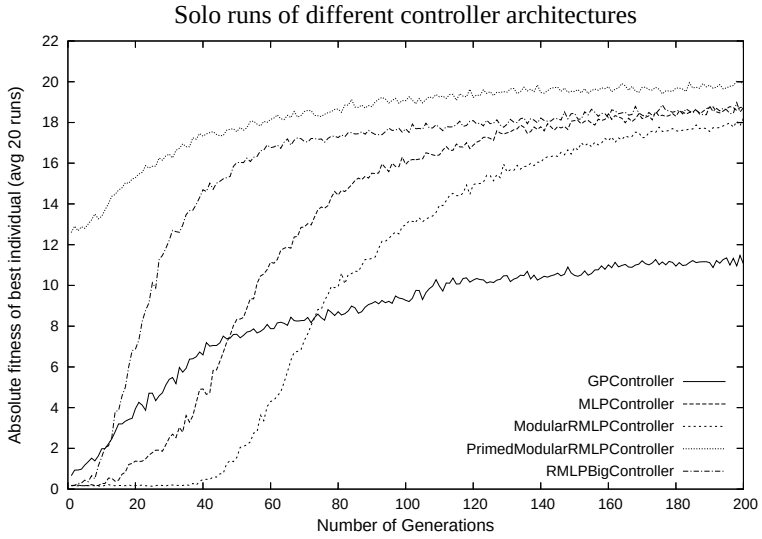


Figure 7.3: Solo evolution of diverse architectures, 20 runs.

independently. This was done using both solo evolution and single-population co-evolution. We then tried using both the generational and steady-state algorithms to compare controller architectures. The controller architecture that was found to perform best was then used to seed all populations of both multi-population evolutionary algorithms. The idea here was to investigate diversification between populations, and the extent to which multi-population competition helps evolve complex general behaviour.

Solo evolution of controller architectures

Our first set of experiments concerned the evolution of controllers for the single-car version of the task. All nine controller architectures described above were evolved in single-architecture populations using a standard 15+15 evolution strategy for 200 populations. These experiments were all repeated for 20 runs. In figure 7.3 we have plotted the fitness growth of the five different controller architectures, including one based on RMLPs (Elman-style recurrent networks). Several things can be learned from this figure: one is that the GP controllers consistently reach a much lower final fitness than the other architectures, even though they learn quite fast in the first few generations. The PrimedModular controllers perform slightly better than the others in the end, but this is probably because they have effectively evolved for longer, the lower layer being pre-evolved. Other than that, RMLPBig (large recurrent network) learns fastest of the controllers.

In figure 7.4 we compare the five different controller architectures based on monolithic (non-modular) RMLPs. What is remarkable here is how similar their ultimate performance is. The only difference of any note is in their learning speed, and here we see a clear relation to the size of the network: the larger the network is (more inputs and larger hidden layer) the faster

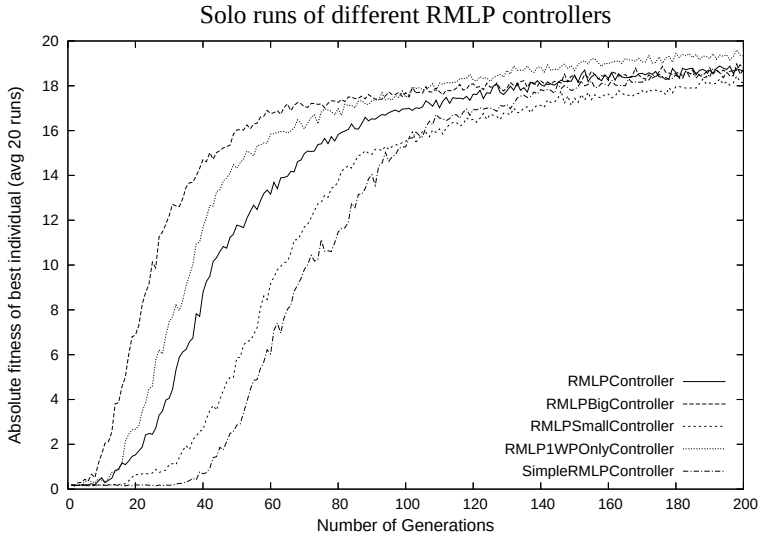


Figure 7.4: Solo evolution of different rmlp-based architectures, 20 runs.

it learns.

Single-population co-evolution

Next, we ran a number of experiments where we co-evolved controllers for the two-car version of the task, using single-population co-evolution with populations of 30 individuals containing only one controller type each. So in these runs, the controllers only ever compete against other controllers of the same architecture. We did 20 runs for each controller architecture, using both the generational and the modified N-strikes-out steady-state co-evolutionary algorithms.

Generational: In figure 7.5 we plot the fitness of the best controller of each generation for the same five controller architectures as in figure 7.3. As in the solo evolution, the GP controllers start out as fast learners but are by far the worst of the lot at the end of 200 generations. Unlike in the solo runs, we see a very clear superiority of the modular controllers over the other architectures. At the end of the runs, the primed modular controllers do better than the non-primed, but the fitness for the non-primed ones is still increasing.

Looking at the RMLP-based controllers (figure 7.6) we see no difference in ultimate fitness and little difference in learning speed. Again, the larger networks learn somewhat faster.

Steady-state: The results of the steady-state runs were very similar, as can be seen from figure 7.7 (we have omitted the graph for rmlp-only comparisons out of space considerations). The primed modular controller still performs almost twice as well as the GP controller. One difference is that the non-primed modular controller learns much faster with steady-state than with generational co-evolution.

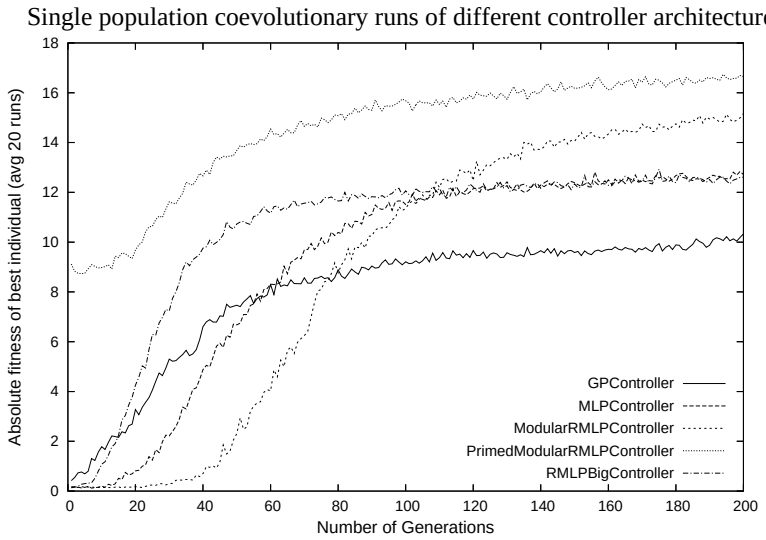


Figure 7.5: Single-population generational co-evolution of diverse architectures, 20 runs.

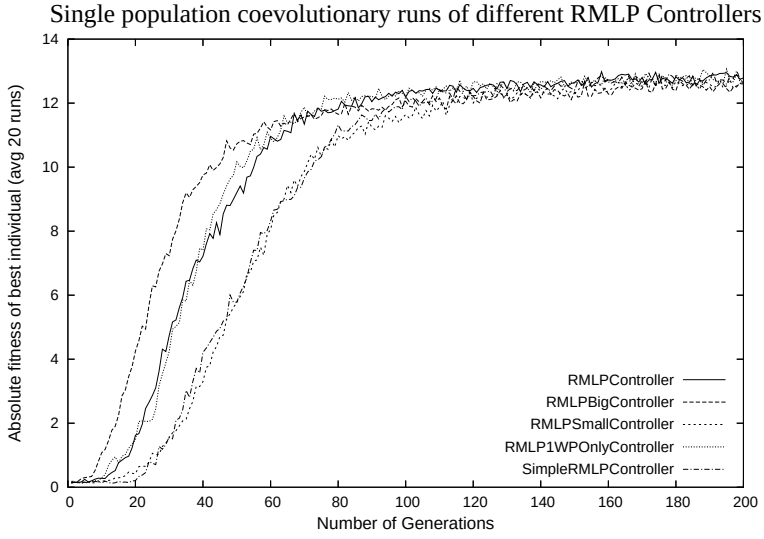


Figure 7.6: Single-population generational co-evolution of different rmlp-based architectures, 20 runs.

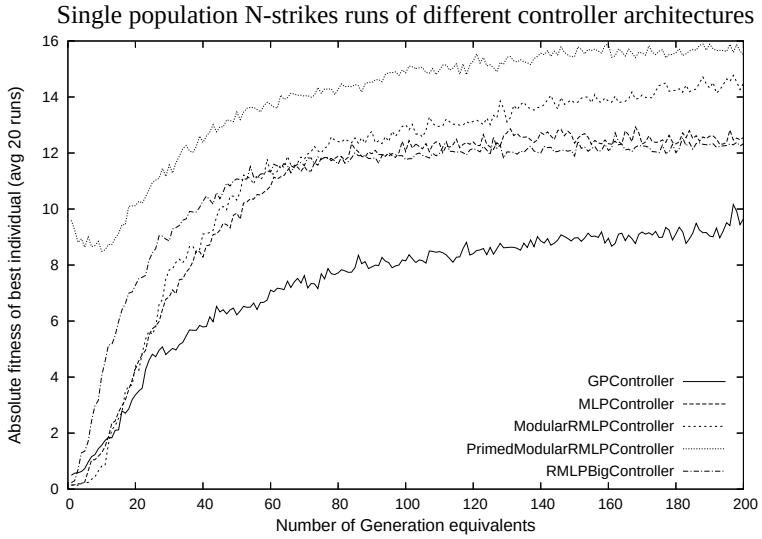


Figure 7.7: Single-population steady-state co-evolution of diverse architectures, 20 runs.

Multi-architecture multi-population co-evolution

And so, at last, we come to the multi-population co-evolution. We used nine populations for these experiments, one population for each controller architecture. In these runs, which lasted for 500 generations (or steady-state equivalents) each individual of each population is tested against individuals of all other populations, thus controllers of all other architectures. At least 20 runs were made for both the generational and steady-state algorithms.

Generational: Figure 7.8 plots the fitness growth of the populations populated by the main different controller architectures (remember that all nine populations were part of the runs, though only five are plotted). Some of what we see here could be expected, given our single-population results. The modular controllers still win (in this case literally) over the other controllers. But what is unexpected is that the GP-based controllers no longer do worst, indeed they perform almost as well as the MLP-based controllers at the very end of the 500 generations. Instead, the RMLPBigControllers perform worst by a significant margin.

This rather surprising result concerning the RMLPBig controller architecture is put into context when looking at figure 7.9, which compares only the RMLP-based controllers. Here we see that the smaller the RMLP-based controller is, the better ultimate fitness it reaches, with the minimalist SimpleRMLPControllers coming out on top.

Steady-state: The steady-state runs paint a similar picture. The main difference between figure 7.10, which shows the fitness growth of the main controller architectures, and figure 7.8 is a slower overall fitness growth, and that the GP controllers as a result don't do any better than the RMLPBig controllers, i.e. not very good at all.

The graph for the RMLP-based controllers has been omitted in order to conserve space, but

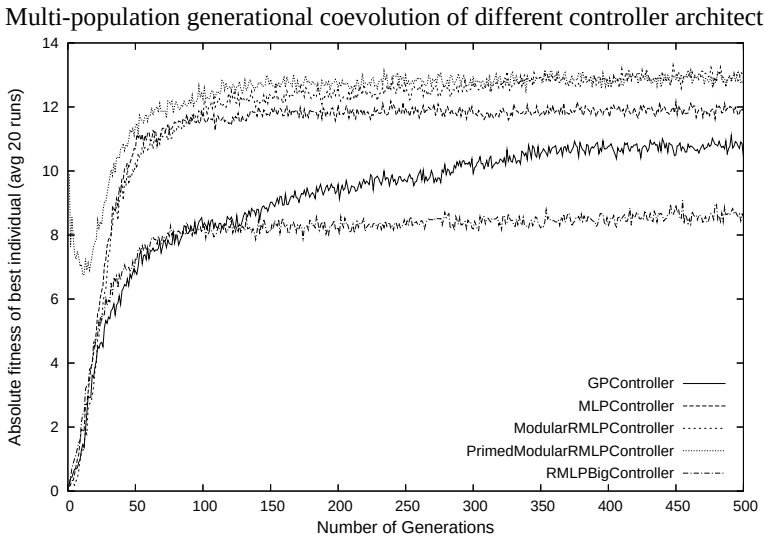


Figure 7.8: Nine-population generational co-evolution of diverse architectures, 10 runs.

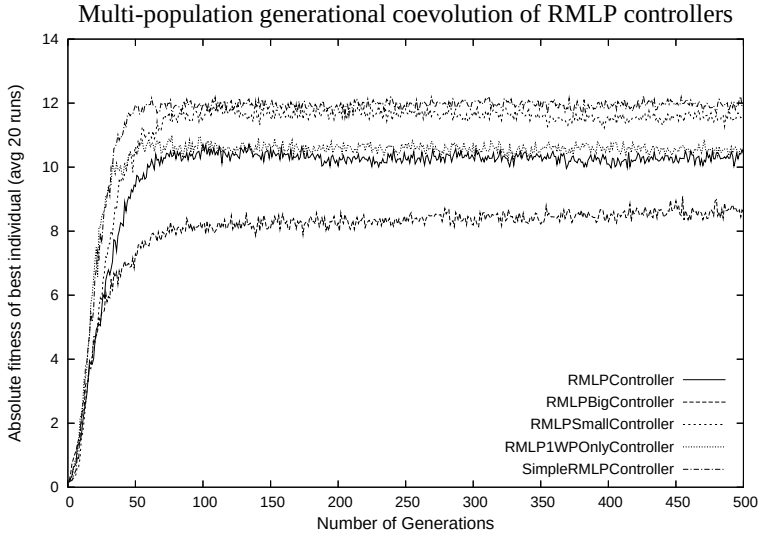


Figure 7.9: Nine-population generational co-evolution of different rmlp-based architectures, 10 runs.

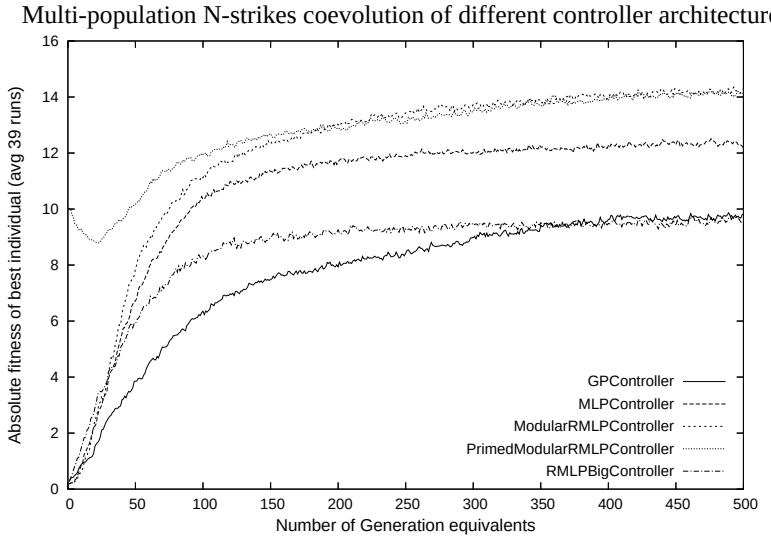


Figure 7.10: Nine-population steady-state co-evolution of diverse architectures.

looks essentially like figure 7.9 but with slower fitness growth.

Single-architecture multi-population co-evolution

In the final set of experiments, we filled all nine populations with RMLPBig controllers, as these are in theory capable of expressing the most complex strategies. We then evolved them for 500 generations, 10 runs each for the generational and steady-state algorithms. We are not showing any graphs of the same type as for the other experiments, as these would be quite uninteresting: all the populations reach the same fitness (of their best individuals) on average, with little differences between the populations in an individual run. This fitness is around 8 or 8.5, virtually the same as the fitness of the RMLPBig controllers in the multi-architecture runs.

Comparison of best controllers

At this point, the reader will probably wonder which of the experiments above actually produced the best controllers. As always with co-evolution, this question is not straightforward to answer. But we've tried to answer this in two ways: by measuring the competition score of representatively high-performing controllers (the best we could find from a limited probe) from each experiment, and by testing the same controllers against each other in two-car racing.

Competition score is the score used to rank submitted controllers in the league table of the CIG Car Racing Competition. It is calculated by letting the controller race 500 trials on its own, and 500 trials each against a rather low-performing hard-coded heuristic controller and a medium-performing MLP-based controller. Table 7.3 shows the competition score breakdown for the selected high-performing controllers. Judging from these scores, the multi-population runs yielded the best overall controllers, the single-population runs slightly less good and the

Controller	solo	heuristic	fixed	competition score
n-pop gen MLP	16.86	12.01	12.27	13.71
n-pop gen Modular	16.49	9.52	**13.37**	13.13
n-pop nstr MLP	17.40	**11.85**	12.73	**13.99**
n-pop nstr Modular	16.59	10.54	12.96	13.36
1-pop gen Modular	16.38	6.96	13.20	12.18
1-pop gen RMLPBig	14.90	11.30	11.89	12.69
1-pop nstr Modular	15.19	10.06	10.16	11.80
1-pop nstr RMLPBig	16.64	11.08	11.68	13.13
Solo Modular	**18.50**	2.80	3.20	8.17
Solo RMLPBig	17.35	4.91	6.54	9.60

Table 7.3: Competition score of a variety of high-performing controllers

solo evolutionar runs really rather bad controllers. The best controller found seem to be based on a MLP, but the differences are not great between the architectures.

Table 7.4 shows the results of direct competition between these controllers. The differences are sometimes quite dramatic, as when the solo-evolved RMLPBig gets 8 points lower fitness than the single-population-evolved modular controller. If a clear winner has to be picked, it is the modular controller evolved with generational multi-population co-evolution, which does not lose significantly to any other controller. Generally, the same dominance pattern of multi-population over single-population over solo evolution persists.

7.2.4 Discussion

We set ourselves a handful of questions to solve at the beginning of the section, and even though we can draw quite a few conclusions from our experiments, all the questions have not been answered. To begin with our conclusions, we found that when two cars were involved, the modular controllers outperform all the other controller architectures, and they do quite well otherwise as well. For solo- and single-population evolution larger, more complex controllers learn faster, whereas this is not the case for multi-population co-evolution. It thus seems that to the extent that multi-population co-evolution is useful for comparing the performance of different controller architectures, it is so in a rather different way than single-population co-evolution. What is clear about the multi-population co-evolution is that (for our particular problem instance) it produces generally better controllers than either solo evolution or single-population co-evolution, corroborating one of our main hypotheses.

Why large networks seem to learn faster than small networks is an interesting question, and our results are somewhat at odds with the received wisdom in the among neuroevolution researchers. For example, the NEAT algorithm starts out with minimally connected networks, and incrementally complexifies these. The stated reason for this is that the search space should

	n-pop gen MLP	n-pop gen Modular	n-pop nstr MLP	n-pop nstr Modular	1-pop gen Modular	1-pop gen RMLPBig	1-pop nstr Modular	1-pop nstr RMLPBig	Solo RMLPBig
n-pop gen MLP	0.00	-0.75	0.09	-0.42	-0.81	1.20	2.37	1.17	4.58
n-pop gen Modular	0.75	**0.00**	**0.71**	**0.35**	0.09	1.91	2.65	**1.90**	5.48
n-pop nstr MLP	-0.09	-0.71	0.00	-0.53	-0.31	1.08	2.81	1.50	4.64
n-pop nstr Modular	0.42	-0.35	0.53	0.00	**0.17**	1.06	**2.82**	1.52	4.31
1-pop gen Modular	**0.81**	-0.09	0.31	-0.17	0.00	**2.10**	0.68	1.77	3.57
1-pop gen RMLPBig	-1.20	-1.91	-1.08	-1.06	-2.10	0.00	1.56	0.36	4.15
1-pop nstr Modular	-2.37	-2.65	-2.81	-2.82	-0.68	-1.56	0.00	-1.51	**8.02**
1-pop nstr RMLPBig	-1.17	-1.90	-1.50	-1.52	-1.77	-0.36	1.51	0.00	3.88
Solo RMLPBig	-4.58	-5.48	-4.64	-4.31	-3.57	-4.15	-8.02	-3.88	0.00

Table 7.4: Score differences in competitions between controllers (average of 500 games). A positive value means that the controller of the current row beats the controller of the current column.

be reduced, which is assumed to lead to better or faster learning [123]. However, our results trigger the suspicion that minimal networks might not have any advantages at all over larger networks, except for the lesser computation required in propagating values through the network.

We are currently not sure about why the more complex controllers seem to learn faster. It is entirely possible that this is a domain-specific effect, linked e.g. to the mutation magnitude employed - for a given mutation magnitude, there is larger chance that a Gaussian mutation will lead to a radical change in at least one dimension in a high-dimensional search space than a low-dimensional one. Further experimentation is needed in order to test whether this effect persists over different parameter settings and domains.

A major unsolved puzzle is why the RMLPBig controller fared so badly, and generally, why RMLP-based controllers did better the simpler they were, in the multi-population experiments. Our main hypothesis here is that because the more complex controllers learn faster, they settle for a particular strategy sooner than the others, and that this particular strategy is not a very good one because the other controllers have not yet developed any useful strategies. Once learned, these strategic niches function as local optima, as learning a generally better strategy would require first unlearning the current mediocre strategy. The slow-learning simple controllers instead find themselves competing against several more well-developed strategies as soon as they unfold their wings, and are forced to learn more general strategies. If this is true, and the effect generalises to other problems, this "slow learner's advantage" could be a valuable tool in our understanding of co-evolutionary dynamics. We are currently thinking of the best way to test this hypothesis.

What we have not had time (and space) to investigate here is the behavioural diversity between the evolved controllers from the different experiments. Thus, we don't know whether the superior performance of the multi-generation co-evolution is because of increased diversity between populations, but we still think this is the case. This needs to be investigated further in a forthcoming paper, using both quantitative and qualitative measures.

We are also careful to point out that we are not claiming the superiority of multi-population competitive co-evolution over two-population co-evolution in the general case. At least not just yet. The experiments in this section will first have to be validated through independent repetition in different domains and with parameter settings.

7.3 Personalized racing track creation

In this section, we are discussing the evolution of fun racing tracks for human players. As stand-ins for the human players (who would certainly not be interested in driving the same

track hundreds of times with only slight variations) we use models of their driving styles in the evalution of the tracks' fitness. The driving style models are exactly those acquired in section 6.1, and this section is also based on the same two papers as that other section.

In order to evolve a racing track based on a player model, we must know what it is for a racing track to be fun, how we can measure this property, and how the racing track should be represented in order for good track designs to be in easy reach of the evolutionary algorithm. We have not been able to find any previous research on evolving tracks, or for that sake any sort of computer game levels or environments. However, Ashlock et al.'s paper on evolving path-finding problems is worthy to mention as a an example of an approach that could possibly be extended to certain types of computer games [5].

We have discussed some theories about what makes computer games fun in section 3.1. Of special interest here is the theory of Malone that fun depends on challenge, fantasy and curiosity. At least challenge and curiosity could potentially be measurable. It seems clear that a racing track should not be too challenging, and not too easy either. As for the curiosity, or "optimal level of informational complexity", this could mean that the track should be varied so that different types of challenges alternate when you drive around it.

An observation of our own, confirmed by the opinions of an unstructured selection of non-experts, is that tracks are fun where it is possible to drive very fast on straight sections, but it is necessary to brake hard in preparation for sharp turns, turns which preferably can be taken by skidding. In other words, it's fun to almost lose control. However, it is possible that this is a matter of personality, and that different players attach very different values to different fun factors. Some people seem to like to be in control of things, and people have very different attention spans, which should mean that some people would want tracks that are easier to learn than others. Identifying different player types and being able to select a mix of fun factors optimal to these players would be an interesting project, but we are not aware of any empirical studies on that subject.

7.3.1 Fitness functions

Developing reliable quantitative measures of, and ways of maximising, all the above properties would probably require significant effort. For these experiments we chose a set of features which would be believed not to be too hard to measure, and designed a fitness function based on these. The features we want our track to have for the modelled player, in order of decreasing priority, is the right amount of challenge, varying amount of challenge, and the presence of sections of the track in which it is possible to drive really fast. The corresponding fitness functions are:

- f_1: the negative difference between actual progress and target progress (in this case defined

as 30 waypoints in 700 timesteps),

- f_2: variance in total progress over five trials of the same controller on the same track,
- f_3: maximum speed.

7.3.2 Track representation

In our earlier paper we evolved fixed-length sequences of track segments. These segments could have various curvatures and decrease or increase the breadth of the track. While this representation had the advantage of very good evolvability in that we could maximise both progress and progress variance simultaneously, the evolved tracks did look quite jagged, and were not closed; they ended in a different point than they started, so the car had to be "teleported" back to the beginning of the track. We therefore set out to create a representation that, while retaining evolvability, allowed for smoother, better-looking tracks where the start and end of the track connect.

The representation we present here is based on b-splines, or sequences of Bezier curves joined together. Each segment is defined by two control points, and two adjacent segments always share one control point. The remaining two control points necessary to define a Bezier curve are computed in order to ensure that the curves have the same first and second derivatives at the point they join, thereby ensuring smoothness. A track is defined by a b-spline containing 30 segments (the full genome thus consisting of a 60-dimensional array), and mutation is done by perturbing the positions of their control points through adding small numbers from a Gaussian distribution.

It can be noted that even though this representation makes nonsensical and undrivable tracks possible, and even probable, such tracks will be strongly selected against and quickly weeded out of the population, precisely because they are undrivable.

The collision detection in the car game works by sampling pixels on a canvas, and this mechanism is taken advantage of when the b-spline is transformed into a track. First thick walls are drawn at some distance on each side of the b-spline, this distance being either set to 30 pixels or subject to evolution depending on how the experiment is set up. But when a turn is too sharp for the current width of the track, this will result in walls intruding on the track and sometimes blocking the way. The next step in the construction of the track is therefore "steamrolling" it, or traversing the b-spline and painting a thick stroke of white in the middle of the track. Finally, waypoints are added at approximately regular distances along the length of the b-spline. The resulting track (see fig.6.1) can look very smooth, as evidenced by the test track which was constructed simply by manually setting the control points of a spline, through

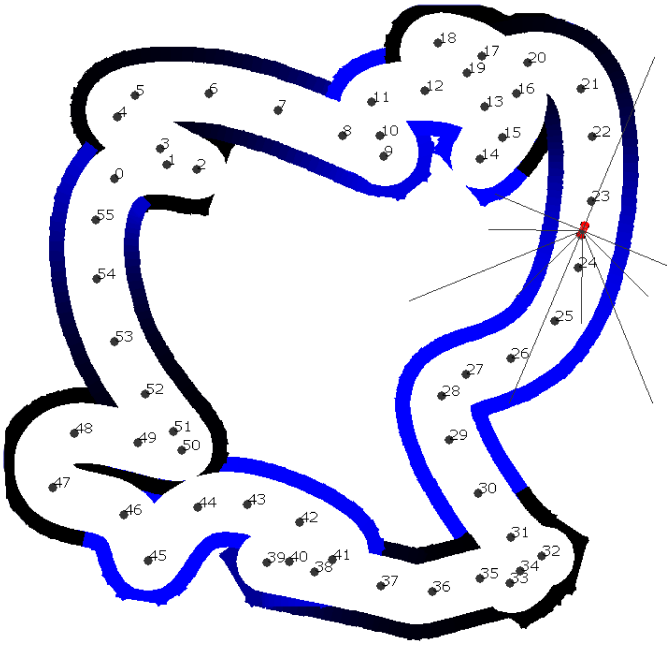


Figure 7.11: Track evolved using the random walk initialisation and mutation.

a process of human trial-and-error.

7.3.3 Initialisation and mutation

In order to investigate how best to leverage the representational power of the b-splines, we experimented with several different ways of initialising the tracks at the beginning of the evolutionary runs, and different implementations of the mutation operator. Three of these configurations are described here.

Straightforward

The straightforward method starts from an initial track shape forming a rectangle with rounded corners. Each mutation operation then perturbs one of the control points by adding numbers drawn from a gaussian distribution with standard deviation 20 pixels to both x and y axes.

Random walk

In the random walk experiments, mutation proceeds like in the straightforward configuration, but the initialisation is different. A rounded rectangle track is first subject to random walk, whereby hundreds of mutations are carried out on a single track, and only those mutations that result in a track on which a generic controller is not able to complete a full lap are retracted. The result of such a random walk is a severely deformed but still drivable track. A population is then initialised with this track and evolution proceeds as usual from there.

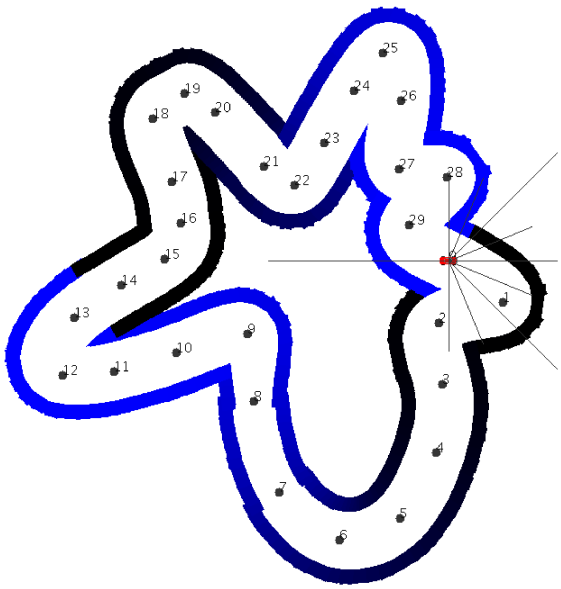


Figure 7.12: A track evolved (using the radial method) to be fun for the first author, who plays too many racing games anyway. It is not easy to drive, which is just as it should be.

Radial

The radial method of mutation starts from an equally spaced radial disposition of the control points around the center of the image; the distance of each point from the center is generated randomly. Similarly at each mutation operation the position of the selected control point is simply changed randomly along the respective radial line from the center. Constraining the control points in a radial disposition is a simple method to exclude the possibility of producing a b-spline containing loops, therefore producing tracks that are always fully drivable.

7.3.4 Results

We evolved a number of tracks using the b-spline representation, different initialisation and mutation methods, and different controllers derived using the indirect player modelling approach.

Straightforward

Overall, the tracks evolved with the straightforward method looked smooth, and were just as easy or hard to drive as they should be: the controller for which the track was evolved typically made a total progress very close to the target progress. However, the evolved tracks didn't differ from each other as much as we would have wanted. The basic shape of a rounded rectangle shines through rather more than it should.

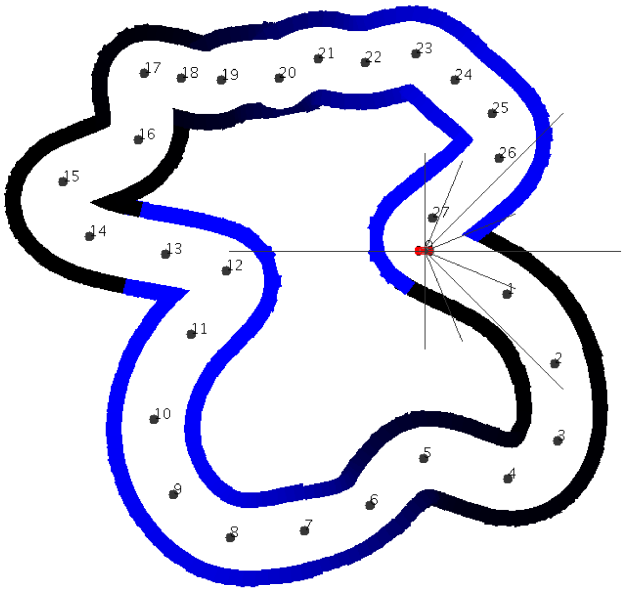


Figure 7.13: A track evolved (using the radial method) to be fun for the second author, who is a bit more careful in his driving. Note the absence of sharp turns.

Random walk

Tracks evolved with random walk initialisation look weird (see 7.11) and differ from each other in an interesting way, and so fulfil at least one of our objectives. However, their evolvability is a bit lacking, with the actual progress of the controller often quite a bit different from the target progress and maximum speed low.

Radial

With the radial method, the tracks evolve rather quickly and look decidedly different (see fig.7.12 and 7.12 depending on what controller was used to evolve them, and can thus be said to be personalised. However, there is some lack of variety in the end results in that they all look slightly like flowers, clear bias of the type of mutation used.

Comparison with segment-based tracks

It is interesting to compare these tracks with some tracks evolved using the segment-based representation from our previous paper. Those tracks (see fig.7.14) do show both the creativity evolution is capable of and a good ability to optimise the fitness values we define. But they don't look like anything you would want to get out and drive on.

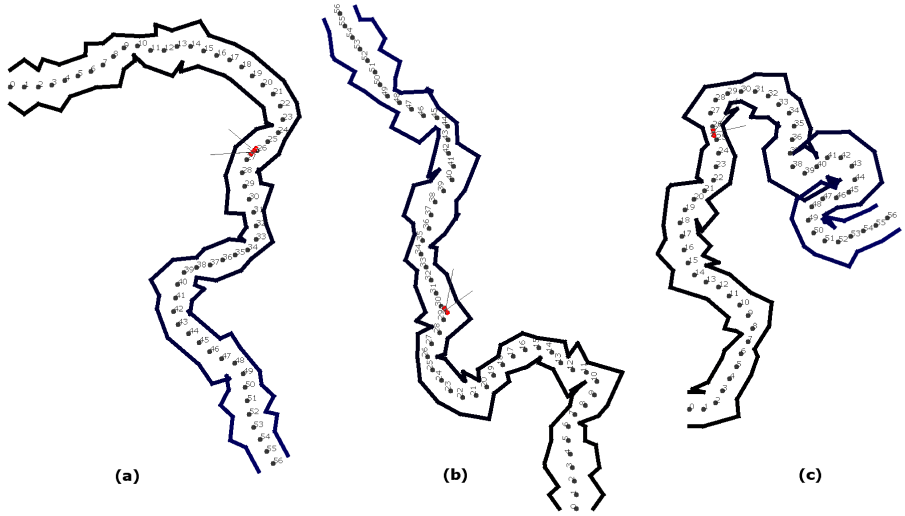


Figure 7.14: Tracks evolved using the segment-based method. Track (a) is evolved for a weak player, and tracks (b) and (c) for a good player. Tracks (a) and (b) are evolved using all three fitness functions defined above, while track (c) is evolved using only progress fitness.

7.3.5 Discussion

Based on the player models at hand we seem to be able to evolve tracks that satisfy our own fitness criteria reasonably well. These tracks also look rather interesting, at least in our opinion. However, whether they are actually more fun to drive than tracks generated randomly or without a player model is not obvious. To find out, we would need to conduct empirical studies with a large number of human subjects. This would indeed be interesting research in its own right, but we currently don't have time or resources.

Further, there is much work let to do before we have a track representation and evolutionary method that can compete with hand-designed tracks. Currently, the tracks either look a bit like flowers, or very jagged. Adding a third dimension to the racing makes devising a good representation even more challenging.

7.4 Summary

This chapter presented three series of experiments on evolving controllers or structures with an unclear and open-ended fitness criterion; we don't know exactly what we want, but we are looking for ways to get there, and might know when we are on the right way. The two first sections concern competitive co-evolution. We explored the absolute-relative fitness spectrum, and introduced multi-population co-evolution with different controller architectures. The wealth of interesting and occasionally counter-intuitive results from especially the second section points to that the combination of competitive co-evolution with game agent control holds considerable promise, and that this enterprise is still in its initial, explorative stage.

The third section concerned the evolution of racing tracks based on previously acquired player models. This area is even more underexplored, and the results in this section are highly tentative (they also suffer from the problem of being very hard to validate without the use of studies on humans). At the same time, this project could potentially be very rewarding, in that it could open up a new application field for computational intelligence in games.

Chapter 8

Conclusions

We have in this thesis presented a large number of experiments that apply evolution, and occasionally other reinforcement learning or supervised learning algorithms, to different games in different manners with different goals and outcomes. Writing a single conclusion chapter to all this might seem a bit contrived. On the other hand, not having a conclusion wouldn't be right either.

Thus, this chapter will contain two main sections, summarizing the main contributions and open directions for future research from two different perspectives: that of computational intelligence and that of game AI. Within each section, our contributions will be discussed in the context of a number of open research topics.

A final section will summarise and discuss future research directions.

8.1 The computational intelligence perspective

In this section, we view the experiments of the preceding chapters in the perspective taken in chapter 2. In the following, we shall go through a number of topics and issues central to that chapter, and discuss how the experiments in this thesis can contribute to our understanding of them.

8.1.1 Robot reality and simulation

In section 2.3.1 we discussed the problems associated both with evolving on real robots and with acquiring robot simulations good enough to meaningfully evolve behaviour in. Throughout this thesis we have demonstrated how computer games can provide complex dynamic environments good enough to evolve (and otherwise learn) interesting behaviour in. We have also have argued qualitatively for the suitability of computer games as environments in which to evolve complex

general behaviour in section 3.1, suggesting that computer games to a rather large extent can replace robotics in evolutionary robotics.

However, sometimes the goal is not to learn behaviour in general, but rather to learn to control a particular physical robot. In section 6.2 we presented a general approach for modelling the dynamics of robots and evolving controllers in simulation that could be transferred back to the physical robot. Central to that approach is the multi-model evolution, where the controllers are tested with several models developed using different learning algorithms and/or representations.

The most obvious way to continue the research described here is to show that really complex behaviour can be evolved in computer games. For this we probably need to switch from home-made computer games to complex commercial games, and from the simplified input we currently use to high-dimensional visual inputs. (At least if we take the symbol grounding problem and importance of closing the sensorimotor loop seriously; while it might be very possible to evolve good game solutions using the internal representations of the game, this would arguably not amount to evolving *intelligence*.) Another path to take is to try to model the dynamics of more complex robots (or possibly robots in more complex environments) and see if multi-model controller evolution scales up. This path is currently being taken for the micro-helicopters in the UltraSwarms project.

8.1.2 Incrementality and modularity

As discussed in sections 2.4.2 and 2.4.3, decomposing the controller structurally and/or functionally, and decomposing the learning of it temporally, can be very useful from the standpoint of efficiently learning effective controllers. In this thesis, no truly new theories about incrementality or modularity have been presented, but instead we have demonstrated the applicability of the concepts to a few domains (in at least slightly novel ways) and so corroborated existing theories.

In sections 5.4.2, 5.4.1 and 7.2 we showed the superiority of modular neural network-based controllers over non-modular controllers. In the case of the Cellz game, the modularity was of a symmetrical or convolutional kind which we have not seen used in control learning before. In the helicopter learning experiments, the difference in performance between modular and non-modular controller representations was qualitative rather than quantitative, in that the non-modular networks would not learn to fly at all. The same experiments also convincingly showed that there are "wrong" modularizations that don't improve evolvability at all, and demonstrated the usefulness of combining modularity with incrementality. In the case of the multi-population co-evolution experiments, the extensive list of controller architectures compared proved that the superior performance of the carefully modularized architectures are not due to an unlucky choice

of non-modular architecture.

The experiments with incremental generalization and specialization of driving skills in section 5.2 demonstrate the application of incremental approaches in a novel way, as the task is not fundamentally changed as much as aspects of the task is removed and taken away; an interesting result here is that "general" driving skills can be incrementally evolved, pointing to the usefulness of incremental evolution for evolving truly general intelligence.

As the benefits of incrementality and (the right sort of) modularity are well established, these concepts are probably to be regarded as tools rather than future research topics in their own right. A related reserach topic is instead to show that the right type of tasks, i.e. computer games, naturally have an inherent incremental structure suitable to learning complex general behaviour. Another future research topic is to develop algorithms that automatically find suitable modularisations for given problems. However, unpublished research by the author suggests that this is not easy at all. The currently most promising direction here is to use some sort of memetic algorithm, where selection takes place on two different time scales: on a larger time scale, a suitable structure for the function representation (e.g. modularisation of the neural network) is evolved. Each evaluation of the modular structure would then involve a search of the parameter space of the function representation on a smaller time scale (e.g. local search through a hill climber for neural network weights). Such an algorithm would almost certainly have to trade slower learning speed for simpler problems for better final fitness for more complex problems.

8.1.3 Controller architectures and learning methods

Sections 2.4.4, 2.4.5 and 2.4.7 discuss the relative merits of different controller architectures and different types of reinforcement learing algorithms. These sections also made it clear that there is little consensus on these topics. Unfortunately, this thesis will not provide such a consensus, but might take us a little bit on the path toward true insight into this important issue.

Most of the experiments in this thesis compare controller architectures or learning algorithms in one way or another. Apart from the points about modularity, a number of main conclusions can be drawn:

First of all, all the non-modular controller architectures we have tried in this thesis have had rather similar learning abilities. There have quantitative differences in both learning speed and eventual performance of the best learned controllers, but these differences have been quantitative rather than qualitative, and not really dramatical (in the extremes, one architecture might eventually reach twice the fitness of another, but never ten times the fitness). When there has been a qualitative difference in learning abilities, this has always been down to inappropriate

inputs to the controller (such as in section 5.1), or to non-modular controllers not being up to the job.

Especially, in the two sections where genetic programming and neuroevolution were compared (5.3.1 and 7.2), we found gp-based controllers to be able to learn just the same tasks as neural network-based controllers. However, the best neural network-based controllers always quantitatively outperformed the best gp-based controllers. Usually, the gp-based controllers learned faster, but this effect was not as pronounced.

In those two sections we also compare (potentially) stateful with reactive controllers. We found no significant performance difference between the two classes. This could be because the tasks we used don't really require the use of internal state, but more likely it is because something else in our experimental setup prevents the evolution of solutions that harness the full power of recurrent neural networks and object-oriented genetic programming. Also, we have not analyzed the evolved solutions further to look for potential differences in driving style, or measured the use of state.

In section 7.2 we also compare neural network-based controllers of different sizes, i.e. with different number of hidden neurons. The results are somewhat surprising, in that the controllers of different sizes all seem to eventually reach the same fitness, but the larger controllers get there faster. We are currently not aware of any theory that can explain this.

Finally, in section 5.3.2 we compare evolution with temporal difference learning, and state-value control with action-value and direct control (the rest of the controller learning experiments in this thesis use direct control exclusively). The results, that state-value control outperforms action-value and direct control, and that temporal difference learning learns faster than evolution but is less reliable and ultimately reach lower fitness levels, are in line with our own ideas and some - but not all! - published studies on similar domains.

With disagreement over these topics abundant, many more such comparative studies need to be done in the future, and theory needs to be developed that is supported by their results. Eventually, this research might lead to reinforcement learning algorithms that combine the best features of evolution and temporal difference learning, and which dynamically select the most appropriate controller representation. A rather straightforward way of doing this, which in various incarnations have been suggested or (to a limited extent) explored by various researchers is to evolve populations of controllers which use some form of td-learning to further learn during their lifetime. A more radical approach would be to start by looking at what information we have (local reinforcements, global fitness measures), what methods for change we have (stochastic variation, recombination, gradient descent in model space, instance-based learning etc.) and try to synthesise a new algorithm from these parts. Quite possibly, evolution itself could be

employed in the design of the learning algorithm, e.g. by giving the various components of the learning algorithm as primitives to a genetic programming system.

8.1.4 Co-evolution

The topics of section 2.4.6 are explored in two sections in the innovation chapter, namely 7.1 and 7.2. In these sections we competitively co-evolve car controllers using single-population co-evolution in the track-based racing game, and multi-population co-evolution in the point-to-point racing game, respectively.

The main positive conclusion of the first of these sections is that different mesures of relative and absolute progress measures in the fitness measure leds to interestingly different behaviour (the videos of sneaky evolved drivers pushing each others off the track have proved quite popular on on-line video sharing sites). On the other hand, the best competitively co-evolved controllers for the two-car version of the track-based racing game were nowhere near as human-competitive as solo-evolved controllers on the solo racing version of the game. The co-evolved controllers are just not good enough; this might be due to the limited amount of sensors or limited size of the neural networks, or to pathologies of the co-evolutionary algorithm. Obviously, there is much scope for improving the performance of such controllers in new experiments, possibly using the insights gained fromt the work described in the next section.

In that section, we show that multi-population competitive co-evolution really works, in that overall better controllers are evolved than those that are evolved using single-population co-evolution or solo evolution. The results also point to a few surprising results, especially that there seems be a correlation between larger size neural networks, faster learning and *lower* eventual fitness - in the multi-population case, but not the single-population case! Further investigating these effects in particular and the use of multi-population competitive co-evolution in general is currently a top research priority.

8.2 The game AI perspective

In the last section, we asked what games can do for us (well, for computational intelligence). In this section, we will ask what we can do for games, thereby adopting the perspective of chapter 3. The three potential contributions to computer games development that can be found in this theses are the methods for generalization and specialization, for generation of diverse opponents, and last but not least for personalized content creation.

8.2.1 Generalization and specialization

In section 5.2 we demonstrated that it is possible to incrementally evolve not only generally good driving behaviour, but also driving behaviour that is specialized to drive very well on particular tracks, at the expense of some general driving ability. Furthermore, while incremental generalization takes a good number of generations, the specialization process is much faster.

One obvious application of this is in racing games which contain a track editor, with which the player can design his own racing tracks (several such games exist, e.g. *TrackMania Nations*, which currently doesn't provide computer-controlled opponents on user-created tracks). The method would be to first evolve general controllers that can race all tracks that can be developed using the track editor (subject to some constraints such as the tracks being drivable at all). When the player then designs a track, the specialization process can then be used to quickly develop opponents that can give the player a match on his own track. The drivers don't need to be evolved for maximum speed, they could instead be evolved to e.g. match the speed of the driver.

Another use of the specialization process could be to measure the difficulty of the player-designed track. It could for example be defined as being equal to the time it takes to specialize to full fitness (starting from a number of different general controllers), or simply the fitness of the specialized controller after a certain number of generations.

This method could in principle be extended to many other types of games where it is possible to talk about playing skill in general and in specific cases, mainly agent games but possibly also management games.

8.2.2 Generation of diverse opponents

In sections 7.1 and 7.2 we present different methods of evolving controllers which differs significantly not only in driving performance but also in driving style. One way of doing this is to tune the fitness function, mixing different amount of relative and absolute fitness. Another way is to use multi-population competitive co-evolution, where the controllers in the different populations are forced into different ecological niches. Yet another way, which is not explored in this thesis, would be to use multi-objective evolutionary algorithms, and define a number of behavioural objectives.

In driving games, having a diverse set of opponents can certainly make the game more fun; there is more variety, and thus more to learn. This is also true of almost any game with more than one NPC, agent and management game alike.

8.2.3 Personalized content creation

The possibly most innovative experiments in this thesis are reported in sections 6.1 and 7.3. In these experiments, players' driving styles are modelled, and tracks are evolved for maximal entertainment of the modelled players. This could give rise to significant add-ons to existing racing (and other) games, or inspire the creation of new types of games. The methods could be combined with generative encodings for environments, to evolve entertaining cities, battle maps, levels, clothing, etc. And who would not want a racing game where the track you are driving never repeats, but just always contains the sort of parts you really enjoy driving?

The experiments on personalized content creation in this thesis are definitely exploratory rather than mature, but the scope for future research along the lines drawn up there is immense.

8.3 Summary and future research directions

Computational intelligence and computer games can be combined in a great variety of ways. In this thesis, a taxonomy has been given of approaches to computational intelligence in games, and existing research within the nascent field of computational intelligence and games has been positioned within this taxonomy.

Three distinct approaches to computational intelligence in games have been identified: optimization, imitation and innovation. Within each approach computational intelligence can be used for the purposes of enhancing computer games, and likewise, computer games can also be used to enhance computational intelligence by providing stimulating problems for new and existing algorithms. We want to emphasise that with stimulating problems we mean not only benchmarks for machine learning algorithms, but also the sort of environments in which complex general intelligence might emerge, consonant with the original goal of evolutionary robotics.

A number of experiments have been presented within all three approaches, with several experiments designed to be of interest both for computational intelligence research and for game development. The majority of the experiments use simulated (and in one case real) car racing as experimental environments, but experiments have also been presented that use two other physics-based agent games.

The three most significant (or at least least insignificant) contributions to science in this thesis, according to the author, has been listed in section 1.1. In short, they are the personalized content creation idea (along with a proof of concept player modelling and track evolution implementation) discussed in sections 6.1 and 7.3, the multi-model approach to dynamics modelling for controller evolution discussed in section 6.2, and the multi-population competitive co-evolution algorithm discussed in section 7.2.

Plenty of possibilities for taking the research presented in this thesis further has been discussed in this chapter as well as in the experimental chapters. It is our belief that the potential for evolution of complex general intelligence inherent in the car racing problem has not been exhausted yet, and that there is much good research left to do by scaling up to a slightly more complex version of the problem (e.g. allowing for more cars simultaneously on the track), allowing for higher-dimensional input, using more powerful function representations (such as complexifying neural networks) and more sophisticated co-evolutionary algorithms. And when the best possible racing car controller for a sophisticated racing game, winning against a variety of good opponents with widely differing strategies, we can move on to the next game with even more potential - which could be an evolution of car racing, as is Wipeout or Grand Theft Auto, allowing for already evolved controllers to incrementally develop.

But when we have evolved the best possible racing car controller we will have already have achieved a lot - almost certainly we would have generated more complex general intelligence that has ever been automatically generated before. Such an achievement would at the very least force various research communities to take computer games more seriously, and possibly also game developers to take computational intelligence research more seriously.

However, in addition to grand visions of the future, much groundwork remains to be done. Many of the experiments described in this thesis have led to interesting results, pointing to possible new insights into the nature of evolutionary learning. These include the incremental evolution of general and specific control, the cancellation of exploitable model deficiencies through multi-model evolution, the slow-learner's advantage in multi-population competitive co-evolution, and several others. However, in no case have we shown that the validity of these results extend significantly beyond the particular experiments in which they were achieved.

In order to validate these results under more general sets of circumstances, they either have to be supported by theory, or by exhaustive experiments under systematically varied conditions (preferably both). As theoretical analysis of the efficacy of evolutionary algorithms have so far had a very limited scope (e.g. to the author's knowledge, there is no convergence proof for an evolutionary algorithm for any remotely interesting problem), we will have to rely on experimentation in order to validate our results. What needs to be done is first to repeat the relevant experiments while systematically varying the parameters, e.g. redoing the multi-population competitive co-evolution experiments while varying population size, selection pressure, mutation rate, etc. Then, the same experiments would have to be re-done in different domains, e.g. using a board game instead of car racing for competitive co-evolution. Only when this is done, and the outcomes of the additional experiments corroborate those of the original ones, should we be prepared to accept the (relatively) general validity of the discovered phenomena.

Bibliography

[1] Pieter Abbeel, Morgan Quigley, and Andrew Y. Ng. Using inaccurate models in reinforcement learning. In *International Conference on Machine Learning (ICML) Pittsburgh PA, USA*, 2006.

[2] Alexandros Agapitos and Simon M. Lucas. Evolving a statistics class with object-oriented genetic programming. In *Proceedings of EuroGP*, 2007.

[3] Alexandros Agapitos, Julian Togelius, and Simon M. Lucas. Evolving controller for simulated car racing with object-oriented genetic programming. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO)*, 2007.

[4] Alexandros Agapitos, Julian Togelius, and Simon M. Lucas. Multiobjective techniques for the use of state in genetic programming applied to simulated car racing. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2007.

[5] D. Ashlock, T. Manikas, and K. Ashenayi. Evolving a diverse collection of robot path planning problems. In *Proceedings of the Congress On Evolutionary Computation*, pages 6728–6735, 2006.

[6] Christian Baekkelund. Academic ai research and relations with the games industry. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, 2006.

[7] Jerome H. Barkow, Leda Cosmides, and John Tooby. *The Adapted Mind: Evolutionary Psychology and the Generation of Culture.* Oxford University Press, 1996.

[8] Randall D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 1995.

[9] H.-G. Beyer and H.-P. Schwefel. Evolution strategies: A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[10] Cristopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[11] J. Bongard, V. Zykov, and H. Lipson. Resilient machines through continuous self-modeling. *Science*, 314:1118–1121, 2006.

[12] Josh C. Bongard. Evolving modular genetic regulatory networks. In *Proceedings of the Congress on Evolutionary Computation*, 2002.

[13] J. Borenstein and Y. Koren. The vector field histogram and fast obstacle-avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 7:278–288, 1991.

[14] Rodney Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[15] Bobby D. Bryant. *Evolving Visibly Intelligent Behavior for Embedded Game Agents*. PhD thesis, Department of Computer Sciences, University of Texas, Austin, TX, 2006.

[16] Bobby D. Bryant and Risto Miikkulainen. Exploiting sensor symmetries in example-based training for intelligent agents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2006.

[17] Gunnar Buason, Nicklas Bergfeldt, and Tom Ziemke. Brains, bodies, and beyond: Competitive co-evolution of robot controllers, morphologies and environments. *Genetic Programming and Evolvable Machines*, 6(1):25–51, 2005.

[18] Mat Buckland. Interview with jeff hannan, Publication date unknown.

[19] Seth Bullock. Co-evolutionary design: Implications for evolutionary robotics. Technical Report CSRP384, 1995.

[20] Raffaele Calabretta, Andrea Di Fernando, Gunter P. Wagner, and Domenico Parisi. What does it take to evolve behaviorally complex organisms? *BioSystems*, 2002.

[21] Raffaele Calabretta, Stefano Nolfi, Domenici Parisi, and Gunter P Wagner. Duplication of modules facilitates functional specialization. *Artificial Life*, 6:69–84, 2000.

[22] Werner Callebaut and Diego Rasskin-Gutman, editors. *Modularity: Understanding the Development and Evolution of Natural Complex Systems*. MIT Press, 2005.

[23] Angelo Cangelosi, Domenico Parisi, and Stefano Nolfi. Cell division and migration in a genotype for neural networks. Technical report, Institute of Psychology, CNR, Rome, 1993.

[24] Benoit Chaperot and Colin Fyfe. Improving artificial intelligence in a motocross game. In *IEEE Symposium on Computational Intelligence and Games*, 2006.

[25] P. S. Churchland, V. S. Ramachandran, and T. J. Sejnowski. A critique of pure vision. In Christof Koch and Joell Davis, editors, *Large-scale neuronal theories of the brain*. MIT Press, 1994.

[26] Andy Clark. *Being There: Putting Brain, Body and World Together Again*. MIT Press, 1998.

[27] Dave Cliff. Computational neuroethology: a provisional manifesto. In *Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, pages 29–39, 1991.

[28] Dave Cliff. Neuroethology, computational. In Michael A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*, pages 737–741. MIT Press, 2003.

[29] Dave Cliff, Inman Harvey, and Phil Husbands. Artificial evolution of visual control systems for robots. In M. Srinivasan and S. Venkatesh, editors, *From Living Eyes to Seeing Machines*. Oxford University Press, 1997.

[30] Nicholas Cole, Sushil J. Louis, and Chris Miles. Using a genetic algorithm to tune first-person shooter bots. In *Proceedings of the IEEE Congress on Evolutionary Computation*, page 139145, 2004.

[31] Peter Cowling. Writing ai as sport. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, 2006.

[32] Peter Cowling, Richard Fennell, Robert Hogg, Gavin King, Paul Rhodes, and Nick Sephton. Using bugs and viruses to teach artificial intelligence. In *Proceedings of The International Conference on Computer Games: Artificial Intelligence, Design and Education (CGAIDE)*, 2004.

[33] P. J. Darwen. Why co-evolution beats temporal difference learning at backgammon for a linear architecture, but not a non-linear architecture. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 1003–1010, 2001.

[34] Richard Dawkins and John R. Krebs. Arms races between and within species. *Proceedings of the Royal Society of London B*, 205:489–511, 1979.

[35] Peter Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2001.

[36] Renzo De Nardi, Julian Togelius, Owen Holland, and Simon M. Lucas. Evolution of neural networks for helicopter control: Why modularity matters. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[37] Jörg Denzinger, Kevin Loose, Darryl Gates, and John Buchanan. Dealing with parameterized actions in behavior testing of commercial computer games. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG)*, pages 37–43, 2005.

[38] Ezequiel A. Di Paolo. Homeostatic adaptation to inversion of the visual field and other sensorimotor disruptions. In *Proceedings of the Conference on the Simulation of Adaptive Behavior (SAB)*, pages 440–449, 2000.

[39] M. Dorigo, V. Trianni, E. Sahin, R. Gross, T. H. Labella, G. Baldassarre, S. Nolfi, J.-L. Deneubourg, F. Mondada, D. Floreano, and L. M. Gambardella. Evolving self-organizing behaviors for a swarmbot. *Autonomous Robots*, 17(2-3):223–245, 2004.

[40] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[41] Jeffrey Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[42] Andrea Di Fernando, Raffaele Calabretta, and Domenico Parisi. Evolving modular architectures for neural networks. In R French and J Sougn, editors, *Proceedings of the sixth Neural Computation and Psychology Workshop: Evolution, Learning, and Development*, pages 253–262, London, 2000. Springer Verlag.

[43] Dario Floreano, Toshifumi Kato, Davide Marocco, and Eric Sauser. Coevolution of active vision and feature selection. *Biological Cybernetics*, 90:218–228, 2004.

[44] Dario Floreano and Francesco Mondada. Evolution of plastic neurocontrollers for situated agents. In P. Maes, M. Mataric, J. Meyer, J. Pollack, H. Roitblat, and S. Wilson, editors, *From Animals to Animats IV: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, Cambridge, MA, 1996. MIT Press-Bradford Books.

[45] Jerry Fodor. *The modularity of mind*. The MIT Press, 1983.

[46] David B. Fogel. *Blondie24: playing at the edge of AI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[47] J.C. Gallagher, Randall D. Beer, K. S. Espenschied, and R. D. Quinn. Application of evolved locomotion controllers to a hexapod robot. *Robotics and Autonomous Systems*, 18:59–64, 1996.

[48] Matt Gilgenbach. Fun game ai design for beginners. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, 2006.

[49] Faustino Gomez and Risto Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.

[50] Faustino Gomez and Risto Miikkulainen. Active guidance for a finless rocket through neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2003.

[51] Faustino Gomez, Juergen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning (ECML)*, 2006.

[52] Thore Graepel, Ralf Herbrich, and Julian Gold. Learning to fight. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*, 2004.

[53] F. Gruau. *Neural network synthesis using cellular encoding and the genetic algorithm.* PhD thesis, Ecole Normale Superieure de Lyon, 1994.

[54] Frederic Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 318–325. Morgan Kaufmann, 1993.

[55] Stevan Harnad. The symbol grounding problem. *Physica D*, 42:335–346, 1990.

[56] Inman Harvey, Phil Husbands, and Dave Cliff. Issues in evolutionary robotics. In *Proceedings of the Conference on Simulation of Adaptive Behavior (SAB)*, 1993.

[57] Inman Harvey, Phil Husbands, Dave Cliff, Adrian Thompson, and Nick Jakobi. Evolutionary robotics: the sussex approach. *Robotics and Autonomous Systems*, 20:205–224, 1997.

[58] Ralf Herbrich. (personal communication), 2006.

[59] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In *Proceedings of the ninth annual international conference of the Center for Nonlinear Studies on Self-organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks on Emergent computation*, pages 228–234, 1990.

[60] Owen E. Holland. *Machine Consciousness.* Imprint Academic, 2003.

[61] Gregory S. Hornby, Hod Lipson, and Jordan B. Pollack. Evolution of generative design systems for modular robots. In *IEEE Conference on Robotics and Automation*, 2001.

[62] Gregory S. Hornby and Brian Mirtich. Comparing diffuse and true coevolution in a physics-based world. Technical Report TR-98-11, 1999.

[63] Michael Husken, Christian Igel, and Marc Toussaint. Task-dependent evolution of modularity in neural networks. *Connection Science*, 14:219–229, 2002.

[64] Kwang-Young Im, Se-Young Oh, and Seong-Joo Han. Evolving a modular neural network-based behavioral fusion using extended vff and environment classification for mobile robot navigation. *IEEE Transactions on Evolutionary Computation*, 6:413–419, 2002.

[65] Nick Jakobi. Evolutionary robotics and the radical envelope-of-noise hypothesis. *Adaptive Behavior*, 6(2):325–368, 1997.

[66] Nick Jakobi. Harnessing morphogenesis. In *Proceedings of Information Processing in Cells and Tissues*, 1997.

[67] D. H. Janzen. When is it co-evolution? *Evolution*, 34(3):611–612, 1980.

[68] Dan-Anders Jirenhed, Germund Hesslow, and Tom Ziemke. Exploring internal simulation of perception in mobile robots. In *Proceedings of the Fourth European Workshop on Advanced Mobile Robots*, pages 107–113, 2001.

[69] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[70] Jerome Kodjabachian and Jean-Arcady Meyer. Evolution and development of modular control architectures for 1d locomotion in six-legged animats. *Connection Science*, 10(3-4):211–237, 1998.

[71] Raph Koster. *A theory of fun for game design.* Paraglyph press, 2005.

[72] C. Kotnik and J. Kalita. The significance of temporal-difference learning in self-play training: TD-rummy versus EVO-rummy. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 369 – 375, 2003.

[73] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[74] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[75] John R. Koza. *Genetic Programming III: Darwinian Invention and Problem Solving.* MIT Press, 1999.

[76] John E. Laird and Michael van Lent. Human-level ai's killer application: Interactive computer games. In *Proceedings of the Seventeenth National Conference on Artificial*

Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, pages 1171–1178, 2001.

[77] Simon M. Lucas. Cellz: A simple dynamic game for testing evolutionary algorithms. In *Proceedings of CEC 04*, 2004.

[78] Simon M. Lucas. Exploiting reflection in object-oriented genetic programming. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2004.

[79] Simon M. Lucas and Thomas P. Runarsson. Temporal difference learning versus co-evolution for acquiring othello position evaluation. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2006.

[80] Simon M. Lucas and Julian Togelius. Point-to-point car racing: an initial study of evolution versus temporal difference learning. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[81] Thomas W. Malone. What makes things fun to learn? heuristics for designing instructional computer games. In *Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems*, pages 162–169, 1980.

[82] Hugo Marques and Owen Holland. Minimal architectures for embodied imagination. In *Proceedings of Brain Inspired Cognitive Systems 2006*, Lesvos, Greece, 2006.

[83] Hugo Marques, Julian Togelius, Magdalena Kogutowska, Owen Holland, and Simon M. Lucas. Sensorless but not senseless: prediction in evolutionary car racing. In *Proceedings of the IEEE Symposium on Artificial Life*, 2007.

[84] James Matthews. Interview with jeff hannan, 2001.

[85] Warren McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[86] Maurice Merleau-Ponty. *Phenomenology of Perception.* Routledge, 1945.

[87] B Mettler, M B Tischler, and T Kanade. System identification modeling of a small-scale unmanned rotorcraft for flight control design. *Journal of the American Helicopter Society*, 47(1):50–63, 2002.

[88] Thomas Miconi and Alastair Channon. The n-strikes-out algorithm: A steady-state algorithm for coevolution. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1639–1646, 2006.

[89] Microsoft. Terrarium, 2005.

[90] Chris Miles and Sushil J. Louis. Towards the co-evolution of influence map tree based strategy game players. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2006.

[91] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[92] Francesco Mondada, Edoardo Franzi, and Paolo Ienne. Mobile robot miniaturisation: A tool for investigation in control algorithms. In *Proceedings of the 3rd International Symposium on Experimental Robotics*, 1993.

[93] Alberto Moraglio and Julian Togelius. Geometric particle swarm optimization for the sudoku puzzle. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO)*, 2007.

[94] Alberto Moraglio, Julian Togelius, and Simon M. Lucas. Product geometric crossover for the sudoku puzzle. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2006.

[95] David E. Moriarty and Risto Mikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1-3):11–32, 1996.

[96] Monroe Newborn. *Kasparov Vs. Deep Blue: Computer Chess Comes of Age*. Springer, 1997.

[97] Alva Noe. *Action in Perception*. MIT Press, 2005.

[98] Stefano Nolfi. Evolutionary robotics: Exploiting the full power of self-organization. *Connection Science*, 10(3-4), 1998.

[99] Stefano Nolfi. Evolving robots able to self-localize in the environment: the importance of viewing cognition as the result of processes occuring at different timescales. *Connection Science*, 14(3):231–244, 2002.

[100] Stefano Nolfi. Power and the limits of reactive agents. *Neurocomputing*, 42(1-4):119–145, 2002.

[101] Stefano Nolfi and Dario Floreano. Coevolving predator and prey robots: Do "arms races" arise in artificial evolution? *Artificial Life*, 4:311–335, 1998.

[102] Stefano Nolfi and Dario Floreano. *Evolutionary robotics*. MIT Press, Cambridge, MA, 2000.

[103] Stefano Nolfi and Domenico Parisi. Evolving non-trivial behaviors on real robots: an autonomous robot that picks up objects. In *Proceedings of the 4th Congress of the Italian Association of Artificial Intelligence*, pages 243–254, 1995.

[104] Andrew Parker. *In the blink of an eye*. Gardner books, 2003.

[105] Gary B. Parker and Matt Parker. Evolving parameters for xpilot combat agents. In *Proceedings of The IEEE Symposium on Computational Intelligence and Games (CIG)*, 2007.

[106] Matt Parker and Gary B. Parker. The evolution of multi-layer neural networks for the control of xpilot agents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[107] Rolf Pfeifer and Josh Bongard. *How the Body Shapes the Way We Think: A New View of Intelligence*. MIT Press, 2006.

[108] Jordan B. Pollack and A. D. Blair. Co-evolution in the successful learning of backgammon strategy. *Machine Learning*, 32:225–240, 1998.

[109] Dean A. Pomerleau. Neural network vision for robot driving. In *The Handbook of Brain Theory and Neural Networks*, 1995.

[110] Steffen Priesterjahn, Kramer, Alexander Weimer, and Andreas Goebels. Evolution of human-competitive agents in modern computer games. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2007.

[111] Steve Rabin. *AI Game Programming Wisdom 3*. Charles River Media, 2006.

[112] Hilary Rose and Steven Rose, editors. *Alas, Poor Darwin: Arguments Against Evolutionary Psychology*. Harmony, 2000.

[113] Christopher Rosin and Richard Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1), 1996.

[114] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Cognition: Explorations in the Microstructure of Cognition*. MIT Press, 1986.

[115] Thomas P. Runarsson and Simon M. Lucas. Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go. *IEEE Transactions on Evolutionary Computation*, 9:628 – 640, 2005.

[116] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM Journal*, 3(3):210–229, 1959.

[117] Nicol N. Schraudolph, Peter Dayan, and Terrence J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing 6*, pages 817–824. Morgan Kaufmann, San Francisco, 1994.

[118] Abdul A. Siddiqi and Simon M. Lucas. A comparison of matrix rewriting versus direct encoding for evolving neural networks. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 392–397, 2006.

[119] B. F. Skinner. *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century-Crofts, 1938.

[120] B. F. Skinner. Teaching machines. *Science*, 128(3330):969–977, 1958.

[121] Pieter Spronck. *Adaptive Game AI*. PhD thesis, University of Maastricht, 2005.

[122] Kenneth Stanley. Practical issues in evolving neural network controllers for video game agents. In *IEEE Symposium on Computational Intelligence and Games Tutorial CD*, 2007.

[123] Kenneth O. Stanley. *Efficient evolution of neural networks through complexification*. PhD thesis, Department of Computer Sciences, University of Texas, Austin, TX, 2004.

[124] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6):653–668, 2005.

[125] Kenneth O. Stanley, Nate Kohl, Rini Sherony, and Risto Miikkulainen. Neuroevolution of an automobile crash warning system. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005)*, 2005.

[126] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[127] Kenneth O. Stanley and Risto Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9:93–130, 2003.

[128] Kenneth O. Stanley and Risto Miikkulainen. Evolving a roving eye for go. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2004.

[129] Kenneth O. Stanley and Risto Mikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Resarch*, 21:63–100, 2004.

[130] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 1998.

[131] Ivan Tanev, M. Joachimczak, H. Hemmi, and K. Shimohara. Evolution of the driving styles of anticipatory agent remotely operating a scaled model of racing car. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005)*, pages 1891–1898, 2005.

[132] Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2006)*, pages 1321 – 1328, 2006.

[133] A. Teller. The evolution of mental models. In *Advances in Genetic Programming*, pages 199–219. MIT Press, 1994.

[134] G. Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[135] G. Tesauro. Comments on "Co-Evolution in the Successful Learning of Backgammon Strategy". *Machine Learning*, 32(3):241–243, 1998.

[136] Julian Togelius. Evolution of the layers in a subsumption architecture robot controller. Master's thesis, University of Sussex, Brighton, 2003.

[137] Julian Togelius. Evolution of a subsumption architecture neurocontroller. *Journal of Intelligent and Fuzzy Systems*, 15:15–20, 2004.

[138] Julian Togelius, Peter Burrow, and Simon M. Lucas. Multi-population competitive co-evolution of car racing controllers. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2007.

[139] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Making racing fun through player modeling and track evolution. In *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 2006.

[140] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Towards automatic personalised content creation in racing games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[141] Julian Togelius, Renzo De Nardi, Hugo Marques, Richard Newcombe, Simon M. Lucas, and Owen Holland. Nonlinear dynamics modelling for controller evolution. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO)*, 2007.

[142] Julian Togelius and Simon M. Lucas. Evolving controllers for simulated car racing. In *Proceedings of the Congress on Evolutionary Computation*, 2005.

[143] Julian Togelius and Simon M. Lucas. Forcing neurocontrollers to exploit sensory symmetry through hard-wired modularity in the game of cellz. In *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games CIG05*, pages 37–43, 2005.

[144] Julian Togelius and Simon M. Lucas. Arms races and car races. In *Proceedings of Parallel Problem Solving from Nature*. Springer, 2006.

[145] Julian Togelius and Simon M. Lucas. Evolving robust and specialized car racing skills. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[146] Edward P. K. Tsang, Sheri Markose, Hakan Er, Abdel Salhi, and Giulia Iori. Eddie in financial decision making. *Journal of Management and Economics*, 2000.

[147] Elio Tuci, Matt Quinn, and Inman Harvey. An evolutionary ecological approach to the study of learning using a robot based model, 2003.

[148] Eric Vaughan. Bilaterally symmetric segmented neural networks for multi-jointed arm articulation. http://www.droidlogic.com/sussex/papers.html, 2003.

[149] Richard A. Watson and Jordan B. Pollack. Coevolutionary dynamics in a minimal substrate. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 702–709, 2001.

[150] Ludwig Wittgenstein. *Philosophical Investigations*. Blackwell, 1953.

[151] Krzysztof Wloch and Peter J. Bentley. Optimising the performance of a formula one car using a genetic algorithm. In *Proceedings of Eighth International Conference on Parallel Problem Solving From Nature*, pages 702–711, 2004.

[152] Georgios N. Yannakakis and John Hallam. Towards capturing and enhancing entertainment in computer games. In *Proceedings of the Hellenic Conference on Artificial Intelligence*, pages 432–442, 2006.

[153] Georgios N. Yannakakis and Manoulis Maragoudakis. Player modeling impact on player's entertainment in computer games. In *User Modeling*, pages 74–78, 2005.

[154] Giorgios N. Yannakakis. *AI in Computer Games: Generating Interesting Interactive Opponents by the use of Evolutionary Computation*. PhD thesis, University of Edinburgh, 2005.

[155] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[156] V. Zykov, J. C. Bongard, and H. Lipson. Evolving dynamic gaits on a physical robot. In *Late Breaking Papers for the Genetic and Evolutionary Computation Conference*, 2004.