University of Sussex

School of Engineering and Informatics

BSc(hons) Computer Science and Artificial Intelligence

Evolving Sumo Fighting Robots in a Simulated 2D Environment

Jamie Mayer

May 2017

Signed:

Jamie Mayer

Acknowledgements

I would like to thank:

- Dr Christopher Buckley for agreeing to and supervising the project, providing sound advice and guidance throughout the project.
- Dr Luc Berthouze for the knowledge gained during the Neural Networks module that ultimately assisted me in the project.
- Amie Buttarazzi for her continued moral support, patience and unwavering faith in my ability to persevere in the face of adversity.
- Nareesa Sulaiman for her advice and assistance in prioritising tasks and organisation over the course of the year.

Contents

List of Figures

Abstract

The project looks at Evolutionary Robotics as a way of developing Continuous Time Recurrent Neural Networks (CTRNN) as simple robot controllers for complex, dynamic behaviour in a simulated environment. The field was very popular during the early 1990s but fell out of favour due to a number of problems. As robotic machinery becomes ever more popular the study delves back in to the field to see if it may present a viable option for developing novel solutions. An artificial simulation is created and agents are evolved to fight in a sumo style tournament exploring different parameters to the Genetic Algorithm. Agents are first trained to stay in the arena boundary and are then further evolved to fight. Although the study is ultimately inconclusive, the objective of training agents to fight is achieved, showing that evolutionary robotics still has potential to develop novel solutions in a hands off way.

Introduction

The objectives of this study are:

(1) - To explore the concept of evolutionary robotics in a simulated environment using a sumo style competition as the problem domain.

(2) - To implement a novel way of assessing fitness of the population by means of direct competition based on performance in the problem domain where the winning solution moves forward in to the next generation of population members and the loser is replaced by a mutated version of the winner. This is contrasted with the idea of co-evolving two separate populations.

(3) - To investigate whether a return to evolutionary robotics could potentially hold the key to creating novel solutions in today's technological climate.

This report shall cover:

- Introduction
- Professional Considerations
- System Requirements
- Methodology
- Results
- Conclusion

This section shall discuss the following:

- Motivations
- What is evolutionary robotics?

- Overview of the study

1.1 Motivations

The development of controllers for robots by hand has been shown to be increasingly difficult as the desired level of complexity is increased [1]. A similar study to this has been conducted by Sharabi and Sipper [3] in which Genetic Programming (GP) was used to evolve a program to act as the robot controller for a Sumo fighting agent. The original study evolved the controllers in a real world setting using a top down camera to give an agent's current position to the as an input along with other on board sensors and actuators on each bot. Two populations were coevolved using dynamic fitness functions to produce desired behaviour. This study takes a lot of the elements of the original while making some changes. The combination of a scaled down genetic algorithm using a static fitness function, coupled with a simulated environment allow for a decreased computational cost, making the entire study easier and faster to process. A top down camera that was used in the original study that gave total visibility of the problem state as an input has been replaced by individual sensors on each agent, giving each bot its own partial visibility of the problem sate as this has been deemed to be more biologically plausible allowing for a closer replication of what can be witnessed in the natural world. A recurrent neural network was chosen as the model for a robot controller as it has many advantages over GP. Mainly in that the high level language needed for GP constrains the behaviour of the bot to concepts that are directly describable by the syntax of a high level language which may cause much coarser granularity in the fitness landscape of the problem. Secondly by using concepts such as recursion in a GP application, there is no guarantee that the program will ever halt unless further constraints are added [1].
Evolutionary Robotics, while becoming popular in the 1990s and early 2000s fell out of favour due to a number of problems. A major one of these was that it scales very poorly as the complexity and size of the task increases, requiring additional computational power that was not readily available at the time. It did however provide novel solutions to the problems that it was applied to and despite these known issues, is worth revisiting with things such as competitive robotics (such as Robocup) growing in popularity, along with more every day problems in business and engineering becoming more and more automated by robotic machinery. Evolutionary Robotics may provide an inexpensive way of developing novel solutions to these complex problems in a hands off manner.

1.2 Introduction to Evolutionary Robotics

In the early days of Evolutionary Robotics [1], a population of robot controllers were generated on a computer. Each of these controllers would then be manually uploaded onto a physical robot which would then attempt to perform the desired task. The performance of the robot would then be passed back to the computer and evaluated for fitness against a task specific fitness function[6]. Population members with higher fitness values have a higher probablity of reproducing and continuing into the next generation while solutions deemed to have poor fitness values have a higher probabilty of being removed from the population entirely and 'dying out'.

In 1992 Cliff, Husbands and Harvey[1] proposed a method for developing controllers using evolutionary computation and suggested that design by hand becomes prohibitively difficult when complexity of desired behaviour is increased (Interestingly, a similar point was made by Alan Turing in 1950 [7]). The methodology that they introduced used an artificial neural network to act as the robotic controller, rather than an explicit program. In this paper, they also made an argument for the benefits of using a simulation over a real world setting, mainly concerned with the time constraints involved with real world evolutions. Although this study explicitly concerns only agents with a fixed morphology, another benefit worthy of note is that simulations allow not just the evolution of robot controllers but also the physical morphology of the agent itself [5].

Traditional approaches to controller development made only modest progress while being composed of fragile and very computationally expensive methods. Cliff et al, suggest that this may be linked to the Functional Decomposition ideology at the time, I.E. the idea that a robotic agent should follow the routine of "Sense, Plan, Act", now known as "Good Old Fashioned Artificial Intelligence" (GOFAI), and that these three things could be analysed independently of one another. They argued that developing a robotic agent by hand for useful tasks, while technically possible had become unfeasible.

ER uses the idea that desired behaviour in a robotic agent can be specified within a fitness function, making it a "hands off" approach to controller development without explicitly specifying at a low level, how the inputs of the agent should map to the outputs. This should result in iterative emergence of desired behaviour over a course of *n* generations. Rather than the classical decompositional approach taken by GOFAI, where systems can be represented by a set of discrete symbols goverened by rules and syntax to determine structure, ER is rooted in the concept of Dynamical Systems (DS). The main contrast between the two approaches is that DS is more concerned with

creating a model to represent the continuous change within the system itself, using differential calculus to describe it.
For the purposes of this study, the methodology originally suggested by Cliff, Husbands and Harvey [1] will be somewhat adhered to for the most part.

1.3 Overview of the project

The study evolves simulated robotic agents to fight in a sumo style competition, they are evolved with the objective of staying inside the confines of a circular sumo arena while forcibly removing another agent from the same confines, thus winning the competition.
The project is made up of the following main parts:

- Building a simulated environment.
- Training a single agent to stay within the arena.
- Pitting the agents against one another two at a time to fight in a sumo arena.

These aspects of the project will be covered in more detail in the methodology section of this report.

Professional Considerations

It is of the utmost importance that the project adheres to the code of conduct laid out by the British Computing Society (BCS) [19]. The BCS code of conduct has 4 sections and these shall each be addressed here.

Section 1 - Public Interest
You shall

- a) have due regard for public health, privacy, security and well-being of others and the environment.
- b) have due regard for the legitimate rights of Third Parties.
- c) conduct your professional activities without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement
- d) promote equal access to the benefits of IT and seek to promote the inclusion of all sectors in society wherever opportunities arise.

This section is not relevant as far as this project is concerned. There is no risk of public health, privacy, security or well-being being compromised as the study takes place in a simulated environment. There is no third party involvement or opportunity for discrimination of any other people or denying access to the benefits of IT to any sectors in society.

Section 2 - Professional Competence and Integrity
You Shall

- a) only undertake to do work or provide a service that is within your professional competence.
- b) **NOT** claim any level of competence that you do not possess.
- c) develop your professional knowledge, skills and competence on a continuing basis, maintaining awareness of technological developments, procedures, and standards that are relevant to your field.
- d) ensure that you have the knowledge and understanding of Legislation and that you comply with such Legislation, in carrying out your professional responsibilities.
- e) respect and value alternative viewpoints and, seek, accept and offer honest criticisms of work.
- f) avoid injuring others, their property, reputation, or employment by false or malicious or negligent action or inaction.
- g) reject and will not make any offer of bribery or unethical inducement

Over the period of this project the author has not and shall never claim any level of competence beyond that which they possess. The undertaking of the project in and of itself fulfils the requirement laid out in subsection C as by carrying out the studies involved in this project professional knowledge, skills and competence relevant to the field concerned are being developed. Subsection D is irrelevant to this study as there is no legislation applicable to the area concerned. Subsection E shall be met by attending regular weekly meetings with the project supervisor where any views presented shall be respected and valued and constructive criticisms shall be sought, acknowledged and where appropriate, incorporated into the project. Subsection F is not relevant as by running studies in a simulation there is no risk of injury to others, destruction of property or employment by false, malicious or negligent action or inaction. The author does not foresee any risk of bribery, corruption or unethical inducement and should any situation involving these things arise they shall be swiftly and firmly rejected.

Section 3 - Duty to relevant authority
You shall

- a) carry out your professional responsibilities with due care and diligence in accordance with the Relevant Authority's requirements whilst exercising your professional judgement at all times.

- b) seek to avoid any situation that may give rise to a conflict of interest between you and your Relevant Authority.

- c) accept professional responsibility for your work and for the work of colleagues who are defined in a given context as working under your supervision.

- d) **NOT** disclose or authorise to be disclosed, or use for personal gain or to benefit a third party, confidential information except with the permission of your Relevant Authority, or as required by Legislation

- e) **NOT** misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others.

For the purposes of the project, the term *Relevant Authority* (RA) has been deemed to be the School of Engineering and Informatics and in the wider view, the University of Sussex. The project shall be conducted in line with the requirements of the RA and professional judgement shall be exercised at all times. Any conflicts of interest shall be actively avoided between the author and the RA and where such conflicts of interest do arise they shall be brought to the attention of the RA immediately. The author accepts full professional responsibility for the work carried out over the course of the project and shall have no colleagues under their direct supervision. Subsection D is not relevant as no confidential information shall be used or accessible for the purposes of the project. The author shall not misrepresent or withhold information on the performance of products, systems or services or take advantage of the lack of relevant knowledge or inexperience of others.

Section 4 - Duty to the profession
You Shall

- a) accept your personal duty to uphold the reputation of the profession and not take any action which could bring the profession into disrepute.

- b) seek to improve professional standards through participation in their development, use and enforcement.

- c) uphold the reputation and good standing of BCS, the Chartered Institute for IT.

- d) act with integrity and respect in your professional relationships with all members of BCS and with members of other professions with whom you work in a professional capacity.

- e) notify BCS if convicted of a criminal offence or upon becoming bankrupt or disqualified as a Company Director and in each case give details of the relevant jurisdiction.

- f) encourage and support fellow members in their professional development.

As this project is not being conducted in a professional capacity, this subsections A,B,D,E, and F are not relevant. Subsection C is also not applicable as the author is not a member of the BCS.

Requirements Analysis

As the system has no explicit user, there are no ’user requirements’ of which to discuss. As a result of this, this section shall list the system requirements.

3.1 System Requirements

3.1.1 Functional Requirements

- The system shall provide a simulated environment in which an arena and robotic agents including sensors will be represented.
- The system shall allow a way of robotic agents moving according to the outputs of its controller
- The system shall implement robot controllers in the form of a Continuous Time Recurrent Neural Network
- The system shall be implemented in the Python programming language.
- The system shall implement a Genetic Algorithm for evolving the robot controllers.
- The system shall show the outcome of the simulation on the screen once the maximum number of generations has been reached.
- The system shall evolve a population of robot controllers for keeping the agents inside of the arena boundary.
- The system shall further evolve the agents that have been evolved to stay inside the arena, to push an adversary agent outside of the arena while itself remaining inside.
- The system shall be able to render the trajectories of agents to the screen and save these as figures for the final report.

- The system shall make use of the Pymunk physics library.
- The system shall make use of the Pygame graphics library.

3.1.2 Non-Functional Requirements

The system shall:

- The system shall provide a way of evolving robot agents to compete in a sumo style tournament inside of a simulated environment.
- The system should be complete by the final project hand in date of Wednesday 10th May 2017.
- Development of the system shall adhere to the BCS code of conduct.

Methodology

This chapter shall discuss the methodology and implementation of the study. It will detail the specifics of;

- **Simulation** - The implementation of the simulation environment and the choices made in this process along with justification for these choices.

- **Artificial Neural Network** - Theory and implementation details of the Artificial Neural Network used as robot controllers.

- **Genetic Algorithm** - The details on the Genetic Algorithms used for the study and justifications as to why these were used.

4.1 Simulation

Due to the large amount of time needed to evolve robot controllers in the real world along with the time limited nature of the project, the study shall take place in an artificial simulated environment. This allows for more efficiency in use of time along with the ability to easily repeat parts of the study if needed.

4.1.1 Selecting a library

The decision was made at an early stage to build the simulation by using a library already in existence which would provide the building blocks to enable the implementation of the custom simulation. The reasoning behind the decision to take this route was that of utilising the limited time in the most efficient manner. Building a simulation from scratch would be extremely time consuming and would ultimately detract from the purpose of the study. For the same reasons, a two-dimensional simulation was chosen over a three-dimensional simulation. Although three spatial dimensions could

potentially yield more realistically plausible results, the added realism would have come with a cost of added complexity which would consume more time to implement to a satisfactory standard.
Research was carried out as to which library, or combination of libraries would best serve the purpose of the study. A list of criteria was drawn up allowing various libraries to be assessed for suitability to the project.
The library **must**;

- Allow for the rendering of two-dimensional graphics to screen, representing the agents and their sensors along with some representation of the arena.

- Posses the capability for said graphics to be switched off during evolution of large populations.

- Have some degree of physics functions as standard such as friction.

- Be written in an implementation language that the author is familiar with as learning a new programming language would be a poor use of time.

After extensive research into these types of libraries it became clear that most of these criteria were predominately met by libraries created for writing interactive video game software. A total of four of these presented as viable options. These were;

Slick2d [8]
For use with Java. With very aesthetically pleasing visuals, Slick2D is designed for creating simple two-dimensional games in a object oriented style. Has no apparent physics functionality as standard.

Duality [9]
For use with C#. Again designed for game creation, however too complex for the scope of this project with no easily accessible API documentation.

Allegro [10]
For use with C++. A popular two-dimensional game library for beginners. A physics module for this library does exist, however appears to be rather

complicated to install and use.

Pymunk [11]
For use with Python. Based upon the chipmunk [12] physics library for use with C++. Pymunk is a physics library that is closely linked with Pygame library [13]. Pygame handles the rendering of graphics to the screen while Pymunk allows for the use of physics when objects interact. Comes with various types of collision handlers as well as allowing things such as sensors.

Pymunk was the library that most closely matched the criteria defined above and because of this, was chosen as the one to use and the study written using the Python 2.7 version of the Anaconda python distribution [14].

4.1.2 Building Blocks of the Simulation

This section will look at the objects and functions which when put together form the bulk of the simulation. The two main components that make up the simulation are; the *Bots* and the *World*. The bots shall be discussed first before moving on to the world.

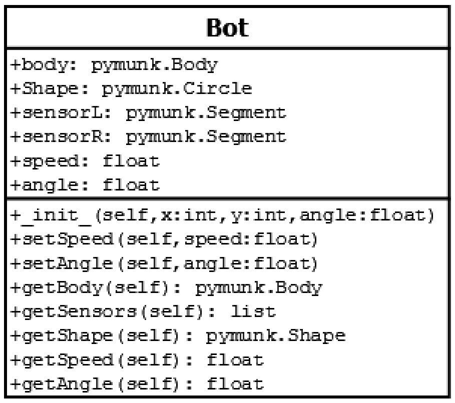
Bots

The bots are objects that represent the agents. Each bot is an instantiation of the *Bot* class, created for the project. A UML like description of the class can be seen in figure 4.1. The class field variables are;

- **body**: A Pymunk *body* object. Bodies are assigned a *shape* object and react to collisions, are affected by forces and gravity and have a finite mass. A shape is moved by assigning a velocity to its associated body along with an angle of orientation.

- **shape**: A Pymunk *Circle* object. A subclass of the Pymunk *Shape* class, intended by the authors of Pymunk to be treated as an abstract class (The two others being *Segment* and *Poly*. Renders a circle to the screen based upon the given parameters such as radius and location.

- **Sensors R and L** Two Pymunk*Segment* objects that are attached to the body. The *Shape* class has a boolean instance variable that allows for them to be marked as a sensor, meaning that collisions will not be



```
Bot
+body: pymunk.Body
+Shape: pymunk.Circle
+sensorL: pymunk.Segment
+sensorR: pymunk.Segment
+speed: float
+angle: float
+_init_(self,x:int,y:int,angle:float)
+setSpeed(self,speed:float)
+setAngle(self,angle:float)
+getBody(self): pymunk.Body
+getSensors(self): list
+getShape(self): pymunk.Shape
+getSpeed(self): float
+getAngle(self): float
```

Figure 4.1: A UML like description of the bot class.

processed but will be registered, allowing data from the collision to be accessed.

- **Speed** a float that determines the velocity of the body, which when moving also causes any attached shapes to move along with it. The velocity vector has two components x and y. Which are determined using the following formula:

  $$x = speed \times cos(\theta)$$

  and

  $$y = speed \times sin(\theta)$$

  where θ is the angle of agent's angle of orientation.

- **angle** a float that determines the angle of orientation of the agent.

The functions are all accessors and mutators of the bot apart from the constructor **_init_** which takes the speed, starting location and angle of each agent and assigns these to the instance variables. Although this is not the most 'pythonic' way of coding, it is what best suited the project.

The agents themselves do not have representations of left and right motors as a real world agent may have but rather the velocity and angle are controlled by directly setting these variables within the body of the bot object. This was deemed as the best way of implementing this feature as while Pymunk does have constructs that vaguely represent motors, none of them were deemed suitable to this study and so directly controlling the agents movements using velocity and angle was the most appropriate way of implementing movement controls.

World

The world itself is generated inside of the function **sim**(), called inside of the GA. The parameters for **sim**() are as follows;

- **w,theta** and **randIn** - These values to be used inside the neural network controller and shall be discussed in that section of the report.

- **pos** a tuple containing 3 values;
  - x - the x coordinate of the starting position.
  - y - the y coordinate of the starting position.
  - θ - the starting angle of the agent.

- **displayOn** - a boolean value to determine if the simulation should be rendered on the screen. This is only used once the GA has finished evolving a population and the fittest solution is rendered to the screen.

The function creates a Pymunk *space* object which holds all other objects in the world and is essentially the space in which everything takes place and allows elements to be added to it. A bot object is initialised and added to the space.
It was required to write two collision handler classes for this project; The first handles bot to bot collisions and the second handles sensor to bot collisions. These classes use collision handlers that come with Pymunk with the sensor to bot handler setting the sensor values later passed to the controller as inputs.
Friction is added to the simulation by setting a dampening value on the space. This prevents agents from continually speeding up and allows them to slow down when velocity values are reduced. The library does come with a concept of 'friction' but was poorly documented and is for use on shapes rather than the

world. As such dampening was a viable alternative that was used to give a similar effect.
The function then renders a circle to the screen, located in the centre of the space. This represents the boundary of the sumo arena and has a radius of 250 pixels and a thickness of 5 pixels.
A *'for'* loop is then entered for the duration of the time that the simulation has been set to run for where each loop represents one discrete time step. On each time step, a set of inputs is fed into the Neural Network and a set of outputs are returned. The inputs are, the sensor values, which for this task should both have a constant value of 0 (as no other agent is inside the arena to trigger a signal), *DistPen* which is a value calculated on how close the bot is to the arena boundary (if the bot is within 5 pixels of the boundary, it holds a value of 1 and 0 otherwise), **and two constant values of** 1. The agent's velocity and angle are then updated according to these outputs and the visualisation re-rendered if applicable before the next loop begins. During the loop, a record of the agent's speed, time spend inside the arena and a record of if the agent left the arena are recorded and upon completion of the function, these values are returned the GA that called it.

Sim Algorithm

```
sim(w,$bias$, randIn, displayOn, pos)
$initialise$
$outputs[0] \leftarrow [bot1.speed, botDist, randIn[0], randIn[1]$
for $i \leftarrow 1$ to $timeLimit$ do
    $inputs \leftarrow [sensorRight, sensorLeft, DistPen, 1, 1]$
    $outputs[i] \leftarrow ctrnn(inputs, w, bias, outputs[i-1])$
    $bot.setSpeed(outputs[0])$
    $bot.setAngle(outputs[1])$
end
if $botOutOfBounds$ then
    $oob \leftarrow$ True
else
    $timeSpentInside++$
end
return $outputs, oob, timeSpentInside$
```

This is a very abstracted pseudo algorithm of the sim function. The controller has been abstracted into a method call for the purposes of this illustration as the details of this shall be described in the next section of this report.

4.2 Neural Network Controller

The robotic controller used in this study will be a Continuous Time Recurrent Neural Network (CTRNN) [1] [2]. A CTRNN is especially strong in this area due to it being a dynamical system with temporal features. The CTRNN is able to process uniformly sized time slices while also having recurrent connections that allow it some degree of memory as previous outputs perpetuate inside of the hidden layers of the network via the recurrent connections. The main operation of the CTRNN model can be described as follows;

$$\frac{dy_{i+1}}{dt} = -y_i + tanh[\sum w_{ij} y_i + \theta + I_i]$$

Where y_i is the output of the network at time step i, w is the matrix of weights associated with the connections between nodes, θ is a bias value and I_i is the input to the network at time step i. The activation function is a $tanh()$ sigmoid function.
As inputs the network takes;

- Right sensor Value

- Left sensor Value

- Distance Penalty - calculated on each time step, has a value of 1 if the agent has a distance to the boundary of < 5 and 0 otherwise.

- Two dummy inputs which have a constant value of 1. This adds more complexity to the network allowing for more dynamic behaviour by adding extra recurrent values which can be seen in figure 4.2.

These inputs to the network take the form of a $5 \times T$ input matrix I where T is the number of time steps in the simulation. The column I_i is determined on each time step i and is as follows;

$$I_i = \begin{bmatrix} Sensor1Val \\ sensor2Val \\ botDistPen \\ 1 \\ 1 \end{bmatrix}$$

The outputs of the network are stored in a matrix Y. On the first iteration, y_0 is created and filled with the agent's current speed, angle, distance to centre and the two *randIn* values. These values are generated alongside the population of weights during the first iteration of the Genetic Algorithm and act as initial starting values for the recurrent connections from the last two output nodes. As the CTRNN is a dynamical system, starting conditions make a large amount of difference as these *randIn* values perpetuate in the memory of the network causing different outputs through time. When mutation occurs inside of the GA, the *randIn* values associated with mutated population member also get mutated to the same degree. The use of such dummy nodes allows for more complexity in the outputs of the network when compared to just the three meaningful nodes as there is more interaction occurring between nodes. This can be seen in fig 4.2.

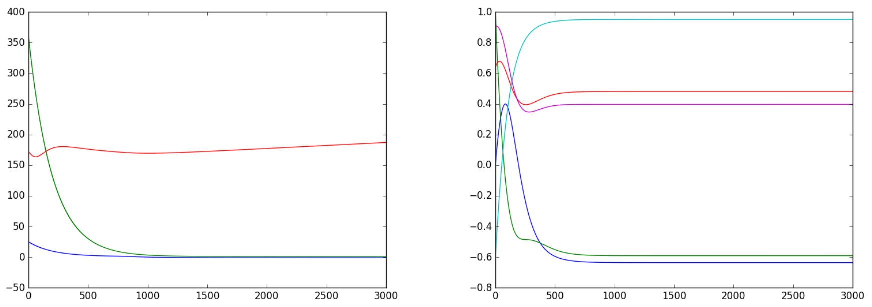


Figure 4.2: Examples of network outputs over time, with 3 nodes on the left and 5 on the right.

By using more nodes, more information is inside the network and thus there are more factors that determine the outputs of the other nodes. This equates to added complexity, which may contribute to more complex behaviour from the agents.

These outputs are then appended to the output matrix Y and the first two are scaled to values between min and max speed and min and max angle before being fed back into the agent and assigned as the speed and angle respectively.

A visual representation of the network can be seen in figure 4.3. In this figure, the nodes on the far left are the input nodes, the middle nodes represent the hidden layer containing the $tanh()$ activation function and the nodes on the far right are the output nodes. The small circle represents the bias, the value which is given to each of the nodes in the hidden layer. These are initialised to 0 in the first population and are mutated later. The solid connections are the feed forward connections while the dashed, backwards directional lines are recurrent connections. These are the connections that are associated with the weights, evolved in the genetic algorithm.

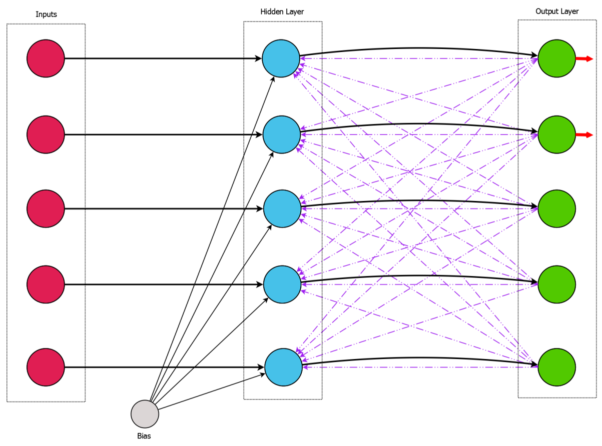


Figure 4.3: A visual representation of the network structure used in the study.

4.3 Genetic Algorithm

This section will cover the implementation specifics of the two Genetic Algorithms (GAs) used in this study. The approach taken is that of splitting the evolution into two main stages. The first of which concerns the task of evolving a single agent to stay within the confines of the arena and the second to make the agents engage in combat. These will be discussed in this order.

4.3.1 Evolving to Stay Inside the Arena

This GA evolves a single agent to stay within the confines of the sumo arena with no other agent present. The population consists of a number of 5×5 weight matrices according to the population size, P, giving a $5 \times 5 \times P$ overall population matrix. This is randomly drawn from a Gaussian distribution with mean 0 and $\sigma = 1$. The randIn was drawn the same way with a number of $2 \times 1 \times P$ values. The biases are initialised to a vector of zeroes of size P.
The main mechanics of the GA take the form of a Steady State GA. With a population size P, on each iteration, 2 members are selected at random,

compared for fitness and the loser replaced by a new population member. This is then repeated P times which equates to one generation in the traditional Generational GA, in which the entire population is replaced each generation. For the sake of simplicity, P iterations of the GA will be referred to as a *Generation* throughout this report. This methodology has benefits in that it is more memory efficient than a traditional Generational GA(Which requires 2 matrices, one for storing the current population and a second for the next generation of the population) and in some ways more biologically plausible in the sense that, in nature a species does not have events whereby the entire current population dies out and is replaced by a new population. Births and deaths happen simultaneously and independently of one another.
Selection takes place using binary tournament selection [15]. On each iteration, two population members are selected at random and their fitness scores are compared; with the winner (higher fitness) being selected into to the next generation and the loser (lower fitness) having all of its genes overwritten by a mutated version of the winner.This approach is that of the Microbial GA [20] proposed by Inman Harvey. In the Microbial GA, the probability of solutions being selected is directly proportional to its fitness. In other words, the fittest population member will be selected every generation having a probability of $p = 1$ of being selected, the least fit will always be replaced, with a probability of $p = 0$ and the population member with the median fitness will be selected with probability $p = 0.5$. Tournament selection has the additional benefit of a $O(n)$ time complexity along with the inherent suitability for parallel processing. An attempt was made to implement a parallel version of the algorithm, however due to limitations of the Microsoft Windows Operating System on which much of the study took place it was deemed a non-viable option.

During mutation all of the population member's genes (i.e the weights), randIn and biases are altered by $\pm 1\%$. This mutation rate was selected as the weights are very small values and setting a mutation rate that is too high can result in non convergence with population members becoming too spread out over the fitness landscape, resulting in what is essentially a random search. There is no sexual reproduction. Exchange in genes between population members resembles the concept of bacterial conjugation, whereby the DNA of one population member is directly replaced by that of another.

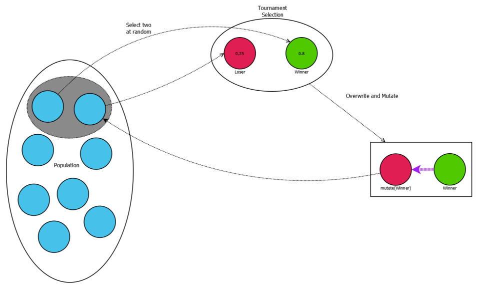


Figure 4.4: Visual representation of the GA select/mutate cycle

The fitness function is used as a way of assessing fitness and assigning a fitness value to a population member. The value assigned is based upon the following criteria;

- If the agent went out of bounds
- Mean speed
- Time spent inside the arena

If the agent did not go out of bounds then a value of 1 is added to the fitness. As this is the main task that the agents are being evolved for at this stage, it gets the highest weighting inside the fitness function. The value directly proportional to the mean speed is added to the fitness value. As a maximum/minimum speed of ± 25 is encoded into the simulation, a mean speed of 25 increments the fitness value by 0.5, a mean speed of 0 increments the value by 0 and a mean speed of -25 decrements the value by 0.5 This served the purpose of not allowing the agent to stay still in order to stay within the arena and to encourage the agent to move in a mostly forward direction while not completely putting a penalty on slight backwards movements that may be beneficial in defensive manoeuvres. Time spent inside the arena was returned as a value to which 1 was added for each time step that the agent stayed inside. This was used as a way of differentiating between members of a population in which none remained inside. The ones that remained inside longer were deemed to be fitter. This scaled in a similar way to the mean speed whereby a maximum vale of 200 is returned if the agent stayed in completely, this is then scaled such that if the agent remained inside, a value of 0.5 is added to the overall fitness and 0 if the agent spent no time inside the area whatsoever(Which is impossible) with values in between, scaled accordingly, meaning that an agent that spent half of the time inside will get a score of 0.5. It may seem that using both a check for out of bounds and using the time spent inside, that the out of bounds check is redundant as by staying in the entire time, the maximum value will be given. However, the time spent inside value is there to differentiate between solutions that did not quite manage to stay inside and the out of bounds check gives an extra boost to the agents that remained inside the whole time. This avoids the scenario where an agent achieves a high mean speed and doesn't remain inside the arena being given a similar fitness value to one that remained inside but only had a moderate mean speed. Using an out of bounds check tips the balance in the favour of remaining inside while still applying pressure on maintaining a reasonably high mean speed. A visual representation of the GA can be seen in fig 4.4 and the pseudo code follows.

Stay Inside Algorithm

```
$popSize \leftarrow s$
$numGems \leftarrow m$
$pop \leftarrow randn(5, 5, popSize)$
$biases \leftarrow zeroes(popSize)$
$randIn \leftarrow randn(2, popSize)$
for $i \leftarrow 0$ to $popSize * numGens$ do
    $rand1 \leftarrow randInt(0, popSize)$
    $rand2 \leftarrow randInt(0, popSize)$
    while $rand1 == rand2$ do
        $rand2 = randInt(0, popSize)$
    end
    $f1 \leftarrow fitness(sim(pop[rand1], biases[rand1], randIn[rand1])$
    $f2 \leftarrow fitness(sim(pop[rand2], biases[rand2], randIn[rand2])$
    if $f1 > f2$ then
        $pop[rand2] \leftarrow muatate(pop[rand1])$
        $biases[rand2] \leftarrow mutate(biases[rand1])$
        $randIn[rand2] \leftarrow mutate(randIn[rand1])$
    else
        if $f2 > f1$ then
            $pop[rand1] \leftarrow mutate(pop[rand2])$
            $biases[rand1] \leftarrow mutate(biases[rand2])$
            $randIn[rand1] \leftarrow mutate(randIn[rand2])$
        end
    end
end
```

As this is only a pseudo algorithm, some of the lower level details have been abstracted for the benefit of clarity and conciseness. Where the $f1$ and $f2$ values are assigned, the *sim()* function is called twice placing the agent at each of the two starting positions. The GA is ran for $P \times M$ iterations, where P is the population size and M is the number of generations to run for.

Fitness Function

```
Data: oob,timeInside,Speed[]
$fitness \leftarrow 0$
if not oob then
    $fitness += 1$
end
$fitness += mean(speed)/50$
$fitness += timeSpentInside/400$
return $fitness$
```

4.3.2 Evolving Two Agents to Fight

Once an agent had been successfully evolved to to stay inside the arena, a second population of agents were created. This contained the successfully evolved agent along with a number of mutated versions of it and the biases and randIn values, also mutated accordingly. An illustration of this can be seen in fig 4.5. These members and their associated values are passed into a **sim**() function that has been modified to processes two agents but is identical in every other way to the function used in the previous step. The population members passed into the simulation are chosen at random and the simulation is ran twice per pair with one agent starting in the first position and the other agent in the second starting position on the first run and then switched for the second run. After each run, both agents are assessed for fitness according to the new fitness criteria.

The function **fightFit**() assesses the population members by looking at;

- Number of times the agent hit the other
- If one agent was knocked out by the other

As the agents have already been evolved to stay inside the arena, there exists an edge case in which both agents will remain in the arena for the same length of time while not colliding at all. In this case a mechanism is needed to resolve the the fitness values in the situation that they are identical. In the event that this event occurs, a small value is added randomly to one of the bots. While not a particularly elegant method of solving this problem, it is effective nonetheless.
This GA works in a similar manner as the one used to evolve agents to remain inside the arena. It follows the fundamentals of a Steady State GA with binary tournament selection while using the elements from the Microbial GA to

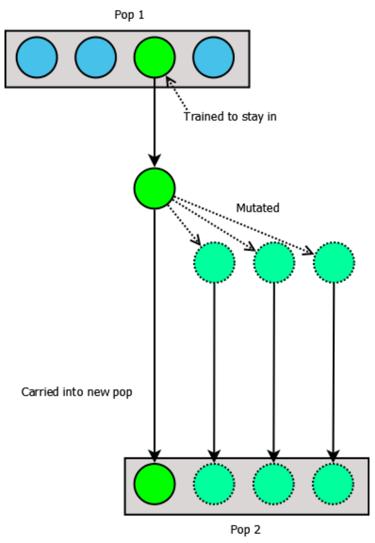


Figure 4.5: Illustration of second generation creation

overwrite the loser of the tournament with a modified version of the winner of the tournament. The algorithm can be found below. While the concept of this phase of the project is to take the winner of each fight as the winner of the competition, a fitness function is still required to determine which agent won. Secondly, a decision must be made in the event that the agents both remain inside of the arena but never meet. The fitness function serves this purpose.

4.3.3 FightFit Algorithm

```
Data: timeInside[],numHits[]
$fitness \leftarrow []$
$fitness[bot1] \leftarrow timesHitOther$
$fitness[bot2] \leftarrow timesHitOther$
if $bot2KO$ then
    $fitness[bot1] += 100$
end
if $bot1KO$ then
    $fitness[bot] += 100$
end
$fitness[bot2] += numHits[bot2]/100$
if $fitness[bot1] == fitness[bot2]$ then
    $fitness[rand()] += 0.00001$
end
```

4.3.4 FightGA Algorithm

```
$oldPop, oldBiases, oldRandIn, fittest \leftarrow loadData()$
$pop.append(oldPop[fittest])$
$biases.append(oldBiases[fittes])$
$randIn.append(oldRandin[fittest)$
**for** $i \leftarrow 0$ **to** $popSize - 1$ **do**
    $pop.append(mutate(oldPop[fittest]))$
    $biases.append(mutate(oldBiases[fittest]))$
    $randIn.append(mutate(oldRandin[fittest]))$
**end**
**for** $i \leftarrow 0$ **to** $popSize * numGens$ **do**
    $cand1 \leftarrow randInt(0, popSize)$
    $cand2 \leftarrow randInt(0, popSize)$
    **if** $cand1 == cand2$ **then**
        $cand2 \leftarrow randInt(0, popSize)$
    **end**
    $fit1, fit2 \leftarrow$
     $sim(pop[cand1], pop[cand2], biases[cand1], biases[cand2] randIn[cand1], randIn[cand2])$
    **if** $fit1 > fit2$ **then**
        $pop[cand2] \leftarrow mutate(pop[cand1])$
        $biases[cand2] \leftarrow mutate(biases[cand1])$
        $randIn[cand2] \leftarrow mutate(randIn[cand1])$
    **else**
        **if** $cand2 > cand1$ **then**
            $pop[cand] \leftarrow mutate(pop[cand2])$
            $biases[cand] \leftarrow mutate(biases[cand2])$
            $randIn[cand1] \leftarrow mutate(randIn[cand2])$
        **end**
    **end**
**end**
```

Results and Discussion

The size of the population can increase the diversity of the potential solutions within the population. The larger the population size, the more likely it is that the initial population contains an optimal solution. As the size of the population grows arbitrarily large, the probability of finding an optimal solution approaches 1.[16].
Tests were initially conducted with a population size 15 running for 10 generations. This produced mediocre results with a mean fitness of 1.2 across 3 trials. The two best solutions to come from these tests both stayed in the arena however were scored poorly on other mean speed. The first appeared to move in a wave like pattern around the boundary of the arena and could prove to become a good solution given more time to evolve. The second scored poorly on mean speed as it approached the boundary and slowed to a halt, meaning that while it stayed within the confines of the arena, it was a poor solution in the wider scheme. This can be seen in fig 5.1 with the generational fitness plot being shown in figure 5.2. Figure 5.2 has been plotted with a y axis starting a 0.4 to highlight the slight increases in fitness at the top of the plot that would not have been visible with a y axis starting at 0. The disparity between the top two fitness plots and the bottom one, shows the importance of the initial starting conditions, as a population seeded with very weak solutions will take longer to converge on an optimal solution, depending factors such as the mutation rate.

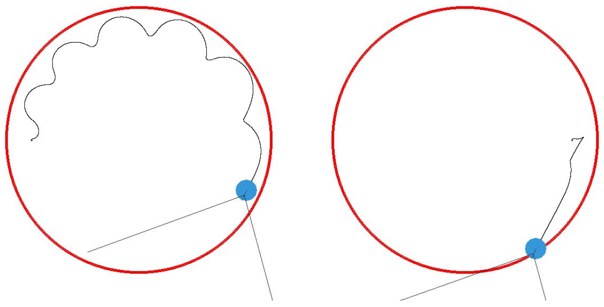


Figure 5.1: The fittest solution from the first test. The outer circle represents the arena boundary while the smaller circles represent the agents with the lines being the trajectory the agent took.

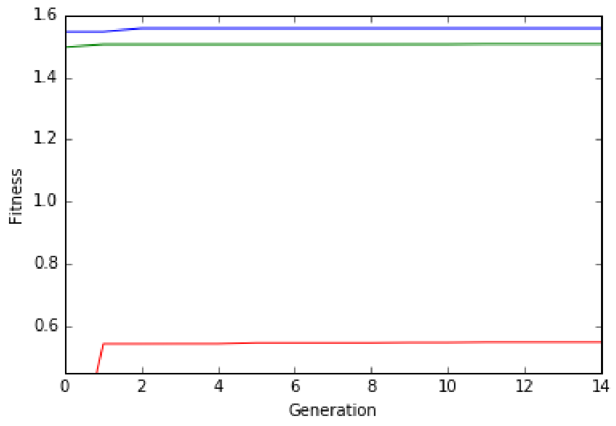


Figure 5.2: Plot of 3 runs of GA with population size 10 for 15 generations.

The number of generations was then increased to 40 with a population size of 20. As can be seen in figure 5.3, increasing both the size of the population and the number of generations increased the mean fitness to 1.52 over 3 trials. For the test with the highest final fitness, the value steadily increased, while for the other 2 tests, fitness increase appeared to stagnate.

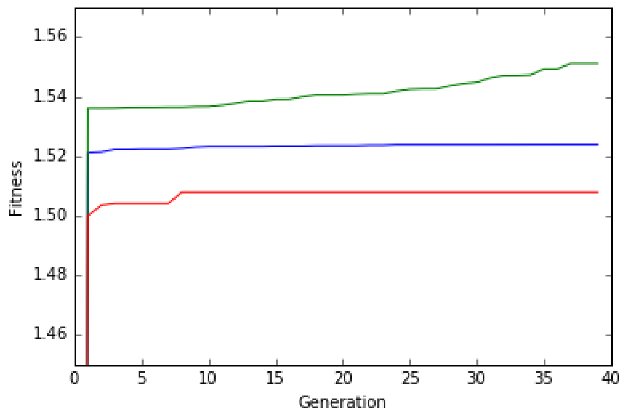


Figure 5.3: Plot of 3 runs of GA with population size 20 for 40 generations. Y axis has begins at 1.4 so as to highlight small increases in fitness that were not visible with a y axis starting from 0.

The solution that produced the best robot trajectories can be seen below in figure 5.4. While the fitness increase of the larger population size and number of generations was not overly large, the trajectories did appear to improve with more complex behaviour being displayed by the agents, with the fittest remaining inside the arena from both starting points.

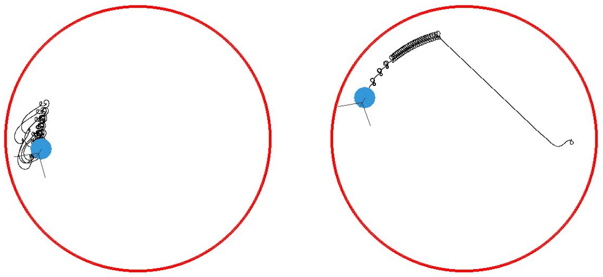


Figure 5.4: Trajectories of the best solution from 40 generations with population size 20

Finally, the population size was increased to 30 and the number of generations to 60. The results can be seen below in figure 5.5. It appears that during these tests, fitness increases rapidly early on, however begins to stagnate and increases in fitness are not as steep. The trajectories for the fittest solution can be seen in fig 5.6. As in previous runs, both agents stayed within the arena, with the trajectory from the first starting position being the better of the two. The agent moves in a forward direction and begins to go in a circle parallel to the boundary. In the second, the agent appears to turn around before movement grinds to a halt. The mean fitness over 3 trials was 1.5.

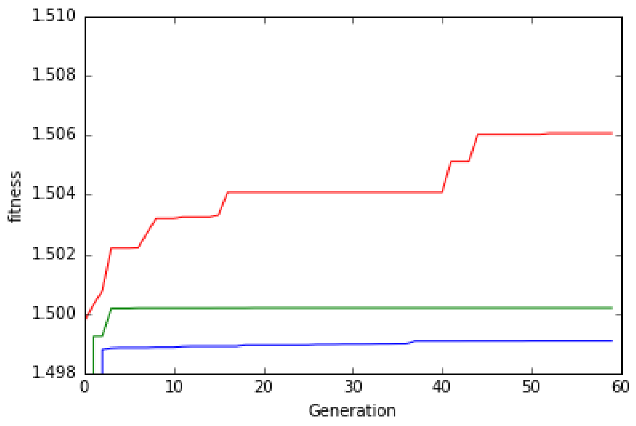


Figure 5.5: Plot of fitness values for GA with population size of 50 and 100 generations.

Based upon the results of these tests, it appears that a population size of 20 over 40 generations is most suitable for this study. Although one trajectory for the larger test was interesting, the time taken to run a large number of simulations with these parameters does not justify the results in a time limited study. Were more time allowed for this, it would ideally have been explored along with larger tests.

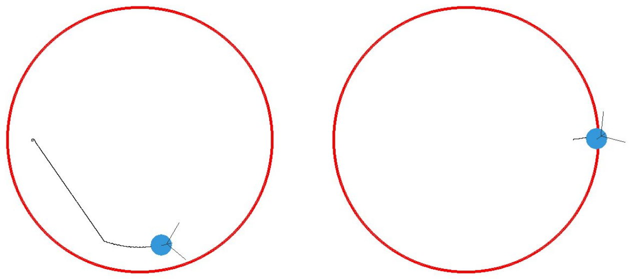


Figure 5.6: Trajectories for the fittest solution after 100 generations with a population size of 50.

Once a good solution had been reached, that stayed inside of the arena, the final task was to evolve it further to engage in combat with an adversarial agent also inside the arena. Following the steps mentioned in the methodology section, an agent was evolved to stay inside the arena and then used to spawn a new population of candidate solutions with a population size of 10 and ran for 20 generations. The trajectories of which can be seen in figure 5.7

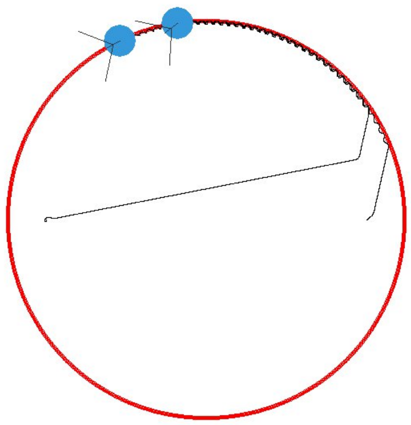


Figure 5.7: Trajectories of fittest and second fittest after 20 generations with population size of 10.

When generating the figures for trajectories, the fittest agent is put up against the solution with the second highest fitness. In figure 5.7, Both agents appear to be moving around the inside of the boundary, however no combat is taking place. This was generally expected from such a low population size coupled with a low number of generations and thus these values were increased and the simulation ran again.
For the second test, the population size was increased to 20 and number of generations to 40. This combination gave the best overall results when training to stay inside the arena and it was thought that this may also be the case for this second phase of the study. The results can be seen in figure 5.8.

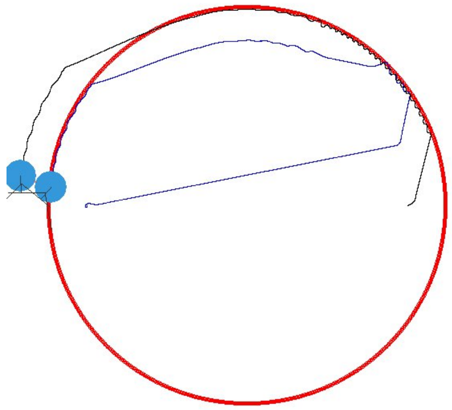


Figure 5.8: Results of running GA with a population size 20 for 4 generations.

After training for 40 generations with a population size of 20, performance was improved. Contact was made between the two agents where the trajectories meet. Where they diverge and then become somewhat parallel is where the winning agent is forcing the losing agent out of the arena and moves in a way such that it will not allow the other agent back into the arena. Although in terms of sumo, this is not entirely relevant to the outcome of the trial, it is an interesting behaviour all the same. A clear evolutionary step has been taken between the first and second trials.

So far, the agents evolved to fight are offshoots of the same agent evolved to stay inside the arena with some small mutation. While it has been demonstrated that this is sufficient enough to begin to see some fighting behaviour, further steps were taken to investigate if this could be improved further.

The first step taken was to increase the size of the sensors on the agents as a way of giving them a longer line of sight. It was hypothesised that this would enable the agents to 'see' the other, in sooner, from further away and that this would cause them to alter their trajectory earlier and engage each other more quickly. Sensor size was increased by a factor of 4. The population size and number of generations remained at 20 and 40 respectively.

As seen in figure 5.9, increasing the size of the sensors appeared to have an impact upon the trajectories of the agents. When compared with those in figure 5.8, the winning agent appears to make much more aggressive and faster turns, making the first turn to its left sooner than it did in the previous test. The behaviour remained somewhat the same in that the winning agent pushed the loser out of the arena and took actions such that it did not allow it back inside. However, this was expected as all agents are mutations of the same population member, which when examined in terms of the fitness landscape means that candidate solutions are all very close to one another in fitness space and may converge on a local optimum.

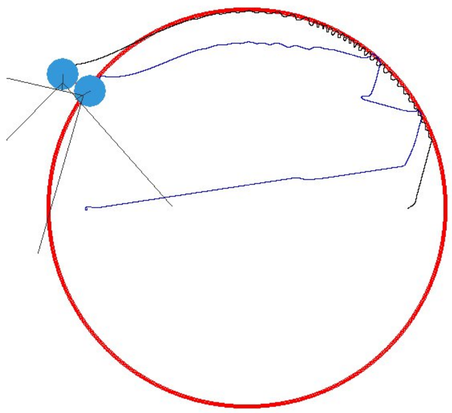


Figure 5.9: Results of increasing the sensor size by a factor of 4.

As a measure to inject some diversity into the population, 5 solutions were trained to stay inside the arena and the new population trained to fight used the fittest of these as seed members whereby they were each added to the new population themselves and every other member was a mutation of one of these with an equal distribution. For example, in a population of size 20, 5 of the population members would be the agents that were trained to stay inside and the remaining space would be filled by 3 mutations of each of the original 5. While running this test, the length of the sensors were again increased by a further 50%. Results of this change show that while some diversity has been added to the gene pool, the duplication used creates a situation where one type of agent becomes dominant over the others. To eliminate this further it is hypothesised that evolving *N* agents to stay inside the arena and using each one of these as a unique member of the new population evolved to fight would create a much more diverse population, covering a wider area of the fitness landscape. Unfortunately there was not enough time to implement this at the late stage of the of project that it was discovered but would be an area to be considered in future work.

The results seen in figure 5.10, show one agent on a much more curved trajectory than in previous examples as the agent alters course but also accelerates when it senses the other. The winner, in this example the agent that started on the right hand side, appears to defensively dodge the attacker and then proceeds to push it out of the arena, showing that a defensive strategy has been evolved rather than an offensive. As this is difficult to convey in a static image, a video recording has been included in the appendix of this report which links to a dropbox url containing the video. An interesting point to note is that for some solutions the sensors appear to induce an excitatory response in movement, meaning that sensor activation causes the agent to accelerate while in others they have evolved a more inhibitory response causing movement to slow down and eventually cease. Again, as this is difficult to illustrate using a static image figure, a link has been provided in the appendix to a dropbox containing a video recording. It seems that agents with more excitatory reactions to inputs engaged the other agent more often than those with inhibitory sensors, which in some cases completely froze until the other agent was out of sensor range. Further more, as another illustration of the benefits of increasing the size of the sensors, the same simulation was ran again using the same agents, but with the sensor size reduced to what the originally were. When doing this, the trajectories were entirely different, with agents never making contact and no fight occurring. This provides further evidence to support that the size of the sensors has a direct impact on the behaviour of the agents. This can be seen in figure 5.11.

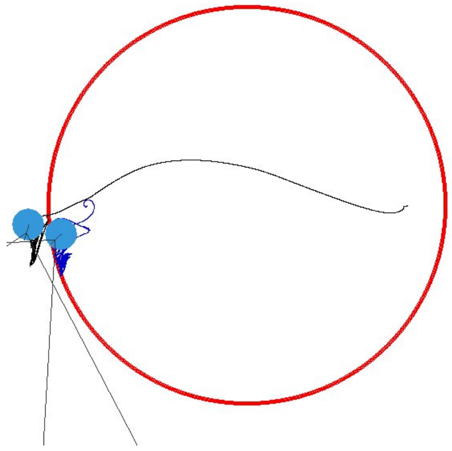


Figure 5.10: Trajectories of fittest solution when population is seeded with 5 different solutions trained to stay inside the arena.

An interesting point was noticed when the agent trajectories were cross examined with the traces of the outputs from the controller over time. An example of this can be seen in figure 5.12. In this figure, a distinct oscillation can be seen in both the trajectories and the network outputs. In the top example, what appears to be a chaotic movement begins, before a pattern of oscillation arises and then settles down. A similar behaviour occurs in the second example with the difference being a small period of what appears to be a constant linear output before the oscillation begins.

As these figures have been generated from agents already trained to stay inside the arena, it would be of interest to compare these to both (i) untrained agents which have been randomly generated and (ii), agents that are actively engaging in combat. These figures can be seen below in figs 5.13 and 5.14.

These figures show that the agents that have been evolved to both stay in the arena and to fight, both show some form of oscillation in the network outputs. These oscillations can also be seen in the trajectories of the agents and indicate that this is an important feature for a controller to develop. In the second example, there are clear spikes while the sensors are sending a signal. This explains why behaviour is radically different during the time when the presence of another agent is sensed.

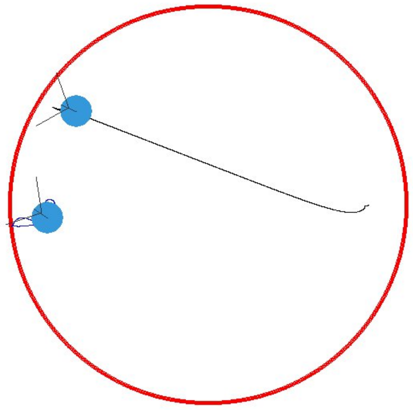


Figure 5.11: Agent trajectories of the same agents used in 5.10, with smaller sensors

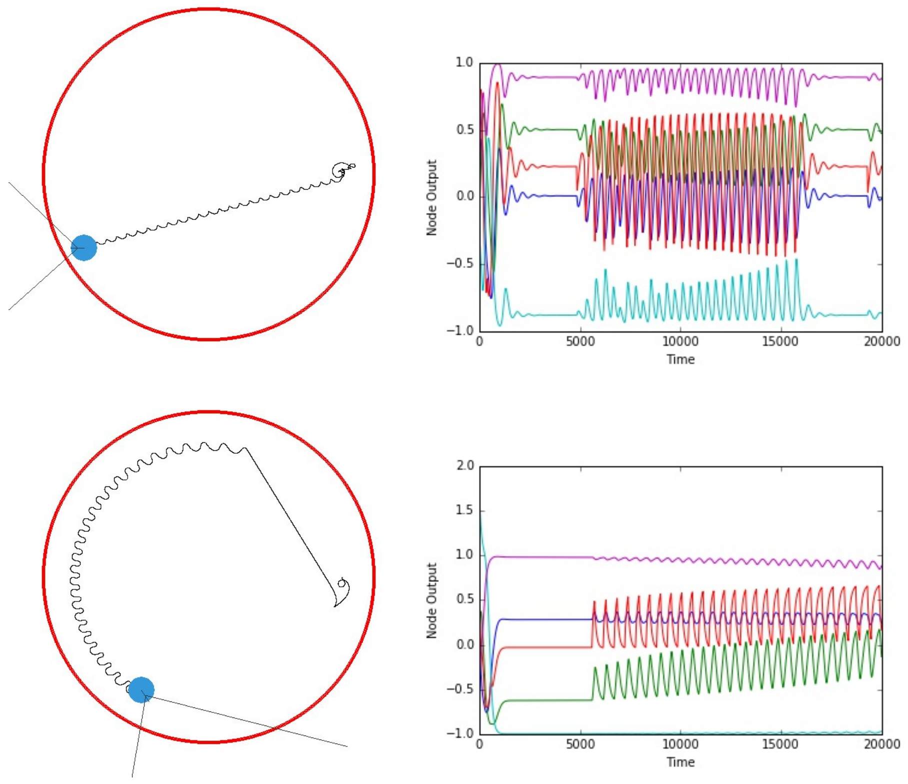


Figure 5.12: Agent trajectories which have been evolved to stay inside the arena on the left with the corresponding network outputs on the right hand side.

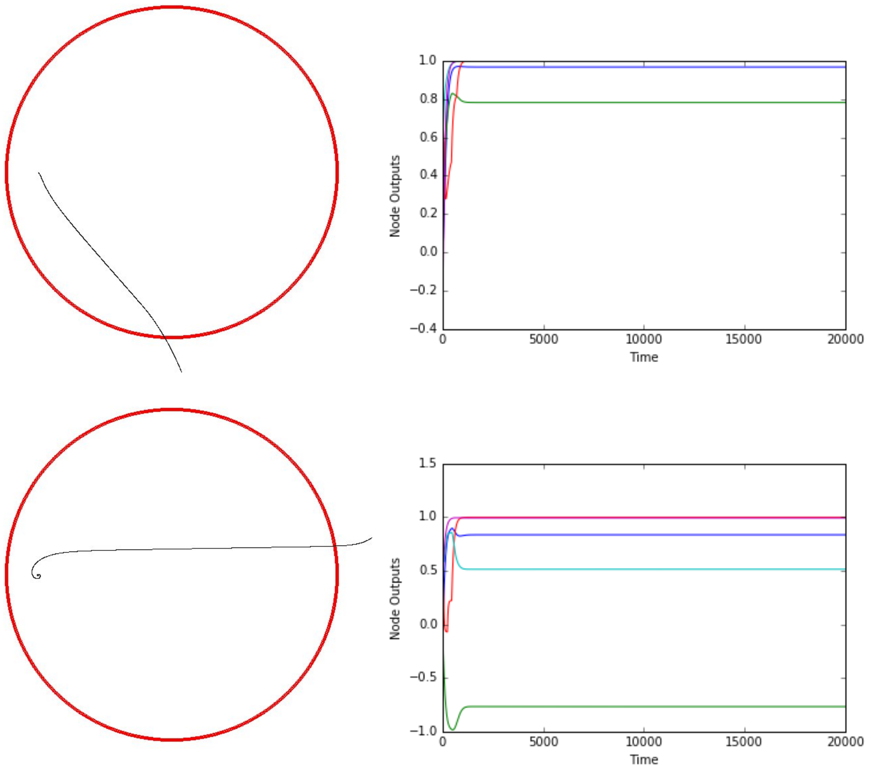


Figure 5.13: Non evolved agent trajectories on the left with corresponding network outputs on the right hand side

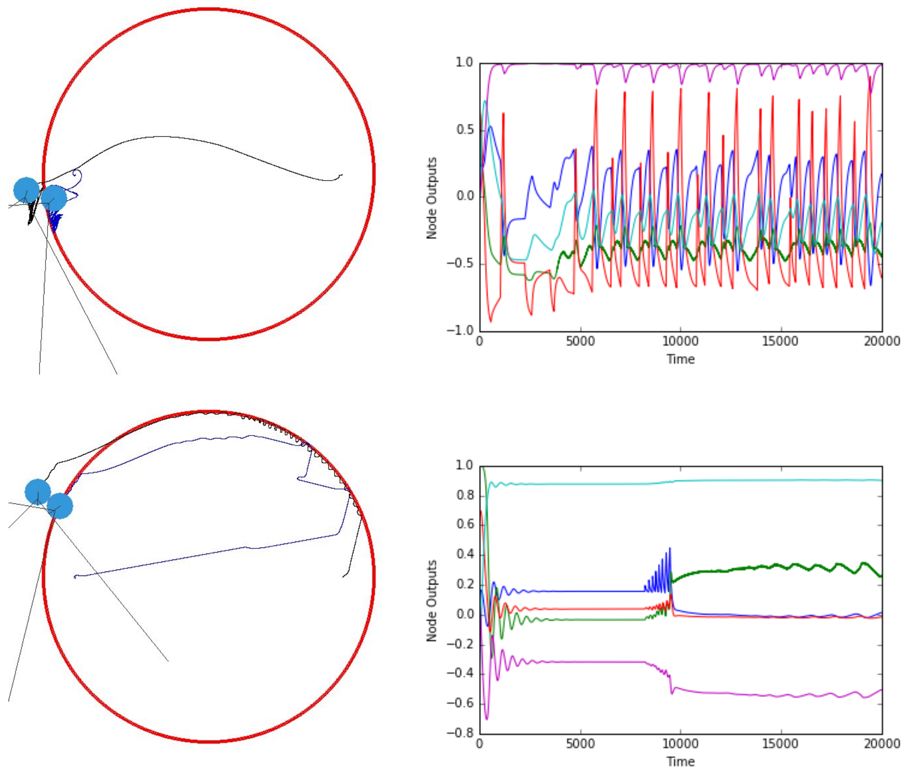


Figure 5.14: Trajectories and Network outputs of the agents successfully evolved to fight.

Conclusion

6.1 Discussion and Evaluation

In the beginning of the report the project objectives were laid out as follows:

- To explore the concept of evolutionary robotics in a simulated environment using a sumo style competition as the problem domain.

- To implement a novel way of assessing fitness of the population by means of direct competition based on performance in the problem domain where the winning solution moves forward in to the next generation of population members and the loser is replaced by a mutated version of the winner. This is contrasted with the idea of co-evolving two separate populations.

- To investigate whether a return to evolutionary robotics could potentially hold the key to creating novel solutions in today's technological climate.

By running various experiments while changing certain free parameters and analysing the impact that those changes had upon the overall solutions produced, evolutionary robotics has been examined using a sumo style competition as the problem domain.
The way that population members are selected during the fighting evolution stage of the project fulfils the second objective. Although there is technically a 'fitness function' that exists, it is merely used as a way of determining which agent won the fight, I.E if one agent knocked the other out of the arena.
For the third objective, results appear to be inconclusive. While the evolution of robotic controllers for the specified task was marginally successful, the study has failed to answer the question of suitability of a return to evolutionary robotics holding the key to robot development in today's world. Further work

would need to be carried out in order to give a meaningful answer on this and this is discussed in the section of this chapter entitled *Further Work*. One area that this study has highlighted, mainly from using libraries meant for video games, is that evolutionary methods may be suitable for creating systems inside of a video game where some degree of artificial intelligence is required. Such things like non player characters and adversarial opponents may lend themselves to being developed in this manner. Again, more work would be need to be carried out to investigate this further but this study provides a small amount of evidence to support this hypothesis.

As a whole, the project was semi-successful, as while some objectives were met, there were issues that presented themselves.

Firstly, the capability of the entire project rests upon how well implemented the simulation is. Any fundamental errors in the simulation translate into errors for the overall project. This was the most challenging area of the project by a vast margin and it is vital that this is not understated. While Pymunk [11] and Chipmunk [12] that is based upon are both solid libraries that have wide use in games, they were not designed for the purposes that this study has used them for. The main challenge that using this library presented was that while the web page has a section of APIs and access to source code, it was at times unclear or ambiguous. Certain data types are poorly explained in documentation for both of these libraries. This caused a lack of information meaning that the simulation was implemented partly on guesswork and as such was not as accurate as it should have been. One main area that this was most prominent was the collision handlers. When a collision occurs an *Arbiter* object is generated containing information about the collision, however the methods contained within this object are very vaguely document. Another point of note is that objects contain certain methods that are nowhere to be seen within the library documentation, meaning that guesses had to be made from looking at code from example programs provided or where this wasn't available pure conjecture. This coupled with the fact that by building a simulation I was deep in unfamiliar waters, around half of the time spent on the project was spent on this task. When taking a step back, certain sacrifices had to be made in the form of moving on with other areas of the project with an imperfect simulation model.

The second area that presented a huge challenge, was the fitness function used when training an agent to stay within the arena. Choosing what information to select on and what weighting each contributing factor should have proved to be a long and arduous process which, at the end of the project still hadn't been completely perfected. While some progress was seen, and agents were successfully evolved to stay within the arena, the behaviour shown wasn't

overly complex. Similar to the last point, time constraints played a big part in this area being somewhat under developed. Had less time been spent on the simulation, this may not have as prominent.
If the study were conducted a second time with the knowledge of facts stated above there would most definitely be some changes;

- Although most of the literature states that using a simulation is the least time consuming and most efficient way of approaching evolutionary robotics, in this case taking a real world approach may have saved an enormous amount of time. As a significant amount of time was spent just building a sub-standard simulation environment, using a real world approach would have allowed more time to have been spent on the actual research topic rather than concentrating on fundamentals of simulation building. For this approach, the robots used would have been Lego Mindstorm robots using the Robot C language. By marking a circle on a large enough floorspace and leaving the robots plugged in to a desktop machine over a long period of time, trials could have been ran whereby the agent feeds the information back to the desktop machine after each run and returns to an initial starting position to begin the next trial. By using this methodology, the robots could have been left unsupervised for long periods of time to evolve. Clearly doing this would introduce complications of its own, time taken being the main one, however I believe that this would pale in comparison to the time spent and frustration caused at the simulation that was built.

- Another alternative to the simulation used would have been to have used an already existing piece of software that simulates robotics, however when searched for, only 3D applications designed for this purpose exist. As using 3D simulations would have scaled the task up in size of complexity, this was avoided in favour of keeping complexity low. Secondly it was thought that by implementing a more custom built environment, more control could have been exerted over the specifics of the simulation.

- As stated previously, the behaviour demonstrated was not very complex, even though the end objective of making the agents fight was somewhat of a success. I hypothesize that by adding more neurons to the CTRNN controller, more complex behaviour could have been achieved. It would have been interesting to have run further tests exploring the impact of adding more neurons to the network.

6.2 Further Work

Were more time have been allocated to improve upon the study and extend the work, there are two main things that would be implemented;

(1) I would be interested in adding an element of ranged combat to the agents allowing them to fire projectiles at the other in order to knock them out of the arena. It would be interesting to see if this also creates more complex defensive strategies by dodging these projectiles. By only allowing the agents to fire in the position they are facing, it may allow a dynamic whereby one agent sneaks upon the other from behind and forces it out. Although this no longer fits the sumo style competition originally laid out in the introduction, a personal curiosity makes adding this dynamic interesting. There are examples on the Pymunk documentation page that show that this type of functionality can be added. When an attempt was made to incorporate this towards the end of the project, the poor documentation again proved to be a hindrance to this preventing any solid implementation.

(2) Exploring the idea of replacing the CTRNN controller with that of a Long Short Term Memory (LSTM) recurrent neural network. An LTSM is a recurrent artificial neural network that allows for weight updates using gradient descent as it allows credit assignment to be carried out during the process of back propagating the error. An advantage LSTM has over CTRNN is that memory inside of a CTRNN can diminish over time, causing signals to 'die out' or memory to leak. LSTM does not have this problem due to the way it uses a gating system to store previous outputs. LSTM has been evolved using the method of NeuroEvolution of Augmenting Topologies (NEAT) introduced by Kenneth Stanley [17]. NEAT allows for both optimisation and increase in complexity at the same time. Meaning that not only are the weights evolved but also the actual topology of the network. This is in contrast to the fixed topology of 5 neurons in the hidden layer of the CTRNN used in this project. NEAT has been shown to be successful in evolving LSTM [18] networks and as such I would very much like to explore this methodology and assess its viability in the domain of evolutionary robotics.

Bibliography

[1] Inman Harvey, Philip Husbands and Dave cliff. *Issues in Evolutionary Robotics*
users.sussex.ac.uk/p̃hilh/pubs/issuesinER.pdf

[2] Dave Cliff,Philip Husbands and Inman Harvey. *Evolving Visually Guided Robots*
users.sussex.ac.uk/p̃hilh/pubs/evolvingVguided.pdf

[3] Sharabi, Shai and Sipper, Moshe. *GP Sumo: Using Genetic Programming To Evolve Sumobots* Genetic Programming and Evolvable Machines. Vol 7, No. 3, p 211-230. 2006

[4] Rodney Brooks *A Robust Layered Control System For A Mobile Robot*
IEEE Journal of Robotics and Automaton, Vol RA-2, No. 1, 1986

[5] Bongard, J & Lipson, H. *Evolved Machined Shed Light on Robustness and Resilience*
Proceedings of the IEEE, 102(5), pp.899-914. 2014

[6] Floreano D & Keller L *Evolution of Adaptive Behaviour in Robots by Means of Darwinian Selection*
PLoS Biology,8(1), p.e1000292. 2010

[7] Alan Turing *Computing Machinery and Intelligence*
Mind , LIX(236) pp.433-460. 1950

[8] Slick2D *2D Java game library* http://slick.ninjacave.com/ 2017.

[9] Duality *A 2D Game Development Framework* http://duality.adamslair.net/ 2017.

[10] Allegro *A game programming library* http://liballeg.org/ 2017.

[11] Pymunk *Pymunk 5.2.0 Documentation* http://www.pymunk.org 2017.

[12] Chipmunk *Chipmink 2D Physics* `https://chipmunk-physics.net/` 2017.

[13] Pygame *pygame.org* `http://www.pygame.org/` 2017.

[14] Anaconda *Continuum* `https://www.continuum.io/DOWNLOADS` 2017.

[15] Miller, Brad.L and Goldberg, David E. "A Comparative Analysis of Genetic algorithms, tournament selection, and the effects of noise." *Complex Systems.* p.193-212. 1995.

[16] Rylander, S.G.B. Optimal population size and the genetic algorithm. *Population,*100(400). p.900 2002

[17] Stanley, K.O. and Miikkulainen, R "Evolving Neural Networks through Augmenting Topologies". *Evolutionary Computation 10.2,* pp99-127, 2002

[18] Rawal, A and Miikkulainen,R. Evolving deep lstm=based memory networks using an information maximization objective. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference* pp. 501-508. ACM, 2016

[19] BCS Code of conduct. *British Computing Society* `http://www.bcs.org/category/6030` 2017

[20] Harvey, Inman "The Microbial Genetic Algorithm" *European Conference on Artificial Life.* Springer Berlin Heidelberg. pp.126-133. 2009

Appendix

videos

Example Video recording of excitatory response to sensor signal:
`https://www.dropbox.com/s/64iq77v012civ3l/20170507_233418.mp4?dl=0`

An early example of evolving an agent to stay within the arena:
`https://www.dropbox.com/s/l95wdhav001h3p2/20170328_170106.mp4?dl=0`

An example of two agents fighting, where one has an excitatory sensor response and the other has an inhibitory sensor response:
`https://www.dropbox.com/s/5pfsfrr5wtxwawd/20170507_170416.mp4?dl=0`

An early example of two agents fighting, with one agent chasing the other down:
`https://www.dropbox.com/s/6pkonhtwfci5rda/20170406_171113.mp4?dl=0`

An example where two agents with inhibitory sensor responses are put against one another, resulting in a stalemate situation:
`https://www.dropbox.com/s/spm4lhgkx0bzy70/20170510_160526.mp4?dl=0`