Computational Intelligence applied to Computer Games

Benoit Chaperot

A thesis submitted in partial
fulfillment of the requirements of
the University of the West of Scotland
for the degree of Doctor of Philosophy
Supervisor: Prof. Colin Fyfe

June 22, 2010

Abstract

This thesis investigates the use of Computational Intelligence techniques in the context of a computer game. Such techniques include artificial neural networks, genetic algorithms and statistical techniques such as bagging, boosting and the cross entropy method. Our aim is to create an AI which exhibits intelligence; currently, intelligence is often equated with adaptation or learning. In the context of computer games, game players are often frustrated by the lack of adaptation in the computer games. Thus in providing adaptation in a computer game, we are enabling the AI to exhibit more intelligence so that the game player will have enhanced enjoyment from playing the game. The problem we address is to train an artificial intelligence to ride a bike in a motocross game as fast as possible. Thus we are attempting to optimise the parameters of our techniques to achieve good performance at solving this particular problem and we evaluate different techniques to change these parameters and control the bikes.

Thus we experiment with several different types of artificial neural network and show that the multilayered perceptron (MLP) trained by the backpropagation algorithm is the most successful and is able to compete against a good human player. Other architectures of artificial neural networks are very much less successful with this problem. We also show little success with ensemble methods such as bagging and boosting.

We also show that the parameters of the multilayered perceptron can be optimised using genetic algorithms but, although the subsequent MLP's find novel solutions not found by the human player, these solutions tend to be less effective than those found by backpropagation when we take into account comparable training times. With longer training times, the genetic algorithm solutions can be as effective as the backpropagation-trained networks.

Finally we show that the technique of cross entropy is a very good optimiser of these parameters and we also develop a genetic algorithm variant suggested by reinforcement learning.

Acknowledgement

I would like to express my gratitude to my supervisor Prof. Colin Fyfe for his invaluable support and guidance throughout the course of the PhD. Colin was always highly approachable and helpful. In a supervisory capacity I could not have asked for anyone better and am extremely grateful.

I would like to thank my colleagues and friends at UWS and Darkskyne for their support and allowing me to run my experiments on their computers.

I would like to thank my parents and my wife Jing Zhou for supporting me throughout my studies.

In memory of my dad Alain Chaperot.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this introductory chapter we give a brief introduction to the subject of our research. Then we discuss the structure and outline of this thesis. Finally we summarise the main research contributions of this thesis.

1.1 Video Games and Artificial Intelligence

Video games offer a good environment to experiment with data mining and advanced artificial intelligence techniques. Thanks to complex simulated physics (we will use the term "physics simulation" in the remainder of the thesis) and 3D graphics, current video games offer very rich environments with complex problems to solve.

The environments offered by video games are also much safer than real-life environments for experiments in AI. There are normally no serious consequences, like physical damage or injury, if an experiment goes wrong, because the environments used are virtual.

Video games can also strongly benefit from good AI; good AI can enrich the player experience by making him/her feel he/she plays with or against intelligent characters and this can add content to the game.

There is also a need in the research community for common platforms in order to evaluate and compare techniques and algorithms, and to test those techniques and algorithms on the same common problems. This has been discussed during various conferences at which papers ([Chaperot and Fyfe 2005],[Chaperot and Fyfe 2006*b*],[Chaperot and Fyfe 2006*a*] and [Chaperot and Fyfe 2007]) were presented.

1.2 Traditional versus Advanced Artificial Intelligence

There are two kinds of Artificial Intelligence (AI) that can be used in video games:

- **Traditional AI**, also called conventional, logical or symbolic AI. This is the kind of AI traditionally used in the game industry [Rabin 2008][DeLoura et al. 2008][Champandard 2009][GameDevAI 2009][Jimenez 2009]. The game developer fully describes the behaviour of the non-player characters (NPC) with a set of "if, then , else" statements or a finite state machine (FSM), using a programming language. A FSM is a model of behaviour composed of a finite number of states, transitions between those states, and actions. The main advantage of such techniques is that one can look inside the technique and determine on what basis a decision has been made. The main disadvantage is that the AI programmer tends to have to identify all situations which are likely to arise in use and program some response to such situations in advance. This has led to the development of techniques which can adapt while in use.
- **Computational Intelligence (CI)**, also called Advanced AI, Distributed AI or non-symbolic AI. The game developer does not directly describe the behaviour of NPC's; instead he creates structures and algorithms that can then learn, adapt or evolve from examples or experience. This kind of AI encompasses techniques like
  - Genetic Algorithms (GA): these techniques are based on the view of evolution as a problem solver i.e. all living things on earth today are here because their ancestors solved the twin problems of how to live to maturity and how to find a mate (for sexual species).
  - Artificial Neural Networks (ANN) : these techniques try to emulate the success of human brains in making sense of the environment in which we live. There are a great many different architectures of artificial neural networks but the most commonly used is the multilayered perceptron which we use extensively throughout this thesis.
  - Artificial Immune Systems: provide another means of adaptation or learning based on the fact that the immune system adapts to antigens by providing antibodies which attack optimally the antigens.

- Probabilistic modelling: different from the above techniques in that it is not based on biological models, these techniques are often known as machine learning. However many researchers use both these techniques and some of the above techniques to tackle a single problem.
- Fuzzy Logic/sets: Crisp sets are binary objects: each element is either within or outwith a set but with fuzzy sets we allow partial membership of a set. Fuzzy logic manipulates this partial membership in order to provide solutions to human-type problems.

These examples of Computational Intelligence are not exhaustive since new techniques such as Swarm Intelligence are being created continuously and are being added to the set of techniques known collectively as Computational Intelligence.

1.3 Structure of the thesis

Chapter 2 reviews some of the existing computer games in terms of:

- their relation to artificial intelligence.
- the physics model which they use.

We consider that these are two major aspects of computer games and we introduce our own computer game, **Motocross The Force**, in order to compare it with these existing games.

Chapter 3 reviews different techniques of computational intelligence. We begin with the multilayered perceptron, an artificial neural network which is trained by supervised learning; in particular, we review the backpropagation algorithm. We also review other architectures for artificial neural networks, specifically Kohonen's Self-Organizing Map, radial basis functions and the topographic product of experts. We also review the genetic algorithm and discuss how it may be used to optimise parameters in the multilayered perceptron. We also discuss the ensemble methods of bagging and boosting. Both have been praised within the statistics community for their almost magical qualities in that they seem to leverage the use of the data and extract more information from a data set than other methods can.

Chapter 4 gives experimental results for each of these techniques in the context of our computer game and shows that the multilayered perceptron

can be trained using backpropagation to be almost as good in terms of riding efficiency as a good human player. The genetic algorithm on the other hand requires much more processing time to achieve this but it also finds solutions to the problem of riding the bike which are very different from the solutions found by a human rider. Our results with the alternative architectures of neural networks and with the ensemble methods were less impressive and we discuss the reasons for this. On the other hand, the technique of cross entropy optimisation did produce some very good results in this context.

Chapter 5 responds to the need for a common platform on which to experiment with AI methods in computer games. There are for example, SDK's such as "Unreal" which is a first person shooter game and allows comparison of techniques but one problem with this is that the developer can only use scripts (rather than more native programming languages such as C++) which limits the amount of flexibility in the development. Also we are specifically interested in creating an SDK pertaining to bike riding. **Motocross The Force** is re-designed to split the AI functionality from the game functionality; this allows any researcher to implement his/her own method for augmenting intelligence within the AI and then simply plug-in the new .dll into the system and generate comparative results.

Finally in Chapter 6 we introduce an improved evolutionary algorithm which is more robust and stable than the standard genetic algorithm and provides the best solution in the thesis.

We conclude with a review of the thesis and suggest future directions for research.

1.4 Contribution of the research

This thesis introduces a new computer game into the research community. While the emphasis of the thesis is on the computational intelligence methods used in the research, we must also point out that the game developed, **Motocross The Force**, is comparable with similar industry-created games in terms of e.g. its graphics or physics simulation. This game has now been re-developed for the research community as a potential test-bed for novel AI techniques. One of the main contributions of this thesis is to present a new SDK which the computer game AI community can use to compare different methods of controlling the bikes.

The overarching aim of the research undertaken in this project is to optimise a set of parameters which control the bike using computational in-

telligence techniques. The architecture of the controlling machine is mostly the multilayered perceptron (see Chapter 3), therefore the problem is to optimise the weights of the multilayered perceptron in order to have the bikes raced around the tracks as fast as possible. To that end, we investigate the use of a variety of techniques (see below) with which we adapt the parameters (weights) of the multilayered perceptron in order to optimise the speed. We also investigate alternative architectures (i.e. not the multilayered perceptron) with the same aim of training these machines to race as fast as possible.

Other contributions of the thesis are based on a comparative study of a number of different techniques already in the computational intelligence literature but not previously applied to this type of game. Thus in Chapter 4, we compare:

Backpropagation training of Artificial Neural Networks This standard supervised training method is shown to be trainable to a level close to that attained by a human expert.

Genetic Algorithms These techniques are also used to train (optimise the parameters of) artificial neural networks. Although this method does not quite match the efficiency of the backpropagation technique, it does find novel ways to ride the bike unlike the backpropagation technique which is simply trained to emulate the human rider.

Ensemble methods which combine multiple classifiers or regressors. We introduce a novel parameterisation which enables us to perform boosting or anti-boosting by changing a single parameter. These techniques were ineffective, however, in the context of the motocross game.

Alternative architectures We have also investigated alternative architectures of artificial neural networks and shown that these techniques do not perform quite as well as the multilayered perceptron.

Cross Entropy Method We have investigated the use of cross-entropy to optimise the parameters of the AI and shown that this technique is working well.

New evolution In Chapter 6 we have developed a new optimisation technique which improves upon the standard genetic algorithm by using a technique inspired by the reinforcement learning method of ϵ-greedy policies.

We wish to highlight the fact that any external researcher can now create his/her own AI and compare results with those in this thesis. This is perhaps one of the most lasting outcomes from this thesis.

Chapter 2

Review of Computer Games

In this thesis we experiment with existing and innovative Artificial Intelligence techniques to control motorbikes in a motocross game. Most video games feature modelled opponents or non-player characters making use of some form of Artificial Intelligence. We are more particularly interested in racing games or games making use of continuous analogous inputs, turning left and right, accelerating or decelerating, moving an object up or down.

There are two major aspects which we wish to highlight in this section:

1. Video games and artificial intelligence.

2. Video games and physics simulation.

2.1 Video Games and Artificial Intelligence

In this section, we review AI techniques in games and programs relevant to this research.

2.1.1 Games not making use of Advanced Artificial Intelligence

The vast majority of video games nowadays do not use advanced AI techniques, but use traditional AI techniques.

MX vs ATV Unleashed

Figure 2.1: Screenshot taken from the game MX vs ATV Unleashed.

This game is one of the best motocross games available for PC and consoles, with a variety of different environments, vehicles and challenges to choose from [MXVSATV 2005]. This game features traditional AI. The motocross track is split into lanes like on a motorway and each computer controlled bike stays in its own lane in order to avoid collision with other bikes. This behaviour is neither the most natural nor the most efficient way to race along a motocross track. The most natural and efficient way to race along a motocross track is normally to take the shortest path, which may involve taking the inside in bends, while attempting to avoid collision with other bikes.

There are also markers along the track, placed by the track designers, to tell the computer controlled bikes what must be the target velocities on portions of the track, when to accelerate and when to decelerate, and what

are the appropriate portions of the track to restart on after a crash. All bikes also have exactly the same behaviour.

Figure 2.1 shows an example screenshot from this game.

Flatout Ultimate Carnage

Figure 2.2: Screenshot taken from the game Flatout Ultimate Carnage.

This game is one of the best racing games available for the XBOX 360 [FlatOut 2007]. This game also features traditional AI but the AI seems more sophisticated than in the game **MX vs ATV Unleashed**.

In this game there are two or three racing lanes along the track, and non-player character (NPC) cars switch from one lane to another lane while attempting to overtake each other, if pushed by another car or following a collision. In this game there are many objects with which the cars can collide and there are also many jumps which make the driving unpredictable. The track also often splits into two or three alternative paths, with one path being generally faster according to the vehicle used. There are also markers along the track, placed by the track designers, to tell the NPC cars what must be the target velocities, when to accelerate and decelerate, what are the appropriate portions of the track to restart on after a crash. NPC's

have various behaviours, partly because the vehicles driven have different characteristics like weight, power, traction and maximum speed.

NPC's do not attempt to avoid accidents, as opposed to more simulation oriented games like **Project Gotham Racing 3**; accidents are part of the gameplay and part of the fun to play the game.

Figure 2.2 shows an example screenshot from this game.

Conclusion

The advantage of such a simplistic AI is that it is totally predictable; it is not likely that one non player character (NPC) has an undesirable behaviour once in a while. For example, one NPC would not decide to have an accident, once in a while, with another character; one NPC would not decide to stop racing, stop on the side of the track or go off the track for some unknown reason.

The disadvantage is also that it is totally predictable; after a few laps it is possible for the player to predict exactly what the NPC's are going to do, and it is possible for the player to win a race simply by adopting a natural behaviour, like taking the inside in bends. The NPC's also do not adapt or learn from experience; their behaviour is constant.

Some techniques are used to make the game slightly more exciting, like increasing or decreasing the performance of one or two NPC's at some times during a race, according to the human player performance; these techniques add an artificial twist to the game but the excitement gained can not compare for some players with the excitement of competing for real against intelligent, continually adapting characters.

2.1.2 Games making use of Advanced Artificial Intelligence

Forza Motorsport

Figure 2.3: Screenshot taken from the game Forza Motorsport.

Forza Motorsport is a car racing video game for the XBOX and has been released in May 2005 [Forza 2005]. Its AI system was the result of a unique collaboration between Microsoft Games Studios, Redmond, USA, and the Machine Learning and Perception group at Microsoft Research, Cambridge, UK [Forza2 2005].

One research objective was to introduce a new "fun" feature to the game based on exploiting "machine learning" techniques. This led to the development of the "Drivatar" concept, wherein the game can learn a probabilistic

model of the player's style of driving. The model is embodied in an AI avatar which can mimic and reproduce the player's individual driving characteristics independent of any particular track or car. The player can then employ this avatar to race for them in certain modes in the game, and can also set up their own customised races against arbitrary "Drivatars": perhaps against their own (for the ultimate personalised challenge) or those of friends or even celebrities.

This is a commercial program and the source code is not available. It is not possible to find a lot of literature about the exact techniques and mechanisms used in the game. It is believed the game uses multi-layered perceptron ANN's (see Section 3.1) to control the computer controlled cars, and the back propagation algorithm to train these artificial intelligences from the recording of human players playing the game.

A screenshot is shown in Figure 2.3.

Simplerace

Julian Togelius and Simon M. Lucas have used various small games, to experiment with artificial intelligence and evolution techniques [Togelius 2007]. The games involved racing a car along a track, controlling an helicopter or hunting for food (**Cellz**). All these games took place in a physics-based environment simulated in discrete time, where the goal involved reaching certain points in space within a specified time. The commands returned from the controllers in **Cellz** are continuous whereas the commands returned from the controllers in the other games are discrete.

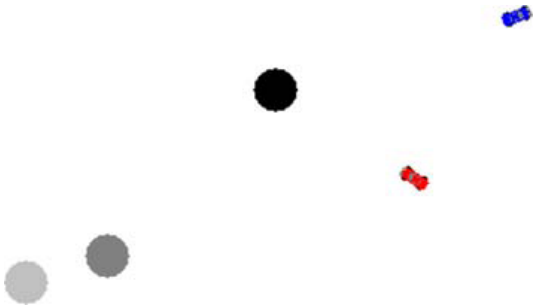

Figure 2.4: Screenshot taken from the game Simplerace.

A car racing competition was arranged as part of the 2007 IEEE Congress on Evolutionary Computation using the **Simplerace** game [Togelius et al. 2008]. **Simplerace** is a tactical racing game for one or two

players (each of which can be a program or human controlling the car via the keyboard), based on a two-dimensional physics model, where the objective is to drive a car so as to pass as many randomised way points as possible. The full Java source code for the game is available freely online to allow researchers to use the code for benchmarking their own algorithms.

In this simulation, a car is simulated as a 20 x 10 pixel rectangle, operating in a rectangular arena of size 400 x 300 or 400 x 400 pixels. The car's complete state is specified by its position $\mathbf{s}$, velocity $\mathbf{v}$, 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 very simple equations of motion.

Represented among the entries to this competition were a wide variety of computational intelligence methods, including genetic algorithms, evolution strategies, neural networks, genetic programming, temporal difference learning, fuzzy logic and force field control. Several of the contributions featured novel uses and combinations of these techniques. Succeeding at the competition task required being able to control an agent in a dynamic environment, but excelling in it required a certain degree of tactics.

A screenshot is shown in Figure 2.4. The interface between the game and the controllers has similarities with the interface in the AI SDK detailed in Chapter 5.

TORCS

Another car racing competition was arranged as part of the IEEE WCCI 2008 conference [Loiacono et al. 2008] using the Open Racing Car Simulator (**TORCS**) [Torcs 2009].

The **TORCS** game features:

1. 42 different cars, 30 tracks, and more than 50 computer controlled opponents to race against.

2. Good 3D graphics with lighting, smoke, skidmarks and glowing brake disks.

3. Good physics simulation with damage model, collisions, tire and wheel properties (springs, dampers, stiffness, ...), aerodynamics (ground effect, spoilers, ...).

Figure 2.5: Screenshot taken from the game TORCS.

4. Different types of races from simple practice session up to the championship.

The user can also develop his/her own computer-controlled driver (also called a robot) in C or C++. The interface between the game and the controllers has similarities with the interface in the AI SDK detailed in Chapter 5.

Five teams took part in the competition and four out of the five participating teams described the architecture and training of their controllers in [Loiacono et al. 2008]. Some of these controllers were CI-based and involved learning while other controllers were non-CI-based and were therefore non-learning. In this edition of the competition, all of the submitted controllers were outperformed by the non-CI, non-learning best controllers that come with the game. It was suggested that the controllers developed by the **TORCS** developers had access to more state information not directly

available through the competition API.

A screenshot is shown in Figure 2.5.

Smart Sweepers

One other interesting program is **Smart Sweepers** by Mat Buckland [Buckland 2005*a*][Buckland 2004][Buckland 2002]. In this simple program, virtual minesweepers are trained to find and collect land-mines scattered about a very simple 2D world. This program is not really a video game, but it makes use of ANN's to control vehicles.

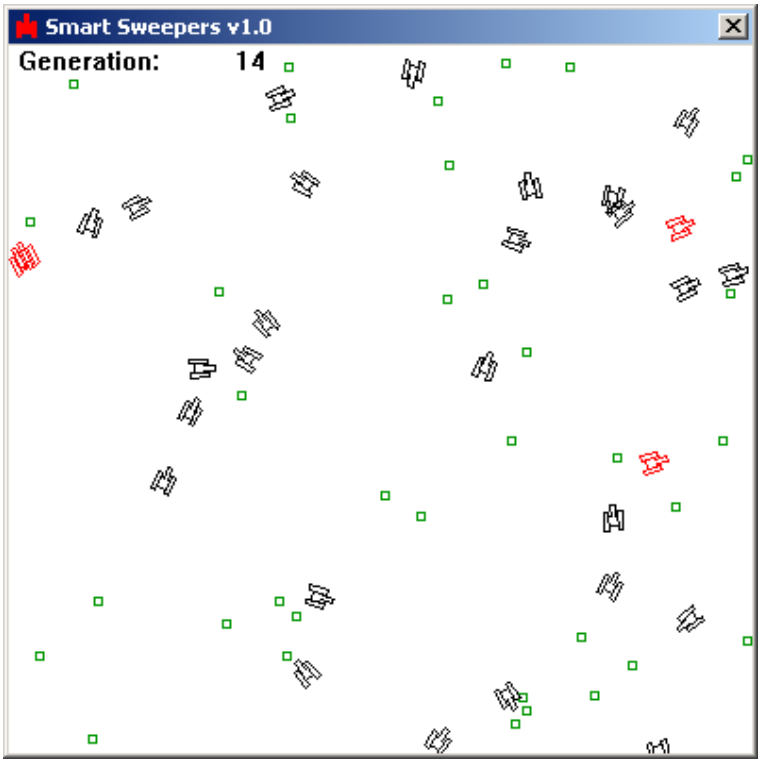

Figure 2.6: Screenshot taken from the program Smart Sweepers.

The display is very simple (Figure 2.6). The minesweepers are represented by the objects that look like tanks and the land-mines are represented by the green dots. Whenever a minesweeper finds a mine it is removed and another mine is randomly positioned somewhere else in the world, thereby ensuring there is always a constant amount of land-mines on display. The minesweepers drawn in red are the best performing minesweepers the program has evolved so far.

The minesweepers are tanks and are controlled by adjusting the speed of the left and right tracks. By applying various forces to the left and right sides of minesweepers it is possible to give them a full range of movements. The networks have two outputs, one to designate the speed of the left track, and the other to designate the speed of the right track.

The networks are trained using a genetic algorithm.

This program is very interesting because the source code is available and the techniques used are very well explained in detail. This program with its documentation [Buckland 2005*b*] have been used as a starting point for the advanced AI in the game **Motocross The Force**.

Pong

[McGlinchey 2003] uses the rather old game of Pong (a type of tennis game) in order to investigate whether neural networks can be used to play the game in the same way that humans play this game. Some training data was created by recording human players playing the game, then the training data was used to train artificial intelligence based on Kohonen's Self Organising Map. The system was successful in playing in a similar style to that of the original player, and also played with a similar level of skill.

This was extended using the Generative Topographic Mapping [Bishop et al. 1997] in [Leen and Fyfe 2005] and a comparative study was performed with the results of [McGlinchey 2003]. A further comparison using the Topographic Product of Experts (see Chapter 3) was performed in [Fyfe 2005]. All three of these mappings - SOM, GTM and ToPoE - are so-called topographic mappings in that they attempt to maintain close relations on the manifold found between closely related data points.

Conclusion

The advantage of Advanced AI is that it is not predictable. The AI is able to evolve and adapt to new situations. This can make the game more interesting to the human player by having intelligent characters in the game evolving and adapting to the game at the same time as the human player.

The disadvantage of Advanced AI is also that it is not predictable. From the game developer point of view, the Advanced AI can be seen as a black box; it is not possible for the game developer to be sure that undesirable behaviours won't be produced once in a while; and when undesirable

behaviours are produced, it is not easy for the game developer to understand it and to fix it.

Advanced AI is more time consuming to develop and less reliable than traditional AI; these are the reasons Advanced AI is not the kind of AI traditionally used in the game industry.

2.2 Video Games and Physics Simulation

Video games can offer rich environments to experiment with artificial intelligence techniques. One way to increase the richness of the environment is to use good Physics Simulation.

With Physics Simulation, entities in the game, like characters and vehicles, can move and act realistically just as they do in the real world; this can contribute to the player's immersion into the game. If an entity in a game is moving in a non-realistic manner, this breaks the player's immersion and decreases satisfaction with the game.

The entities' behaviours can be complex and unpredictable, the vehicles can be very fast and hard to control; vehicles can skid in turns and do jumps; these complex and rich behaviours take an important part in the "fun" to play a game and improve the player's gaming experience.

The complex behaviours can also create complex problems to solve for artificial intelligence techniques. In this section we review simulation techniques in games and programs relevant to this research.

2.2.1 Rigid Body Simulation

Rigid body simulation, also known as physics simulation, is a method for simulating mechanical systems. It is generally present as a piece of software (library), used as part of another piece of software (in this case a video game). Physics libraries can be commercial middleware like Havock Physics [Havok 2005] and Nvidia PhysX [PhysX 2005], or open source like the Open Dynamics Engine (ODE) [Ode 2006]. The binary Nvidia PhysX is now totally free and available for immediate download by developers with much better performance, stability and functionalities than ODE. PhysX supports hardware acceleration using PhysX Processing Units (PPU) and now modern Nvidia Graphics Processing Units (GPU); Modern GPU's have a highly parallel structure that makes them more effective than general-purpose CPU's

for a range of complex applications such as physics simulation; the simulation can run faster. Running the simulation on the GPU also gives the benefit to offload calculations from the CPU, allowing it to perform other tasks instead, potentially resulting in a smoother gaming experience.

2.2.2 Games not making use of Rigid Body Simulation

Many racing games do not use or do not fully use Rigid Body Simulation.

Motocross Madness 2

Motocross Madness 2 is a motocross game, released in year 2000; it was very popular and still is a fun game to play.

The game [MCM2 2000] does not really make use of rigid body simulation; instead the game makes use of simple physics rules; the bike and biker can be seen as one entity or one rigid body, a point evolving on a terrain; this entity can jump, skid, absorb shocks from the terrain. The turns can be very sharp and not realistic; shocks are absorbed by the entity until a threshold after which the bike crashes.

The character animations are predefined, i.e. all the characters moves come from motion capture or have been generated by a human animator. When the character turns, he stretches his leg inside the turn like real riders do; when the character is in the air, he can perform a stunt like stretching both arms and legs; when the bike crashes, the bike and bikers become separate entities, the character slides on the terrain and animates unrealistically and irrespective of the obstacles he meets on the terrain.

A closer look at the wheels allows the user to see that the wheels do not really touch the ground, the wheels slide on the ground when the bike turns. The game does not qualify as a satisfactory rigid body simulation because there is really only one body used for both bike and character when the bike is in control and one body per bike and per character when the bike is out of control and crashes. To that extent, the simulation is too rigid body. For the game to qualify as a modern rigid body simulation, each part of the character body and bike would need to be represented using separate rigid bodies.

MX vs ATV Unleashed

This game is one of the best motocross games available for PC and consoles, with a variety of different environments, vehicles and challenges to choose from, as described in the previous section. It has been developed by Rainbow Studios, the developers of **Motocross Madness 2**.

Body parts of the character and parts of bike are represented using separate rigid bodies; the user can see the bike and character animate while evolving along a track. A closer look at the wheels allows the user to see that the wheels do not really touch the ground, the wheels slide on the ground when the bike turns. The control model is in fact very similar to the one in the previous game **Motocross Madness 2**.

Rigid body simulation is used for aesthetic purposes and has little effect on the control of the bike and the gameplay. The game has been released on platforms with much more processing power than was available when the previous game, **Motocross Madness 2** was released. The extra processing power was used for improving the graphics rather than the physics simulation.

Gran Tourismo 5 Prologue

This Playstation 3 game [GT5P 2008] showcases the automotive experience that is imminent with **Gran Turismo 5**.

The game looks very good, and makes good use of the graphical capabilities of the Playstation 3.

There are many parameters the user can set, especially in terms of the simulation model and driving aids.

A closer look at the game allows the user to see that the game does not really use physics simulation. If the user drives a car at full speed head on into another car, the two cars stop unrealistically and seem unable to flip onto a side. Similarly, if the user attempts to perform a doughnut (manoeuvre where the car is rotating around a set of wheels in a continuous motion), the skid marks left on the road seem to be perfectly circular, even if the road is not flat.

The physics engine is a very complex physics model representing the physics of cars; this complex model takes many parameters into consideration (e.g. the characteristics of cars, whether driving aids are on or off) and gives the user the illusion of driving a real car. In extreme conditions (accidents, skidding) this complex physics model may prove to be impractical,

not realistic, and break.

The physics engine allows the user to easily experience many aspects of driving, and have the impression of driving well, at the expense of accuracy and realism of the simulation. This game is more an arcade game, and less a simulator game, when compared to other games like **Project Gotham Racing 3**.

The full game (**Gran Tourismo 5**) is due in 2010. In [Remo 2009], the producer of the series, Kazunori Yamauchi, explains that an entirely new game engine has been developed for **Gran Tourismo 5**. Hence **Gran Tourismo 5 Prologue** was using the legacy game engine from **Gran Tourismo 4**, at the time available on Playstation 2, with a lot less processing power than nowadays consoles and PC; this explains the limited physics simulation featured in **Gran Tourismo 5 Prologue**.

2.2.3 Games making use of Rigid Body Simulation

Many racing games make full use of Physics Simulation. These games include **Project Gotham Racing 3**, **Flatout Ultimate Carnage** and **Forza Motorsport**. In these games, the wheels and the chassis are represented using separate rigid bodies or entities; the wheels are attached to the chassis using simulated suspensions, and the tyres touch the ground using simulated tyre friction.

In racing games making full use of physics simulation, all the car behaviours are generated implicitly by the simulation and not explicitly and deterministically by the program. For example, when the player commands the vehicle to turn, no part of the program explicitly turns the vehicle as a result of the turn command; instead the program explicitly turns the direction of the front wheels and, because of physics simulation and simulated friction between the wheels and the ground, the vehicle turns.

2.3 Motocross The Force

Figure 2.7: Screenshot taken from the game Motocross The Force.

Motocross The Force is a motocross game featuring terrain rendering and rigid body simulation applied to bikes and characters [Chaperot et al. 2009]. The game has been developed in conjunction with Eric Breistroffer (2D and 3D artist), and Thibault Saint Olive (track designer). The game has been released and is now available for download [Chaperot 2009]. A screenshot is shown in Figure 2.7.

There are various interesting aspects in using artificial neural network methods in a motocross game. Because the design of an ANN is motivated by analogy with the brain, the rationale for their use in the current context is that entities controlled by ANN are expected to behave in a human or animal manner, and these behaviours can add some life and content to the game. The human player has also the possibility to create new tracks. ANN's have the capability to perform well and extrapolate when presented with new and

different sets of inputs from the sets that were used to train them; hence an ANN trained to ride a motorbike on a track should be able to ride the same motorbike on another similar track. ANN's have the capability to train and evolve their behaviours. ANN's may be able to perform with good lap times on any given track while still retaining elements of human behaviour.

No gameplay mechanism is enforced in the game; the player can decide not to race and just ride the bike and explore the environment. The game features a clock and the player can also decide to race along the track, competing against the clock or against NPC's.

The controls for the human player are the same as the controls for the NPC's. The two basic controls are:

- accelerate and brake.
- turn left and right.

The game has evolved with time; two optional controls have been added to the basic controls; these controls are especially useful for controlling the bike when it is in the air before landing:

- rotation of the bike forward and backward.
- rotation of the bike left and right.

The controls can thus be summarised by four floating point values in the range {-1 to 1}; if a character decides not to use the optional controls, it can leave them to 0 (default value).

There is one marker known as a waypoint which marks the position and orientation of the centre of the track, every metre along the track. These waypoints are used to ensure bikes follow the track and we will talk about positions in waypoint space when giving positions with respect to the waypoints. For example, for the evolutionary algorithms, the score is calculated as follows:

- **vPassWayPointBonus** is a bonus for passing through a waypoint.
- **vMissedWayPointBonus** is a bonus/penalty (i.e. normally negative) for missing a waypoint.
- **vCrashBonus** is a bonus/penalty (i.e. normally negative) for crashing the bike.

- **vFinalDistFromWayPointBonusMultiplier** is a bonus/penalty (i.e. normally negative) for every metre away from the centre of the next waypoint.

All the experiments with AI techniques are done in the context of this motocross game. During the experiments, many parameters are found empirically; the parameters found may not always be the most optimal but together prove to produce good results. Many parameters given in this thesis are given for indication purposes and to help the reader in reproducing the experiments.

Body parts of the character and parts of bike are represented using separate rigid bodies. One big difference between this game and other games like **MX vs ATV Unleashed** is that in this game rigid body simulation is not only used for aesthetic purposes but is used for simulating the bike and takes part in the gameplay. As opposed to other motorbike games, the two wheels touch the ground and the bike and rider fully respond to bumps on the track.

Some driving aids have been added to allow human players and AI to more easily play the game; for example, forces are added to the bike to help the bike maintain its balance. Even with the driving aids the game is more difficult to play than most other motocross games. The game does not really qualify as a simulator because the simulation is not very realistic. The game makes use of physics simulation; the physics simulation creates rich bike behaviours that enrich and improve the player's gaming experience; it was assumed that it was more enjoyable to play a game that is not totally predictable, where it is not possible to take exactly the same lanes and do the same jumps, like in real life, rather than a game where it is possible to do exactly the same things lap after lap.

The bike physics is illustrated in Figure 2.8.

Figure 2.8: Bike Physics: The bike and biker as seen in the game and their associated collision and dynamic objects used in the simulation.

2.4 Conclusion

The processing power available on game platforms is always increasing, with the processing power nearly always dedicated to displaying nice graphics and smooth animations. Graphics and animations are what the consumers see first, and base their buying decisions on.

Other characteristics of the game, like good gameplay, good physics simulation and AI, are more difficult to evaluate in a short amount of time, and are generally allocated less resources by the game developers.

Very simple simulation and simple simulated models can sometimes be a choice. In arcade games, fun and gameplay are preferred to simulation and realism; this allows children to play a game like a car racing game that would otherwise only be playable by adults if the simulation was perfectly accurate.

Arcade games also have the advantage of requiring less processing power than simulators, so the same game can run and the gameplay be the same on platforms with very different technical capabilities.

Video games and physics simulation offer complex problems to solve for artificial intelligence techniques; this becomes even more true as the complexity of the physics models increases as does the simulated environment featured in the games.

All games reviewed in this chapter offer complex problems to solve for artificial intelligence techniques. The game **Motocross The Force** is

not any better than any of the other games; it has been chosen for testing Computational Intelligence techniques in this thesis mainly because the game has been developed by the student, the full source code was available and there was no limit for developing and testing Computational Intelligence techniques for the game.

Chapter 3

Literature Review

3.1 Artificial Neural Networks

Artificial Neural Networks [Bishop 1995, Hertz et al. 1992, Hecht-Nielsen 1991, Haykin 1994, Arbib 1995] are a form of Advanced AI or Computational Intelligence. The game developer does not directly describe the behaviour of NPC's; instead he creates structures and algorithms that can then learn, adapt or evolve from examples or experience. More generally, ANN's are general non-parametric techniques where the parameters inside the networks are given values according to information derived from the data.

3.1.1 Multilayered Perceptrons

Artificial neural networks are usually software simulations which are models at some level of real brains. Multilayered perceptrons (MLP) networks are the first type of neural networks investigated in this thesis. Other types of neural networks [Fyfe 2005] may be equally useful for the task of controlling computer bikes in the motocross game.

Models from this type of ANN are made of layers of neurons. Activity in the network is propagated forwards via weights from the input layer, $\mathbf{x}$, to the hidden layer where some function of the net activation is calculated. Then the activity is propagated via more weights to the output layer, $\mathbf{y}$, where some function of the net activation may also be calculated.

The neurons in the input layer are passive in that they merely hold the activation corresponding to the information to which the network must

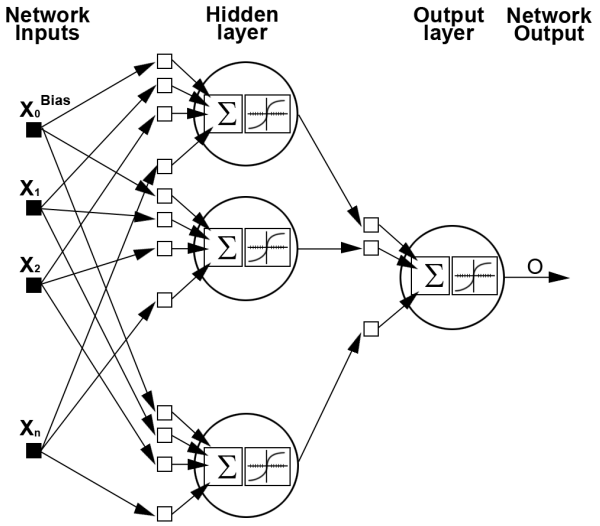

Figure 3.1: A Multilayered Perceptron Network.

respond. In our case this will be local information about the terrain which the artificial rider is currently meeting.

The neurons in the hidden layer are so-called as they cannot directly communicate in any way with the external environment. They only communicate with neurons in the input layer and in the output layer.

The neurons in the output layer communicate the response of the network to the external environment. In our case the response will be the actions which are required to ride the artificial motorbike, corresponding to the appropriate response to the local information about the terrain which the artificial rider is currently meeting.

Each neuron is a simple processing unit, with a number of inputs; one input can be a bias which is a non-zero input which allows the neuron to produce an output even if the sum of all other inputs is equal to zero. The activation function is typically a sigmoid function and can be a logistic or a tanh() function. Both of these functions satisfy the basic criterion that they are differentiable. The differentiable property is important so that an algorithm such as the backpropagation algorithm can be used to train the network. In addition they are both monotonic and have the important property that their rate of change is greatest at intermediate values and least at extreme values. This makes it possible to saturate a neuron's output at one or other of their extreme values.

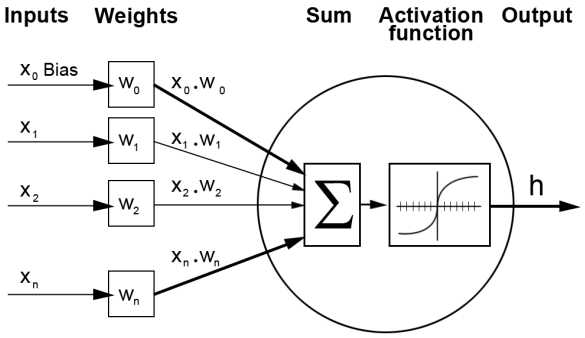

Figure 3.2: A neuron which corresponds to a single unit of a multilayered perceptron network.

An MLP network and a neuron are illustrated in Figures 3.1 and 3.2.

The final point worth noting is the ease with which their derivative can be calculated:

- if $f(x) = \tanh bx$, then $f'(a) = b(1 - f(a) * f(a))$
- if $f(x) = \frac{1}{1+\exp(-bx)}$, then $f'(a) = bf(a)(1 - f(a))$

Activation is passed from inputs to hidden neurons through a set of weights, W. At the hidden neurons, a non-linear activation function is calculated, e.g. $m \tanh \frac{act}{r}$ where r is a parameter which controls the slope of the sigmoid function. Let us have $N+1$ input neurons [1], H hidden neurons, and O output neurons. Then the calculation at the hidden neurons is:

$$act_i = \sum_{j=0}^{N} W_{ij} x_j, \forall i \in 1, ..., H \tag{3.1}$$

$$h_i = \tanh \frac{act}{r} \tag{3.2}$$

where x_0 is the bias, a constant non-zero input,and h_i is the firing of the i^{th} hidden neuron. This is then transmitted to the output neurons through a second set of weights, V, so that:

$$act_i = \sum_{j=0}^{H} V_{ij} h_j, \forall i \in 1, ..., O \tag{3.3}$$

$$o_i = act_i \tag{3.4}$$

[1] The additional 1 corresponds to the bias term.

where h_0 is the bias, a constant non-zero input, h_i is the firing of the i^{th} hidden neuron and o_i is the i^{th} output from the network.

Thus activation is passed from inputs to outputs. The whole machine tries to learn an appropriate mapping so that some function is being optimally performed. Such networks use supervised learning to change the parameters, W and V i.e. we must have a training data set which features inputs and the corresponding desired outputs or correct answers. One common supervised learning technique is the backpropagation algorithm (see later).

A non-linear activation function could also be calculated in the output layer; the network could also have more than one hidden layer though it can be shown theoretically that the single hidden layer is enough to model any function with only a countable number of discontinuities. Given that we wish a continuous response in the main from our network with only a small number of discontinuities (e.g. when the decision to jump over 2 hills rather than 1 hill is taken), we can easily make the decision that a single layer of hidden neurons is sufficient for our purposes.

3.1.2 The Backpropagation Algorithm

Let the P^{th} input pattern be $\mathbf{x}^P$, which after passing through the network evokes a response $\mathbf{o}^P$ at the output neurons. Let the target value associated with input pattern $\mathbf{x}^P$ be $\mathbf{t}^P$. Then the error at the i^{th} output is $E_i^P = t_i^P - o_i^P$ which is then propagated backwards (hence the name) to determine what proportion of this error is associated with each hidden neuron. The algorithm is:

1. Initialise the weights to small random numbers.
2. Choose an input pattern, $\mathbf{x}^P$, and apply it to the input layer.
3. Propagate the activation forward through the weights till the activation reaches the output neurons.
4. Calculate the δ's for the output layer $\delta_i^P = (t_i^P - o_i^P) f'(Act_i^P)$ using the desired target values for the selected input pattern.
5. Update the hidden to output layer weights with $\Delta_P w_{ij} = \gamma . \delta_i^P . o_j^P$, where o_j^P is the output of the j^{th} hidden neuron to the P^{th} pattern.
6. Calculate the δ's for the hidden layer(s) using $\delta_i^P = \sum_{j=1}^{O} \delta_j^P w_{ij} . f'(Act_i^P)$.

7. Update the input to hidden weights in the network according to $\Delta_P w_{ij} = \gamma.\delta_i^P.o_j^P$ where o_j^P is the value of the j^{th} input neuron in the P^{th} pattern.

8. Repeat steps 2 to 6 for all patterns.

with $\mathbf{w}_{ij}$ the j^{th} weight of the i^{th} neuron in a layer, and $\mathbf{o}_j^P$ the j^{th} input to the neuron (i.e. j^{th} input to the network if the neuron is in the first hidden layer, output of the j^{th} neuron in the previous layer otherwise).

3.1.3 Kohonen's Self-Organizing Map

We mentioned in Chapter 2 that McGlinchey used a self-organising map in the game of Pong. This motivates us to try the same architecture on the motocross game.

Kohonen's Self-Organizing Map (SOM) [Kohonen 1995] is one of the most popular ANN's. It is a topology preserving mapping technique, based on a form of unsupervised learning, known as competitive learning. The SOM was introduced as a data quantisation method but has found at least as much use as a visualisation tool.

A topographic mapping (or topology preserving mapping) is a transformation which captures some structure in the data so that points which are mapped close to one another share some common feature while points which are mapped far from one another do not share this feature. The data space is generally high dimensional and the feature space is generally two-dimensional.

Unsupervised learning means that no human intervention is needed during the learning process; the network can create mappings from data space to feature space and find interesting features of the data without any human labelling or mapping example.

Competitive learning, also used for vector quantisation, is a process where nodes in feature space (neurons in the output layer) are competing over taking responsibility for data samples, and only one node (the winning neuron) takes full responsibility for a data sample. The neurons are arranged in neuron space with some structure e.g. typically if we imagine a one dimensional neuron space, the neurons are equally placed in a line or if we imagine a two dimensional neuron space, the neurons may be at the corners of a regular grid. During the learning process, not only are the weights into the winning neuron updated but also the weights into its neighbours where such

neighbours are identified in neuron space *not* data space. Kohonen defined a neighbourhood function $f(i, i*)$ of the winning neuron $i*$. The neighbourhood function is a function of the distance between i and $i*$ in neuron space. A typical function is the Difference of Gaussians function which, because of its shape, is also known as the Mexican hat function; thus if unit i is at point r_i in the output layer then:

$$f(i, i*) = a \exp(\frac{- \parallel r_i - r_{i*} \parallel^2}{2\sigma_a^2}) - b \exp(\frac{- \parallel r_i - r_{i*} \parallel^2}{2\sigma_b^2}) \tag{3.5}$$

With $a > b$ and $\sigma_a < \sigma_b$.

Using this function (3.5), neurons which are close to the winning neuron in the output layer are also dragged towards the input data while those neurons further away are pushed slightly in the opposite direction.

The algorithm is:

1. Select at random an input data sample.

2. There is a competition among the output neurons. That neuron whose weights are closest to the input data sample wins the competition:

$$\text{winning neuron}, i* = \arg\min(||x - w_i||) \tag{3.6}$$

   Where $||x - w_i||$ represents the Euclidean distance between the weights of a neuron i and the input data sample. Other distance functions can also be used.

3. Now update all neurons' weights using:

$$w_{ij} = w_{ij} + \Delta w_{ij} \tag{3.7}$$
$$\Delta w_{ij} = \alpha(x_j - w_{ij}) * f(i, i*) \tag{3.8}$$

   Where:

$$f(i, i*) = a \exp(\frac{- \parallel r_i - r_{i*} \parallel^2}{2\sigma_a^2}) - b \exp(\frac{- \parallel r_i - r_{i*} \parallel^2}{2\sigma_b^2}) \tag{3.9}$$

4. Go back to step 1.

3.1.4 Radial Basis Functions

Radial Basis Functions networks [Haykin 1994] have a similar functionality but different architecture to MLP networks. The input layer is simply a receptor for the input data. The crucial feature of the RBF network is the function calculation which is performed in the hidden layer. This function performs a non-linear transformation from the input space to the hidden layer space. The hidden neurons' functions form a basis for the input vectors and the output neurons merely calculate a linear (weighted) combination of the hidden neurons' outputs. An often-used set of basis functions is the set of Gaussian functions whose mean and standard deviation may be determined in some way by the input data. Therefore if $\phi(\mathbf{x})$ is the vector of hidden neurons' outputs when the input pattern $\mathbf{x}$ is presented and if there are M hidden neurons, then:

$$\begin{aligned} \phi(\mathbf{x}) &= (\phi_1(\mathbf{x}), \phi_{\mathbf{2}}(\mathbf{x}), ..., \phi_{\mathbf{M}}(\mathbf{x}))^{\mathbf{T}} && (3.10) \\ \text{where } \phi_i(\mathbf{x}) &= \exp(-\lambda_i ||\mathbf{x} - \mathbf{c_i}||^{\mathbf{2}}) && (3.11) \end{aligned}$$

where the centres $\mathbf{c}_i$ of the Gaussians will be determined by input data. Note that the terms $||\mathbf{x} - \mathbf{c}_i||$ represent the Euclidian distance between the inputs and the i^{th} centre. The output of the network is calculated by:

$$y = \mathbf{w}.\phi(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) \quad (3.12)$$

where $\mathbf{w}$ is the weight vector from the hidden neurons to the output neuron.

The training of RBF networks is done using an algorithm very similar to the back propagation algorithm as described above; the main difference is only the weights for the output neurons have to be trained; hence the training is much faster. However the training time with MLP was already very fast and was not an important issue.

3.1.5 Topographic Products of Experts

In this section, we introduce a new topology preserving mapping we call the Topographic Products of Experts (ToPoE) [Fyfe 2007].

The most common topographic mappings are Kohonen's Self-Organizing map (SOM, above) [Kohonen 1995] and varieties of multidimensional scaling [Hastie et al. 2001]. Like SOM, ToPoE can be used as a visualisation tool. As opposed to SOM, ToPoE has the advantage that it explicitly

dispenses with the quantisation element; while its centres may lie on a manifold, the user interpolates the projections of data points between the centres to infer the shape of the manifold.

In a product of experts, all the experts take responsibility for all the data: the probability associated with any data point is the (normalised) product of the probabilities given to it by the experts.

We envisage that the underlying structure of the experts can be represented by K latent points, $t_1, t_2, ..., t_K$ which have some structure to their geometry in the same way which we saw with the neurons in Kohonen's Self Organising Map. To allow local and non-linear modelling, we map those latent points through a set of M basis functions, $f_1(), f_2(), ..., f_M()$. This gives us a matrix ϕ where $\phi_{kj} = f_j(t_k)$.

Thus each row of ϕ is the response of the basis functions to one latent point, or alternatively we may state that each column of ϕ is the response of one of the basis functions to the set of latent points. One of the functions, $f_j()$, acts as a bias term and is set to one for every input. Typically the others are Gaussians centred in the latent space. The output of these functions are then mapped by a set of weights, W, into data space. W is $M \times D$, where D is the dimensionality of the data space, and is the sole parameter which we change during training.

We will use w_i to represent the i^{th} column of W and ϕ_j to represent the row vector of the mapping of the j^{th} latent point. Thus each basis point is mapped to a point in data space, $\mathbf{m}_j = (\phi_j W)^T$.

We may update W either in batch mode or with online learning. To change W in online learning, we randomly select a data point, say $\mathbf{x}_i$. We calculate the current responsibility of the j^{th} latent point for this data point,

$$r_{ij} = \frac{exp(-\gamma {d_{ij}}^2)}{\sum_K exp(-\gamma {d_{ik}}^2)}; \tag{3.13}$$

where $d_{pq} = ||\mathbf{x}_p - \mathbf{m}_q||$, the Euclidean distance between the p^{th} data point and the projection of the q^{th} latent point (through the basis functions and then multiplied by W). If no centres are close to the data point (the denominator of (3.13) is zero), we set $r_{ij} = \frac{1}{K}, \forall j$.

Define ${m_d}^{(k)} = \sum_{m=1}^{M} w_{md}\phi_{km}$, i.e. $m_d^{(k)}$ is the projection of the k^{th} latent point on the d^{th} dimension in data space. Similarly let $x_d^{(n)}$ be the d^{th} coordinate of $\mathbf{x}_n$.

[Fyfe 2007] shows that a learning rule which maximises the likelihood

of the data under the model is:

$$\Delta_n w_{md} = \sum_{k=1}^{K} \eta \phi_{km} (x_d^{(n)} - m_d^{(k)}) r_{kn}; \tag{3.14}$$

where we have used Δ_n to signify the change due to the presentation of the n^{th} data point, $\mathbf{x}_n$, so that we are summing the changes due to each latent point's response to the data points. Note that, for the basic model, we do not change the ϕ matrix during training at all.

3.2 Genetic Algorithms

Genetic algorithms (GA) [Mitchell and Forrest 1993, Dasgupta 1993] became popular after the seminal work of Holland [Holland 1981] in the 1970's and 80's. His algorithm is usually known as the simple GA now since many of those now using GA's have added bells and whistles [Rechenberg 1994]. Holland's major breakthrough was to code a particular optimisation problem in a binary string, a string of 0's and 1's. He then created a random population of these strings and evaluated each string in terms of its fitness with respect to solving the problem. Strings which had a greater fitness were given greater chance of reproducing and so there was a greater chance that their chromosomes (strings) would appear in the next generation. Holland showed that the whole population of strings eventually converged to satisfactory solutions to the problem.

Notice that the population's overall fitness tends to increase as a result of the increase in the number of fit individuals in the population. However, there may be just as fit (or even fitter) individuals in the population at time t-1 as there are at time t; there is a strong stochastic element to the process of evolution. Thus, when we are discussing evolution, we can only make statements about populations rather than individuals.

We can identify the problem of finding appropriate weights for the MLP as an optimisation problem and have this problem solved using the GA i.e. we are using exactly the same architecture for the artificial neural network as in the previous chapter but are optimising the network's weights using the genetic algorithm: we code the weights as floating point numbers and use the algorithm on them with a score or fitness function.

The algorithm is:

1. Initialise a population of chromosomes randomly.

2. Evaluate the fitness of each chromosome (string) in the population.

3. For each new child chromosome:

   (a) Select two members from the current population. The chance of being selected is proportional to the chromosomes' fitness.

   (b) With probability, C_r, the crossover rate, cross over the numbers from each chosen parent chromosome at a randomly chosen point to create the child chromosomes.

   (c) With probability, M_r, the mutation rate, modify the chosen child chromosomes' numbers by a perturbation amount.

   (d) Insert the new child chromosome into the new population.

4. Repeat steps 2-3 till convergence of the population.

The desired characteristics of evolved individuals are expressed in the fitness or score function, used for the evaluation of individuals in the populations. The algorithm applied to the motocross game is illustrated in Figure 3.3.

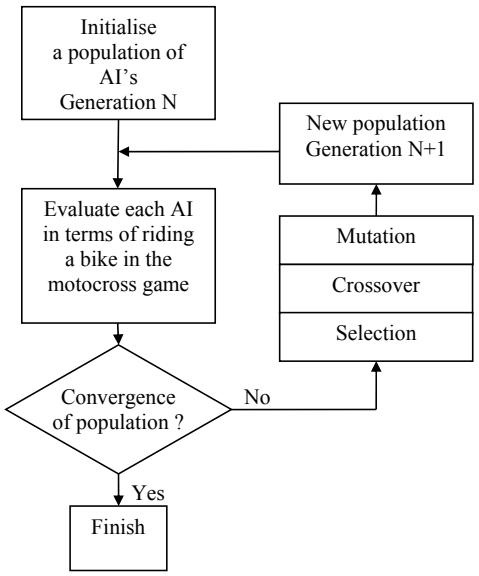

Figure 3.3: GA applied to the motocross game.

3.3 Ensemble Methods

Recently a number of ways of combining predictors have been developed e.g. [Breimen 1999; 1997, Friedman et al. 1998, Heskes 1997]. Perhaps the simplest is bagging predictors. The term "bagging" was coined by joining bootstrapping and aggregating; we are going to aggregate predictors and in doing so we are bootstrapping a system. We note that the term "bootstrapping" was derived from the somewhat magical possibilities of "pulling oneself up by one's bootstraps" and the process of aggregating predictors in this way does give a rather magical result - the aggregated predictor is much more powerful than any individual predictor trained on the same data [Breimen 1999; 1997, Friedman et al. 1998, Heskes 1997]. It is no wonder that statisticians have become very convincing advocates of these methods.

3.3.1 Bagging

Bootstrapping [Breimen 1999] is a simple and effective way of estimating a statistic of a data set. Let us suppose we have a data set, $D = \{\mathbf{x}_i, i = 1, ..., N\}$. The method consists of creating a number of pseudo data sets, D_i, by sampling from D with uniform probability with replacement of each sample. Thus each data point has a probability of $(\frac{N-1}{N})^N \to 0.368$ as $N \to \infty$ of not appearing in each bootstrap sample, D_i. For example,

1. if $N = 100$, $(\frac{N-1}{N})^N = 0.366$;
2. if $N = 1000$, $(\frac{N-1}{N})^N = 0.3677$;
3. we are dealing with data with $N \approx 10^5$, $(\frac{N-1}{N})^N = 0.3679$.

Each predictor is then trained separately on its respective data set and the bootstrap estimate is some aggregation (almost always a simple averaging) of the estimate of the statistic from the individual predictors. Because the predictors are trained on slightly different data sets, they will disagree in some places and this disagreement can be shown to be beneficial in smoothing the combined predictor. Typically, the algorithm can be explained as follows:

1. Create N bags by randomly sampling from the data set with replacement.
2. The probability of any single piece of data to be in any particular bag is approximately 0.631.

3. ANN's are trained on the bags separately.

4. The trained ANN's are then presented with an input and the outputs of the ANN's are combined.

We can use a number of different methods to combine the outputs from the different bags. We chose to use as the combination operator, in spirit similar to [Heskes 1997], the following:

$$O_{ANN} = O_{AVE} * (1 - w) + O_{WIN} * w; \tag{3.15}$$

With O_{AVE}, the average of all ANN's outputs, O_{WIN}, the output of the most confident ANN, which is the output with the largest magnitude, and w the winning parameter varying from 0 to 1. This allows us to vary the combination method by varying the single parameter w.

3.3.2 Boosting

There has been recent work identifying the most important data samples [Vapnik 1995]; and presenting the ANN more with the most important data samples (boosting [Friedman et al. 1998]) in order to concentrate training on these more difficult data. Therefore the supervised method (for example backpropagation or tree-based classification) is initially asked to learn to output the correct answer (the target response) for all of the data. It will succeed with some but may fail with others. These others are then concentrated upon for further training of the supervised method.

There are a number of ways to implement the boosting algorithm. Algorithmically an error is calculated as the difference between the desired/target output and the output given by the ANN. Subsequently, the ANN is presented more with training samples which produce large errors. We will also investigate a technique we call anti-boosting: with anti-boosting, the ANN is presented more with samples which produce small errors i.e. we are emphasising those samples which seem to be easiest for the backpropagation algorithm to learn.

We investigate the effect of different types of training data. For example, some parts of the track are relatively easy and the rider can accelerate quickly over these while other parts are far more difficult and so more care must be taken. The latter parts are also those where most accidents happen. Our first conjecture is that training the neural network on these more difficult parts might enable it to concentrate its efforts on the difficult sections

of the track and so a training routine is developed in which each training sample has a probability to be selected for training the ANN proportional to the error produced the last time the sample was presented to the ANN. This allows us to train the ANN with more difficult situations.

3.4 The Cross Entropy Method

The cross entropy method has been well introduced in [de Boer et al. 2004] and was motivated as an adaptive algorithm for estimating probabilities of *rare events* in complex stochastic networks. In such a situation, a Monte Carlo simulation which draws instances from the true distribution of events would require an inordinate number of draws before enough of the rare events were seen to make a reliable estimate of their probability of occurring. It was soon realised that the cross entropy method can also be applied to solving difficult combinatorial and continuous optimisation problems with a simple modification of the method. Generally speaking, the basic mechanism involves an iterative procedure of two phases:

1. draw random data samples from the currently specified distribution.
2. identify those samples which are, in some way, "closest" to the rare event of interest and update the parameters of the currently specified distribution to make these samples more representative in the next iteration.

In this section, we wish to apply the method of cross entropy to the N-persons Iterated Prisoner's Dilemma, an abstract mathematical game which has close links to the formation of oligopolies [Wang et al. 1999].

The Cross Entropy method is best approached from the perspective of its use in estimates of statistics concerning rare events such as the probability measure associated with the rare event. We discuss this first before going on to apply the method to the field of optimisation.

3.4.1 Rare Event Simulations

Let $l = (S(\mathbf{x}) > \gamma)$ be the event in which we are interested and typically we will be interested in problems in which l is very small. We could use Monte Carlo methods to estimate l but if l is very small this would lead to a very large number of samples before we could get reliable estimates of l. The cross

entropy method uses *importance sampling* rather than simple Monte Carlo methods: if the original pdf of the data is $f(\mathbf{x})$, then we require to find a pdf, $g(\mathbf{x})$, such that all of $g()$'s probability mass is allocated in regions in which the samples are close to the rare-event. More formally, we have the deterministic estimate

$$l = \int I_{\{S(\mathbf{x})>\gamma\}} f(\mathbf{x}) d\mathbf{x} = \int I_{\{S(\mathbf{x})>\gamma\}} \frac{f(\mathbf{x})}{g(\mathbf{x})} g(\mathbf{x}) d\mathbf{x} = E_{g()} \left[I_{\{S(\mathbf{X})>\gamma\}} \frac{f(\mathbf{X})}{g(\mathbf{X})} \right]. \tag{3.16}$$

where I_L is the indicator function describing when L in fact occurred. An unbiased estimator of this is

$$\hat{l} = \frac{1}{N} \sum_{i=1}^{N} I_{\{S(\mathbf{X_i})>\gamma\}} \frac{f(\mathbf{x}_i)}{g(\mathbf{x}_i)} = \frac{1}{N} \sum_{i=1}^{N} I_{\{S(\mathbf{X_i})>\gamma\}} \mathbf{W}(f(\mathbf{x}_i), g(\mathbf{x}_i)) \tag{3.17}$$

where $\mathbf{W}()$ is known as the likelihood ratio.

The best $g()$ in (3.16) is $g^*(\mathbf{x}) = \frac{I_{\{S(\mathbf{x})>\gamma\}} f(\mathbf{x})}{l}$, which would have the same shape as $f()$ but all its probability mass in the interesting region. This is illustrated in Figure 3.4. Note that for the optimal $g()$, $\int_{\mathbf{x}:S(\mathbf{x})>\gamma} g*(\mathbf{x}) d\mathbf{x} = 1$ while $\int_{\mathbf{x}:S(\mathbf{x})>\gamma} f(\mathbf{x}) d\mathbf{x} = l$.

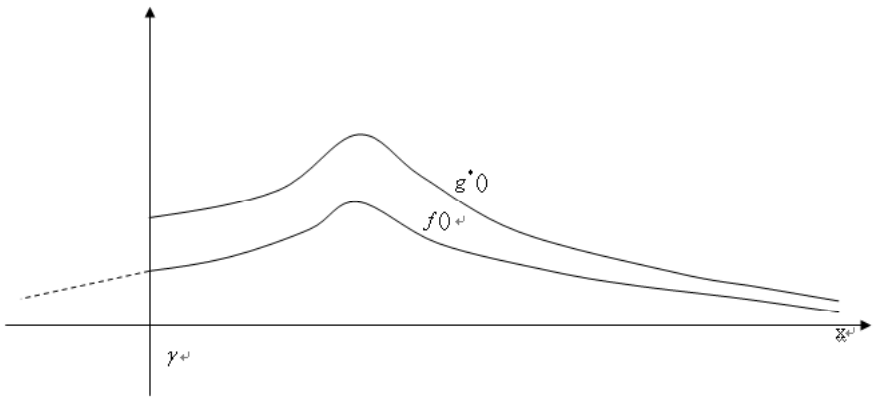

Figure 3.4: The original distribution f() has probability mass outwith the region we are interested in but the importance sampling distribution has only domain $\mathbf{x} : S(\mathbf{x}) > \gamma$.

However we don't know l. (This is exactly what we are trying to get). So what we do is to pick a family of pdfs $g(\mathbf{x}, \mathbf{v})$, parameterised by $\mathbf{v}$ ($\mathbf{v}$ will be the mean and variance for Gaussian).

We now wish to minimise Kullback Leibler divergence between g^* and $g()$,

$$\min KL(g^*, g) = \int g^*(\mathbf{x}) \ln g^*(\mathbf{x}) d\mathbf{x} - \int g^*(\mathbf{x}) \ln g(\mathbf{x}, \mathbf{v}) d\mathbf{x} \tag{3.18}$$

So we maximise the cross entropy $\int g^*(\mathbf{x}) \ln g(\mathbf{x}, \mathbf{v}) d\mathbf{x}$.

We pick $\mathbf{v}$ to max $\int \frac{I_{\{S(\mathbf{x})>\gamma\}} f(\mathbf{x})}{l} \ln g(\mathbf{x}, \mathbf{v}) d\mathbf{x}$, which is the same as

$$\max \int I_{\{S(\mathbf{x})>\gamma\}} f(\mathbf{x}) \ln g(\mathbf{x}, \mathbf{v}) d\mathbf{x}, \tag{3.19}$$

where we have discarded l a constant. But getting an optimal $g(\mathbf{x}, \mathbf{v})$ for a particular γ may not be an easy task. Therefore we create a set of γ_t for which we estimate the corresponding $\mathbf{v}_t$. The γ_t are chosen such that

$$P(\mathbf{x} : S(\mathbf{x}) > \gamma_t) > P(\mathbf{x} : S(\mathbf{x}) > \gamma_{t+1}) \tag{3.20}$$

i.e. at each iteration, the events are becoming more rare under $f()$. Therefore

$$\max \int I_{\{S(\mathbf{x})>\gamma\}} f(\mathbf{x}) \ln g(\mathbf{x}, \mathbf{v}) d\mathbf{x} \tag{3.21}$$

$$= \max_{\mathbf{v}_t} \int I_{\{S(\mathbf{x})>\gamma_\mathbf{t}\}} \frac{f(\mathbf{x})}{g(\mathbf{x}, \mathbf{v}_{t-1})} \ln g(\mathbf{x}, \mathbf{v}_t) g(\mathbf{x}, \mathbf{v}_{t-1}) d\mathbf{x} \tag{3.22}$$

$$= \max_{\mathbf{v}_t} E_{g(\mathbf{x}, \mathbf{v}_{t-1})} \Big\{ I_{\{S(\mathbf{x})>\gamma_t\}} \mathbf{W}(f(\mathbf{x}), g(\mathbf{x}, \mathbf{v}_{t-1})) \ln g(\mathbf{x}, \mathbf{v}_t) \Big\} \tag{3.23}$$

This is deterministic but we are working with samples. So we pick $\mathbf{v}_t$ to maximise

$$\max_{\mathbf{v}_t} \frac{1}{N} \sum_{i=1}^{N} I_{\{S(\mathbf{x}_i)>\gamma_t\}} \mathbf{W}(f(\mathbf{x}_i), g(\mathbf{x}_i, \mathbf{v}_{t-1})) \ln g(\mathbf{x}, \mathbf{v}_t) \tag{3.24}$$

For example, if $g(\mathbf{x}, \mathbf{v}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(\frac{\mathbf{x}-\mu}{\sigma})^2}$, we find minimum of

$$\frac{1}{N} \sum_{i=1}^{N} I_{\{S(\mathbf{x}_i)>\gamma_t\}} \mathbf{W}_{t-1} \Big\{ \ln(\sigma) + \frac{1}{2\sigma^2} (\mathbf{x} - \mu)^2 \Big\} \tag{3.25}$$

We calculate the derivative of this with respect to the parameters, and set this equal to 0, to determine

$$\hat{\mu} = \frac{\sum_{i=1}^{N} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{t-1}) X_i}{\sum_{i=1}^{N} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{\mathbf{t-1}})} \tag{3.26}$$

$$\hat{\sigma^2} = \frac{\sum_{i=1}^{N} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{t-1}) (\mathbf{x}_i - \hat{\mu})^2}{\sum_{i=1}^{N} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{\mathbf{t-1}})} \tag{3.27}$$

The "cross entropy method" is so-called since we minimise the Kullback-Leibler divergence between the data distribution and the importance sampling distribution.

3.4.2 Algorithm for Rare Events

Then the simplest algorithm [de Boer et al. 2004] depends on working within a family of pdfs whose parameters we update i.e. let $f(\mathbf{x}) = f(\mathbf{x}, \mathbf{u})$, $\mathbf{u}$ being a parameter of the family to which $f()$ belongs; then the basic algorithm is

1. Define $\hat{\mathbf{v}}_0 = \mathbf{u}$. Set t=1.
2. Generate random samples, $\mathbf{X}_1, ..., \mathbf{X}_N$ from $f(\mathbf{x}, \hat{\mathbf{v}}_{t-1})$.
3. Calculate $S(\mathbf{X}_1), ..., S(\mathbf{X}_N)$ and order them. Let $\hat{\gamma}_t$ be the $1-\rho$ sample quantile, above which we identify the "elite" samples.
4. Use the same samples to calculate

$$\hat{\mathbf{v}}_t = \frac{\sum_{i=1}^{N} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{t-1}) h(\mathbf{x}_i)}{\sum_{i=1}^{N} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{\mathbf{t-1}})} \tag{3.28}$$

   where $h(\mathbf{X}_i)$ is capturing some statistic of the elite samples: for example, if we have a Gaussian random variable, h() would be defining the mean and variance of the distribution.
5. If $\hat{\gamma}_t = \gamma$, continue; else $t = t + 1$ and return to 2
6. Generate a sample $\mathbf{X}_1, ..., \mathbf{X}_{N_1}$ from $f(\mathbf{x}, \hat{\mathbf{v}}_t)$ and estimate

$$l = \frac{1}{N_1} \sum_{i=1}^{N_1} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{t-1}) \tag{3.29}$$

Note that, although step 4 looks formidable, it is actually only counting the fraction of samples which satisfy the current criterion.

3.4.3 Cross Entropy Method for Optimisation

For optimisation, we need to turn the problem into the so-called *associated stochastic problem*(ASP) firstly. The basic method is

- Generate random samples from the associated stochastic problem using some randomisation method.
- Update the parameters (which will typically be parameters of the pdf generating the samples) to make the production of better samples more likely next time. For a Gaussian distribution, this results in updates only to the mean, μ, and covariance, Σ.

Note that, unlike the rare event simulations, we do not have a base parameterisation to work to and hence no need to have the $W(\mathbf{X}_i, \mathbf{u}, \hat{\mathbf{v}}_{t-1})$ term in the calculation. Of course we could have this term defined in terms of $\hat{\mathbf{v}}_t$ and $\hat{\mathbf{v}}_{t-1}$ but [Rubinstein and Kroese 2004] show that this is not essential and indeed tends to introduce unnecessary noise into the convergence.

We usually wish to maximise some performance function $S(\mathbf{x})$ over all states $\mathbf{x}$ in data set $\aleph$. Denoting the maximum by γ^*, we have

$$\gamma^* = \max_{\mathbf{x}\in\aleph} S(\mathbf{x}) \tag{3.30}$$

Thus, by defining a family of pdfs $\{f(.\,;\mathbf{v}), \mathbf{v} \in \nu\}$ on the data set $\aleph$, we follow [Rubinstein and Kroese 2004] to associate with (3.30) the following estimation problem

$$l(\gamma) = \mathbf{P}_v(S(\mathbf{X}) \geq \gamma)) = \mathbf{E}_v I_{\{S(\mathbf{x})>\gamma\}} \tag{3.31}$$

where $\mathbf{X}$ is a random vector with pdf $f(.\,;\mathbf{v}), \mathbf{v} \in \nu$. To estimate l for a certain γ close to γ^*, we can make adaptive changes to the probability density function according to the Kullback-Leibler cross-entropy. Thus we create a sequence $f(.\,;\mathbf{v}_0), f(.\,;\mathbf{v_1}), f(.\,;\mathbf{v_2}), \ldots$ of pdfs that are optimised in the direction of the optimal density and for the fixed $\hat{\gamma}_t$ and $\hat{\mathbf{v}}_{t-1}$, we derive the $\hat{\gamma}_t$ from the following program

$$\max_{\mathbf{v}} \hat{D}(\mathbf{v}) = \max_{\mathbf{v}} \frac{1}{\mathbf{N}} \sum_{i=1}^{N} I_{\{S(\mathbf{X}_i)>\hat{\gamma}_t\}} \ln f(\mathbf{X}_i;\mathbf{v}) \tag{3.32}$$

One advantage that this representation has is that it is very simple to change the base learner. If we use Gaussian distribution, we need to estimate the mean and variance of the elite samples as

$$\hat{\mu} = \frac{1}{N_{elite}} \sum_{i=1}^{N_{elite}} \mathbf{X}_i \tag{3.33}$$

$$\hat{\Sigma} = \frac{1}{N_{elite}} \sum_{i=1}^{N_{elite}} (\mathbf{X}_i - \hat{\mu})(\mathbf{X}_i - \hat{\mu})^T \tag{3.34}$$

[Rubinstein and Kroese 2004] shows that if we have a finite support discrete distribution such as the Bernoulli distribution, then we can have the elements of the probability vector updated according to

$$\hat{p}_{t,ij} = \frac{\sum_{k=1}^{N_{elite}} I_{S(x_k)>\hat{\gamma}_t} I_{x_{ki}=j}}{\sum_{k=1}^{N_{elite}} I_{S(x_k)>\hat{\gamma}_t}} \tag{3.35}$$

where we use $\hat{p}_{t,ij}$ is the estimated probability that the i^{th} element of the probability vector will equal j at iteration t. This is the method Colin Fyfe used to solve the **Iterated Prisoner's Dilemma** problem. He used the smoothing technique of [Rubinstein and Kroese 2004] so that the parameter vector actually used at time t was

$$\tilde{\mu}_t = \alpha\hat{\mu}_t + (1-\alpha)\tilde{\mu}_{t-1} \tag{3.36}$$

where $\hat{\mu}_t$ is the outcome of the calculation (3.33) and $\alpha = 0.2$.

Chapter 4

Experiments

4.1 Setup

We will use the same setup (inputs/outputs) for most experiments except for the Symbolic AI experiment.

4.1.1 Inputs

An example of how the track appears to human game players is shown in Figure 4.1.

There is one marker known as a waypoint which marks the position and orientation of the centre of the track, every metre along the track. Waypoints can be thought of as being gates the bike crosses while following the track. The current waypoint is the waypoint the bike is supposed to cross next in order to follow the track.

We define "forward space" to be the spatial coordinate system given by the velocity vector of the bike and the Up (vertical) vector. Previously we had defined all operations in "bike space", defined by the coordinates of the bike however we found that the bike was moving and rotating a lot along the track. It appeared that instead of expressing the position of the waypoints in bike space, it was better to express these in forward space.

The inputs to the ANN are:

- orientation of the bike, in forward space, given by two 3D vectors.
- positions of the centre of the track, at distances

Figure 4.1: Inputs of the AI: positions of the centre of the track, used as inputs to the ANN.

{0,2,5,7,10,15,20,25,30,35,40,50,60} metres, in forward space, given by thirteen 3D vectors.

- distance between the bike and the ground.
- velocity of the bike, given by a scalar.
- a bias, set to 10.

There are two main advantages in using the forward space instead of the bike space to transform ground samples:

1. It does not rotate in time in relation to the ground as much as the bike transform, so it allows the ANN to more easily identify input patterns for ground samples.

2. Because the velocity direction information is now contained in the forward space used to transform ground samples, it is now possible to express the velocity as a scalar and not a vector and save two inputs for the ANN.

The inputs to the networks are mixed and can be of very different magnitude; for example one input can be the height of a waypoint in centimetres and vary in the range {-10000,10000} while another input may be one component of the orientation of the bike and vary in the range {-1,1}. For untrained ANN's, large inputs are seen as more significant, i.e. carry more weight in the final output, than smaller inputs. This becomes even more significant when we consider alternative architectures to multilayered perceptron networks. It has proved to significantly improve ANN's operation to normalise all inputs in the range {-10,10}.

1. The normalisation allows untrained ANN's to initially see all inputs as equally significant; this proves to improve the training.

2. The normalisation reduces the risk of computational problems inside the ANN's that would normally occur during operations between single precision floating point numbers of very different magnitude.

The normalisation and bias values have been found experimentally to produce good results. Other values would also probably produce equally good results.

To normalise the inputs in the range {-10,10}, the minimum and maximum values of each input are found from the samples in the training set; then for each sample each input is normalised in the range {-10,10} using:

$$i' = 20\frac{i - i_{MIN}}{i_{MAX} - i_{MIN}} - 10 \tag{4.1}$$

4.1.2 Outputs

The outputs from the ANN are:

1. turn left/right.

2. target velocity.

3. rotation of the bike forward/backward.

4. rotation of the bike left/right.

It was not totally obvious what would be the best output to expect from the ANN's. If the outputs for the ANN are exactly the same as the controls for a human player, then one output is the acceleration control. The ANN is required to do some derivation work: for a given situation, the ANN has to determine what the optimal velocity is, then determine if it must accelerate or brake in order to reach that optimal velocity. This is not an easy task. If the ANN is trained using the back propagation algorithm, then there might not be training samples where the bike is to accelerate in one part of a track after recovering from an accident. Similarly if the bike is normally braking in a particular portion of the track, and the derivation work is not done well and the bike is travelling at a low velocity, then the bike may brake and come to a stop. Now if the output from the ANN is the target velocity, then there is no more derivation work to be done by the ANN. The derivation work is done through simple computation. This in theory allows the ANN to concentrate on determining what is the most optimal velocity in a given situation.

There are also problems associated with this technique: one problem is because the ANN only determines the target velocity, important inputs such as the bike balance may not be taken into consideration well in the final acceleration decision. Another problem is because one input to the ANN is the current velocity, one ANN trained using back propagation may reproduce at output the velocity at input. This identity mapping is not useful in this context.

The output of the ANN to be used in the evaluation of the target velocity is $\mathbf{O_{Vel}}$ (output 2) and its value is in the range {-1,1}.

The target velocity $\mathbf{V_{Target}}$ (to be used in the evaluation of the acceleration/deceleration decision) is evaluated as:

$$V_{Target} = \frac{V_{Max} * (O_{Vel} + 1)}{2} \tag{4.2}$$

with $\mathbf{V_{Max}}$ the bike maximum velocity (32 m/s).

The acceleration/deceleration decision is evaluated as:

$$Acc = C_1 * \frac{V_{Target} - V_{Cur} * (Dir * V)}{V_{Max}} \tag{4.3}$$

with $\mathbf{V_{Cur}}$ the current velocity, $\mathbf{Dir} * \mathbf{V}$ the dot product between the forward direction of the bike and the velocity vector direction (used to prevent

problems when the bike is travelling backwards), and $\mathbf{C_1}$ an acceleration multiplier (experimentally set to 2).

The ANN is trained at evaluating the target velocity. To prevent the ANN reproducing at output the velocity at input, the target velocity used while performing the back propagation algorithm is evaluated as:

$$V_{Target} = V_{Cur} * (Dir * V) + C_0 * V_{Max} * Acc$$

with $\mathbf{V_{Cur}}$ the bike real current velocity in a given situation, **Acc** the associated acceleration decision (from the training set), and $\mathbf{C_0}$ an acceleration multiplier (experimentally set to 0.7).

Having the ANN trained at evaluating a target velocity proved to be successful; the computer controlled bikes perform much better in all situations. The other outputs (1,3,4) are used directly as controls for riding the bike.

We use the same activation function at the outputs as at the hidden neurons, i.e. $m \tanh \frac{act}{r}$, with an activation multiplier m set to 1.33, and an activation response r set to 5. The activation multiplier m set to 1.33 allows the networks to easily output values in the range {-1,1}. Each control for the bike is also in the range {-1,1}, thus the outputs from the network can be used directly as controls for the bike and neurons in the output layer use largely the "linear" range of the activation function. The activation response r set to 5 allows each neuron in the network to be responsive, i.e. a small change in the input can produce a significant change in the output.

The ANN needs to be trained. One common supervised learning technique is the backpropagation algorithm.

4.2 Artificial Neural Networks

4.2.1 Multilayered Perceptrons and the Backpropagation Algorithm

We first consider an artificial neural network trained by the backpropagation method. The ANN's have 48 inputs, 2 hidden layers of 40 neurons and 4 outputs. These numbers of hidden layers and neurons in the hidden layer have been found experimentally to produce good results. Other numbers could also probably produce equally good results.

We are using a supervised method (backpropagation) to update the parameters and so we require training data for which there already exists target actions. We note that these targets (which will be the actions of a good human player on the track) are not necessarily optimal - there may be a better player who could beat the human player - but at least they will be better than random and should be competitive with other humans.

The backpropagation algorithm in the context of this motocross game requires the creation of training data made from a recording of the game played by a good human player. The targets are the data from the human player i.e. how much acceleration/deceleration, left/right turning and rotation (forwards/backwards and left/right) of the bike was done by the human player at that point in the track. The aim is to have the ANN's reproduce what a good human player is doing. The human player's responses need not be the optimal solution but a good enough solution and, of course, the ANN's will learn any errors which the human makes.

Some training data is created by having the author playing the game on track Long for 45 minutes (270218 samples), with an average lap times of approximately 2 minutes 15 seconds. The author could have tried to optimise the training set, for example he could ride in a safe manner, taking extra care on the difficult portions of the track, and avoiding obstacles using extra safe distance, in order for the ANN to learn behaviours that would prevent these accidents; instead, the author played the game in a fast but risky manner.

The backpropagation algorithm is used to train an ANN on newly created training data. The number of iterations is set to 1000000. The learning rate is set to decrease logarithmically from $1 * 10^{-2}$ to $1 * 10^{-5}$. The training is done online at a rate of 5000 iterations a second; this allows the user to observe the ANN as it trains. After training, the average lap time on track Long for a computer controlled bike is found to be 2 minutes 26 seconds.

There is a 11 seconds difference in average lap time between the computer controlled bike and the human controlled bike. The computer controls the bike in a fast and risky manner, similar to the way the human player controls the bike; however the computer is not yet as successful as the human player at controlling the bike and has slightly more accidents, hence the difference in lap times.

Track Long is shown on Figure A.2.

Generalisation

Of course, we do not want to train an AI only to be able to ride the bike on a single track. We wish to train the AI in such a way that its training carries over to other tracks, just as a human player's training and experience would equip him/her to play on other tracks.

To that end, we check the generalisation property of our ANN; we present our ANN (trained using backpropagation with a training set from track Long) with track O and track A. The three tracks have different features. Track O features large hills and long straights that track Long doesn't feature; they also share many common features like turns and small bumps. Track A features many big turns and jumps that track Long doesn't feature.

On track O, the average lap time for the human player is 4 minutes 5 seconds and the average lap time for our ANN is 4 minutes and 20 seconds. On track A, the average lap time for the human player is 2 minutes 53 seconds and the average lap time for our ANN is 3 minutes and 15 seconds.

The ANN has never been trained to ride a motorbike on these particular tracks, but still is able to do so nearly as well as the human player, because it has been trained on another track and the three tracks have many similarities. The difference in lap times between the human player and the computer is even smaller on track O than on the original track, because track O is slightly less challenging.

The generalisation property is confirmed. The tracks used to test the generalisation property are shown on Figure A.3.

Conclusion

We have shown that the backpropagation algorithm can train ANN's to ride motorbikes in nearly the same way and nearly as well as a human player. The AI can be trained in a relatively modest time and will emulate the person who created the training data: if the human takes risks and rides dangerously, the AI will learn to ride the same way; if the human opts for safety and has a cautious ride, the AI will also ride cautiously.

Finally the ANN's trained using BP are also able to generalise from one track to another similar track; this is an essential property for an AI since we do not wish to have to start our training from scratch each time a new environment is encountered. We require that, just as with humans, its experience and success on one set of input data is carried over to be the basis

of success in other data sets..

4.2.2 Kohonen's Self-Organizing Map

In the context of the game, the data is of dimension D, with:

$$D = D_{Input} + D_{Output} \tag{4.4}$$

D_{Input} is the number of inputs; the inputs are the state or situation of the bike, i.e. position, orientation and velocity of the bike relative to the track, and information about the terrain. In the previous chapters, these were the inputs to ANN's. There are 48 inputs.

D_{Output} is the number of outputs; the outputs are the decisions made according to the state or situation of the bike, i.e. accelerate or brake, turn left or right, and 2 rotations of the bike. In the previous chapters, these were the output from ANN's. There are 4 outputs.

During training, the distances are evaluated and the weights are adjusted over the full dimensionality of the data (inputs and outputs) D. Then, when the network is actually used inside the game to control a motorbike, the distances are evaluated using only the first D_{Input} dimensions, and the network output is the remaining D_{Output} dimensions of the winning neuron's weights.

Our SOM network is made of 272 neurons, positioned regularly on a 16 by 17 grid. Experiments showed that increasing the number of neurons increases the training time but does not improve significantly the performance of the network.

Using this SOM network to control a motorbike in the game, the average lap time on track Long is found to be 3 minutes 11 seconds while it was 2 minutes 26 seconds when a MLP network was used.

4.2.3 Radial Basis Functions

The inputs and outputs of the RBF network are the same as those used for the MLP network.

RBF networks require the determination of centres which may be optimised by gradient descent on the error function though this is not guaranteed to converge. Thus an alternative strategy is used in that the centres are chosen randomly from the data set. The average lap time on track Long is

approximately 3 minutes 15 seconds while it was 2 minutes 26 seconds when a MLP network was used.

Similarly the width of the basis functions can be optimised with the training set but again this can lead to non-convergence and so we simply experimented with different values and chose what seemed to be the most appropriate value.

Our RBF network uses 100 centres. Experiments showed that increasing the number of centres increases the training time but does not improve significantly the performance of the network.

It is possible that better performance for the RBF network can be achieved by more carefully determining the centres for the networks. However the time spent determining good centres for the network would increase the time spent to create and train the network, and hence the training speed advantage the RBF had over the MLP would be cancelled.

4.2.4 Topographic Products of Experts

The data set is the same as for SOM, as described in the previous section.

During training, the responsibilities of the experts are calculated over the full dimensionality of the data (inputs and outputs). Then, when the network is trained and actually used inside the game to control a motorbike, the responsibilities of the experts are evaluated using only the first D_{Input} dimensions.

Each output is computed as:

$$O_d = \sum_{k=1}^{K} (\mathbf{m}_d^{(k)}) r_{kn}; \tag{4.5}$$

Our ToPoE network uses 200 latent points and 26 basis functions. Experiments showed that increasing the number of latent points and basis functions increases the training time but does not improve significantly the performance of the network.

The ToPoE network is performing better than the SOM. The average lap time on track Long is approximately 3 minutes 5 seconds while it was 3 minutes 11 seconds when a SOM network was used.

The difference is due to the fact that Kohonen's SOM is a quantisation method; it does not interpolate between projections of latent points into data

space; the method tries to find the best match between a given situation, and situations it has learned, then gives the corresponding decision. The ToPoE network is able to interpolate between many similar situations, and so the final decision is an interpolation of different decisions. However the ToPoE network requires a lot more processing than all of the other methods to learn and to make a decision.

4.3 Genetic Algorithms

As stated in Chapter 3, genetic algorithms can be considered as generic optimisation techniques. The problem to which we apply the genetic algorithm is that of finding the set of parameters (weights) for our multilayered perceptron which allows the bikes to be driven as fast as possible around the various tracks in the game. That is, instead of using the backpropagation algorithm to optimise the weights, we use the genetic algorithm to optimise the weights with the fitness of any particular set of weights being determined by the time taken to complete a circuit: the faster the bike completes a circuit of the track, the more fit the parameters/weights are deemed to be.

Instead of the technique for crossover discussed in Chapter 3 an alternative technique for crossover has been investigated: instead of crossing over the numbers (corresponding to the ANN's weights) from each chosen parent chromosome at a randomly chosen point to create the child chromosomes, numbers from parents are averaged to create the child chromosomes. This seems appropriate because we are working with floating point numbers and not binary digits and is a method which is sometimes used with Evolution Strategies [Rechenberg 1994] which are designed for use with floating point numbers. Initial experimentation revealed that a blend of these two techniques worked best. The particular crossover technique was chosen randomly, with each technique being given equal chance, for each new child chromosome and then applied as usual.

In the current implementation, the NPC's do not see each other. Collisions have been removed between motorbikes; this has two advantages:

1. Collisions between motorbikes do not interfere any more with the evaluation of motorbike driving. A motorbike crashes only because of bad driving and not because another bike crashed into it.

2. This saves some processing power and allows simulating more bikes on a single computer.

In early experiments, the number of generations was set to 100, with a population of 80 ANN, elitism of 2 (number of the fittest chromosomes being passed directly from the parent population to the child population) which is equal to 2.5%, a mutation rate of 0.1, a crossover rate of 0.7, and a perturbation rate decreasing logarithmically from 0.8 to 0.008. These values were found experimentally to give reasonable results but there seems to be no way to derive optimal values for such parameters. In passing we note that these algorithms are multi-parameterized so that if we change one parameter, this may have a knock-on effect on the optimal range of a second parameter. Such search spaces are difficult to evaluate algorithmically and so heuristics are mainly used to find optimal ranges for parameters.

There was a simple reason why the perturbation was set high at the beginning and low at the end: let us consider an ANN attempting to jump bumps; if for example the bike goes at 30 km/h, then the bike can jump over perhaps one large bump; if the bike goes at 45 km/h, then the bike may be able to jump two large bumps at once. However if the bike goes at 35 km/h, then the bike may land on the ascending part of the second bump and is likely to crash. It was assumed that if the perturbation was not set high at the beginning, then an ANN which is successfully jumping one bump would not be able to attempt jumping two bumps at once; any increase in speed would only take it into the crash regime not into the second safe regime. The perturbation was set low at the end of the training, because as we are approaching the solution, we don't want to deviate too much from this solution; all that is required at this stage is some fine-tuning which we get from a much smaller perturbation parameter.

The early experiments showed that most of the improvement happened towards the beginning of evolution, when the perturbation was approximately equal to 0.3. Furthermore, we want the AI to improve continuously, and not stop improving after a given number of generations. Because the neurons make use of a non-linear function (as the activation function), a small change in a weight inside the ANN can still produce a large enough change on the output of the ANN. Some small changes inside the ANN can still allow it to change from one safe regime to another safe regime in the case of doing jumps. This is why the perturbation rate was finally set constant at 0.3.

The intelligence and experience are shared by more than one bike. Originally 6 bikes were on the track and the same number of ANN were thus evaluated for fitness at any given time. By disabling features, like collision between bikes and the sound, it was finally possible to have 10 bikes on the track at the same time.

One constraint while evolving ANN's with the GA is processing time. It takes time to evaluate each and every individual. Some fit individuals can be evaluated as being unfit because of bad circumstances if individuals are evaluated for too short a time, and reciprocally some unfit individuals can be evaluated as being fit. Finally it was empirically found that we had accurate results from the score function and hence valuable evolution when we evaluate all individuals for 10 minutes.

The early population size was 80, with elitism of 2. A large population size allows for diversity in the population; however it takes time to evaluate each generation of population. Similar or better results were obtained by reducing the population size to 20, which allows for more generations and hence more evolution, for the same computer processing time.

The fitness of individuals is evaluated using a score function. The score is calculated as follows:

- **vPassWayPointBonus** is a bonus for passing through a waypoint.
- **vMissedWayPointBonus** is a bonus/penalty (i.e. normally negative) for missing a waypoint.
- **vCrashBonus** is a bonus/penalty (i.e. normally negative) for crashing the bike.
- **vRespawnBonus** is a bonus/penalty (i.e. normally negative) for being respawn by the game engine after failing to cross the next waypoint, or after driving off track, for a given time (9 seconds).
- **vFinalDistFromWayPointBonusMultiplier** is a bonus/penalty (i.e. normally negative) for every metre away from the centre of the next waypoint.

The bonuses/penalties are associated with events; the fitness of each individual is initially set to zero; when an event occurs, the score or fitness of the individual is incremented by the corresponding bonus/penalty; all bikes/individuals are evaluated for the same duration **vTestTime** (evaluation period). The individual who has accumulated the greatest bonuses or the least penalties is evaluated as being the fittest.

Early experiments have shown some interesting behaviours:

- If **vPassWayPointBonus** is too high and **vRespawnBonus** not low enough (penalty not high enough), the ANN tends to learn to instantly

crash the bike immediately after the bike has been spawned, relying on the fact that the game engine spawns the bike on the track in the right direction after a crash.

- If **vCrashBonus** is too low (penalty too high), the ANN tends to learn to ride away from the track, where it is flat and safer.

- If **vTestTime** is too high, then the ANN tends to control the bike in a very slow and safe manner, in order not to crash and not put itself in an unrecoverable situation.

- If **vTestTime** is too low, then the ANN tends to control the bike in a fast but risky manner.

After experimentation, the following values have been determined as suitable:

- **vPassWayPointBonus** = 10

- **vMissedWayPointBonus** = -1

- **vCrashBonus** = -300

- **vRespawnBonus** = -300

- **vTestTime** = 600 seconds

The following values have been determined as suitable for the GA algorithm:

- **vMutationRate** = 0.2

- **vMaxPerturbation** = 0.3 (constant)

4.3.1 Training

Below are the graphs showing the fitness (Figure 4.2) and lap times (Figure 4.3) of ANN's trained using the GA to ride motorbikes on track Long during nearly 5 days. The weights inside the ANN's are initialised with random values in the range {-1,1}.

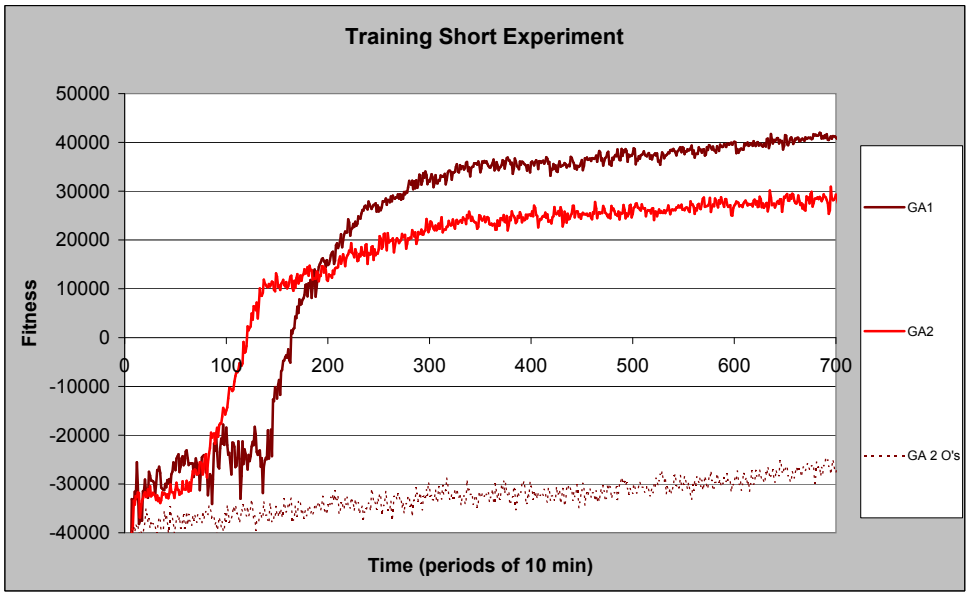

Figure 4.2: Fitness of the AI, GA used for training networks. Track Long.

The time is represented in periods of 10 minutes. This is the time required to evaluate a group of individuals.

The plain lines represent results of two experiments with the same initial conditions. Because of the stochastic nature of the genetic algorithm, we do not obtain exactly the same results each time we run the algorithm; the two experiments (referenced as GA1 and GA2 in Graphs 4.2 and 4.3) are representative of the results which we achieve each time. We can see that the algorithm is working, the fitness is increasing and the lap times are decreasing with time.

The two graphs, fitness and lap times, do not exactly follow each other; the fitness function is a function of how fast the ANN's are riding motorbikes along the track; it is also a function of how good the ANN's are following the track and how good they are at avoiding crashes.

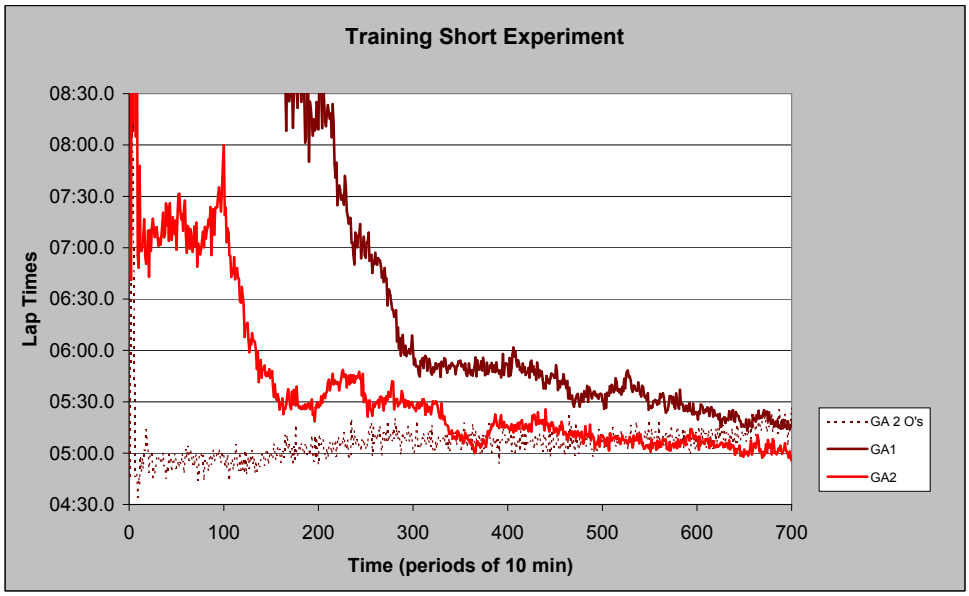

Figure 4.3: Lap Times of the corresponding bikes, GA used for training networks. Track Long.

The ANN's have to give four outputs corresponding to the four controls to ride the motorbikes; two of these outputs, the rotation of the motorbikes, are optional and can be left to their default values, zero; a human player can play the game very well without the need for these additional controls. We investigated whether it was more difficult to train networks to respond with 4 outputs than only 2 outputs to see if networks trained to give 2 outputs would evolve faster than networks trained to give 4 outputs. The doted lines (referenced as GA2O's in Graphs 4.2 and 4.3) represent results of the same experiments with the same initial conditions, except ANN's are giving only 2 outputs. The graphs show that these networks giving 2 outputs do not evolve as fast as the other networks giving 4 outputs. The extra controls were added to make it easier for human players to control the bikes, especially when the bikes are in the air. The experiments show that these controls also make it easier for the computer to control the bikes.

We can see that for the doted lines, the fitness is slowly increasing, but the lap times do not improve much; this means that the improvement in fitness is due to a reduction in the number of crashes or respawns. The

fitness function is such that networks first learn to follow the track and avoid crashes, then learn how to improve lap times by driving faster on portions of the track.

We see that the ANN's are improving but generally do not perform as well as other ANN's trained using backpropagation. After more than 116 hours of training, the average lap time is approximately 5 minutes on track Long, while it was 2 minutes 26 seconds when the Backpropagation Algorithm was used to train a network. It seems that the lap times are improving, reducing at a rate of approximately 0.6 second per hour of evolution. It will take a long time before networks trained using the GA perform as well as other networks trained using the BP algorithm.

During the last year of research, the student had access to better computers (Intel Quad Core 2.4 GHZ, as opposed to AMD Athlon 1.7 GHZ), the program was changed to allow the simulation to run faster than realtime, and to run as a screen saver. On a Intel Quad Core 2.4 GHZ, the simulation ran 4.5 times faster than realtime. Many computers were used at night and weekends for weeks to run the experiments. These longer experiments will be referred to, in the rest of thesis, as the new experiments. The original experiments demonstrate short time evolution and the new experiments demonstrate long time evolution.

The initial conditions were slightly different, with now all evolution experiments starting from the same random population. Two experiments with the GA were carried out for 1666 hours, and the results are shown in Figures 4.4 and 4.5.

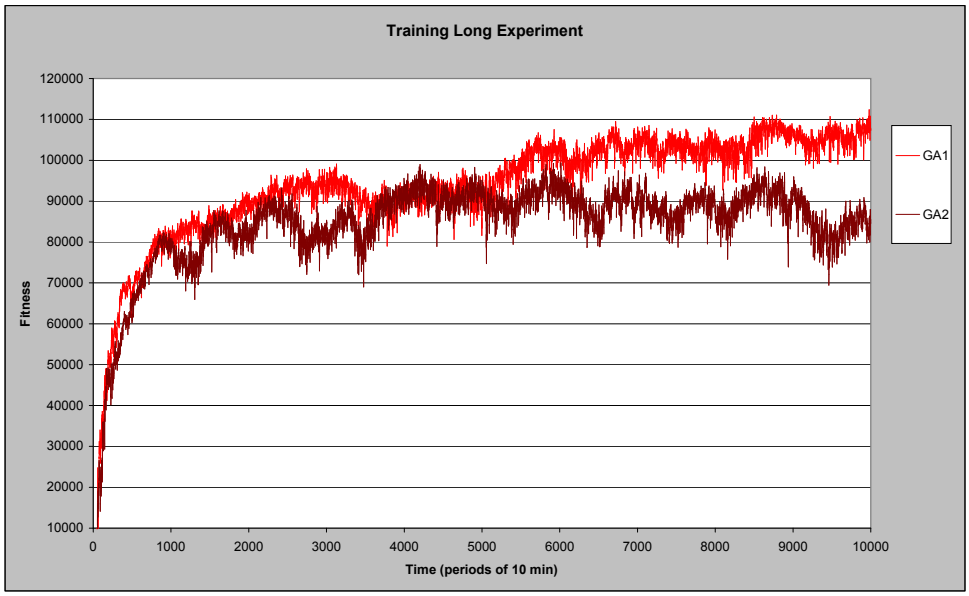

Figure 4.4: New experiments, fitness of the AI, GA used for training networks. Track Long.

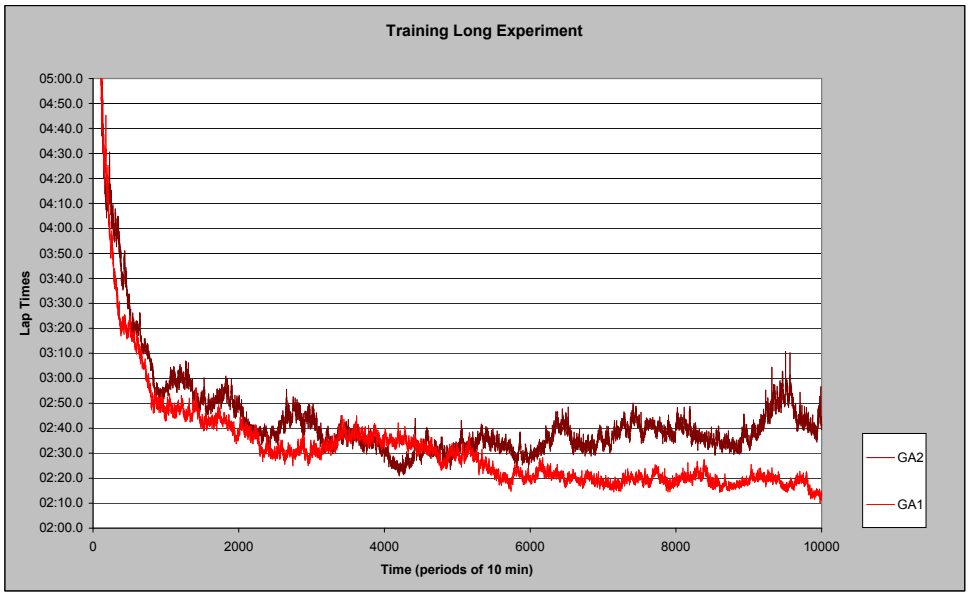

Figure 4.5: New experiments, lap times of the corresponding bikes, GA used for training networks. Track Long.

We can see that after a very long training, 1666 hours, the average lap time are approximately 2 minutes 14 seconds and 2 minutes 40 seconds on track Long, while it was 2 minutes 26 seconds when the Backpropagation Algorithm was used to train a network. In one AI experiment (referenced as GA1 in Figures 4.4 and 4.5), the networks trained using the GA perform better than networks trained using the BP algorithm.

The experiments were run for another 1666 hours; the results are shown in Figures 4.6 and 4.7.

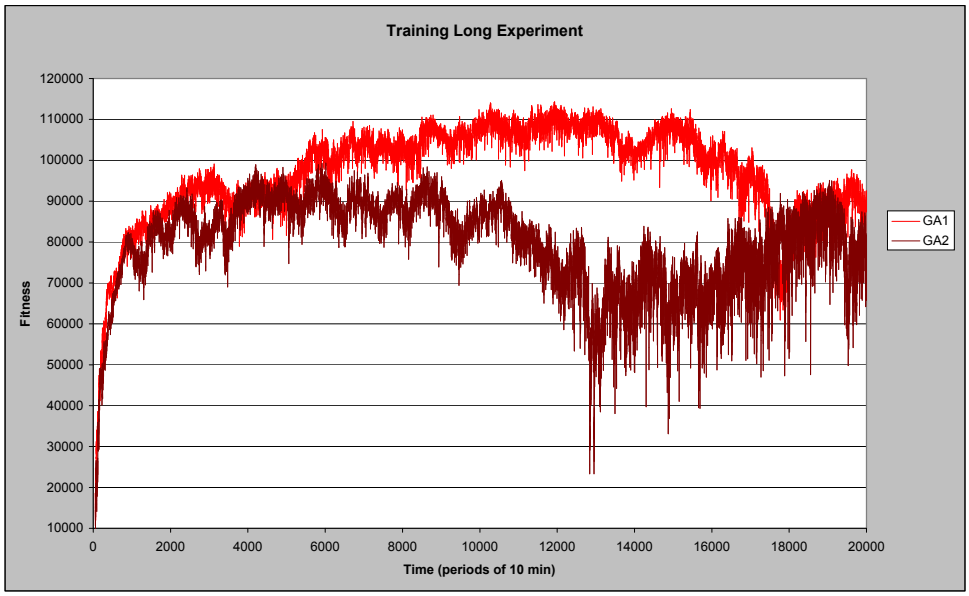

Figure 4.6: Degenerating AI, fitness of the AI, GA used for training networks. Track Long.

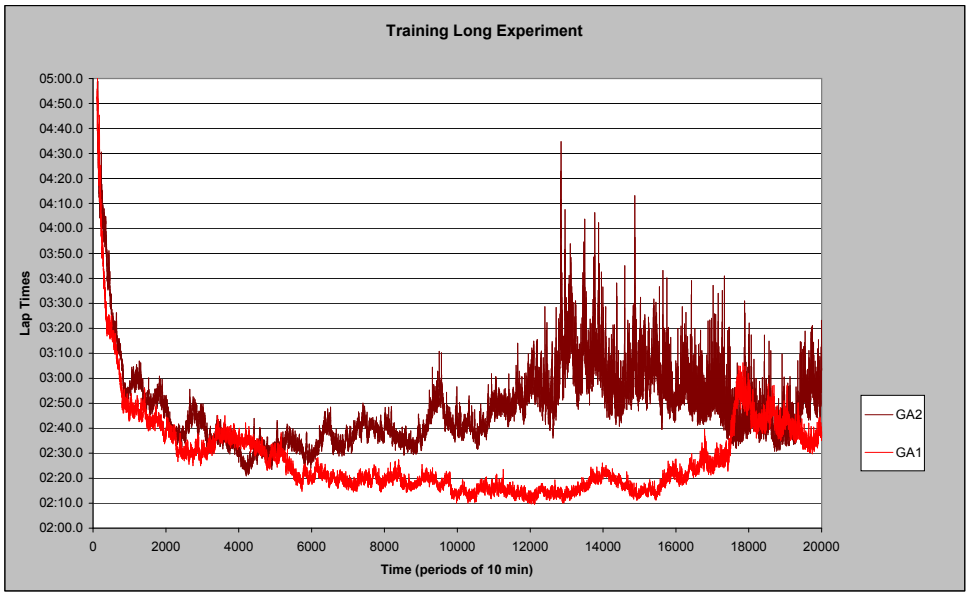

Figure 4.7: Degenerating AI, lap times of the corresponding bikes, GA used for training networks. Track Long.

The fitness tends to increase and lap times tend to decrease; note however from the graphs that the fitness can also decrease from one generation to the next.

The GA makes intensive use of random numbers in the epoch process. Fit individuals are given more chance to reproduce but because random numbers are used in the selection process, it is possible that during one epoch only the worst individuals are selected to reproduce. It is also possible that fit individuals are evaluated as unfit - again the stochastic nature of the experience allows this. In the above graphs, after 10000 generations, the population seems to be slowly degenerating, with individuals after 20000 generations being on average less fit than individuals after 10000 generations.

The degeneration may be due to the perturbation not decreasing with time and the population size being too small. These problems can certainly be minimised using a decreasing perturbation and a larger population size; however increasing the population size would increase the time required to evaluate each population and also it may be difficult to determine how exactly

the perturbation should be decreased with time.

4.3.2 Optimisation

We now experiment with starting the evolution from a population of different ANN's already trained using BP. One major problem with doing crossover with ANN's is that each neuron has a functionality or part of the behaviour (for example turning right), and while doing crossover, the child chromosome may end up having twice the required number of neurons for a given functionality (turning right) and no neurons for another functionality (turning left). An attempt has been made to reorder neurons in the parent ANN's, according to similarities and apparent functionalities, before performing crossover, in order to reduce this problem. This proved not to be successful, and ANN's generated by the crossover of two different ANN's still produced bad random behaviours. Eventually, because of elitism, and because crossover is not always performed, the population converges towards one individual ANN, which is not always the best one, and diversity in the population is lost.

One way to solve the problem is to start with one individual ANN already trained with the backpropagation algorithm, and mutate it to generate a starting population of differently mutated individuals. This proved to be very successful.

An extension of this is to start with many different individuals, give a unique identifier corresponding to a species to each individual, then only perform crossover between two individuals if the two individuals have the same identifier or belong to the same species. During the first epoch, no crossover can take place because all individuals are different and belong to different species; then, because of elitism and mutation, more than one individual from the same species are present in the population, and more and more crossovers can take place between slightly different individuals from the same species, until eventually all individuals in the population belong to the same species. This proved to be even more successful, and this is the technique we now use in this section. In the continuation, we will call each species a sub AI.

Some training data is created by having the author playing the game on 5 different tracks, including track Long, for a total of 51 minutes (308421 samples). 16 ANN's are trained using $\frac{1}{16}$ of the training set and 4 ANN's are trained using the full training set. Four ANN's are trained using the same entire training set because the random weight initialisation means that after training, each sub AI will behave slightly differently from the others.

These 20 trained ANN's are used as a starting population. After a few generations, one ANN eventually wins and all individuals in the population are mutated versions of this winning individual.

Below are the graphs (Figures 4.8 and 4.9) showing the fitness and lap times of ANN's optimised using the GA to ride motorbikes on track Long during a long time, nearly 21 days. The population size was 20. Two experiments have been carried out, both with elitism set to 4.

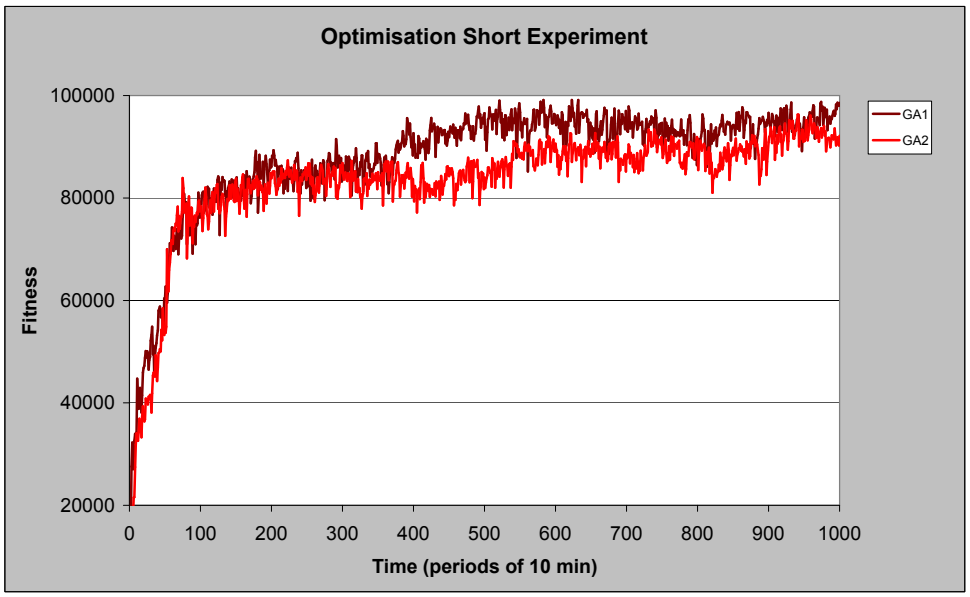

Figure 4.8: Fitness of the AI, GA used for optimising networks. Track Long.

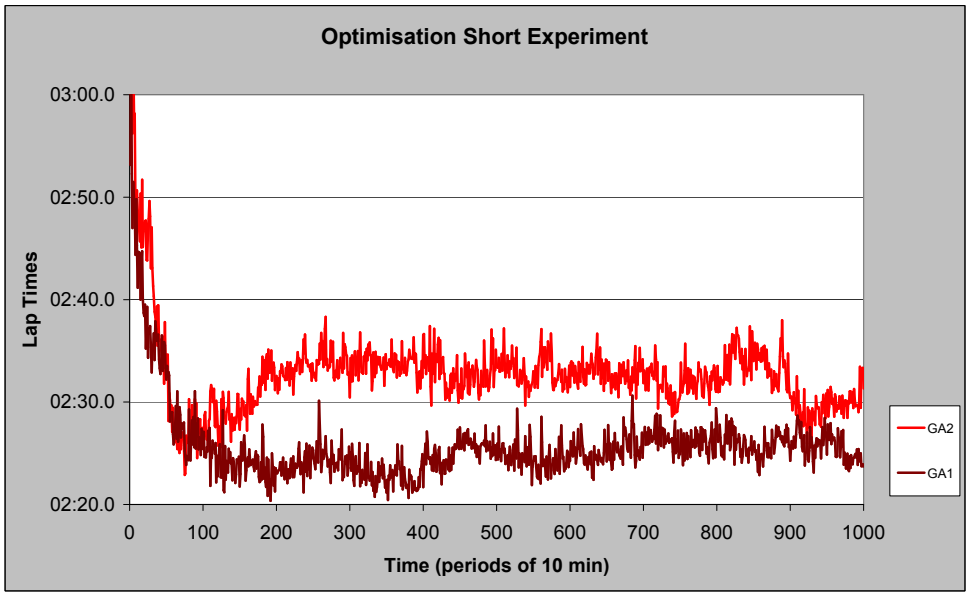

Figure 4.9: Lap Times of the corresponding bikes, GA used for optimising networks. Track Long.

We see that the optimisation is working. The average fitness of the population increases from approximately 15000 to 92000. The average lap times decreases from 3 minutes 5 seconds to 2 minutes 22 seconds, faster than an ANN trained using the BP and training data from track Long (2 minutes 26 seconds). The average performance at the beginning of the evolution is low because the population initially contains many individuals trained to ride motorbikes on very different tracks than track Long. The two experiments have slightly different outcomes: in the first experiment ANN with identifier 9 was the winner, in the second experiment ANN with identifier 6 was the winner.

Some longer optimising experiments are carried out (Figures 4.10 and 4.11) in order to investigate whether the simulations give stable results or if there would be the degeneration effect which we saw previously.

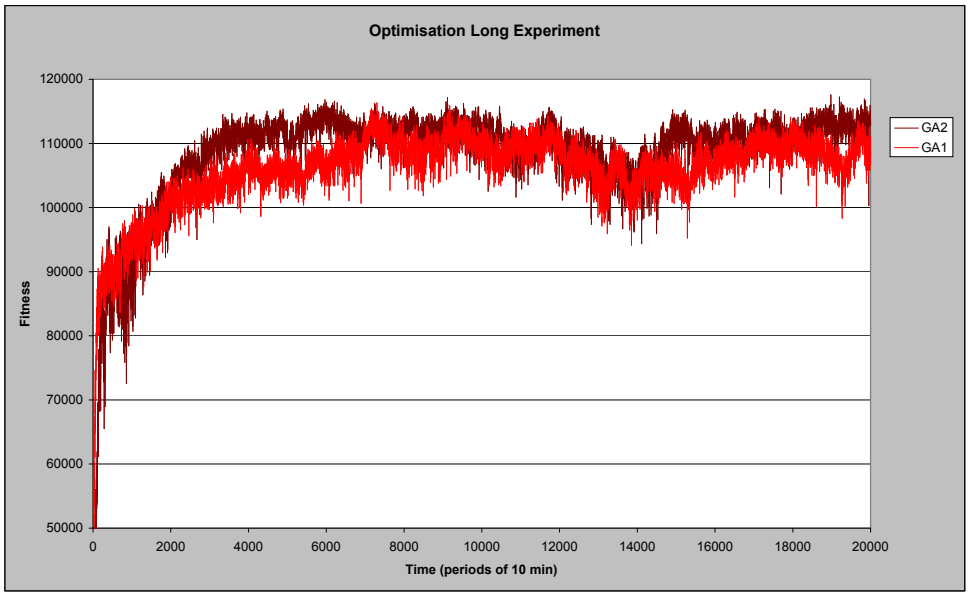

Figure 4.10: New experiment, fitness of the AI, GA used for optimising networks. Track Long.

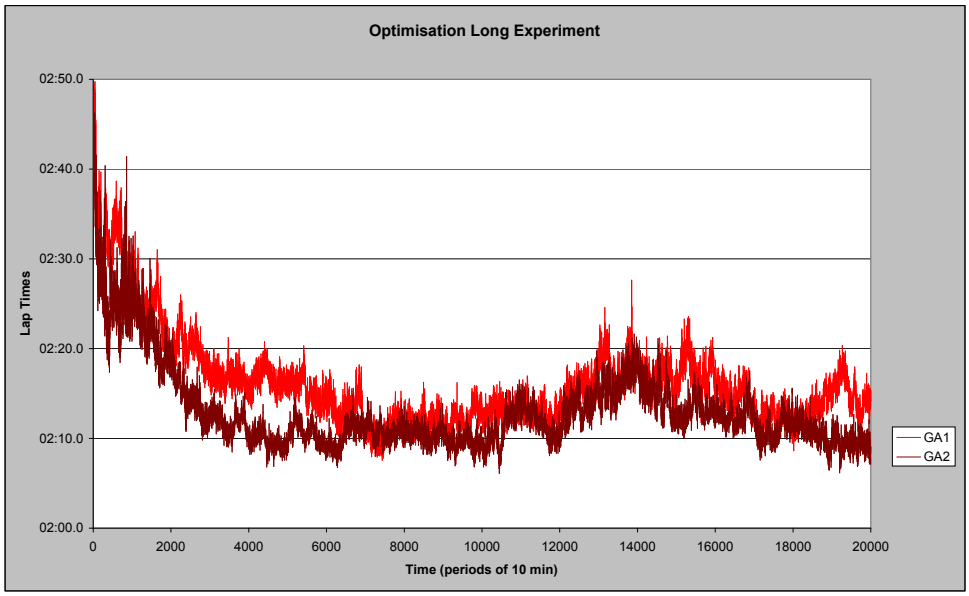

Figure 4.11: New experiment, lap Times of the corresponding bikes, GA used for optimising networks. Track Long.

The experiment is successful, with now bikes performing faster than when the BP algorithm was used to train the networks, 2 minutes 14 seconds and 2 minutes 9 seconds per lap on average at the end of the experiment compared to 2 minutes 26 seconds when the BP algorithm was used to train the network. Also there seems to be less degeneration this time.

4.3.3 Conclusion

The Genetic Algorithm is an optimisation process, which can be used to train ANN's, but it takes a long time for the evolution to take place and the individuals produced are not as fit as individuals produced using a supervised learning technique like the Backpropagation Algorithm.

The GA seem to work better at optimising a population of already trained ANN's. The optimisation takes a long time but at the end the average lap times are better than lap times from individuals trained using only the Backpropagation Algorithm.

New long experiments showed slightly different results, with the population slowly degenerating during training and good performance during optimising.

One big advantage in this training technique is that the technique can produce individuals performing better than a human player.

One disadvantage of this technique is that it takes a long time to perform. Another disadvantage is that the evolution is not very predictable and the evolution can sometimes fail to function properly.

4.4 Ensemble Methods

4.4.1 Bagging

Above, the ANN was trained using training data made from a recording of the game being played by a good human player. The data was made from the recording of the author playing the game on many different motocross tracks. We now investigate whether bagging can improve the AI's performance. As stated earlier, each bag contains the same number of samples as the original data set but the samples are drawn randomly with replacement from the data set. The number of bags is the only parameter in this method: we have typically used 10 bags in the experiments below.

Some early experiments were done using ten ANN's. The early experiments showed that:

1. With $w = 0$, the combined output was a smooth output, and the computer controlled bikes tended to ride in a slow but safe manner.

2. With $w = 1$ (similar to "bumping" [Heskes 1997]), the combined output was a decisive output and the computer controlled bikes tended to ride in a fast but risky manner.

These early experiments showed that this w parameter can allow us to adjust the behaviour of a computer controlled bike, and adjust the performance so that it can match that of the human player.

More experiments were carried out; eight ANN's are trained on eight separate bags from training set 920 and another ANN is trained using the entire training set. Equation 3.15 is used to combine the outputs of the eight

ANN's with various values for w. The results are summarised in the following table (4.1) and graphs (4.12).

AI	Description	Lap Time	Fitness
0	trained on one bag	02:40.3	41378
1	trained on one bag	02:34.9	43609
2	trained on one bag	02:25.5	56529
3	trained on one bag	02:35.8	48222
4	trained on one bag	02:39.7	37439
5	trained on one bag	02:35.5	33523
6	trained on one bag	02:29.4	57035
7	trained on one bag	02:35.1	46433
8	average decision, AI 0 to 7	02:35.4	45321
9	trained on entire set	02:30.8	51908
N/A	average performance, 0 to 7	02:34.5	45521
0b	av{0,7}*1.000+win{0,7}*0.000	02:32.2	48404
1b	av{0,7}*0.889+win{0,7}*0.111	02:31.6	45300
2b	av{0,7}*0.778+win{0,7}*0.222	02:31.2	48621
3b	av{0,7}*0.667+win{0,7}*0.333	02:28.8	52974
4b	av{0,7}*0.556+win{0,7}*0.444	02:30.3	48330
5b	av{0,7}*0.444+win{0,7}*0.556	02:30.7	44299
6b	av{0,7}*0.333+win{0,7}*0.667	02:34.6	42506
7b	av{0,7}*0.222+win{0,7}*0.778	02:36.5	35694
8b	av{0,7}*0.111+win{0,7}*0.889	02:46.4	25961
9b	av{0,7}*0.000+win{0,7}*1.000	02:38.8	33743

Table 4.1: Bagging, training set 920

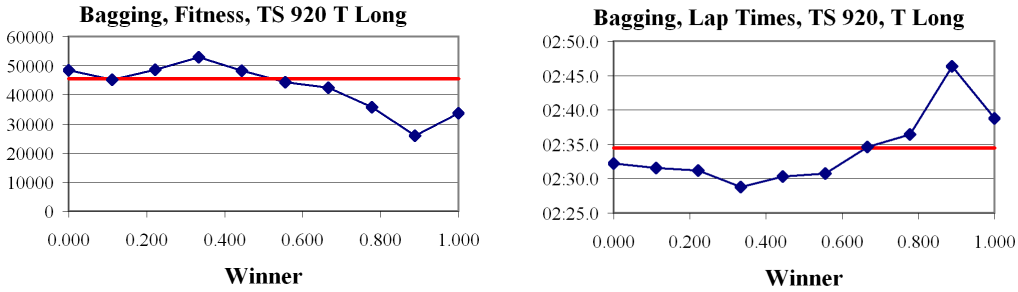

Figure 4.12: Bagging, training set 920. The red line represents the average performance of the first 8 AI's trained on separate bags and the dark blue line represents the performance of the combined 8 AI's, with the winning parameter w varying from 0 to 1.

The tracks used in Training Set 920 are shown on Figure A.1.

The evaluation period in this section is 40 minutes; therefore each laptime and fitness given in this section are average laptime and evaluated fitness of one AI during a 40 minutes evaluation period. In previous sections the evaluation period was 10 minutes. In this section each fitness or score of an AI is evaluated during a 40 minutes evaluation period and then is divided by 4 for easy comparison with the other experiments. Evaluating each AI for a longer time makes the evaluations more accurate.

There are differences in measured performances between the first eight AI's, because each of these AI's is trained using a different bag. There are also differences in measured performances between all AI's because each ANN is initialised using different random numbers, and the racing is not very predictable; fast bikes tend to ride in a risky manner, with a high but not very predictable number of crashes during a race.

As an example, in the previous table, the performance of AI 8 should match exactly the performance of AI 0b; however the performances are different.

One way to obtain more accurate and reliable results is to carry out experiments with a higher number of AI's, for a longer time.

We can see that most AI's trained on bags do not perform as well as one AI trained on the entire training set. The average lap time for AI's 0 to 7 is 2 minutes 35.4 seconds and the performance for an AI trained on the entire training set is 2 minutes 30.8 seconds. When combined with equation

3.15, the combined decisions of AI's 0 to 7 can perform better than an AI trained on the entire training set; with w set equal to 0.333, the average lap time is 2 minutes 28.8 seconds, 2 seconds faster than one AI trained on the entire training set, and 5.7 seconds faster than the average performance of the combined AI's.

The same experiment is carried out with training set Long, containing only training samples from track Long. The results are summarised in the following table (4.2) and graphs (4.13).

AI	Description	Lap Time	Fitness
AI	Description	Lap Time	Fitness
0c	trained on one bag	02:20.7	67265
1c	trained on one bag	02:23.7	63609
2c	trained on one bag	02:21.3	70278
3c	trained on one bag	02:24.6	64660
4c	trained on one bag	02:18.2	74336
5c	trained on one bag	02:18.7	70899
6c	trained on one bag	02:31.2	52389
7c	trained on one bag	02:17.6	63342
8c	average decision, AI 0c to 7c	02:16.3	76832
9c	trained on entire set	02:22.2	71699
N/A	average performance, 0c to 7c	02:22.0	65847
0d	av{0c,7c}*1.000+win{0c,7c}*0.000	02:23.5	67466
1d	av{0c,7c}*0.889+win{0c,7c}*0.111	02:16.5	75986
2d	av{0c,7c}*0.778+win{0c,7c}*0.222	02:19.1	73748
3d	av{0c,7c}*0.667+win{0c,7c}*0.333	02:17.5	78263
4d	av{0c,7c}*0.556+win{0c,7c}*0.444	02:15.1	78328
5d	av{0c,7c}*0.444+win{0c,7c}*0.556	02:18.8	70001
6d	av{0c,7c}*0.333+win{0c,7c}*0.667	02:23.8	63756
7d	av{0c,7c}*0.222+win{0c,7c}*0.778	02:28.8	56841
8d	av{0c,7c}*0.111+win{0c,7c}*0.889	02:34.2	51154
9d	av{0c,7c}*0.000+win{0c,7c}*1.000	02:40.7	41184

Table 4.2: Bagging, training set Long

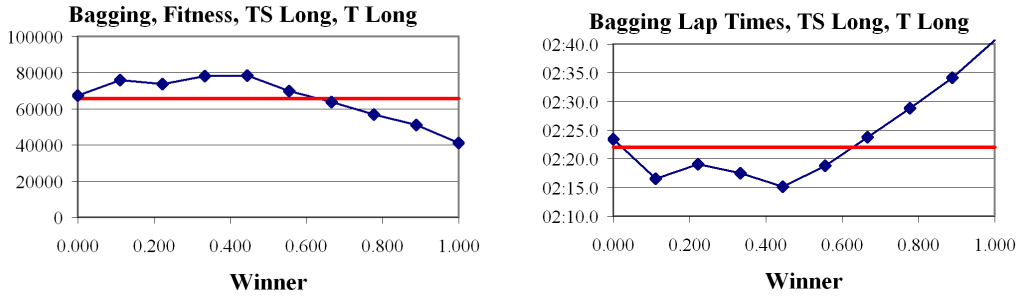

Figure 4.13: Bagging, training set Long. The red line represents the average performance of the first 8 AI's trained on separate bags and the dark blue line represents the performance of the combined 8 AI's, with the winning parameter w varying from 0 to 1.

With training set Long, on average AI's trained on bags perform as well as one AI trained on the entire training set (approximately 2 minutes 22.0 seconds lap time); this is because the training set is very large (270218 samples) and contains more than enough samples to train the ANN's; the recording rate is high (100 samples a second), with very small differences between two adjacent samples; a bike travelling at 100 km/h would only travel 27.78 cm in $\frac{1}{100}$ of a second.

The combined decisions of AI's 0c to 7c can perform better than an AI trained on the entire training set; with w set equal to 0.444, the average lap time is 2 minutes 15.1 seconds, 6.9 seconds faster than an AI trained on the entire training set, and 6.7 seconds faster than the average performance of the combined AI's.

Finally, we want to experiment with training set 920 in order to investigate whether bagging itself is really having an effect. We do this by having each of the eight AI's trained on a separate $\frac{1}{8}$ of the training set and comparing the results with the bagged AI's. The results are summarised in the following table (4.3) and graphs (4.14).

AI	Description	Lap Time	Fitness
0e	trained on 1st 1/8 of set	03:01.4	27532
1e	trained on 2nd 1/8 of set	03:01.4	29659
2e	trained on 3rd 1/8 of set	03:15.8	13884
3e	trained on 4th 1/8 of set	04:22.8	-2142
4e	trained on 5th 1/8 of set	02:57.6	25356
5e	trained on 6th 1/8 of set	02:33.1	48289
6e	trained on 7th 1/8 of set	03:05.9	8980
7e	trained on 8th 1/8 of set	04:32.8	-46355
N/A	average performance, 0e to 7e	03:21.3	13150
N/A	average performance, 0 to 7	02:34.5	45521
0f	av{0e,7e}*1.000+win{0e,7e}*0.000	02:40.7	44195
1f	av{0e,7e}*0.889+win{0e,7e}*0.111	02:51.2	30528
2f	av{0e,7e}*0.778+win{0e,7e}*0.222	02:44.3	32116
3f	av{0e,7e}*0.667+win{0e,7e}*0.333	02:46.6	24488
4f	av{0e,7e}*0.556+win{0e,7e}*0.444	02:59.0	20132
5f	av{0e,7e}*0.444+win{0e,7e}*0.556	03:04.9	3175
6f	av{0e,7e}*0.333+win{0e,7e}*0.667	03:29.9	-10961
7f	av{0e,7e}*0.222+win{0e,7e}*0.778	03:43.8	-23237
8f	av{0e,7e}*0.111+win{0e,7e}*0.889	04:08.2	-34259
9f	av{0e,7e}*0.000+win{0e,7e}*1.000	04:42.5	-49772

Table 4.3: No Bagging, training set 920

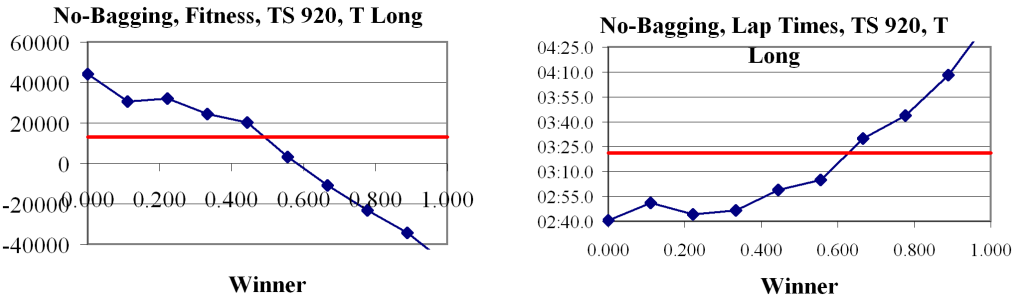

Figure 4.14: No Bagging, training set 920. The red line represents the average performance of the first 8 AI's trained on separate subsets and the dark blue line represents the performance of the combined 8 AI's, with the winning parameter w varying from 0 to 1.

As expected, the average performance of the first eight AI's is lower

than that of AI's trained on separate bags and much lower than that of an AI trained on the entire set. The combined decisions of AI's 0e to 7e can perform better than each individual AI, but not as well as an AI trained on the entire training set; with w set equal to 0, the average lap time is 2 minutes 40.7 seconds, 9.9 seconds slower than an AI trained on the entire training set, but 40.6 seconds faster than the average performance of the combined AI's.

Discussion

Bagging allows us to improve the performance of an AI, by combining outputs from a number of different ANN's, at the expense of more processing requirements. The signal has to be propagated through more than one ANN.

The AI is also adjustable; it may be possible to propagate the signal through various numbers of ANN's and combine the outputs, hence adjusting the performance and the the processing power required by the AI. This can be useful so that the AI can adapt to the performance of a human player, for maximum playing enjoyment, and also prevent the game from slowing down, by reducing at times the processing required by the AI when other parts of the computer program require processing.

4.4.2 Boosting

The first algorithm which we tried was not efficient; this algorithm stored an average error value for each of the training samples, which is not memory efficient, and made use of roulette wheel selection to choose the samples according to the average error, which is not efficient in terms of processing time. There was a time during which the average errors were computed, and a time during which the average errors were used. The algorithm was complicated, was making use of many parameters and was hard to tune.

It appeared that because the learning rate is low, the ANN does not change very much with time, and it is possible to use the instantaneous error, and not the average error. Instead of selecting samples according to the error the sample produces, it is possible to select samples randomly, evaluate the error, and modify the instantaneous learning rate according to this instantaneous error. This new algorithm is much more efficient and produces very similar results.

It was assumed that boosting would be beneficial, however we also

want to experiment with anti-boosting. The instantaneous learning rate γ_i is calculated as:

$$\gamma_i = \gamma_o * \beta_i; \tag{4.6}$$

with γ_o the original learning rate and β_i the instantaneous boosting which is calculated as:

$$\beta_i = e^{\beta_o * abs(\epsilon_i)/\epsilon_a}; \tag{4.7}$$

with β_o the base boosting parameter, varying between -1 and 1, ϵ_i the instantaneous error and ϵ_a the average error over all samples. The instantaneous error is the difference between the desired output and the output given by the ANN. The average error over all samples is calculated incrementally as:

$$\epsilon_a = \epsilon_a * (1 - \alpha) + abs(\epsilon_i) * \alpha; \tag{4.8}$$

with α a small value (0.01).

The experiments were done using training set Long, with the boosting parameter varying between -1 and 1. The performance was measured on track Long. The results are shown in the following graphs (4.15):

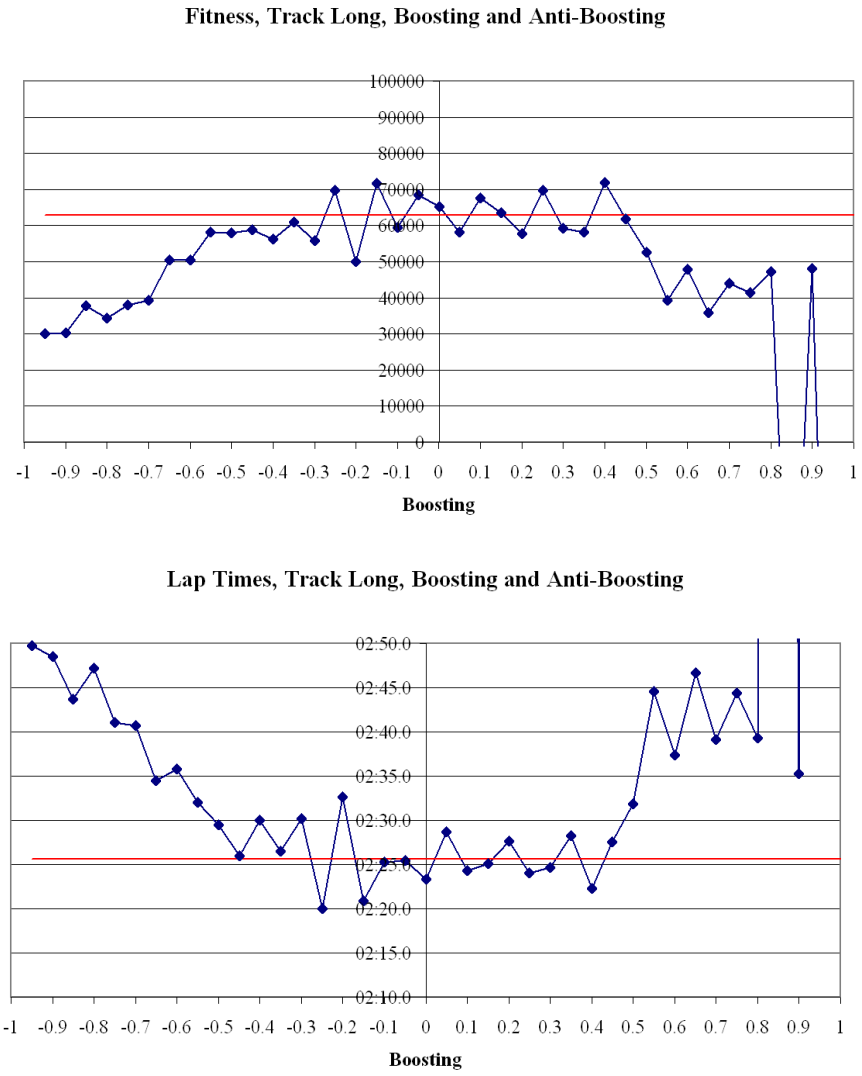

Figure 4.15: Boosting {-1,1} The red line represents the average performance of AI's trained without any boosting or anti-boosting. The dark blue line represents the performance of an AI with the boosting parameter β_o varying from -1 to 1.

The performance does not vary smoothly with the boosting parameter. It seems that the performance may be increased with a boosting parameter somewhere in the range -0.3 to -0.1. This is investigated in the following graphs (4.16):

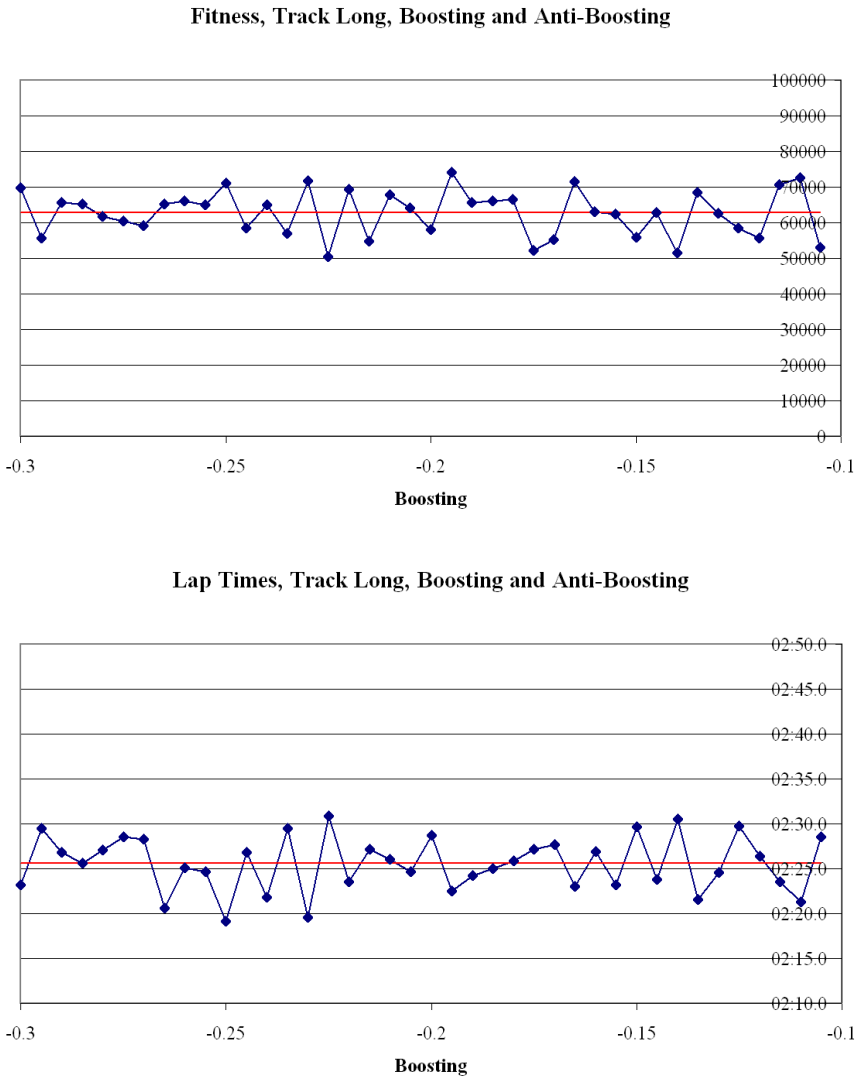

Figure 4.16: Boosting {-0.3,-0.1} The red line represents the average performance of AI's trained without any boosting or anti-boosting. The dark blue line represents the performance of an AI with the boosting parameter β_o varying from -0.3 to -0.1.

What could have been peaks in performance proves to be nothing but noise due to ANN initialisations and the inherent stochasticity within the racing.

Discussion

Boosting is a method designed to emphasise data samples which are more difficult than most to classify or regress on. We have investigated boosting in the context of the motocross game but have been disappointed with the results. Boosting and anti-boosting prove to have a negative effect on the training of the AI; they also add complexity to the training algorithm. Therefore we do not recommend their use for computer games.

4.4.3 Conclusion

We have experimented with bagging and boosting and found neither to be very attractive in the context of this computer game, **Motocross The Force**. In the literature, it is said that bagging reduces error by tackling the variance in a predictor while boosting reduces error by tackling the bias in a predictor. The poor results from boosting in particular seem to suggest that our base predictor, the multilayered perceptron, is indeed powerful enough to learn how to control the bikes in the motocross game. This further strengthens our belief that a single layer of hidden neurons is sufficient for this game.

The results from bagging are more disappointing: we *do* find that the bagged AI predictors are better than predictors trained on only a part of the training set, however even combining these predictors is not more powerful than training an AI on the whole data set and the latter method is less computationally intensive.

Therefore our overall conclusion is that ensemble methods do not seem to be appropriate for this particular game and we conjecture that this will be true for similar games. However this conjecture must be empirically tested by other researchers.

4.5 The Cross Entropy Method

For CE, we want to use a Gaussian distribution, and estimate the mean and variance of the elite individuals weights as detailed in equations (3.33) and (3.34). According to these equations there is a requirement to maintain a population of elite individuals; the New Evolution Technique introduced in Chapter 6 already maintains a population of elite individuals.

Our CE implementation therefore is identical to our NE implementation, except the creation of new individuals. In NE and GA, the creation of new individuals involves:

1. Select two members from the current population. The chance of being selected is proportional to the chromosomes' fitness.

2. With probability, C_r, the crossover rate, cross over the numbers from each chosen parent chromosome at a randomly chosen point to create the child chromosomes.

3. With probability, M_r, the mutation rate, modify the chosen child chromosomes' numbers by a perturbation amount.

For our CE implementation, the creation of new individuals involves equations (3.33) and (3.34).

There is a time during which the population has not yet converged towards one individual; the population contains individuals with different ID's. We know from our experience with the GA that creating child individuals from parents with different ID's (different species) generally produces totally unfit individuals; therefore when the population has not yet converged, equations (3.33) and (3.34) are not used to generate child individuals; only mutation applied on two parents is used to create children.

After the population has converged, the individuals in the population may be very similar and differ only by a few weights; the standard deviation may be non-zero only for a few weights and the CE technique may end up optimising only the few weights which have non-zero standard deviation, and the elite population may not evolve well. To avoid this problem, when a standard deviation is found to be zero, then it can be set to an arbitrary small value. In our implementation, when a standard deviation is found to be zero, then it is set to the average of all originally non-zero standard deviations.

4.5.1 Training

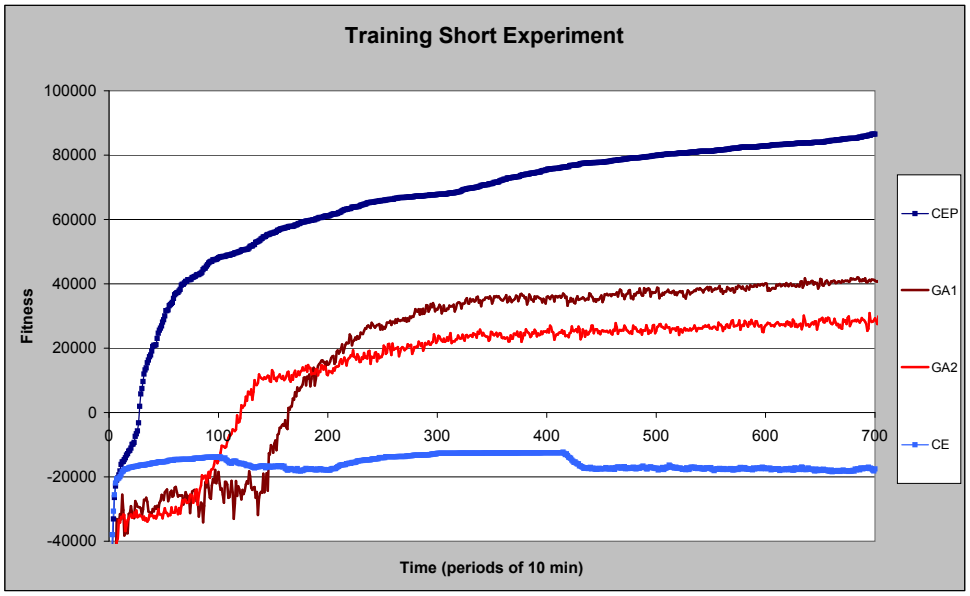

Figure 4.17: Fitness of the AI, Cross Entropy Method used for training networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

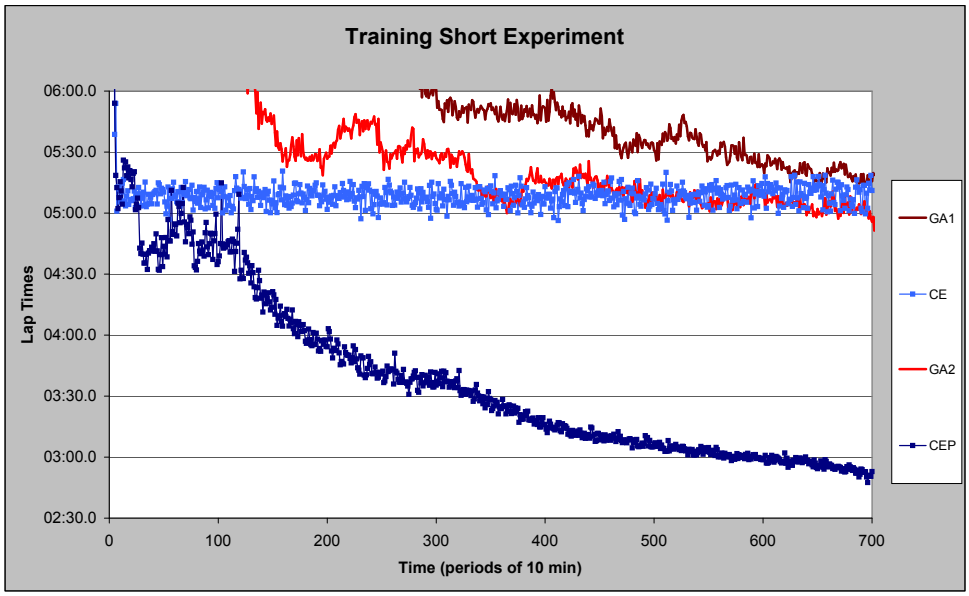

Figure 4.18: Lap times of the corresponding bikes, Cross Entropy Method used for training networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

We can see from these experiments (Figures 4.17 and 4.18) that with the CE Method, the population converges too quickly towards a local optimum. This can be avoided by adding perturbation to the newly created individuals like in the GA. Cross Entropy with Perturbation (CEP) performs better than GA.

Some new longer optimising experiments are carried out in order to investigate the stability of the method.

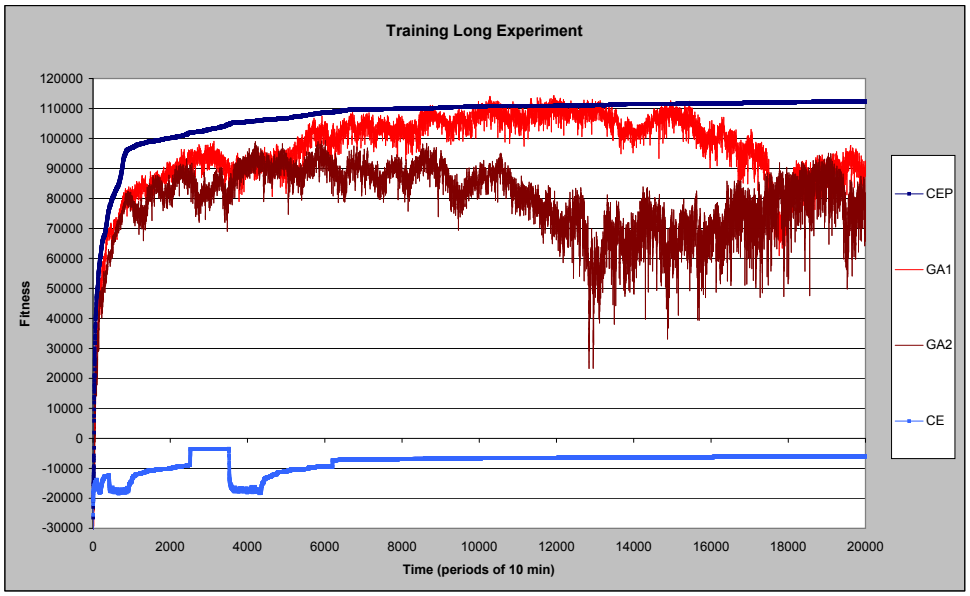

Figure 4.19: New experiment, fitness of the AI, Cross Entropy Method used for training networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

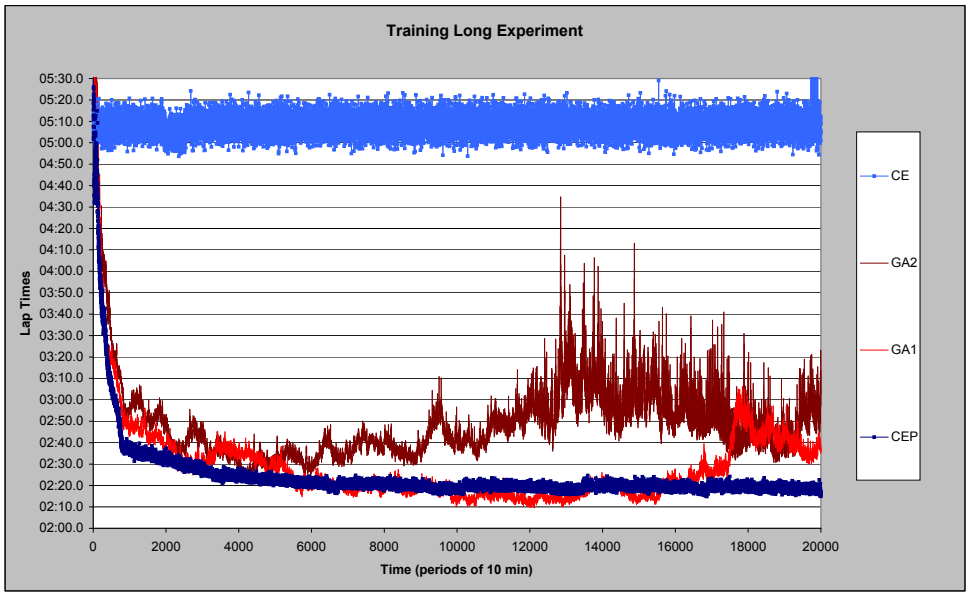

Figure 4.20: New experiment, lap times of the corresponding bikes, Cross Entropy Method used for training networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

We can see from these new experiments (Figures 4.19 and 4.20) that that CEP performed better than GA with the fitness increasing steadily, the lap times decreasing steadily and no degeneration, with average lap times 2 minutes 15 seconds versus 2 minutes 40 seconds for GA.

4.5.2 Optimisation

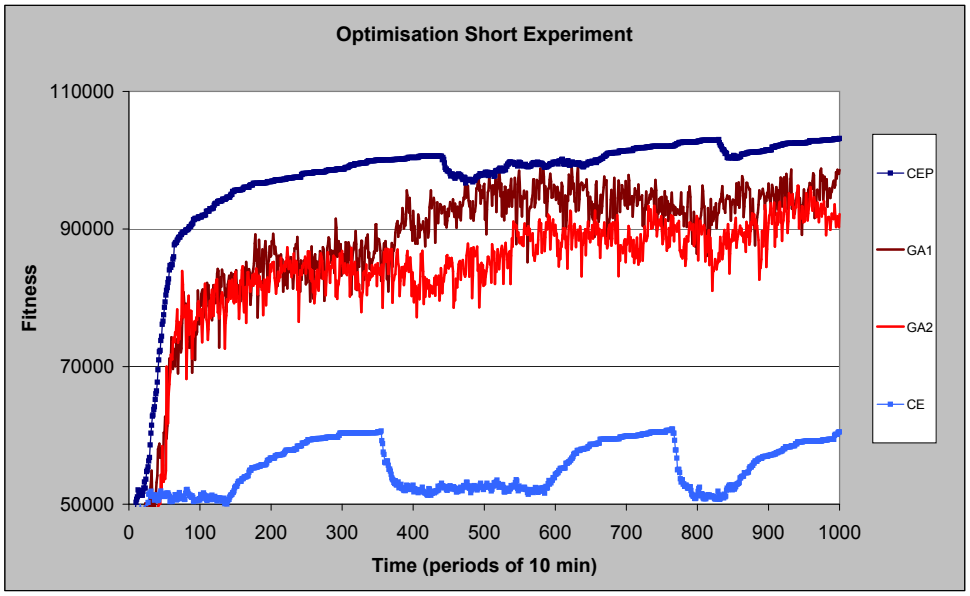

Figure 4.21: Fitness of the AI, Cross Entropy Method used for optimising networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

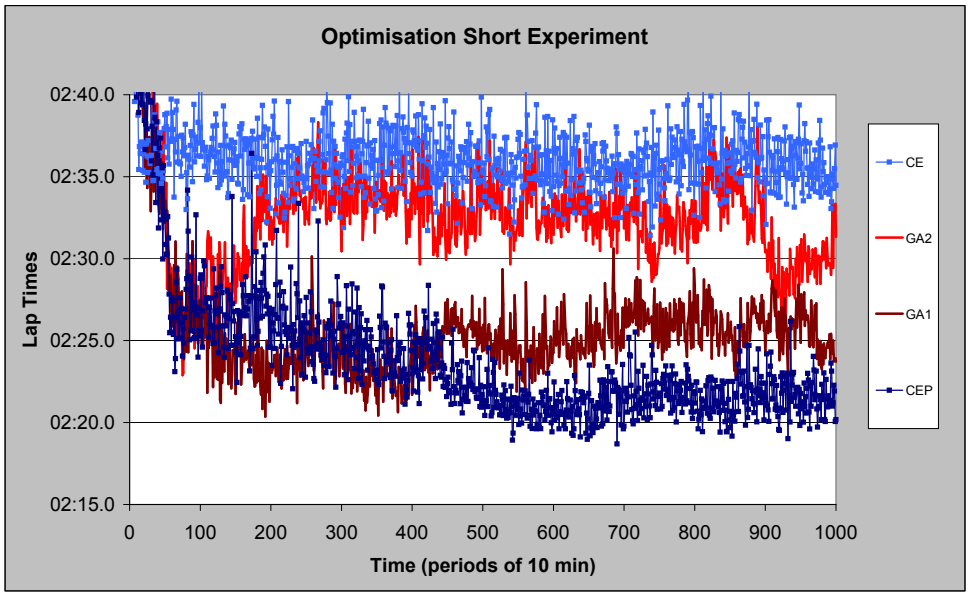

Figure 4.22: Lap times of the corresponding bikes, Cross Entropy Method used for optimising networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

The population converged towards one individual and CE was effectively used, after approximately 20 generations (t=20). This experiment, as shown in Figures 4.21 and 4.22, proved to be successful, with CEP performing better than GA.

Some new longer optimising experiments are carried out.

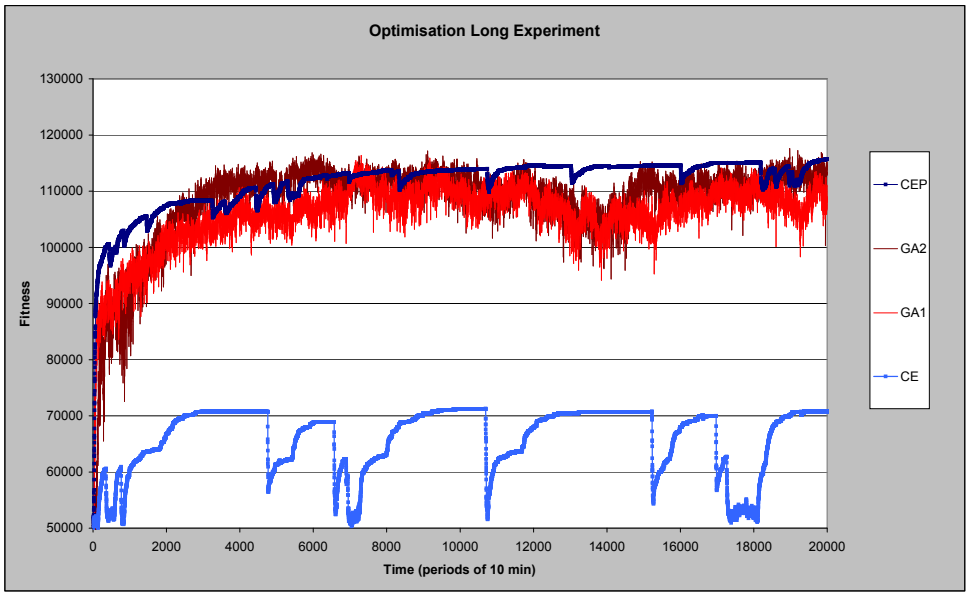

Figure 4.23: New experiment, fitness of the AI, Cross Entropy Method used for optimising networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

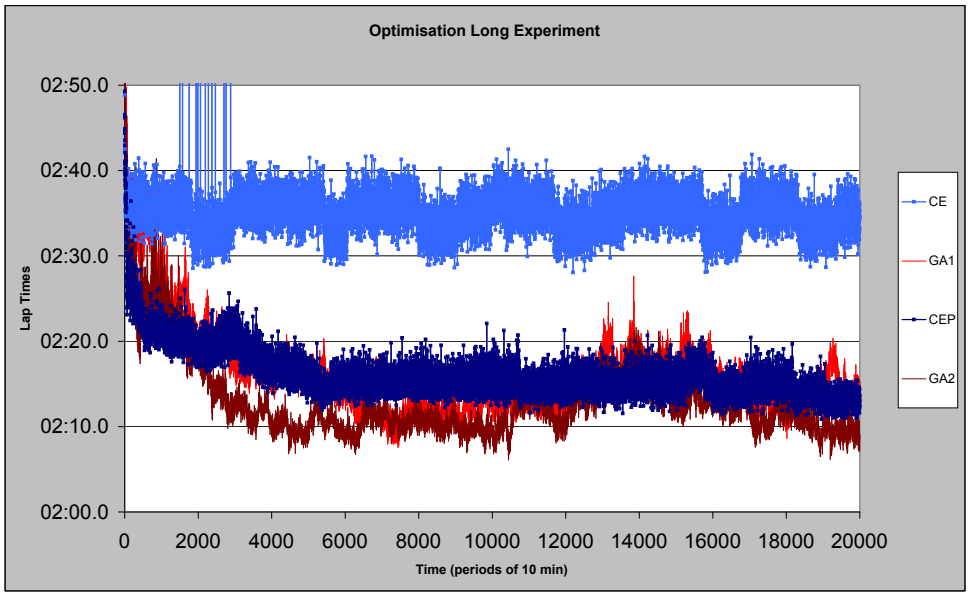

Figure 4.24: New experiment, lap times of the corresponding bikes, Cross Entropy Method used for optimising networks. Cross Entropy in light blue, Cross Entropy with Perturbation in dark blue. Track Long.

We can see from Figures 4.23 and 4.24 that CEP is performing slightly better than GA; there are times when the evolution seems to slightly break (fitness falls sharply) and other times when there seems to be no evolution (fitness constant over a period of time). After the completion of the optimising phase, average lap time is 2 minutes 13 seconds.

4.5.3 Conclusion

Cross Entropy with perturbation performs better and is more steady and predictable than GA. There are still some small breaks in fitness and lap times.

4.6 Symbolic AI

Symbolic AI is the kind of AI traditionally used in the game industry. The game developer fully describes the behaviour of the non-player characters (NPC) with a set of "if, then , else" statements or a finite state machine (FSM), using a programming language.

In the context of the motocross game, the symbolic AI is implemented in a project called Minimal because it is the simplest and more minimalist implementation of an AI and can be used as a starting point to develop a more sophisticated AI DLL for the game.

```
VOID SMinimal::DecisionMaking(SampleData &sampleData,const D3DXMATRIX &ForwardTransform)
{
    //take target waypoint position, 30 metres in front of bike.
    (*s_GetWayPointPosition)(m_Target,sampleData.nCurrentWayPointId,3000.f);

    D3DXVECTOR3 TargetDir;
    Vec3Subtract( &TargetDir, &m_Target, &sampleData.BikePos );
    Vec3Normalize( &TargetDir, &TargetDir);

    D3DXVECTOR3 BikeDir(-sampleData.BikeDir1.x,-sampleData.BikeDir1.y,-sampleData.BikeDir1.z);

    D3DXVECTOR3 Cross;
    Vec3Cross(&Cross, &TargetDir, &BikeDir );
    FLOAT vVelocity = Vec3Length(&sampleData.BikeVel);

    //if bike is facing target.
    if (Vec3Dot(&TargetDir, &BikeDir) > 0.5f)
    {
        //turn towards target.
        sampleData.vLeftX = 1.5f*Cross.z;
        //maintain maximum speed (35m/s), reduce speed if off target.
        sampleData.vRightY = -((3500.f - vVelocity) / 3500.f - fabs(Cross.z));
    }
    else    //bike is not facing target
    {
        if (Cross.z > 0)    sampleData.vLeftX = 1.f;    //if bike is facing left, then turn right.
        else                sampleData.vLeftX = -1.f;   //if bike is facing right, turn left.

        sampleData.vRightY = -((1000.f - vVelocity) / 1000.f);  //maintain speed (10m/s).
    }

    sampleData.vLeftY = 0.f;    //do not rotate bike.
    sampleData.vRightX = 0.f;
    m_Target.z += 100.f;        //rise target 1 metre above ground debug purposes.
}
```

Figure 4.25: AI Class SMinimal, method DecisionMaking.

Figure 4.25 shows some code from the symbolic AI class: we clearly see that it follows the "if, then , else" template described above. The code works as follows:

- a target waypoint position is obtained from the game engine, 30 metres in front of the bike.
- the bike and target direction are calculated.
- the velocity is calculated.
- dot and cross products are calculated between bike and target directions.
- if the bike is facing towards target (the dot product between bike and target directions is more than 0.5).
  - turn towards target, according to cross product.
  - maintain high speed (35m/s), reduce speed if off target.
- else (bike is not facing towards target)
  - maximum turn towards target.
  - maintain low speed (10m/s).
- set (optional) rotation controls to zero.

Using this very simple code, the average lap time on track long is 2 min 48 seconds; it is faster than ToPoE and SOM implementations (3 min 5 seconds and 3 minutes 11 seconds), but slower than a MLP network (2 minutes 26 seconds).

One reason why this implementation does not perform as well as a MLP network is that it takes only one waypoint position to make the decision, as opposed to 13 waypoint positions for the MLP implementation. The AI makes decisions using a lot less information than other implementations. It would be possible to make the AI take more information as input, however the code would become complicated and would require setting many internal parameters; the resulting AI would be hard to adjust and maintain.

It would also be possible to add special markers along the track, to help the AI by telling it what must be the target velocities on portions of the track; this would be easy to implement but would require additional work

from the track designers and would require to be replicated for each new track.

Using evolutionary techniques, it would be possible to have the AI evolve with time as in Sections 4.3.2 and 4.5.2 by evolving the internal parameters. Symbolic AI would have less internal parameters than a MLP networks and as a result would be faster to optimize.

4.7 Conclusion

The experiments with SOM, RBF, and ToPoE networks prove not to be very successful. The bikes are not able to make decisive decisions in terms of accelerating and turning, specially RBF networks, and are performing poorly compared to bikes controlled by MLP networks. A major problem with these kinds of networks is that their internal operations prevent them from differentiating between important and not so important inputs. All inputs to the networks are considered as equally important by these networks if the inputs are normalised, and large inputs are considered more important than smaller inputs if inputs are not normalised. This is different from MLP networks, where neurons in the hidden layer are connected with connections of various strength to the neurons in the input layer, hence allowing the networks to evaluate the difference between important and unimportant inputs, even if inputs are not normalised.

The experiments with Ensemble Methods (Bagging and Boosting) prove also not to be very successful.

Cross Entropy with perturbation performs better and is more steady and predictable than GA.

A simple symbolic AI performs better than SOM, RBF, and ToPoE networks but not as well as MLP. A more complicated symbolic implementation taking more information to make decisions would likely perform better but would be more difficult to implement and maintain.

The experiments are summarised in the following table[1][2][3]:

AI	Lap Time	Training Time
Good Human Player	02:15	Many hours
MLP trained using BP and training data[1]	02:26	A few seconds
Kohonen SOM trained using training data[1]	03:11	A few seconds
RBF trained using training data[1]	03:15	A few seconds
ToPoE trained using training data[1]	03:05	A few seconds
MLP trained using GA	02:40	3333 hours
MLP optimised using GA[2]	02:22	3333 hours
MLP trained using train. data[1] and Bagging	02:15	A few seconds
MLP trained using t. data[1] and Boosting	02:26	A few seconds
MLP trained using CEP	02:15	3333 hours
MLP optimised using CEP[2]	02:13	3333 hours
Symbolic AI	02:48	None
MLP trained using NE[3]	02:07	3333 hours
MLP optimised using NE[2][3]	02:05	3333 hours

Table 4.4: Experiments summary, track Long

[1]The training data is made from the recording of the good human player playing the game.

[2]Optimisation processes start with an MLP originally trained using BP and a training data.

[3]New Evolution technique introduced in Chapter 6.

Chapter 5

AI SDK

In the research community, it happens often that one researcher presents a new algorithm or AI or data mining technique, and presents the technique in the context of one particular problem or application. The applications used are often very different. It appears that there is a need in the research community for common platforms to evaluate and benchmark individual AI techniques. This has been discussed during various conferences at which papers ([Chaperot and Fyfe 2005],[Chaperot and Fyfe 2006*b*],[Chaperot and Fyfe 2006*a*] and [Chaperot and Fyfe 2007]) were presented.

For data mining it is common practice to use standard datasets to evaluate and compare different classification techniques (see for example [Michie et al. 1994]). For video games, there seems to be a need for more common platforms to compare AI techniques.

Splitting the motocross game, between the game engine on one side and the AI on the other side, allows for easier AI code maintenance and implementation, and changes the game into a common open platform for many developers and researchers to test and compare different techniques. The game and SDK have been released and are now available for download at the following address [Chaperot 2009]:

`http://cis.uws.ac.uk/benoit.chaperot`

In the following sections we detail how we have split the game, and how to implement new AI for the game.

5.1 Original Game Architecture

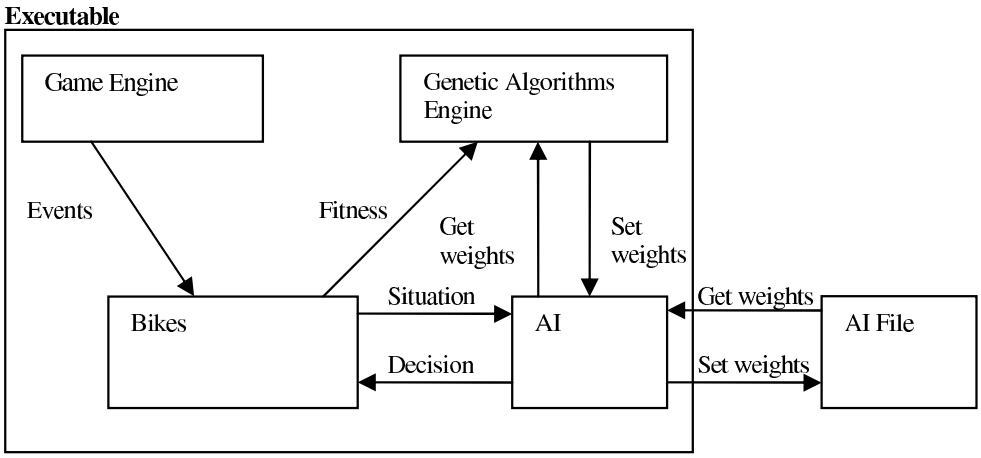

Figure 5.1: Original Game Architecture

The original game architecture is shown in Figure 5.1. We see that the game engine and the AI are contained inside the same executable.

1. The game is composed of one executable and some data files.

2. The AI source code is in the middle of the game engine code; this makes it difficult to read and maintain.

3. There are some intellectual property issues in that only the authors can implement an AI for the game.

4. Only one kind of AI can be used at a time in the game to control motorbikes. Each motorbike can use its own data file. It is not possible to directly compare AI techniques.

5.2 New AI SDK Architecture

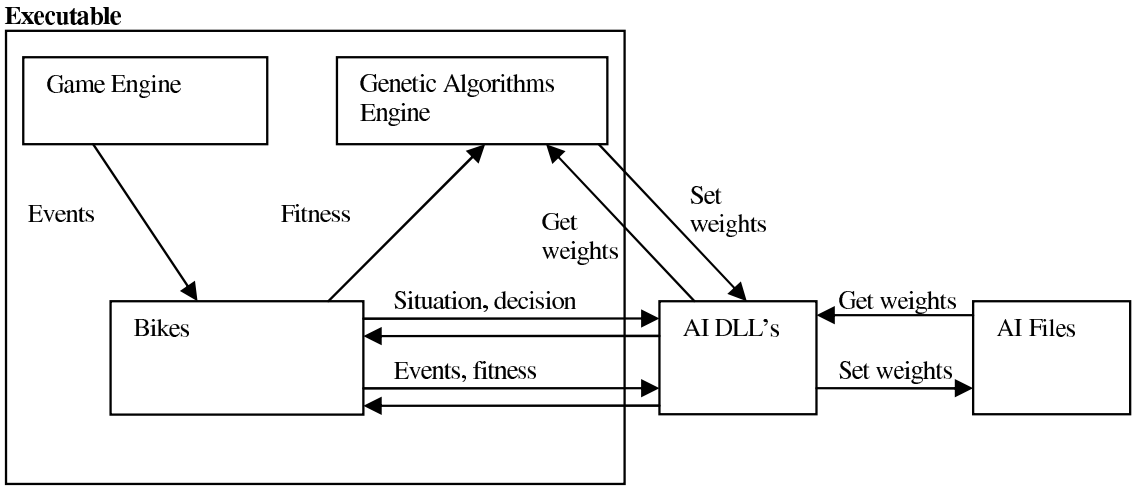

Figure 5.2: Motocross The Force AI SDK Architecture

The new game architecture is shown in Figure 5.2. The game engine is contained inside an executable, and the AI is contained inside separate DLL's. The reader can easily compare points 1 to 4 in this section with the previous section for easy comparison of the two architectures.

1. The game is made of one executable, some DLL's and some data files. DLL stands for Dynamic Link Library. Typically DLL's provide one or more particular functions and a program accesses the functions by creating either a static or dynamic link to the DLL. In the context of the new architecture for the motocross game, each DLL implements one kind of AI and is linked dynamically.

2. The AI source code is separated from the game engine code, with one small DLL project per AI; this makes it easy to read, implement and maintain.

3. The AI part is separated from the game engine and is open source. Anyone can implement an AI for the game.

4. Many kinds of different AI's can be used at a time in the game to control motorbikes. Each motorbike can use its own DLL file and its own data file. It is possible and easy to directly compare AI techniques.

The GA has been left inside the executable so that an external AI can have its internal parameters (weights) optimised by the executable. An external AI can also implement its own GA or optimisation technique and not use the optimisation facilities offered by the executable.

5.3 Game Classes and Structures

Before implementing AI for the game, it is important that the user has a good understanding of the various classes and structures in use in the game.

5.3.1 Track

A track is a course over which races are run. Typically a track is of variable width along its course. The tracks are marked using WayPoints.

5.3.2 WayPoint

WayPoints are markers positioned on the centre of the track, every metre along the track, and are used:

- To give course information to computer controlled bikes, i.e. position, direction and width of the track.
- To monitor the performance: a bonus can be given to a computer controlled bike for passing a WayPoint as described in Sections 2.3 and 4.3.

5.3.3 Game Engine

There is one game engine, it is implemented by the executable. It is everything but the AI, and is responsible for updating the simulation.

5.3.4 AI

There is one AI per computer controlled bike. Each AI can be written to or read from an AI data file. Each AI makes use of an AI DLL. More than one

bike can share the same DLL. An AI is trained to make a decision given a situation.

5.3.5 Situation

The situation is the general state of the bike, position, orientation and velocity relative to the ground.

5.3.6 Decision

These are commands and are nearly the same as the controls for the human player:

- turn left/right.
- target velocity.
- rotation of the bike forward/backward.
- rotation of the bike left/right.

5.3.7 SampleData

SampleData is the main structure used for communication between the game engine (executable) and the AI DLL. Typically the game engine fills the situation fields of the structure and passes the structure to the AI DLL; the AI DLL fills the decision fields of the structure, given the situation, and passes it back to the game engine; the game engine updates the state of the corresponding computer controlled bike and the simulation accordingly.

5.3.8 Training Set

A training set is a structure used for the training of AI. A training set is a collection of **SampleData**'s made from the recording of a human player playing the game. Each sample contains a situation and the corresponding human player's decision. AI's are trained to make the same decision as the human player, given a situation. We saw in the previous chapter that there is a conceptual difference between the way the data is used for supervised compared with unsupervised techniques: the supervised techniques use error

between their outputs and the target responses in order to change their parameters to make their outputs more like the target responses in future; the unsupervised techniques perform some form of averaging over the targets to generate an appropriate output.

5.3.9 Terrain

This is a structure used to give ground height information.

5.3.10 Weight

A Weight is an AI parameter that is to be optimised using for example Genetic Algorithms. Typically a weight is a connection strength between two neurons in an ANN.

5.3.11 Genetic Algorithms

Training is considered to be a type of optimisation. GA is used to improve the AI's performance by modifying AI weights. GA is implemented by the executable.

5.4 Implementing New AI

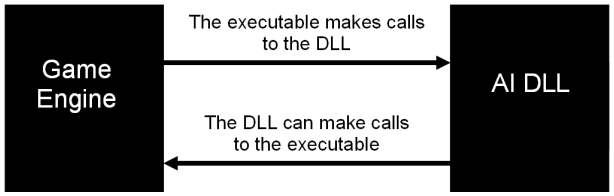

Figure 5.3: Communication between the game engine executable and the AI DLL.

An important feature of the architecture is that the executable calls DLL functions, for example for decision making, but the AI DLL's can also call

executable functions, for example to obtain more information about a situation, before making a decision. It is a two way communication process as shown in Figure 5.3.

5.4.1 DLL Functions

In order to be recognised by the game executable each AI DLL must be placed in a particular folder and implement a set of functions; these functions can be grouped into categories.

General Operation Functions

The general operation functions are:

- Creation: this function is called once before any other DLL function; it creates an AI and returns a void pointer on the newly created AI to the executable.
- Destruction: this function is called once when the program exits; it is responsible for releasing all the memory and resources allocated in the Creation function.
- Decision Making: this function returns a decision to the executable, given a situation.
- Render: this function is called every time the game is rendered; this gives the opportunity to the AI DLL to display AI information; this is mainly used for debugging purposes.

After AI creation, the executable keeps a void pointer to the AI and passes it as a parameter to all subsequent calls to the AI DLL.

Training Functions

These functions are typically used for training the AI using back propagation techniques.

- Generate AI From Training Set: this function is called every time the user wants an AI to learn from a training set. The DLL loads and processes the training set.

- Generate AI From Training Set Update: this function is called once per game update: the DLL updates the training or generation of an AI from a training set; typically there are 25 backpropagation iterations per game update and 100 game updates per second.
- Is Generating AI From Training Set: returns true if the AI is currently training from training set.

Weights Functions

These functions are typically used for training the AI using e.g. genetic algorithms techniques.

- Put Weights
- Get Weights
- Set Generation
- Get Generation
- Save Weights
- Get Number Of Weights

Version Functions

- Get AI Name: returns AI DLL name, one name per AI DLL.
- Get AI Version: returns version of AI implemented.
- Get Debug: returns true if this is a Debug version of AI DLL.

All these functions were found useful to carry out our experiments. To make the architecture simpler, an AI DLL is required to implement all these functions in order to be recognised by the executable. If some functions are not needed (for example the user does not want to use GA), then the user can simply create empty functions.

5.4.2 Executable Functions

The executable makes the following functions available to AI DLL's.

WayPoint Functions

AI DLL's can make calls to the executable to obtain the following interpolated information about WayPoints:

- Transformation matrix (a 4x4 matrix representing a position and an orientation)
- Position
- Direction
- Width

The information is interpolated between WayPoints. The functions take two parameters, a WayPoint index, with one WayPoint every metre along the track, and a distance in centimetres along the track from this WayPoint.

Drawing Functions

AI DLL's can make calls to the executable to draw the following kind of primitives on the screen:

- 3D Vertices
- 3D Lines
- Text
- Rectangles

These functions are used mainly for debug purposes and take a colour as one of their parameters.

Track Functions

AI DLL's can make calls to the executable to obtain information about tracks and terrain:

- Height: a function returns the terrain height at a given position.

- Track creation: the DLL can load tracks; this is useful when processing training sets; a training set can contain training data from more than one track.
- Track unique identifiers: the DLL can check that a track has not changed since the time the training set was generated.

Other Functions

- Get Version: returns version of the executable.
- Forward transform: a function returns a space centred at the origin of the motorbike; the Z axis points up and the Y axis follows the horizontal velocity direction. This space is more convenient than bike space to represent and transform world objects in relation to the bike.

5.4.3 Operation

The new AI system works as follow:

1. The executable loads all DLL's contained in a given directory; if the DLL implements all functions described above, and versions match, then it is kept loaded, otherwise it is unloaded.
2. Each computer controlled bike loads its own AI data file; each AI data file is to be associated with an AI DLL. The association between data files and DLL's is done by matching the name contained in data file header with the names given by the DLL's.
3. The AI is created using the matching AI DLL **CreateAI** function, and all future AI function calls will call the matching AI DLL functions.

5.5 Conclusion

In this chapter, we have reviewed the changes to the structure of the code designed to make this game a suitable testbed for disparate researchers to investigate the use of their particular AI's in the context of this game. We envisage that it is now possible for researchers to implement their own methods in this game and perform comparative studies with the results discussed in the thesis. More details about the SDK can be found in Chapter B.

Chapter 6

A New Evolution Technique

We introduce an improved evolutionary algorithm technique which overcomes some of the deficiencies in the previous methods.

6.1 Presentation

We have seen in Chapter 4 how Genetic Algorithms could be used to train or optimise Artificial Neural Networks to control motorbikes in the motocross game.

There are still problems with this approach:

- Training and optimisation take a long time to perform (typically a few days). It is not desirable that all the improvement in AI performance is lost when the game exits.
- Behaviours that are appropriate on one motocross track may not be appropriate on another track. If an AI spends a long time on a track, it is not desirable that its performance increases only for that particular track and decreases for all other tracks.

To solve these two problems, an AI DLL, using the AI SDK described in Chapter 5, is created. This new AI DLL is called **Evolution** and implements Genetic Algorithm and Cross Entropy Optimisation.

This AI DLL creates:

1. One AI file per AI.

2. One AI file per AI per motocross track.

Each file does not contain information about one AI, but contains information about a population of sub AI's (typically 20 sub AI's per file). Each sub AI is made of:

- Weights, to be optimised, corresponding to connection weights inside an ANN.
- Estimated fitness, to be maximised.
- Number of iterations or length of time the sub AI has been evaluated for.
- Unique identifier (ID), used to only perform crossover between sub AI's with the same ID.

This last item is because crossover between individuals with different ID's (different species) generally produces totally unfit individuals.

The first file is the initial state of the AI, after it has been trained from a training set using the Backpropagation Algorithm, before optimisation. Each sub AI is trained using different bags or portions of the training set, so after the initial training all sub AI's are slightly different from one another.

- Half the sub AI's train on separate and equal size portions of the training set.
- The other half train using separate bags, as described in Section 3.3.1.

All sub AI's are initially given different unique identifiers.

The second file is the current state of the AI; when the game level is loading, the AI attempts to load this second file, so it can continue the evolution on that particular track that was started in a previous game session. If the second file does not exist (no evolution was started on that particular track in any previous game session), the AI loads the first file and starts evolution from the beginning. The AI then saves itself to file using the second file name.

The AI saves itself to file each time there is a change in the population of sub AI's.

The new AI DLL called **Evolution** allows us to experiment with different evolution techniques with one small programming project.

The GA is a generational evolution technique. Each population of chromosomes is replaced by a new population of chromosomes after every generation.

One problem with this technique is associated with the fact that the chance for chromosomes to be selected for reproduction is proportional to the chromosomes' fitness. Random numbers are used and there is a small but non-zero possibility that only the least fit chromosomes are selected for reproduction. This can sometimes have a positive effect on the evolution by preventing the population from converging towards a local optimum solution. However this can at other times have a negative effect when a lot of the evolved training is lost and the population degenerates. This is particularly true when the population size is small. We saw examples of this in chapter 4.

This is what seems to have happened in the following experiment (Graph 6.1), in which the GA was used to evolve a population.

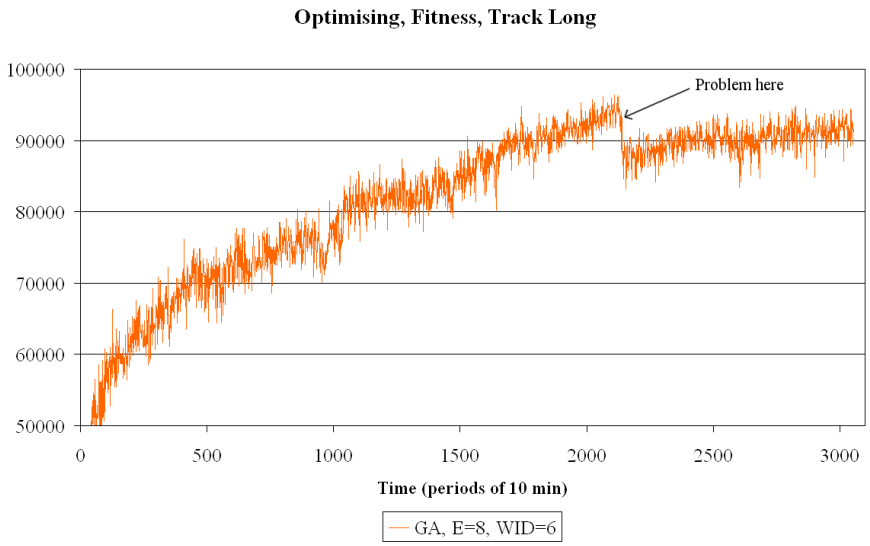

Figure 6.1: Fitness of the AI, GA used for optimising networks; problem at t=2136, the fitness decreases sharply.

Another problem is that the evaluation period is short, and during this short period, fit chromosomes may be evaluated as unfit and reciprocally unfit chromosomes can be evaluated as fit.

Finally, this evolution technique does not keep a record of the best chromosomes generated so far.

To solve this problem we investigated a steady state evolution technique inspired by Reinforcement Learning.

In traditional generational evolution, each individual in the whole population is evaluated for a fixed time and the fittest are picked to breed and form the next generation. In steady state evolution, populations contain competing individuals of all ages, each with the potential for dying or procreation at each stage [School 2003].

In this new technique the population is not totally replaced after every generation; instead each individual is generated and evaluated and only enters the population if it is evaluated as more fit than the worst individual in the population, to replace the worst individual in the population. The population is thus an elite one, providing a record of the best individuals generated so far.

The AI uses these elite individuals most of the time; this allows the AI to perform better than if a newly generated individual was being used. The AI at the same time refines the evaluation of the elite individual being used. The AI sometimes tries newly generated individuals, so that the population can evolve.

The rationale for this methodology is based on Reinforcement Learning in which ϵ-greedy policies are often used: an ϵ-greedy policy is one in which the currently best solution (the best solution found up to this time) is used most of the time but with probability ϵ (usually a small value, say 0.1) a random solution is chosen. This enables a good policy to be exploited mostly but allows for some exploration of alternative, potentially better policies. Thus in the current context, we exploit the currently optimal solutions most of the time but occasionally try out a new random solution.

The rationale for this methodology is based also on the fact that in real populations of individuals, fit individuals live longer that unfit individuals.

6.2 Experiments

The evaluation period is the same as with GA (10 minutes). At the beginning of each evaluation period, each AI randomly picks one of the following three states:

- Evaluator: The best least evaluated individual in the elite population is picked for evaluation (if there are equally least evaluated individuals,

then the best one of these sub AI's is evaluated first). The AI takes this evaluator state if one individual in the population has not yet been evaluated in which case the individual is picked for evaluation. This state is used to refine the fitness evaluation of individuals in the population.

- Explorer: A new individual is created using GA techniques and the elite population. At the end of the evaluation period the newly created individual replaces the worst individual in the population only if it is evaluated as more fit; this way the elite population fitness normally only increases.

- Exploiter: The fittest individual is picked for evaluation. This state is used to refine the fitness evaluation of the fittest individual in the population; in this mode the AI normally performs best.

The AI picks one of the three states with respective probabilities 0.05, 0.9 and 0.05.

It is possible by modifying these numbers to adjust the behaviour of the AI; have the AI to exploit more (better performance), or explore more (better evolution).

6.2.1 Training

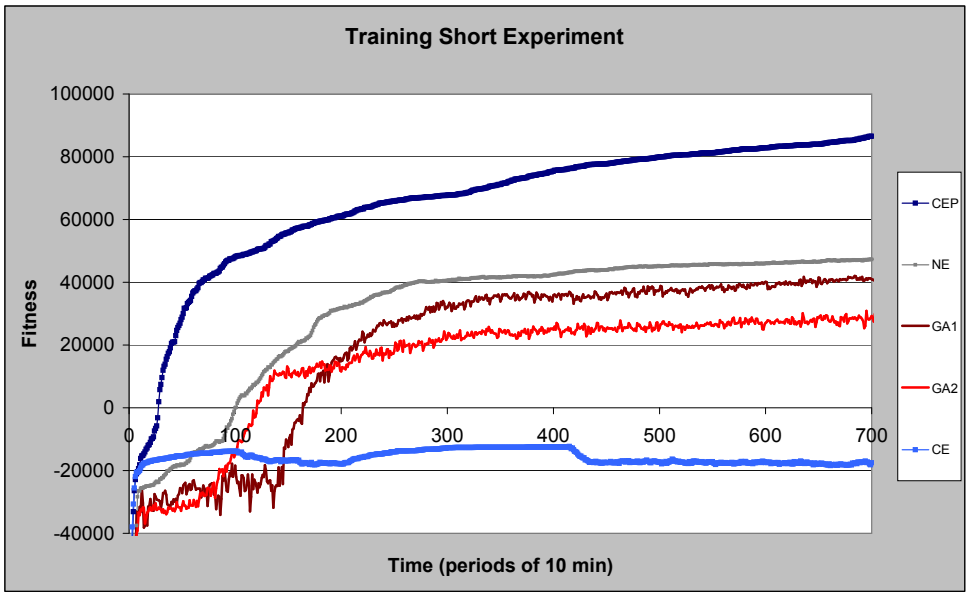

Figure 6.2: Fitness of the AI with the New Evolution Technique used for training networks. New evolution technique in thick grey compared to GA in light red. Track Long.

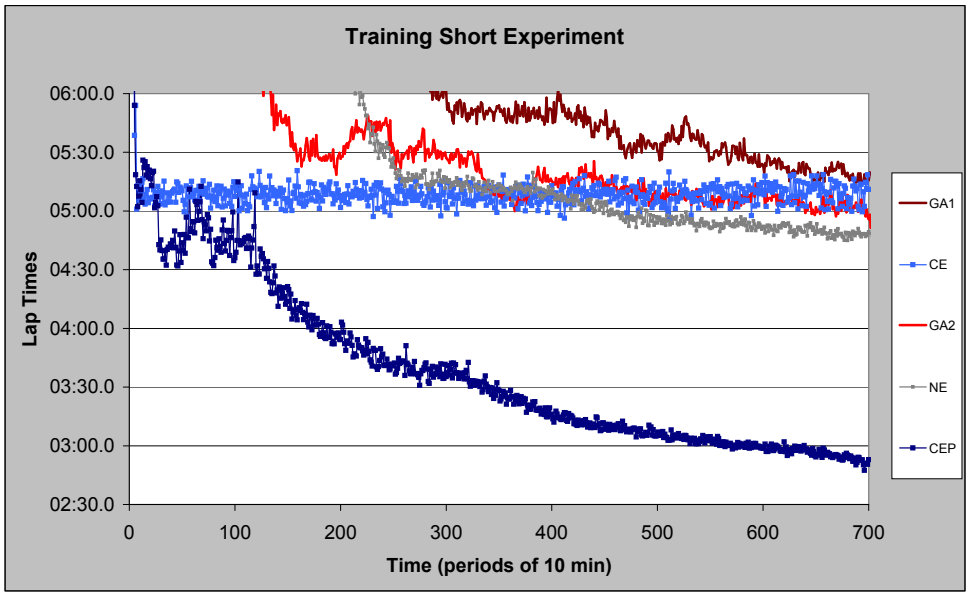

Figure 6.3: Lap Times of the bikes (corresponding to Figure 6.2), New Evolution Technique used for training networks. The New evolution technique in thick grey compared to GA and CEP. Track Long.

This experiment (Figures 6.2 and 6.3) is successful. Some new longer training experiments are carried out (Figures 6.4 and 6.5) in order to investigate whether this technique suffered from the same shortcomings as the standard Genetic Algorithm. In particular, we wish to investigate whether the solution found is stable or is liable to develop the degeneracy which we saw with the simple GA.

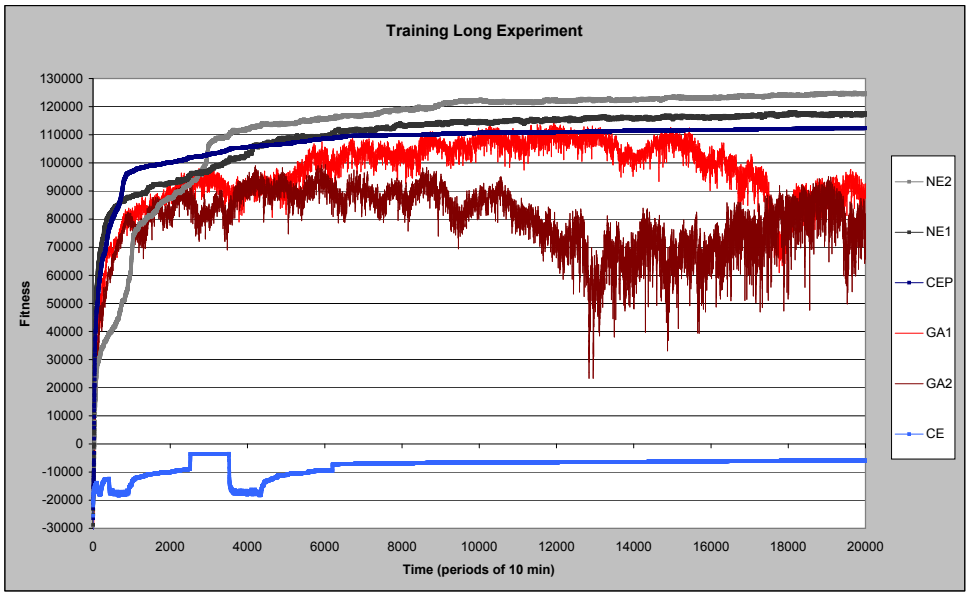

Figure 6.4: New experiment, fitness of the AI, New Evolution Technique used for training networks. New evolution technique in thick grey compared to GA and CEP. Track Long.

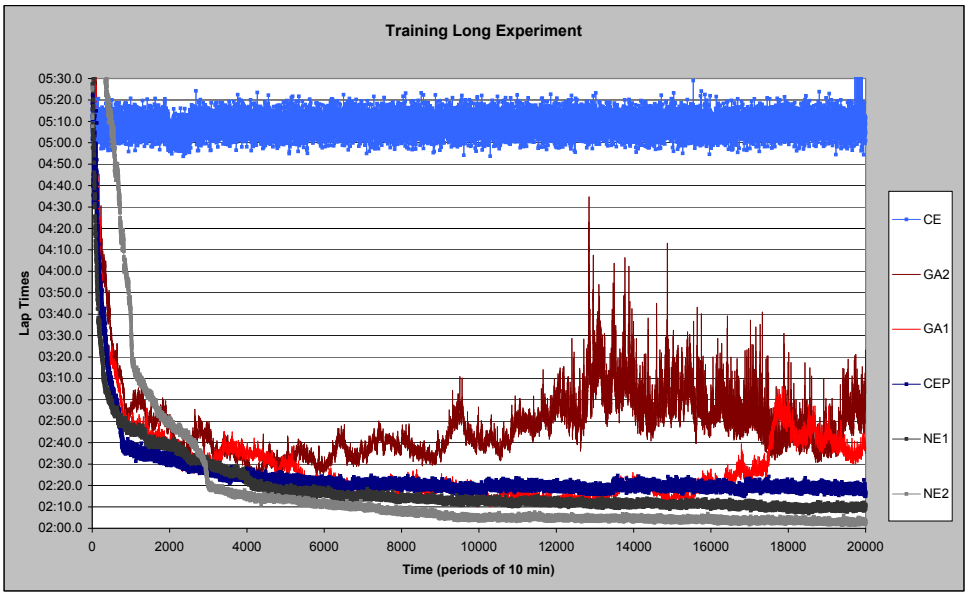

Figure 6.5: New experiment, lap times of the corresponding bikes, New Evolution Technique used for training networks. New evolution technique in thick grey compared to GA in light red. Track Long.

We can see that this New Evolution Technique is working much better than GA at training networks, with the fitness increasing steadily and no degeneration. At the end of training, average lap times are 2 minutes 10 seconds and 2 minutes 3 seconds.

6.2.2 Optimisation

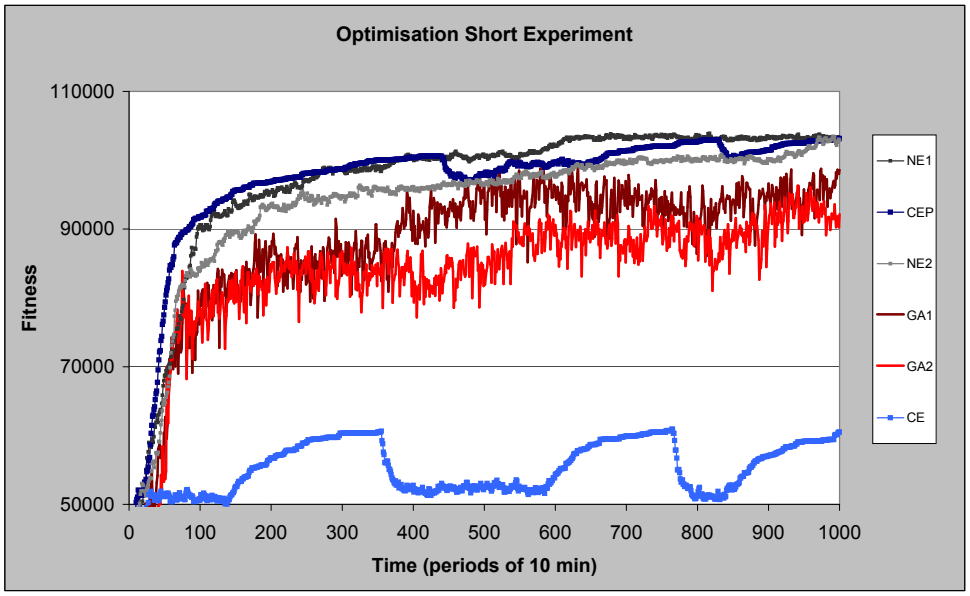

Figure 6.6: Fitness of the AI, New Evolution Technique used for optimising networks. New Evolution Technique in thick grey compared to GA and CEP. Track Long.

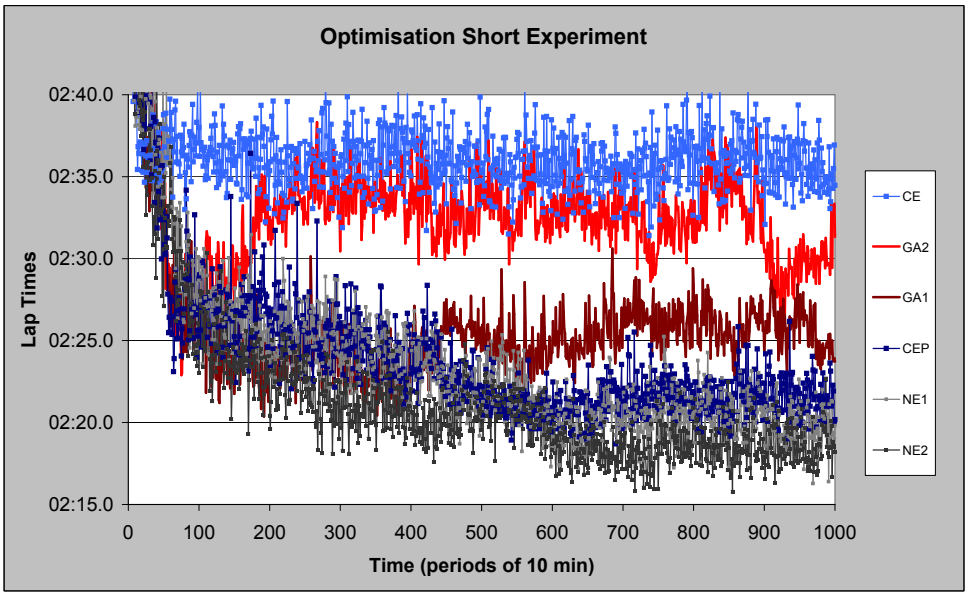

Figure 6.7: Lap Times of the corresponding bikes, New Evolution Technique used for optimising networks. New Evolution Technique in thick grey compared to GA and CEP. Track Long.

Some new longer optimising experiments are carried out.

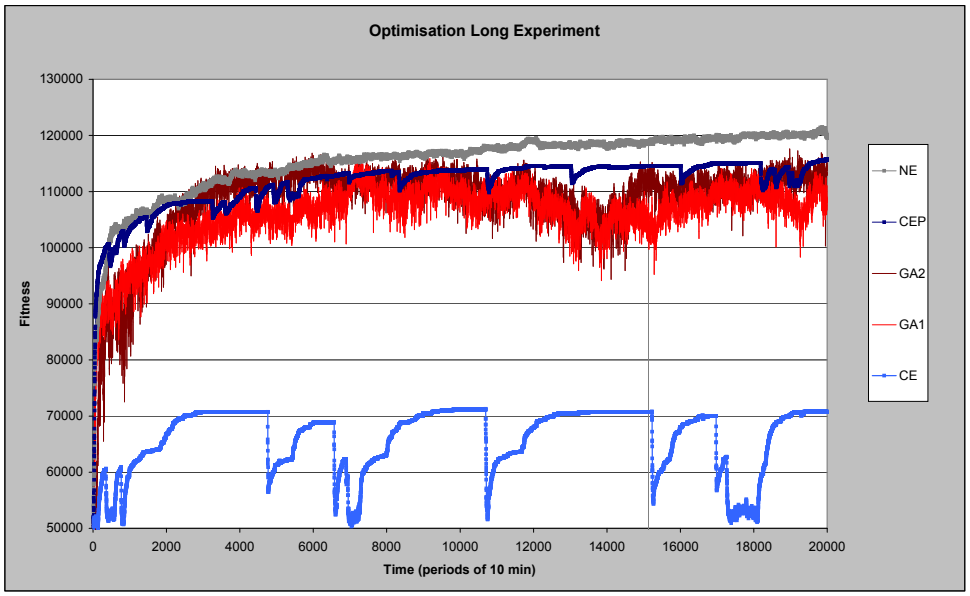

Figure 6.8: New experiment, fitness of the AI, New Evolution Technique used for optimising networks. New experiment, new evolution technique in thick grey compared to GA and CEP. Track Long.

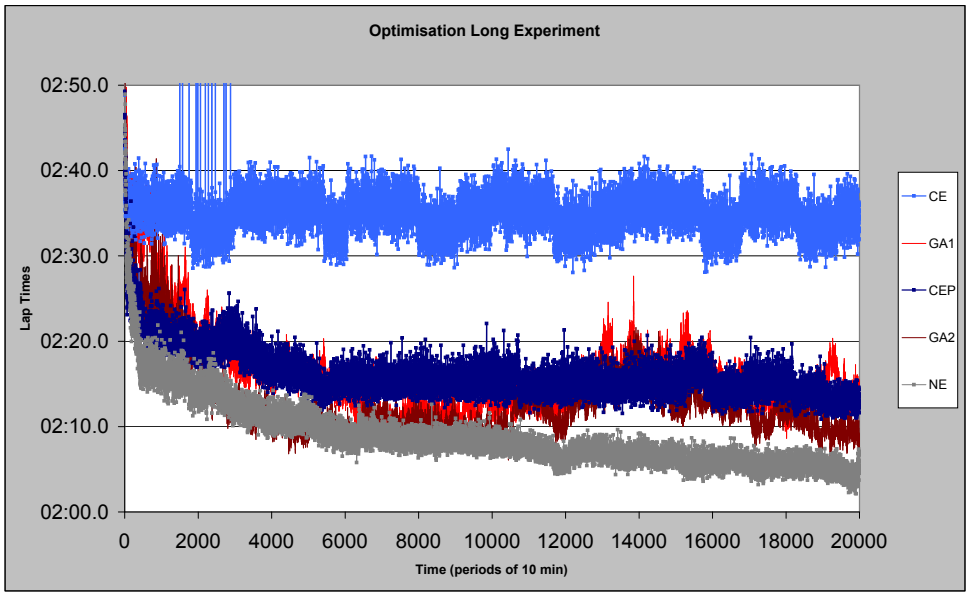

Figure 6.9: New experiment, lap times of the corresponding bikes, New Evolution Technique used for optimising networks. New experiment, new evolution technique in thick grey compared to GA and CEP. Track Long.

We can see from Figures 6.6, 6.7, 6.8 and 6.9 that this New Evolution Technique is working better than the standard GA and CEP at optimizing networks, with the fitness increasing steadily and no degeneration. At the end of optimising, the average lap time is 2 minutes 5 seconds. The computer controlled bikes are now faster than a good human player (2 minutes 15 seconds); the computer can race motorbikes fast for a long time whereas the human player can only race fast for a few laps.

6.2.3 Conclusion

This New Evolution Technique, inspired by Genetic Algorithm and the Reinforcement Learning technique of ϵ-greedy policies, is an optimisation process that seems to perform better than the simple GA and Cross Entropy with Perturbation methods. The end result is an AI performing better than a good human player. Unlike the GA, the evolution is steady, robust and pre-

dictable. However, similar to the GA, the optimisation can take a long time to perform.

Note that in the current implementation it is possible for the user to play the game while the AI of other bikes is being optimised by the evolution technique; if the AI is not optimised it may still perform well but may not be optimal. The optimisation process takes a long time and the user may want the AI to be optimised while he/she is not playing the game.

Chapter 7

Conclusion

In this short final chapter, we give a review of the thesis, highlighting our new work and then suggest strands of future work.

7.1 Review of the thesis

This thesis has presented the results of five years work into the use of AI techniques in the control of a motocross game. We began with a review of similar games and a review of different techniques of computational intelligence.

We have used an artificial neural network, specifically a multilayered perceptron, to control the bikes within the game. A great deal of effort has gone into getting the correct representation of the game world which was optimal for the AI. This representation has to capture the most important aspects of the environment but not be so complex that it interferes with learning which features are most essential to learning how to get round the track optimally. Having developed this representation, we used it consistently throughout the thesis.

The AI has learned (the parameters have been trained) using a number of different techniques, the most noteworthy of which are

1. Backpropagation. This standard supervised technique was shown to be very effective at learning how to ride the bike, its lap times being very close to those achieved by the author. We showed that the trained networks showed good generalisation properties: an AI trained on one track would perform well on tracks other than the training one, provided of course that they were not too different. Of course, we cannot

expect that this method would beat the human player - it is, after all, trained to try to emulate the human and so its responses are incrementally approaching his responses but will never go beyond his responses. Thus we investigated other techniques which have the potential for improving on the human skills.

2. Genetic Algorithms. We used GA to optimise the parameters in the multilayered perceptron. The fitness function or score function only used the effectiveness of the AI riding the bike not the information from a human driving round the track. The GA performed well but required much more processing time to perform to the standard set by the backpropagation network, but it also found solutions to the problem of riding the bike which were very different from the solutions found by a human rider.

3. Ensemble methods. We were somewhat disappointed by the results from bagging and boosting. We developed a new parameterisation of boosting which enabled us to investigate boosting and anti-boosting by changing a single parameter but nevertheless, the results were not encouraging. If our experience is typical of research in this area, these techniques are not useful in the context of computer games.

4. Alternative architectures. We have also investigated different architectures of artificial neural networks - the radial basis function, the self-organising map and the topographic product of experts - and a simple symbolic AI method based on "if then else" rules; these provided a variety of results but not yet able to go beyond those of the backpropagation method.

5. Cross Entropy Method. We have investigated the use of cross-entropy to optimise the parameters of the AI and we have shown that this technique was successful.

6. New evolution. We have developed a new optimisation technique called New Evolution which improves upon the standard genetic algorithm by using a technique inspired by the reinforcement learning method of ϵ-greedy policies; it was shown to improve the genetic algorithm's long term stability.

7.2 Future work

We have suggested that our game, **Motocross The Force**, can be used as a test-bed for future AI developments and that the need for such a test-bed is widely shared. However this game should not be the sole basis on which to evaluate AI developments: there are a variety of computer game genres - first person shooters, empire builders etc - for which different forms of AI might be more appropriate. We therefore would like to see a library of games developed in the same manner as in chapter 5.

We have managed to create an AI which can compete with a human in terms of effectiveness, efficiency, creativity. The AI is good but may not be the most optimal thus there will be ongoing research into improving the quality of the AI and, as we said at the beginning, computer games provide a test-bed which, in many ways, is ideal for experimenting with the new techniques: the world is simpler than the real world, the programmer is in charge of the physics of the environment, there are no major repercussions if something goes wrong and the difficulty of any task can be increased incrementally as the AI becomes more effective.

Future work may include implementing AI for the next version of **Motocross The Force**, which will feature better gameplay and physics simulation, and using the processing power offered by nowadays powerful graphical processing units and multicore CPU's to implement computer vision and more powerful AI. Instead of feeding the AI with a simplified representation of the environment it may be possible to feed it with a full 3D view of the environment very similar to what a human player can see while playing the game. It is probable that next version of **Motocross The Force** will also feature an AI SDK.

References

Arbib, M. A., ed. (1995), *The Handbook of Brain Theory and Neural Networks*, MIT Press.

Bishop, C. (1995), *Neural Networks for Pattern Recognition*, Oxford:Clarendon Press.

Bishop, C. M., Svensen, M. and Williams, C. K. I. (1997), 'Gtm: The generative topographic mapping', *Neural Computation* .

Breimen, L. (1997), Arcing the edge, Technical Report 486, Statistics Dept, University of California, Berkeley.

Breimen, L. (1999), Using adaptive bagging to debias regressions, Technical Report 547, Statistics Dept, University of California, Berkeley.

Buckland, M. (2002), *AI Techniques for Game Programming*, Course Technology PTR.

Buckland, M. (2004), *Programming Game AI by Example*, Jones and Bartlett Publishers.

Buckland, M. (2005*a*), 'Ai junkie'. `http://www.ai-junkie.com`.

Buckland, M. (2005*b*), 'Smart sweepers'. `http://www.ai-junkie.com/ann/evolved/nnt5.html`.

Champandard, A. J. (2009), 'Ai game dev'. `http://aigamedev.com/`.

Chaperot, B. (2009), 'Motocross the force ai sdk'. `http://cis.uws.ac.uk/benoit.chaperot`.

Chaperot, B., Breistroffer, E. and Saint-Olive, T. (2009), 'Motocross the force'. `http://www.jstarlab.com`.

Chaperot, B. and Fyfe, C. (2005), Motocross and artificial neural networks, *in* 'Game Design And Technology Workshop'.

Chaperot, B. and Fyfe, C. (2006*a*), Creating an ai-test platform, *in* 'Game Design And Technology Workshop'.

Chaperot, B. and Fyfe, C. (2006*b*), Improving artificial intelligence in a motocross game, *in* 'IEEE Symposium on Computational Intelligence and Games, CIG'06'.

Chaperot, B. and Fyfe, C. (2007), Topographic products of experts applied to a motocross simulation and simulation stabilisation, *in* 'Game Design And Technology Workshop'.

Dasgupta, D. (1993), Optimisation in time-varying environments using structured genetic algorithms, Technical Report IKBS-17-93, University of Strathclyde.

de Boer, P.-T., Kroese, D. P., Mannor, S. and Rubenstein, R. Y. (2004), 'A tutorial on the cross-entropy method', *Annals of Operations Research* **134**(1), 19–67.

DeLoura, M., Treglia, D., Kirmse, A., Pallister, K., Dickheiser, M. and Jacobs, S. (2008), *Game Programming Gems Series*, Charles River Media.

FlatOut (2007), 'Flatout: Ultimate carnage'. `http://uk.xbox360.ign.com/objects/840/840485.html`.

Forza (2005), 'Forza motorsport'. `http://uk.xbox.ign.com/objects/682/682857.html`.

Forza2 (2005), 'Forza motorsport'. `http://research.microsoft.com/en-us/projects/drivatar/forza.aspx`.

Friedman, J., Hastie, T. and Tibshirani, R. (1998), Additive logistic regression: a statistical view of boosting, Technical report, Statistics Dept, Stanford University.

Fyfe, C. (2005), Local vs global models in pong, *in* 'International Conference on Artificial Neural Networks, ICANN2005'.

Fyfe, C. (2007), Two topographic maps for data visualisation, Technical Report 14, University of Paisley.

GameDevAI (2009), 'Artificial intelligence'. `http://www.gamedev.net/reference/list.asp?categoryid=18`.

GT5P (2008), 'Gran turismo 5 prologue'. `http://uk.ps3.ign.com/objects/949/949777.html`.

Hastie, T., Tibshirani, R. and Friedman, J. (2001), *The Elements of Statistical Learning*, Springer.

Havok (2005), 'Havok physics'. `http://www.havok.com`.

Haykin, S. (1994), *Neural Networks- A Comprehensive Foundation*, Macmillan.

Hecht-Nielsen, R. (1991), *Neurocomputing*, Addison Wesley.

Hertz, J., Krogh, A. and Palmer, R. G. (1992), *Introduction to the Theory of Neural Computation*, Addison-Wesley Publishing.

Heskes, T. (1997), Balancing between bagging and bumping, *in* 'Neural Information Processing Sytems, NIPS7'.

Holland, J. (1981), Genetic algorithms and adaptation, Technical Report 34, University of Michigan.

Jimenez, E. (2009), 'The pure advantage: Advanced racing game ai'. `http://www.gamasutra.com/view/feature/3920/the_pure_advantage_advanced_.php`.

Kohonen, T. (1995), *Self-Organising Maps*, Springer.

Leen, G. and Fyfe, C. (2005), Training an ai player to play pong using the gtm, *in* 'IEEE Symposium on Computational Intelligence and Games'.

Loiacono, D., Togelius, J., Lanzi, P. L., Kinnaird-Heether, L., Lucas, S. M., Simmerson, M., Perez, D., Reynolds, R. G. and Saez, Y. (2008), The wcci 2008 simulated car racing competition, *in* 'IEEE Computational Intelligence and Games 2008'.

McGlinchey, S. J. (2003), Learning of ai players from game observation data, *in* 'GAME-ON', pp. 106–.

MCM2 (2000), 'Motocross madness 2'. `http://uk.pc.ign.com/objects/014/014057.html`.

Michie, D., Spiegelhalter, D. J. and Taylor, C. C., eds (1994), *Machine learning, neural and statistical classification*, Ellis Horwood.

Mitchell, M. and Forrest, S. (1993), 'Genetic algorithms and artificial life', *Artificial Life* . (Santa Fe Working Paper 93-11-072).

MXVSATV (2005), 'Mx vs atv unleashed'. `http://uk.xbox.ign.com/objects/709/709100.html`.

Ode (2006), 'Open dynamics engine'. `http://www.ode.org`.

PhysX (2005), 'Nvidia physx'. `http://www.nvidia.com/object/physx_new.html`.

Rabin, S. (2008), *AI Game Programming Wisdom Series*, Charles River Media.

Rechenberg, I. (1994), Evolutionsstrategie, Technical report, University of Stuttgart.

Remo, C. (2009), 'Yamauchi: Gran turismo 5's development cost hit $60 million'. `http://www.gamasutra.com/php-bin/news_index.php?story=25966`.

Rubinstein, R. Y. and Kroese, D. P. (2004), *The Cross-Entropy Method*, Springer.

School, J. B. (2003), 'Evolving efficient neural network classifiers'. `citeseer.ist.psu.edu/680576.html`.

Togelius, J. (2007), Optimization, Imitation and Innovation: Computational Intelligence and Games, PhD thesis, University of Essex.

Togelius, J., Lucas, S. M., Thang, H. D., Garibaldi, J., Nakashima, T., Tan, C. H., Elhanany, I., Berant, S., Hingston, P., MacCallum, R. M., Haferlach, T., Gowrisankar, A. and Burrow, P. (2008), The 2007 ieee cec simulated car racing competition, *in* 'Genetic Programming and Evolvable Machines'.

Torcs (2009), 'The open racing car simulator'. `http://torcs.sourceforge.net`.

Vapnik, V. (1995), *The nature of statistical learning theory*, Springer Verlag, New York.

Wang, T., Fyfe, C. and Marney, J. P. (1999), A comparison of an oligopoly game and the n-person iterated prisoner's dilemma, *in* 'Fifth International Conference of the Society for Computational Economics,Computing in Economics and Finance, CEF99'. (Special session on evolutionary computation in Economics and Finance).

Appendix A

Tracks

A.1 Tracks

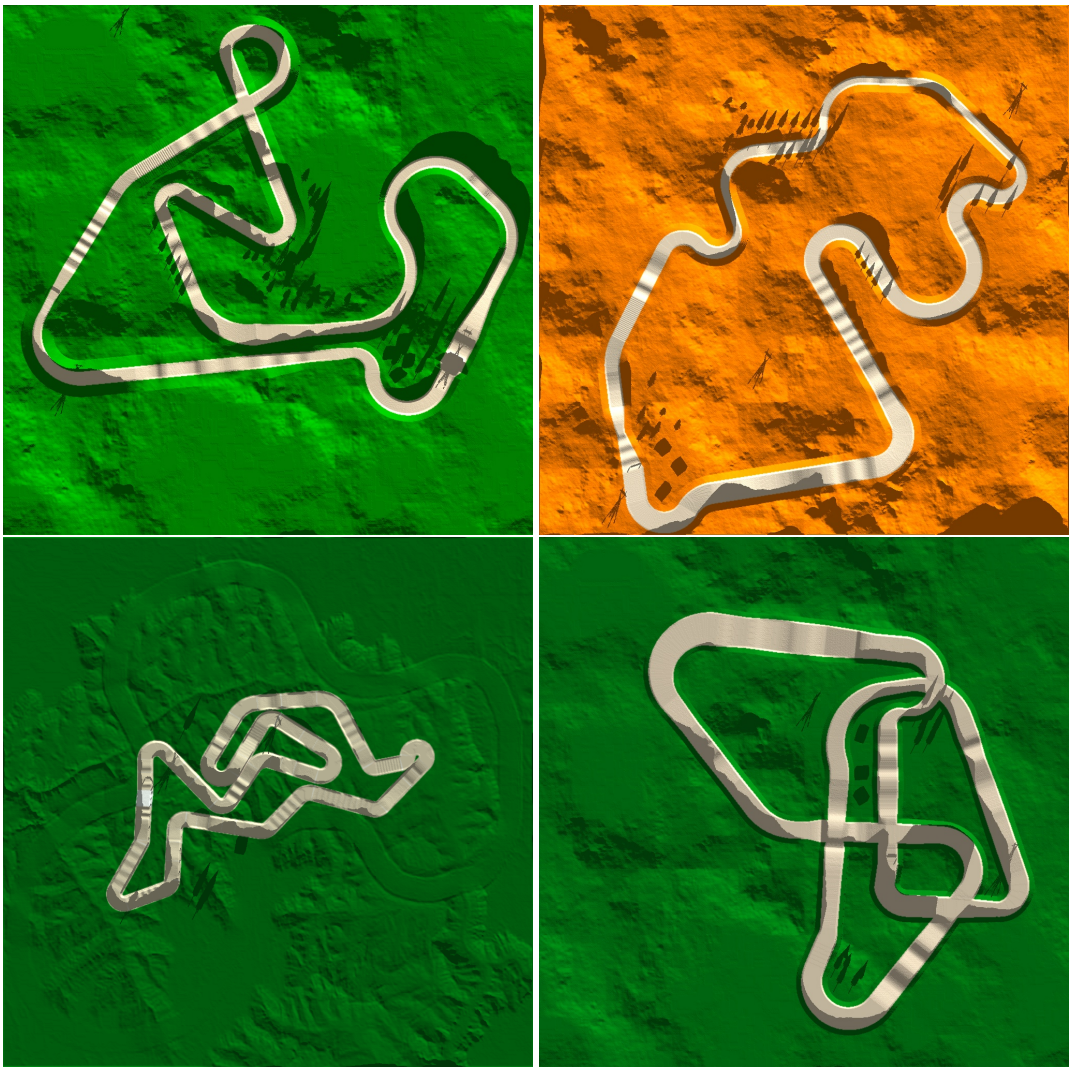

Figure A.1: Tracks Chill (top left), Chill2 (top right), Long (bottom left) and TrackA (bottom right), used in Training Set 920.

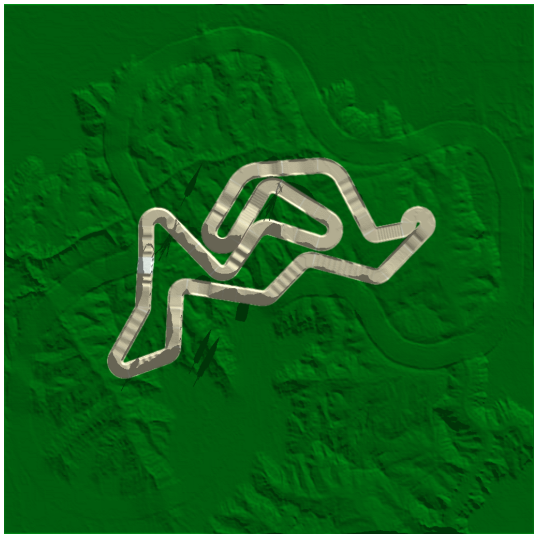

Figure A.2: Track Long used in Training Set Long.

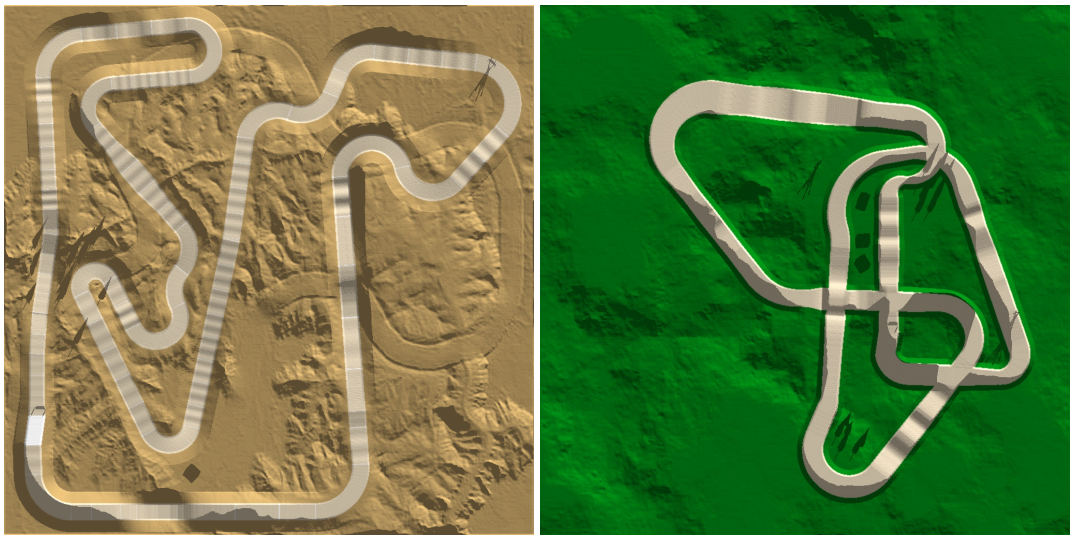

Figure A.3: Track O (left) and track A (right) used to test ANN generalisation property.

Appendix B

AI SDK

B.1 AI DLL

Modules

- **Construction and Destrucion Functions**
- **Operation Functions**
- **Back Propagation Functions**
- **Genetic Algorithm Functions**
- **Other Functions**

Detailed Description

All these functions must be implemented by the AI DLL for the AI DLL to be recognised by the executable.

B.1.1 Construction and Destrucion Functions

Functions

- VOID * **CreateAI** (const WCHAR *strAIFilename, BOOL bReadFromFile)
- VOID **DestroyAI** (VOID *pAI)

Function Documentation

VOID* CreateAI (const WCHAR * *strAIFilename*, BOOL *bReadFromFile*)

Create an AI, given an AI file name (.NN file extension).

Parameters

strAIFilename (in) AI file name.

bReadFromFile (in) set to true if AI DLL actually creates the AI from the file, or false to create a blank AI.

Returns

void pointer on the newly created AI.

See also

SaveWeights (p. 142). **AIFILEHEADER** (p. 156).

VOID DestroyAI (VOID * *pAI*)

Destroy the AI.

Parameters

pAI (in) void pointer on the AI to destroy.

B.1.2 Operation Functions

Functions

- VOID **DecisionMaking** (VOID *pAI, **SampleData** &sampleData, const D3DXMATRIX &ForwardTransform)
- VOID **RenderAI** (VOID *pAI, BOOL bDebug)

Function Documentation

VOID DecisionMaking (VOID * *pAI*, SampleData & *sampleData*, const D3DXMATRIX & *ForwardTransform*)

Make a decision, given a situation.

Parameters

pAI (in) void pointer on the AI.

sampleData (in,out) situation and decision.

ForwardTransform (in) Bike forward transformation matrix.

See also

SampleData (p. 158).

VOID RenderAI (VOID * *pAI*, BOOL *bDebug*)

Render debug information.

Parameters

pAI (in) void pointer on the AI.

bDebug (in) Debug mode; the user can activate Debug mode by pressing (CTRL + SHIFT + D).

See also

Drawing Functions (p. 147)

B.1.3 Back Propagation Functions

Functions

- VOID **GenerateAIFromTrainingSet** (VOID *pAI, const WCHAR *strTSFilename)
- VOID **UpdateAI** (VOID *pAI)

Function Documentation

VOID GenerateAIFromTrainingSet (VOID * *pAI*, const WCHAR * *strTSFilename*)

Load and process a training set; prepare for training from training set.

Parameters

pAI (in) void pointer on the AI.

strTSFilename (in) file name of the training set.

VOID UpdateAI (VOID * *pAI*)

Update the AI; can be used for updating the training from training set (back propagation).

Parameters

pAI (in) void pointer on the AI.

B.1.4 Genetic Algorithm Functions

Functions

- BOOL **CanAIBeOptimisedUsingGA** (VOID *pAI)
- VOID **PutWeights** (VOID *pAI, const vector< float > &weights, INT &cweight)
- VOID **GetWeights** (VOID *pAI, vector< float > &weights)
- VOID **SetGeneration** (VOID *pAI, INT nGeneration)
- INT **GetGeneration** (VOID *pAI)
- VOID **SaveWeights** (VOID *pAI, INT nId)
- INT **GetNumberOfWeights** (VOID *pAI)
- FLOAT **GetFitness** (VOID *pAI)
- VOID **NewFitness** (VOID *pAI)
- VOID **NewEvent** (VOID *pAI, **EventType** event, FLOAT vValue)
- VOID **NewEpisode** (VOID *pAI)

Detailed Description

These are functions for training the AI using Genetic Algorithm or Reinforcement Learning.

Function Documentation

BOOL CanAIBeOptimisedUsingGA (VOID * *pAI*)

Can the AI be optimised using Genetic Algorithms ? If yes then the game engine may attempt to optimise the weights inside the AI using a Genetic Algorithm.

Parameters

pAI (in) void pointer on the AI.

Returns

true if AI is currently training from training set.

FLOAT GetFitness (VOID * *pAI*)

Get fitness

Parameters

pAI (in) void pointer on the AI.

Returns

Fitness

INT GetGeneration (VOID * *pAI*)

Get generation.

Parameters

pAI (in) void pointer on the AI.

Returns

Generation.

INT GetNumberOfWeights (VOID * *pAI*)

Get number of weights.

Parameters

pAI (in) void pointer on the AI.

Returns

Number of weights

VOID GetWeights (VOID * *pAI*, vector< float > & *weights*)

Get weights from AI.

Parameters

pAI (in) void pointer on the AI.

weights (out) weights.

VOID NewEpisode (VOID * *pAI*)

New episode. The bike has been respawned.

Parameters

pAI (in) void pointer on the AI.

VOID NewEvent (VOID * *pAI*, EventType *event*, FLOAT *vValue*)

New event.

Parameters

pAI (in) void pointer on the AI.

event (in) event type.

vValue (in) information relative to event; meaning depends on event type.

See also

EventType (p. 155).

VOID NewFitness (VOID * *pAI*)

New fitness, new chromosome to be evaluated.

Parameters

pAI (in) void pointer on the AI.

VOID PutWeights (VOID * *pAI*, const vector< float > & *weights*, INT & *cweight*)

Put weights into an AI.

Parameters

pAI (in) void pointer on the AI.

weights (in) weights.

cweight (in,out) position

VOID SaveWeights (VOID * *pAI*, INT *nId*)

Save weights.

Parameters

pAI (in) void pointer on the AI.

nId (in) not used.

See also

AIFILEHEADER (p. 156).

VOID SetGeneration (VOID * *pAI*, INT *nGeneration*)

Set generation.

Parameters

pAI (in) void pointer on the AI.

nGeneration (in) generation.

B.1.5 Other Functions

Functions

- const CHAR * **GetAIName** ()
- INT **GetAIVersion** ()
- INT **GetDebug** ()

Function Documentation

const CHAR* GetAIName ()

Get AI name; used to match AI files with AI DLL's.

Returns

AI name.

INT GetAIVersion ()

Get AI version.

Returns

nAIVERSION.

INT GetDebug ()

Get Debug; returns true if this is the debug version of the AI DLL. The release version of the executable uses the release version of the AI DLL and the debug version of the executable uses the debug version of the AI DLL.

Returns

Debug.

B.2 Executable

Modules

- **WayPoint Functions**
- **Drawing Functions**
- **Terrain Functions**
- **Other Functions**
- **Structures, typdefs and enums.**

Detailed Description

Functions implemented by the executable that the AI DLL can call.

B.2.1 WayPoint Functions

Functions

- VOID **GetWayPointTransform** (D3DXMATRIX &Transform, INT nWayPointId, FLOAT vDist)
- VOID **GetWayPointPosition** (D3DXVECTOR3 &Pos, INT nWayPointId, FLOAT vDist)
- VOID **GetWayPointDirection** (D3DXVECTOR3 &Dir, INT nWayPointId, FLOAT vDist)
- FLOAT **GetWayPointWidth** (INT nWayPointId, FLOAT vDist)

Detailed Description

These are functions to get information about WayPoints.

Function Documentation

VOID GetWayPointDirection (D3DXVECTOR3 & *Dir*, INT *nWayPointId*, FLOAT *vDist*)

Get interpolated WayPoint direction at given distance from nWayPointId.

Parameters

Dir (out) WayPoint direction.

nWayPointId (in) WayPoint ID, starting from zero, with one WayPoint every metre along the centre of the track.

vDist (in) Distance from WayPoint ID, in cm, along the centre of the track, used for interpolation.

VOID GetWayPointPosition (D3DXVECTOR3 & *Pos*, INT *nWayPointId*, FLOAT *vDist*)

Get interpolated WayPoint position at given distance from nWayPointId.

Parameters

Pos (out) WayPoint world position (cm).

nWayPointId (in) WayPoint ID, starting from zero, with one WayPoint every metre along the centre of the track.

vDist (in) Distance from WayPoint ID, in cm, along the centre of the track, used for interpolation.

VOID GetWayPointTransform (D3DXMATRIX & *Transform*, INT *nWayPointId*, FLOAT *vDist*)

Get interpolated WayPoint transformation matrix at given distance from nWayPointId.

Parameters

Transform (out) WayPoint Transformation matrix.

nWayPointId (in) WayPoint ID, starting from zero, with one WayPoint every metre along the centre of the track.

vDist (in) Distance from WayPoint ID, in cm, along the centre of the track, used for interpolation.

FLOAT GetWayPointWidth (INT *nWayPointId*, FLOAT *vDist*)

Get interpolated WayPoint width at given distance from nWayPointId.

Parameters

nWayPointId (in) WayPoint ID, starting from zero, with one WayPoint every metre along the centre of the track.

vDist (in) Distance from WayPoint ID, in cm, along the centre of the track, used for interpolation.

B.2.2 Drawing Functions

Functions

- VOID **ImmediateDrawVertex** (D3DXVECTOR3 Pos, FLOAT vSize, DWORD Color)
- VOID **ImmediateDrawLine** (D3DXVECTOR3 A, D3DXVECTOR3 B, DWORD Color)
- VOID **ImmediateDrawText** (const WCHAR *strTxt)
- VOID **ImmediateDrawRectangle** (FLOAT x, FLOAT y, FLOAT w, FLOAT h, FLOAT z, DWORD Color)

Detailed Description

These are functions for drawings things on screen; for debug purposes.

See also

RenderAI (p. 137).

Function Documentation

VOID ImmediateDrawLine (D3DXVECTOR3 *A*, D3DXVECTOR3 *B*, DWORD *Color*)

Draw a 3D line.

Parameters

A (in) First world position (cm).

B (in) Second world position (cm).

Color (in) Color (ARGB).

VOID ImmediateDrawRectangle (FLOAT *x*, FLOAT *y*, FLOAT *w*, FLOAT *h*, FLOAT *z*, DWORD *Color*)

Draw a 2D rectangle. Screen coordinataes are (0,0) top left to (1,1) bottom right.

Parameters

x (in) X position of top left corner.

y (in) Y position of top left corner.

w (in) Width.

h (in) Height.

z (in) Depth (range (0,1)).

Color (in) Color (ARGB).

VOID ImmediateDrawText (const WCHAR * *strTxt*)

Draw text.

Parameters

strTxt (in) Text to display.

VOID ImmediateDrawVertex (D3DXVECTOR3 *Pos*, FLOAT *vSize*, DWORD *Color*)

Draw a 3D Vertex.

Parameters

Pos (in) World position (cm).

vSize (in) Size of vertex (cm).

Color (in) Color (ARGB).

B.2.3 Terrain Functions

Functions

- FLOAT **TerrainGetHeight** (FLOAT x, FLOAT y)
- VOID **TrackCreate** (const WCHAR *strFileName, BOOL bFullCreate)
- INT **TrackGetUniqueID** (INT nId)
- const WCHAR * **TrackGetShortName** ()
- const WCHAR * **TrackGetFileName** ()

Detailed Description

These are functions to create and get information about tracks and terrains.

Function Documentation

FLOAT TerrainGetHeight (FLOAT *x*, FLOAT *y*)

Get terrain height at given horizontal position (Z is up).

Parameters

x (in) X position in world space (cm).

y (in) Y position in world space (cm).

Returns

Terrain height at given position (cm).

VOID TrackCreate (const WCHAR * *strFileName*, BOOL *bFullCreate*)

Create a track, given a track file name.

Parameters

strFileName (in) Track file name.

bFullCreate (in) Set to true to create everything including the visuals, or false to create minimum required for processing training set data.

const WCHAR* TrackGetFileName ()

Get the track file name.

Returns

Track file name.

const WCHAR* TrackGetShortName ()

Get the track short name.

Returns

Short name.

INT TrackGetUniqueID (INT *nId*)

Get one of the ten track unique ID's. Unique ID's are used to uniquely identify tracks; this is useful to ensure a track has not been modified between the time training data has been generated and the time the training data is used.

Parameters

nId (in) Unique ID ID (0,9).

Returns

Unique ID.

B.2.4 Other Functions

Typedefs

- typedef BOOL(CALLBACK * **TPROGRESSBARCALLBACKUPDATE**)(VOID *pUser, WCHAR *strDesc, FLOAT *pRatio, INT nNumIterations)

Functions

- VOID **GetForwardTransform** (D3DXMATRIX &ForwardTransform, const D3DXVECTOR3 &BikeDir1, const D3DXVECTOR3 &BikeVel, const D3DXVECTOR3 &BikePos)
- INT **GetAIVersion** ()
- VOID **AddProgressBar** (**TPROGRESSBARCALLBACKUPDATE** CallBackUpdate, VOID *pUser, INT nIterationsPerUpdate, BOOL bRender)
- VOID **ProfilerStart** (INT nID)
- VOID **ProfilerStop** (INT nID)
- VOID **GetOtherBikesInfo** (VOID *pAI, vector< **BikeInfo** > &bikeInfos)

Typedef Documentation

typedef BOOL(CALLBACK * TPROGRESSBARCALLBACKUPDATE)(VOID *pUser, WCHAR *strDesc, FLOAT *pRatio, INT nNumIterations)

Callback.

Implemented by DLL, called by executable, to be passed as parameter to function AddProgressBar.

Parameters

pUser (in) pointer on user data.

strDesc (out) text to display.

pRatio (out) progress.

nNumIterations (in) number of iterations inside the callback.

Returns

true if process finished, false otherwise.

Function Documentation

VOID AddProgressBar (TPROGRESSBARCALLBACKUPDATE *CallBackUpdate*, VOID * *pUser*, INT *nIterationsPerUpdate*, BOOL *bRender*)

Add Process, render progress bar.

Parameters

CallBackUpdate (in) Process callback.

pUser (in) pointer on user data, to be passed to CallBackUpdate.

nIterationsPerUpdate (in) number of iterations inside the CallBackUpdate.

bRender (in) set to true to render progress bar on screen.

INT GetAIVersion ()

Get AI version.

Returns

nAIVERSION.

VOID GetForwardTransform (D3DXMATRIX & *ForwardTransform*, const D3DXVECTOR3 & *BikeDir1*, const D3DXVECTOR3 & *BikeVel*, const D3DXVECTOR3 & *BikePos*)

Get a space centred at the origin of the motorbike; the Z axis points up and the Y axis follows the horizontal velocity direction. This space is more convenient than bike space to represent and transform world objects in relation to the bike.

Parameters

ForwardTransform (out) Bike forward transformation matrix.

BikeDir1 (in) Bike back direction.

BikeVel (in) Bike velocity (cm/s).

BikePos (in) Bike world position (cm).

VOID GetOtherBikesInfo (VOID * *pAI*, vector< BikeInfo > & *bikeInfos*)

Get Information about other bikes position and velocity.

Parameters

pAI (in) void pointer on the AI of the current bike.

bikeInfos (in,out) a reference to a vector of **BikeInfo** (p. 157).

See also

BikeInfo (p. 157).

VOID ProfilerStart (INT *nID*)

Start the time profiling of a block; after program execution, the profiler reports in file "profile.txt".

Parameters

nID (in) Block Identifier.

VOID ProfilerStop (INT *nID*)

Stop the time profiling of a block; after program execution, the profiler reports in file "profile.txt".

Parameters

nID (in) Block Identifier.

B.2.5 Structures, typdefs and enums.

Classes

- struct **AIFILEHEADER**

  AI file header.

- struct **SampleData**

  Sample.

- struct **BikeInfo**

  Bike Position.

Enumerations

- enum **EventType** {

  EVENT_TYPE_PASSWAYPOINT = 0, **EVENT_TYPE_MISSWAYPOINT** = 1, **EVENT_TYPE_CRASH** = 2, **EVENT_TYPE_RESPAWN** = 3,

  EVENT_TYPE_NEWLAP = 4 }

  Event Enum.

Variables

- const INT **nAIVERSION** = 3

  AI Version.

Detailed Description

These are used for communication between the DLL's and the executable.

Enumeration Type Documentation

enum EventType

Event Enum.

Enumerator:

EVENT_TYPE_PASSWAYPOINT Bike has just passed a waypoint (the associated value is the normalised distance from centre of the next WayPoint).

EVENT_TYPE_MISSWAYPOINT Bike has just missed a waypoint (the associated value is the normalised distance from centre of the next WayPoint).

EVENT_TYPE_CRASH Bike has just crashed (the associated value is zero).

EVENT_TYPE_RESPAWN Bike has has just been respawned (the associated value is the distance fron the previously missed WayPoint (cm)).

EVENT_TYPE_NEWLAP Bike has has just completed a lap (the associated value is the lap time in seconds).

Variable Documentation

const INT nAIVERSION = 3

AI Version.

The AI version given in the file header, by the executable and by the AI DLL must match nAIVERSION.

B.3 Class List

B.3.1 AIFILEHEADER

Public Attributes

- CHAR **pszAiName** [AI_NAME_MAXLENGTH]
- DWORD **nAiVersion**

Detailed Description

AI file header. This file header must be at the begining of every AI file (files with .NN file extension).

Member Data Documentation

DWORD AIFILEHEADER::nAiVersion

AI version; must be equal to nAIVERSION.

CHAR AIFILEHEADER::pszAiName[AI_NAME_-MAXLENGTH]

AI name, with one name per AI DLL; is used to tell the executable which AI DLL to use for a given AI file (.NN file extension).

B.3.2 BikeInfo

Public Attributes

- D3DXVECTOR3 **BikePos**
- D3DXVECTOR3 **BikeVel**

Detailed Description

Bike Position. Structure used for communication between the executable and the AI DLL about other bikes position and velocity.

Member Data Documentation

D3DXVECTOR3 BikeInfo::BikePos

Global position of the bike (cm)

D3DXVECTOR3 BikeInfo::BikeVel

Global velocity of the bike (cm/s)

B.3.3 SampleData

Public Attributes

- INT **nCurrentWayPointId**
- D3DXVECTOR3 **BikePos**
- D3DXVECTOR3 **BikeDir0**
- D3DXVECTOR3 **BikeDir1**
- D3DXVECTOR3 **BikeVel**
- D3DXVECTOR3 **BikeAPos**
- D3DXVECTOR3 **BikeAVel**
- FLOAT **vLeftX**
- FLOAT **vLeftY**
- FLOAT **vRightX**
- FLOAT **vRightY**
- FLOAT **vScore**
- FLOAT * **pInputs**

Detailed Description

Sample. Structure used for communication between the executable and the AI DLL.

Member Data Documentation

D3DXVECTOR3 SampleData::BikeAPos

(in) Bike angular position (rad).

D3DXVECTOR3 SampleData::BikeAVel

(in) Bike angular velocity (rad/s).

D3DXVECTOR3 SampleData::BikeDir0

(in) Bike right direction.

D3DXVECTOR3 SampleData::BikeDir1

(in) Bike back direction.

D3DXVECTOR3 SampleData::BikePos

(in) Bike world position (cm).

D3DXVECTOR3 SampleData::BikeVel

(in) Bike velocity (cm/s).

INT SampleData::nCurrentWayPointId

(in) Current WayPoint ID.

FLOAT* SampleData::pInputs

(in) Pointer on user data.

FLOAT SampleData::vLeftX

(in,out) Turn decision.

FLOAT SampleData::vLeftY

(in,out) Bike rotate around right axis decision.

FLOAT SampleData::vRightX

(in,out) Bike rotate around up axis decision.

FLOAT SampleData::vRightY

(in,out) Accelerate brake decision.

FLOAT SampleData::vScore

(in) Score (not used).