Trinity University
Digital Commons @ Trinity

Computer Science Honors Theses | Computer Science Department

3-31-2013

Machine Learning through Evolution: Training Algorithms through Competition

Valeri Alexiev
Trinity University, valexiev@trinity.edu

Follow this and additional works at: http://digitalcommons.trinity.edu/compsci_honors

Recommended Citation
Alexiev, Valeri, "Machine Learning through Evolution: Training Algorithms through Competition" (2013). *Computer Science Honors Theses*. 33.
http://digitalcommons.trinity.edu/compsci_honors/33

This Thesis open access is brought to you for free and open access by the Computer Science Department at Digital Commons @ Trinity. It has been accepted for inclusion in Computer Science Honors Theses by an authorized administrator of Digital Commons @ Trinity. For more information, please contact jcostanz@trinity.edu.

Machine Learning through Evolution:
Training Algorithms through Competition

Valeri Alexiev

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

March 31, 2013

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Student Copyright Declaration: the author has selected the following copyright provision:

[X] This thesis is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License, which allows some noncommercial copying and distribution of the thesis, given proper attribution. To view a copy of this license, visit http://creativecommons.org/licenses/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

[ ] This thesis is protected under the provisions of U.S. Code Title 17. Any copying of this work other than "fair use" (17 USC 107) is prohibited without the copyright holders permission.

[ ] Other:

Distribution options for digital thesis:

[X] Open Access (full-text discoverable via search engines)

[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

Machine Learning through Evolution:

Training Algorithms through Competition

Valeri Alexiev

Abstract

Machine learning is an important part of most current Artificial Intelligence applications as it allows programs to continually improve their performance without outside help. An important testing ground for machine learning algorithms is learning how to play a particular game. In fact, game-playing programs are considered to have advanced the field of AI significantly because they provide an easy way to measure the performance of an AI agent. This project focuses on the games Connect Four and Robocode. The game-playing agent uses a neural network as a heuristic function in a standard Min-Max tree search algorithm. The neural network is trained using particle swarm optimization, an algorithm based on the flocking behavior of birds and fish, in a tournament-style competition against other neural networks. The only game-specific information provided is whether a particular game was won, lost or resulted in a draw. Even with so little information, the algorithm shows that it's capable of creating game-playing agents that are equal to or better than hand-designed programs. In addition, this project tries to extend this approach to games that are not turn-based perfect information games. The game of RoboCode, an educational game that pits AI tanks against each other, is used as a testing ground for the algorithm. Particle swarm optimization is used to train neural networks that perform the targeting subroutine of the tank. The score of the robot at the end of a battle is used as an indication of the performance of the neural network. Some results of this approach are presented, showing that the algorithm can successfully create aiming networks.

Acknowledgments

I would like to dedicate this thesis to my parents, who have provided me with so many opportunities. I would also like to acknowledge the Department of Computer Science, and especially my thesis advisor Dr. Mark Lewis, for their help and assistance in the creation of this thesis. Last, but not least, I want to thank my fellow Computer Science majors and other friends for their continued support and interest in my work.

Machine Learning through Evolution:
Training Algorithms through Competition

Valeri Alexiev

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

March 31, 2013

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Student Copyright Declaration: the author has selected the following copyright provision:

[X] This thesis is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License, which allows some noncommercial copying and distribution of the thesis, given proper attribution. To view a copy of this license, visit http://creativecommons.org/licenses/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

[ ] This thesis is protected under the provisions of U.S. Code Title 17. Any copying of this work other than "fair use" (17 USC 107) is prohibited without the copyright holders permission.

[ ] Other:

Distribution options for digital thesis:

[X] Open Access (full-text discoverable via search engines)

[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

Machine Learning through Evolution: Training Algorithms through Competition

Valeri Alexiev

Contents

1 Introduction **1**

1.1 Goal of this research 1

1.2 Previous Work in the Area 1

1.3 Contents 2

2 Background Information **3**

2.1 AI and Games 3

2.1.1 Connect Four 3

2.1.2 Robocode 4

2.2 Overview of the Algorithms Used 6

2.2.1 Game Trees 6

2.2.2 Min-Max Algorithm 6

2.2.3 Neural Networks 7

2.2.4 Particle Swarm Optimization 9

3 Connect Four **12**

3.1 Methods 12

3.1.1 Neural Network Architecture 12

3.1.2 Training Procedure . . . 13
3.2 Results and Discussion . . . 14

4 Robocode **21**
4.1 Methods . . . 21
4.1.1 Neural Network Architecture . . . 21
4.1.2 Training Procedure . . . 22
4.2 Stationary robots . . . 23
4.3 Moving robots . . . 24
4.4 Dodging Robots . . . 27

5 Conclusion **29**
5.1 Conclusions . . . 29
5.2 Further work . . . 29

A Connect Four Data **33**

B Robocode Data **36**

C Robocode Testing Robots **38**
C.1 Stationary Bots . . . 38
C.2 Moving Bots . . . 40
C.3 Dodging Bots . . . 41

List of Tables

A.1 Varying iterations for network: 42-11-1; LBest 5; Vmax: 0.2; Inertia: 0.9; Population: 15 33

A.2 Varying inertia for network: 42-11-1; LBest 5; Vmax = 0.2; Population: 15; Epochs: 200 33

A.3 Varying Vmax for network: 42-11-1; LBest 5; Inertia: 0.9; Population: 15; Epochs: 200 34

A.4 Varying networks: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 15; Epochs: 300 34

A.5 Varying population size for network: 42-84-1; LBest 5; Vmax = 0.1; Inertia: 0.5; Epochs: 50 34

A.6 Varying neighborhood size for network: 42-84-1; LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 45; Epochs: 50 35

B.1 Simulation results (EvoBot / Control) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30 36

B.2 Simulation results (EvoBot / Crazy) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30 36

B.3 Simulation results (EvoBot / Crazy) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30 37

B.4 Simulation results (EvoBot / VelociRobot) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30 37

List of Figures

2.1 A Connect Four game with yellow and red indicating Player 1 and 2, respectively. 4
2.2 The anatomy of a Robocode robot. 5
2.3 A screenshot of a battle between two robots. 5
2.4 Two-ply game tree for Tic-Tac-Toe with rotations of the game board removed 7
2.5 Artificial Neural Network 8
2.6 The Sigmoid Function 9
2.7 Example of a particle's velocity update 11

3.1 The result of varying the number of iterations 14
3.2 The result of varying the inertia, w 15
3.3 The result of varying the maximum speed of particles, $Vmax$ 16
3.4 The result of varying the size of the neighborhood of the LBest architecture 18
3.5 The result of varying the size of the population 18
3.6 The result of varying the size and architecture of the neural network 19
3.7 Comparison against other algorithms 20

4.1 The architecture of the aiming network 22

4.2 The result of varying the network size 23
4.3 The result of varying the number of iterations 24
4.4 The result of varying the number of iterations 25
4.5 The result of varying the network size 25
4.6 The result of varying the number of iterations 26
4.7 The result of varying the network sizes 27
4.8 The result of varying the number of iterations 28

Chapter 1

Introduction

1.1 Goal of this research

The goal of this research is to validate the versatility of neural networks trained using Particle Swarm Optimization by implementing them as players in two very different games. This project focuses on the games Connect Four and Robocode and tries to prove that game agents for both of them can be created using the same general approach.

1.2 Previous Work in the Area

Using artificial neural networks to play games is not a novel approach. Grim et al. have trained a probabilistic neural network to play Tic-Tac-Toe using training data generated by a heuristic function [6]. In his book on AI game development, Alex Champandard gives an example of a neural network that handles the aiming for a video game character [2]. Machine learning algorithms have been applied previously to Robocode as well. *Robocode JGAP* is one such project that uses genetic programming to evolve the code for a robot [9]. Training

neural networks using Particle Swarm Optimization (PSO) and an implementation of the approach have been presented in one of the first books about Particle Swarm Optimization [7]. Additionally, in his Masters thesis, Cornelis Franken uses PSO to co-evolve neural networks playing Tic-Tac-Toe and Checkers [5].

1.3 Contents

Chapter 2 covers background information about Connect Four and Robocode, as well as a description of the main algorithms used in this research. Chapter 3 covers how the algorithms are implemented for Connect Four and how varying the parameters of the Particle Swarm Optimization affects the score of the trained networks. Chapter 4 focuses on Robocode, explaining how interfacing with Robocode works and how the training procedure is set up. In addition, I present results of testing the algorithm against both stationary and moving robots. Chapter 5 summarizes the conclusions from this research and sketches out possible directions for future work.

Chapter 2

Background Information

2.1 AI and Games

One of the main goals of Artificial Intelligence is to create algorithms that can outperform humans in a variety of complex tasks. As such, games have always had a special place in AI research. This is because, in general, most games are fairly well understood problems. In addition, game scores are a very obvious indicator of the performance of the algorithm.

2.1.1 Connect Four

Connect Four is a very famous board game that is usually played on a 6x7 board as shown in Figure 2.1. Players take turns placing discs on the board. Discs are played on a particular column and fall to the lowest available position on that column. The goal of the game is to connect four of your own pieces horizontally, vertically or diagonally. Different scholars give different calculations as to the complexity of the game. Kissman and Tromp have calculated that Connect Four has 4,531,985,219,092 positions [8] [14], while Allis estimates the positions to be 70,728,639,995,483 [1]. In his Masters thesis, Allis also proves that the

first player to play wins [1].

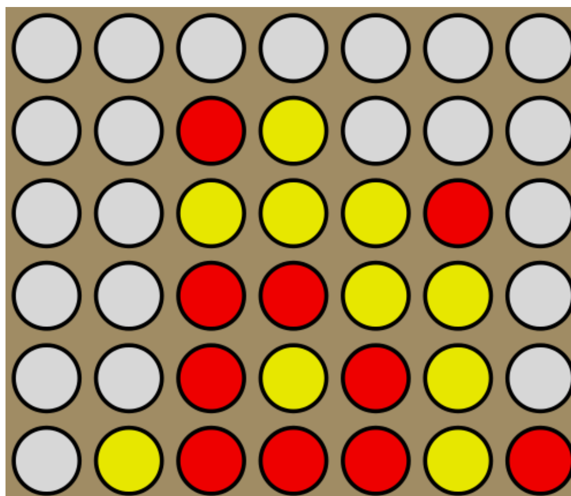


Figure 2.1: A Connect Four game with yellow and red indicating Player 1 and 2, respectively.

2.1.2 Robocode

Robocode is an open source educational game. The goal of the game is to create an AI tank that battles other bots. Because bots can be easily exchanged online, a large community has grown around Robocode. There are even different tournaments based on the file-size limitations of the robots. In order to make downloading robots from the Internet safe, Robocode runs battles in a secure sandbox, limiting the robot's access to the system. All interactions with the robots is provided by specialized classes that observe the battles and fire event handlers as necessary. Even though the secure sandbox can the turned off, the way Robocode is designed makes interfacing with the robots difficult, which causes some problems during the implementation of the training algorithm that will be discussed later.

Each robot is composed of three sections as seen in Figure 2.2: a body that moves the robot, a gun that shoots bullets, and a radar that detects enemies. The source code of

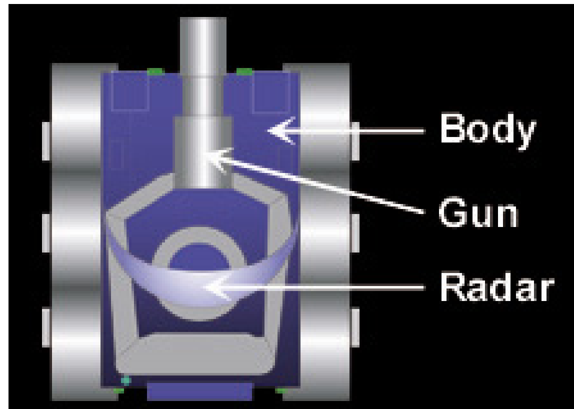


Figure 2.2: The anatomy of a Robocode robot.

each robot consists of a main loop and several event handlers that respond to certain game situations. At the start of a round, each robot is instantiated at a random position on the battlefield. Figure 2.3 shows what a Robocode battle looks like.

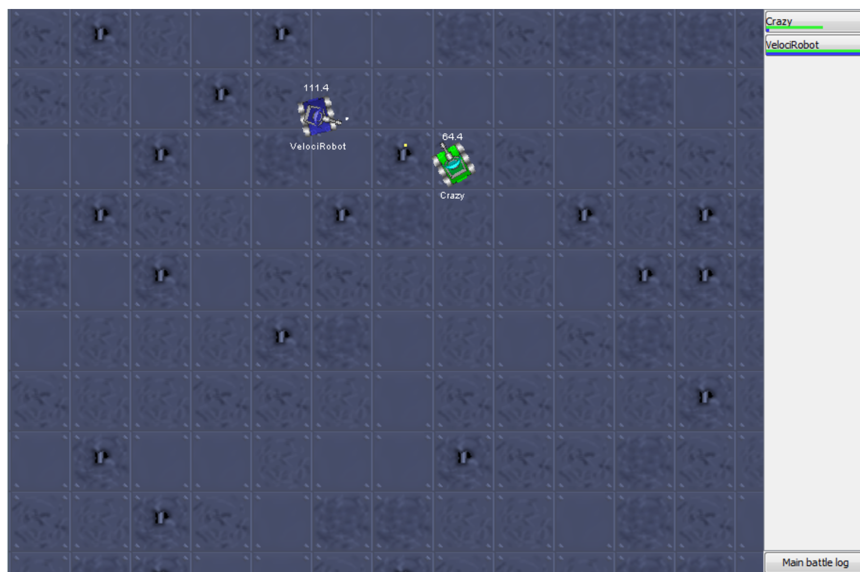


Figure 2.3: A screenshot of a battle between two robots.

Scoring in Robocode is quite detailed, but everything that contributes to the total score of a robot can be divided into 3 areas: Survival, Bullet Damage and Ram Damage [13]. For Survival, each robot that's alive scores 50 points every time an enemy dies, with the last robot standing getting an additional 10 points for each dead robot. For Bullet Damage, robots score 1 point for each point of damage they do to an enemy, with a bonus of 20% if the robot ends up killing its enemy. For Ram Damage, robots score 2 points for each point of damage they do to an enemy, with a bonus of 30% if the robot ends up killing its enemy by ramming into it.

2.2 Overview of the Algorithms Used

2.2.1 Game Trees

A *game tree* is a standard representation of possible moves in perfect-information games [10]. *Perfect-information games* are deterministic games, in which all players have perfect information about the game state. Thus, chess is a perfect-information game, while poker is not, because the players cannot see each other's cards. An example of a game tree for the game Tic-Tac-Toe can be seen in Figure 2.4. In this example only the first two moves are shown, but following the same principles the entire game tree of Tic-Tac-Toe can be expanded and all possible games can be enumerated.

2.2.2 Min-Max Algorithm

When expanding the full game tree is not possible, the *Min-Max algorithm* can be used to play the game intelligently. The Min-Max algorithm uses the concept of game trees in order to minimize the possible loss for a worst case scenario [10]. The algorithm does this by playing in a way that minimizes the loss against a perfectly playing opponent. It does

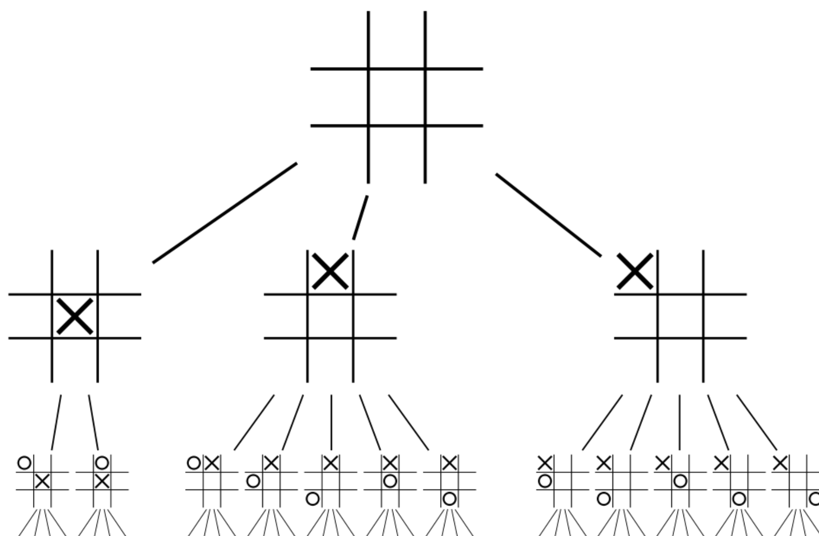


Figure 2.4: Two-ply game tree for Tic-Tac-Toe with rotations of the game board removed

this by using a heuristic function to evaluate the desirability of board states. In this project, the heuristic function used is a *neural network*.

2.2.3 Neural Networks

Neural networks are a computational model inspired by the interconnected nature of the human brain. They are made up of artificial neurons that are rough approximations of their biological equivalent. Neural networks excel at many tasks such as pattern recognition, function approximations, classification, and data processing [10]. Because of that there are a multitude of practical applications, including systems control, speech recognition, automated trading systems, email spam filters, and game-playing. Thus, neural networks are one of the main algorithms used in this project and some knowledge of them is required to understand this research.

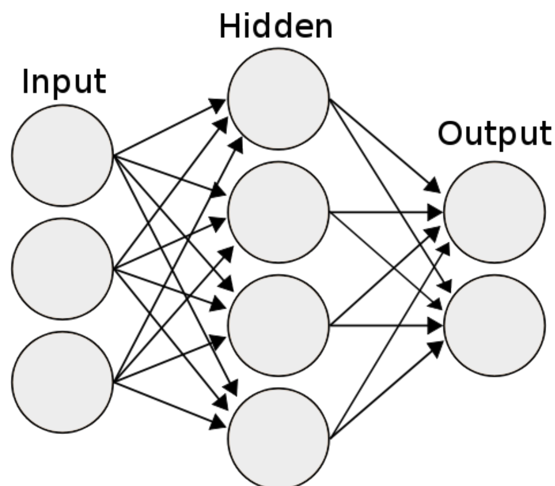


Figure 2.5: Artificial Neural Network

Figure 2.5 shows a simple 3-layer neural network that has 3 input neurons, 4 hidden neurons, and 2 output neurons. This configuration can be summarized as a "3-4-2 network". The network in this image is an example of a *feed-forward* neural network as all layers only transmit information forward. While there are many other types of networks they will not be covered, as they are not used in this project. In order to understand how a neural network works, first we need to understand how the neurons it's composed of function. An artificial neuron has several inputs and one output. Each input has a specific weight associated with it. The output of a neuron is a function of the weighed sum of the inputs. A commonly used function is the sigmoid function shown on Figure 2.6. The neural network implementation used in this research is part of the Encog Machine Learning Framework [4].

The strength of neural networks does not come directly from their architecture or from the functioning of their neurons. It comes from the fact that they can be trained to perform a variety of tasks. Training a neural network constitutes of finding the set of weights that

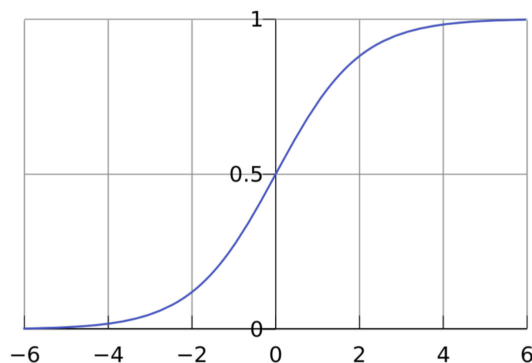


Figure 2.6: The Sigmoid Function

give the best performance. Usually, neural networks start with a random set of weights and, through iterations of a training algorithm, these weights are adjusted towards their optimal values. There is a variety of different training algorithms, but the most popular one is called *backpropagation*. This algorithm uses a technique called gradient descent, in order to adjust the weights, starting at the output layer and propagating to the input layers. The main problem with this training algorithm is that it requires a set of training cases that have already been evaluated. While this is acceptable in many applications, it is difficult to construct such a set for a game. This is why this project uses a different approach and uses an algorithm called *Particle Swarm Optimization*, which does not require a set of training cases.

2.2.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an optimization technique inspired by the movement of organisms in bird flocks and fish schools [7]. Each particle represents a proposed solution to a problem, and its movement represents a search for optimal parameters in the solution space. The movement of a particle is influenced by its previous best position and the best

position of its neighbors. Particle Swarm Optimization has been used fairly successfully on a variety of problems, including training neural networks.

The PSO achieves its success through the application of a simple formula:

$$v(t) = wv(t-1) + c_1 r_1(t)(y(t) - x(t)) + c_2 r_2(t)(z(t) - x(t)) \tag{2.1}$$

This formula is the velocity update of a particle and has three main components. The first component is the momentum component: $wv(t-1)$, where w is the inertia constant, and $v(t-1)$ is the previous value of the particle's velocity. Thus, w indicates how much the current velocity is influenced by the previous velocity. Next is the *cognitive component*: $c_1 r_1(t)(y(t) - x(t))$, where c1 is called the cognitive constant, $r_1(t)$ is a random number, $y(t)$ is the particle's previous best position, and $x(t)$ is the particle's current position. The cognitive constant c_1 scales the attraction that a particle experiences towards its previous best position, while the randomness introduced by $r_1(t)$ ensures that there is a proper balance between exploration and exploitation. The last component of the velocity update equation is the so-called *social component*: $c_2 r_2(t)(z(t) - x(t))$, where c_2 is the social constant, $r_2(t)$ is a random number, $z(t)$ is the group's previous best position, and $x(t)$ is the particle's current position. Similarly to the cognitive component, the social constant c_2 scales the result, while $r_2(t)$ introduces randomness that helps the particles explore the solution space. The values for the cognitive and social constants most often cited in literature are $c_1 = c_2 = 2$ and these values are used in the Particle Swarm Optimization class used in this project [7] [11].

The PSO implementation that I use is a slightly modified version of the algorithm that is available as part of the Encog Machine Learning Framework [4]. The only difference between the two implementations is the neighborhood architectures they use. The neighborhood

architecture of a PSO algorithm determines how the group of a particle is defined and, thus, how we get the term $z(t)$ in Equation 2.1. There are many different neighborhood architectures used in literature, but the two standard ones are *Gbest* and *Lbest*. *Gbest* uses the global best position, while *Lbest* uses a circular ordering of the particles and defines a particle's group to include a number of neighbors on both sides of the particle. While the PSO implementation in the Encog Machine Learning Framework uses *Gbest*, I decided to use *Lbest*, because it has been shown to perform better in multimodal search spaces, which was confirmed by some preliminary tests [7]

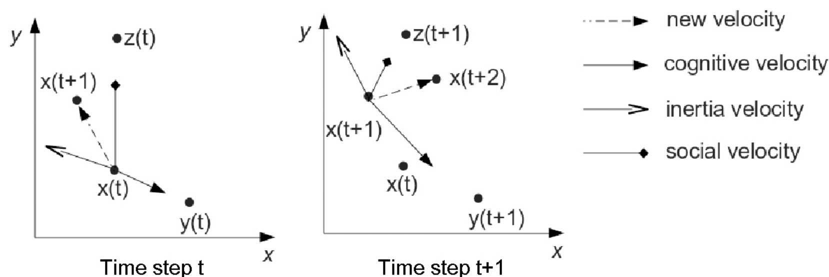


Figure 2.7: Example of a particle's velocity update

To make sense of how the velocity equation works in practice, we can look at Figure 2.7. This example is simplified since it only optimizes 2 variables (x and y), thus the search space is only 2-dimensional. When optimizing the weights of a neural network, every weight is a variable. Thus, for a simple 4-2-1 network, the search space becomes 10-dimensional. Yet, even the smallest Connect Four neural network tested is much bigger than that. Having the architecture 42-11-1, it represents a 473-dimensional optimization problem.

Chapter 3

Connect Four

This chapter focuses on the approach used to create a program that can learn to play Connect Four. Results of the implementation are presented and discussed.

3.1 Methods

3.1.1 Neural Network Architecture

The game agent uses a simple feed-forward artificial neural network as a heuristic function for a Min-Max algorithm that determines the best possible move by expanding the game tree 3 moves ahead.

The neural network used has 42 inputs and 1 output. The inputs represent a standard 6x7 Connect Four board, where a 1 represents the agent's piece, a -1 represents the enemy's piece and a 0 represents an empty position. The output of the neural network represents the desirability of the given board and is the heuristic value used in the Min-Max algorithm.

3.1.2 Training Procedure

The weights of the network are found using Particle Swarm Optimization. During each iteration of the PSO, each network that is a member of the population plays against a Hall of Fame, a collection of networks that is composed of the best performing players found so far in the simulation. The Hall of Fame starts as a copy of the first generation of particles and is continually updated throughout the simulation.

To evaluate each particle, the algorithm tests it against each Hall of Fame member twice: once as player 1, and once as player 2. A draw does not change the score, a win increases it by 1, and a loss decreases it by 2. After the evaluation, if a network has a higher score than the worst-performing Hall of Fame member, the Hall of Fame gets updated with the new network. In order to keep the scores of the Hall of Fame members up to date, they get reevaluated every 5 iterations.

At the end of the training, the top scoring network is validated against a randomized version of a human-designed heuristic in a set of 100 games. Because of the inherent imbalance of the game, the neural network has to play 50 games as player 1 and 50 games as player 2, in order to balance out the score. The original heuristic function is deterministic, which creates problems during validation, because all of the validation games as a certain player would result in the same outcome. In addition, the outcome of the games is arbitrary and does not indicate general Connect Four playing ability. In order to test how well the neural network generalizes, I created a randomized heuristic function that plays in a random manner 30% of the time, and for the rest of the time it uses the deterministic function. This ensures that the heuristic function forces the neural network in new areas of the game tree in each game while still playing (somewhat) intelligently.

3.2 Results and Discussion

In this section, the results of varying the parameters of the algorithm are presented and discussed. Additionally, the results are compared with the performance of other heuristic functions.

As can be seen in Figure 3.1, varying the number of iterations, for which the algorithm runs, causes the score of the resulting network to vary wildly with no discernible trend. This might be the result of overfitting, a situation, in which all the neural networks would get progressively better against each other, but would lose their ability to play against more general players. One way to prevent overfitting would be to include a heuristic function as a permanent member of the Hall of Fame. However, even without such measures, it seems that a maximum number of iterations as low as 300 provide a relatively high score and, since running time is always a consideration, such values should be preferred over higher number of iterations.

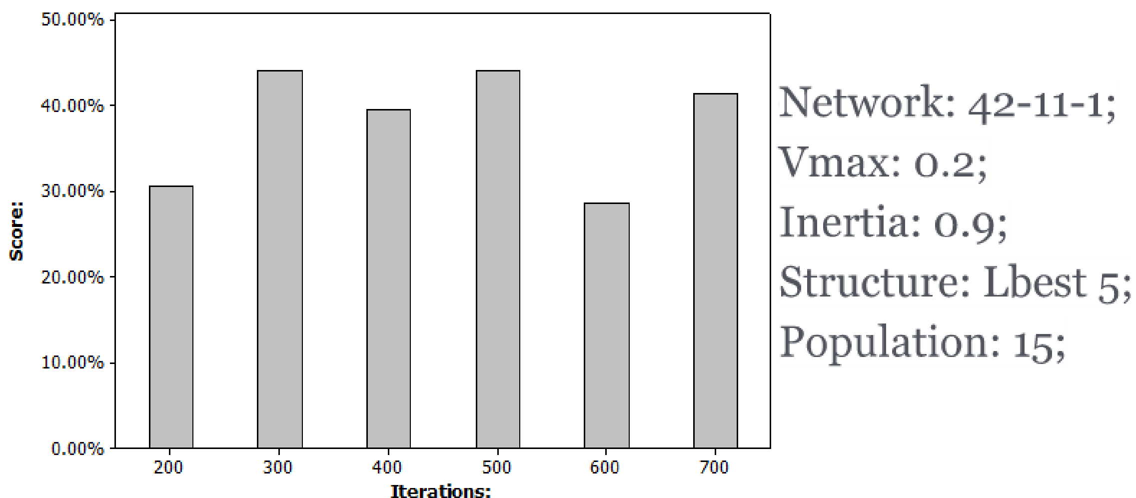


Figure 3.1: The result of varying the number of iterations

Another important parameter is the inertia constant, signified by w in Equation 2.1. The inertia constant determines to what extent does the previous velocity of a particle influence the current velocity. Thus, a high value of the inertia constant would mean that the speed of the particle would not be influenced as much by the newly available information in a iteration. Conversely, a low inertia value would mean that the particle has very little ”memory” of its previous path and, instead, its velocity is mostly determined by the current iteration. In Figure 3.2, we can see that, overall, lower values of the inertia constant are better. It would seem that higher values of w prevent the particles from utilizing the information about the current personal best and the current group best positions. This would mean that the particles would have a harder time converging on a favorable region, which would lead to lower scores overall. From the figure, we can see that $w = 0.5$ returns the best results. This is probably because it strikes the best balance between exploration and exploitation.

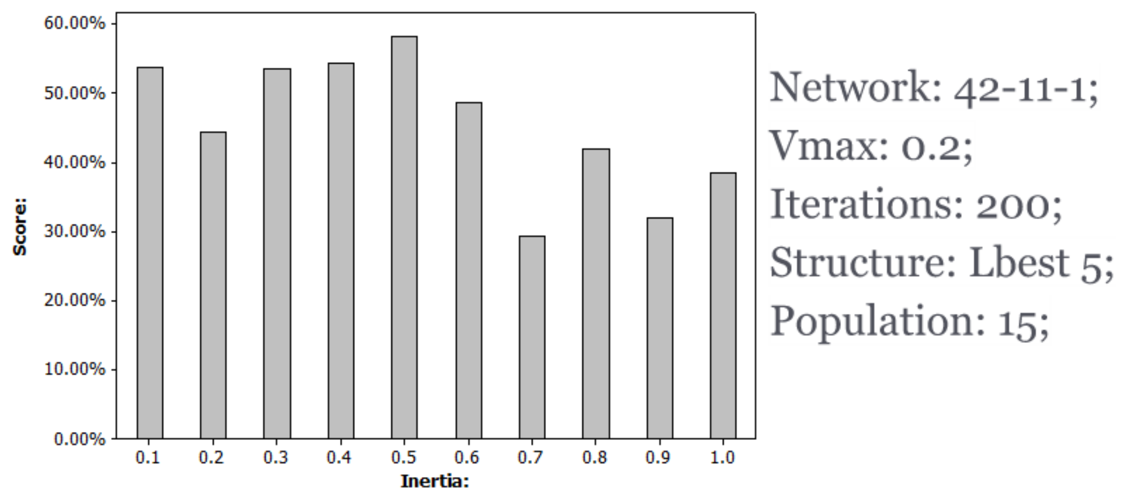


Figure 3.2: The result of varying the inertia, w

The speed of the particles in a PSO tends to explode and cause chaotic behavior when it's not controlled. This is done by specifying a $Vmax$, a maximum speed that the particles can achieve. Figure 3.3 shows how varying the $Vmax$ affects the result of the algorithm. We can see that the higher maximum speeds result in worse performance. This is likely because the particles achieve high acceleration and, without a lower $Vmax$ to limit the speed, they end up "over-shooting" the favorable regions in the solution space. Thus, the particles end up oscillating around good solutions, but never actually converge on them. Similarly, we can see that limiting the speed too much results in worse scores as well, though the performance hit is not as severe as in the case of higher $Vmax$. This is caused by the fact that the particles take a lot longer to move through the solution space and converge on favorable solutions. The results in Figure 3.3 suggest that $Vmax = 0.10$ is the optimal value for this parameter.

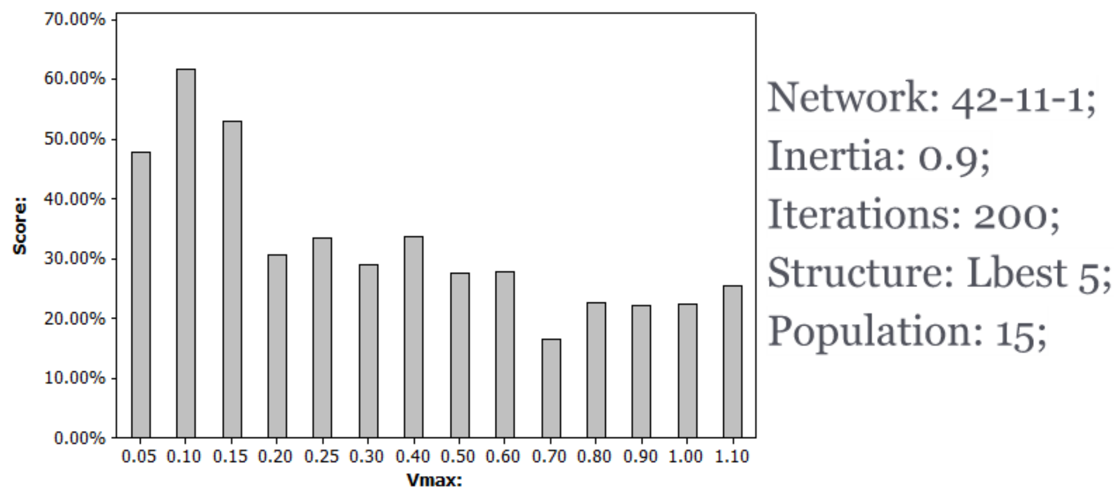


Figure 3.3: The result of varying the maximum speed of particles, $Vmax$

As explained in Section 2.2.4, the algorithm uses a $LBest$ structure to define the group,

from which particles get the information about the group's best position used in the social component of Equation 2.1. Thus, finding the optimal value for the neighborhood size is crucial, as it controls how information about solutions propagates through the population. In Figure 3.4 we can see the results of varying the size of the neighborhood, with the best results achieved with $LBest5$. It might seem counterintuitive that such a low neighborhood size results in a better performing neural networks. After all, one of the strengths of Particle Swarm Optimization is that the particles get information about the highest scoring particles in their neighborhood. While it would seem that limiting the neighborhood size would mean less information about the global best solutions and would, consequently, result in lower scores, this is not the case. Smaller neighborhood sizes prevent the algorithm from prematurely converging on local maxima. That being said, even though the best results are achieved when we use $LBest5$, the graph is clearly bimodal, with a second peak around 23 neighbors, or around 50% of the population size. There is no clear reason why this is the case and further research must be done to find what the connection is between population size and neighborhood size.

The population size of the Particle Swarm Optimization is an important parameter because it determines the number of particles exploring the solution space at any given time. Thus, theoretically, we would like to be able to use a very large population size. However, this slows down the algorithm prohibitively, so we want to find a reasonable population size that results in high scoring neural networks. In Figure 3.5, we can see the results of varying the population size of the algorithm when we use $LBest5$. We can see that, in general, bigger population sizes result in better scoring players, though there is a peak at 45 particles. It is not clear whether this is because 45 particles is somehow an optimal population size or because we have optimized the neighborhood size when the population is 45. The second case could mean that $LBest5$ is uniquely suited for population size of 45,

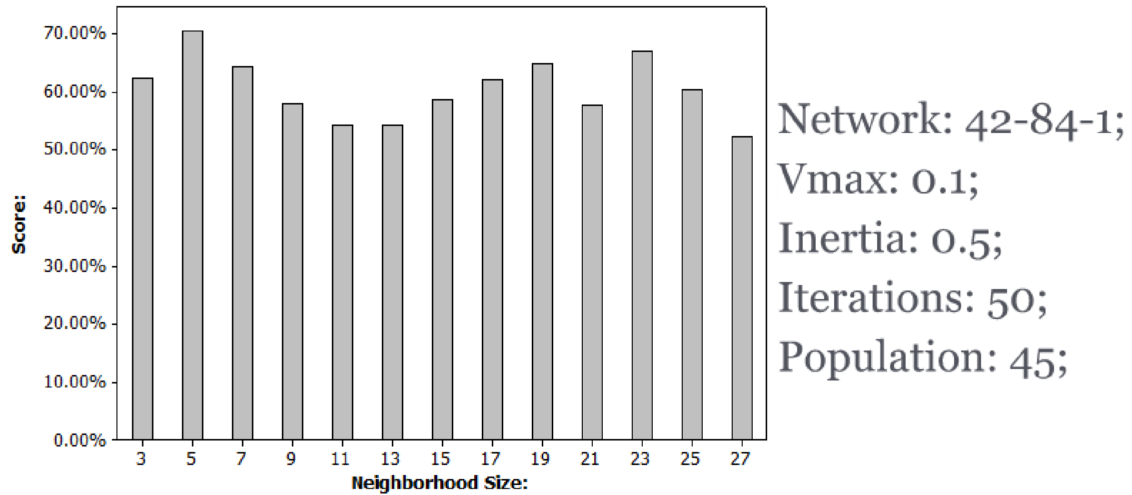


Figure 3.4: The result of varying the size of the neighborhood of the LBest architecture

but applying it to other populations results in suboptimal scores. As noted before, more research needs to be done in this area.

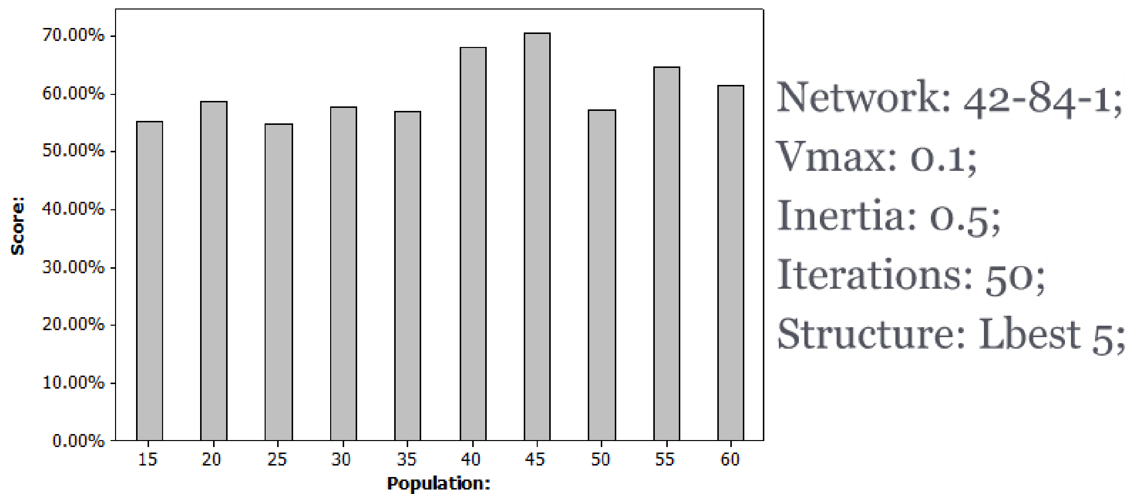


Figure 3.5: The result of varying the size of the population

The neural network size and architecture used is very important for the performance

of the algorithm. A higher number of neurons provide a network with more flexibility and power, and, thus, we would assume that networks with bigger hidden layers would score higher. However, we can see in Figure 3.6 that this is not the case. While the highest scoring network (42-84-1) does have a large hidden layer, we can see that the second highest scoring network (42-21-1) scores better than networks much larger than it. This could result from the fact that larger networks have a lot more weights than smaller ones. For example, a network of size 42-60-1 has around 2500 weights, compared to 42-21-1 network which has only 900 weights. This would mean that it would be much harder for the PSO to find the optimal weights, which would lead to lower scores.

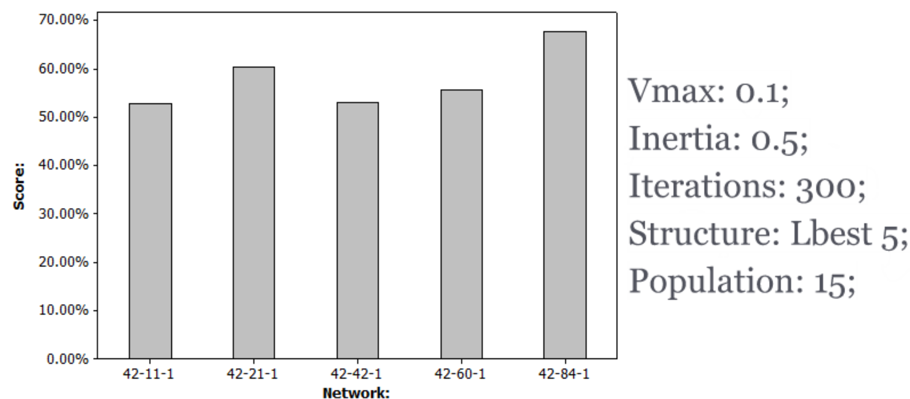
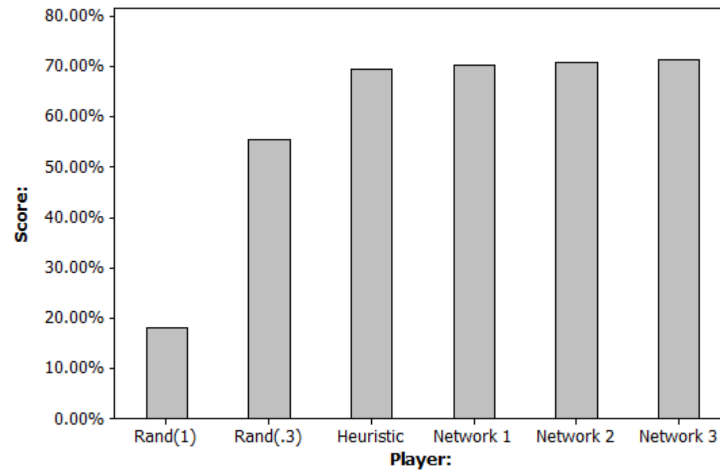


Figure 3.6: The result of varying the size and architecture of the neural network

After investigating what the optimal parameters of the algorithm are, we can compare the resulting networks against other algorithms. Figure 3.7 shows the three top-scoring networks in comparison with other heuristics. *Heuristic* refers to the deterministic heuristic described in Section 3.1.2. *Rand(.3)* refers to an algorithm that plays random a third of the time and the rest uses the heuristic, while *Rand(1)* refers to a player that always plays random. We can see that the top scoring neural networks perform at the same level or even

better than the human-designed heuristic function.



Network 1: 42-84-1; LBest 5; 45 pop; Vmax = 0.1; Inertia = 0.5; Iterations = 50;
Network 2: 42-125-1; LBest 5; 60 pop; Vmax = 0.1; Inertia = 0.5; Iterations = 50;
Network 3: 42-125-1; LBest 5; 45 pop; Vmax = 0.1; Inertia = 0.5; Iterations = 50;

Figure 3.7: Comparison against other algorithms

Chapter 4

Robocode

4.1 Methods

4.1.1 Neural Network Architecture

As described in Section 2.1.2, the code for each Robocode robot is composed of a main loop and several event handlers. This chapter describes the approach to creating and training a neural network that can handle the targeting system that gets called when the radar detects an enemy robot.

Figure 4.1 shows the architecture of the targeting neural net. The inputs include the normalized values for the enemy's current and past bearing, its current and past distance, its current and past energy, and its velocity. These values are part of the information that the Robocode engine provides the robot when an enemy is detected. The network has a single output that is used to turn the robot, in an effort to track the enemy.

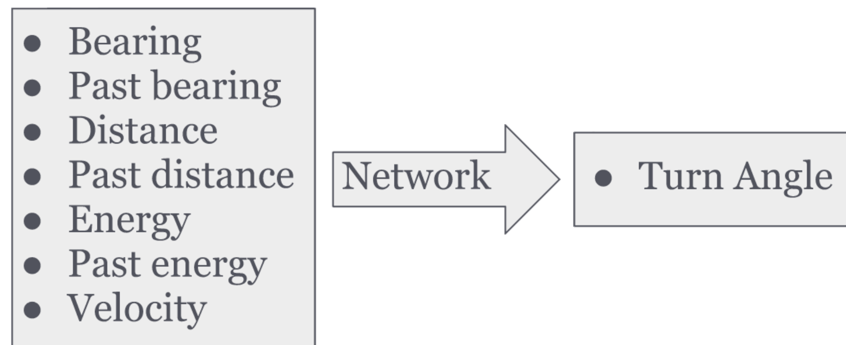


Figure 4.1: The architecture of the aiming network

4.1.2 Training Procedure

During each iteration of the algorithm, every member of the population plays 100 rounds against a testing robot. The testing robot has the exact same code as the robots that are trained with the exception of its tracking system. Upon initialization, the robot that we are training, which we'll call EvoBot, reads from the hard drive the neural network that is being tested and then uses it to track enemies. The score of each PSO particle is equal to the robot's score given by the Robocode engine. Robocode's scoring algorithm is described in detail in Section 2.1.2.

Unlike the Connect Four implementation, in this case the algorithm is not parallelized because of how Robocode instantiates the robots during battle. As mentioned in Section 2.1.2, the design of Robocode severely limits any communication with the robot. Because Robocode loads the robot class by itself and does not provide any means of calling the robot's methods, there is no way to tell EvoBot which neural network should be loaded. This means that in the current implementation of the algorithm, the filename of the neural net is hardcoded in the robot and only one particle is being evaluated at a single time by the PSO.

4.2 Stationary robots

The first robot that the algorithm is tested against is called ControlBot. ControlBot is very simple: it just rotates in one spot until a robot is detected. Since EvoBot shares most of its code with ControlBot, that means that both robots are stationary. This greatly simplifies the problem as it gets rid of any inaccuracies due to motion and makes it impossible for robots to dodge incoming bullets.

Figure 4.2 summarizes some results on how varying the network size affects the score of EvoBot. Interestingly enough, using a small network seems beneficial in this scenario. The larger the network in this example, the lower its score. In addition, the smallest network (7-14-1) is not only the highest scoring one, but it also beats the testing robot by a considerable margin. This result can be explained by the fact that the larger networks need more iterations to be properly trained since the PSO has to optimize more weights to achieve the same result.

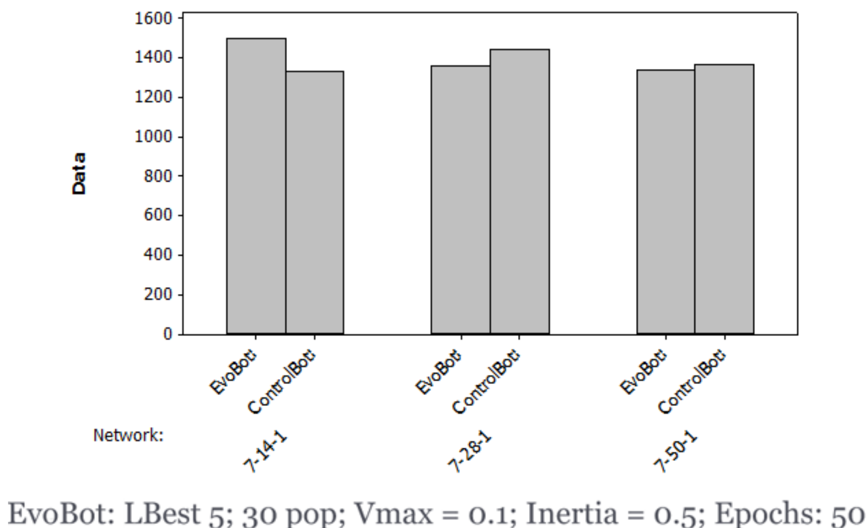


Figure 4.2: The result of varying the network size

Indeed, tests show that increasing the number of iterations seems to increase the score of EvoBot as well. Figure 4.3 shows a clear positive trend and no sign of overfitting.

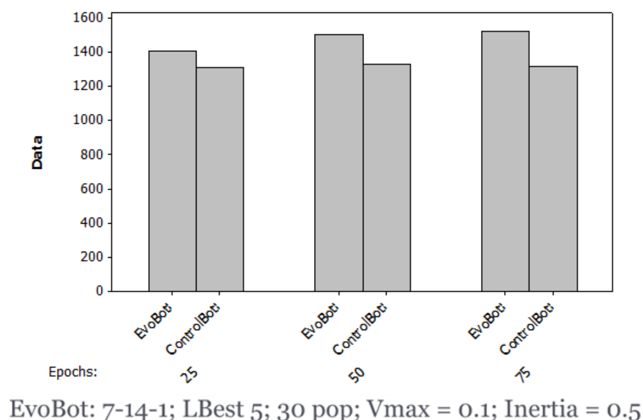


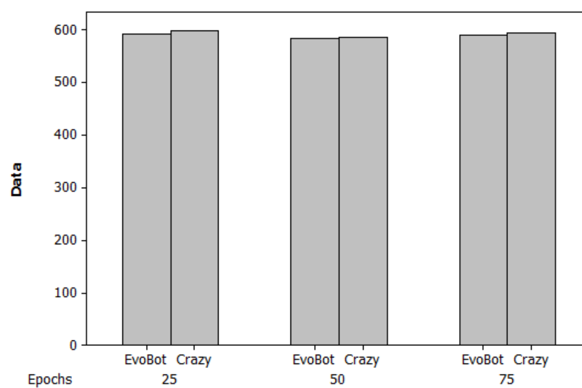
Figure 4.3: The result of varying the number of iterations

4.3 Moving robots

After successfully training robots against the stationary ControlBot, the algorithm is tested against one of Robocode sample robots, called "Crazy". Crazy moves in random patterns across the battlefield. This introduces movement to the problem. Because the bots can now move forward and backward, we introduce an additional Boolean input that shows which direction EvoBot is moving in. This additional input would hopefully provide enough information for EvoBot to be able to properly track its enemies regardless of the direction it's moving in. Unfortunately, this approach wasn't too successful.

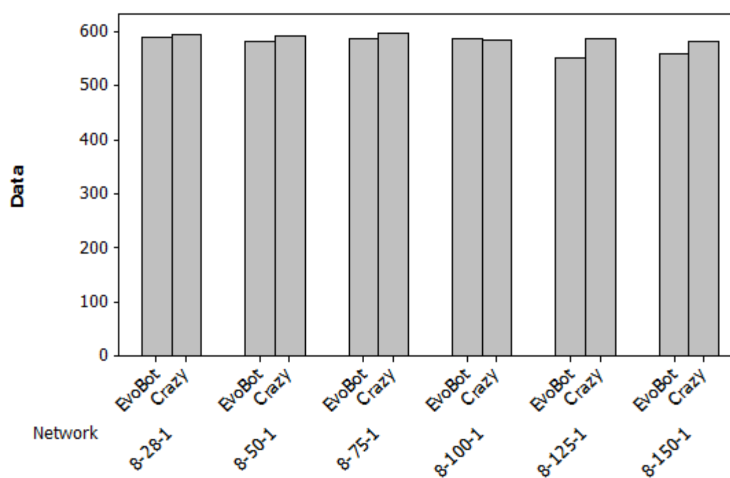
In Figure 4.4 we can see the results of varying the number of iterations on the performance of EvoBot. The trained robot always performed slightly worse than the Crazy robot.

In addition, increasing the number of iterations did not affect the score at all.



EvoBot: 8-28-1; LBest 5; 30 pop; Vmax = 0.1; Inertia = 0.5;

Figure 4.4: The result of varying the number of iterations



EvoBot: LBest 5; 30 pop; Vmax = 0.1; Inertia = 0.5; Epochs: 75;

Figure 4.5: The result of varying the network size

Varying the network size did not result in any significant score improvement. Figure 4.5

shows that none of the tested network sizes performed well. Only the 8-100-1 network performs better than Crazy, but even then it's not a significant difference.

In an effort to increase the score of EvoBot, we revert to the original input/output scheme outlined in Figure 4.1. Instead of using a Boolean input to indicate whether EvoBot is moving forward or back, we reverse the sign of the network output when the robot is going in reverse. This approach achieves considerable success.

Varying the number of iterations produces the results shown in Figure 4.6. As can be seen, all tested networks performed better than Crazy and, while the trend isn't as clear as in Figure 4.3, increasing the number of iterations seems to increase the score of EvoBot.

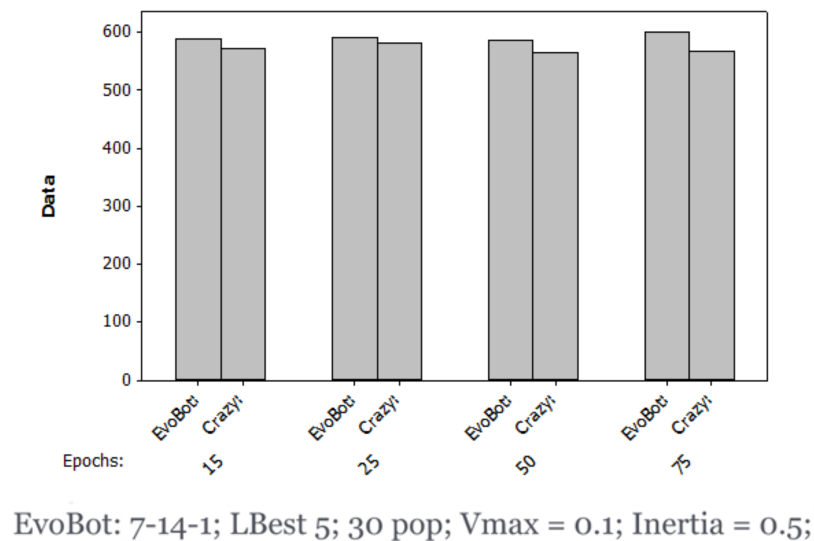


Figure 4.6: The result of varying the number of iterations

Unlike the case of the stationary bots, increasing the number of hidden neurons does improve the score of EvoBot, albeit slightly. The weak positive trend and the fact that all tested network sizes outperformed Crazy can be seen in Figure 4.7.

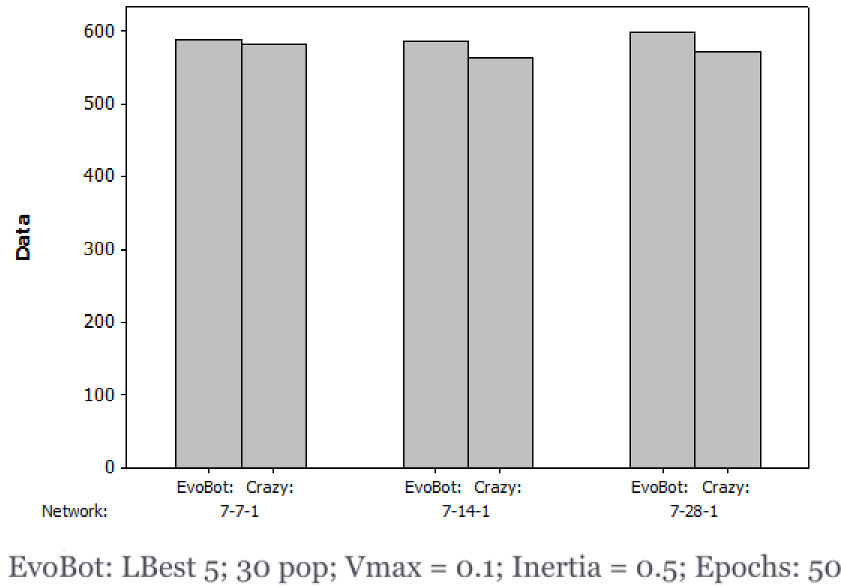


Figure 4.7: The result of varying the network sizes

4.4 Dodging Robots

After the successful performance of the algorithm against a moving robot, I chose the Robocode sample robot *VelociRobot* as the next testing robot. VelociRobot not only moves around the battlefield, but it also tries to dodge attacks by turning when hit by an enemy bullet. As can be seen in Figure 4.8, the performance of EvoBot against this robot is excellent. In fact, EvoBot performs better against VelociRobot, than it does against Crazy. This is likely due to the fact that VelociRobot always dodges in the same way, which is a behavior that the neural network would learn how to respond to quickly. Crazy, on the other hand, always moves in a random pattern, which makes it unpredictable.

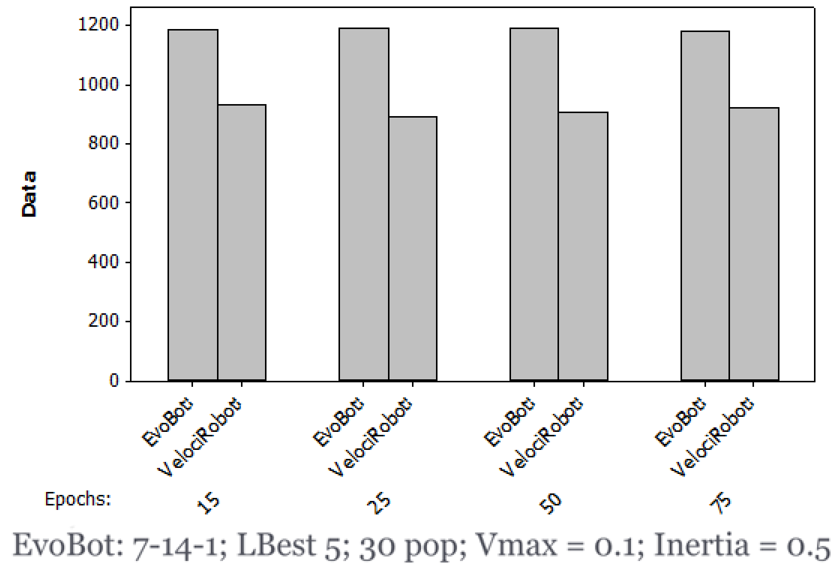


Figure 4.8: The result of varying the number of iterations

Chapter 5

Conclusion

5.1 Conclusions

The results of this research show that it is possible to train a neural network using PSO to compete successfully against traditional algorithms in both turn-based strategic games and real-time reactionary ones. This, of course, means that this approach can be extended beyond games and used on a wide variety of other problems that are not as well understood or that are difficult to solve algorithmically. Most of the results presented also show that the parameters of the algorithm act in a straightforward way that makes intuitive sense. This is important as it makes the search for optimal parameters for different problems easier.

5.2 Further work

There are several areas of this research that can be expanded upon. In connection to exploring how the algorithm behaves when run with different parameters, it would be very enlightening to see what the relation is between the size of the neighborhood in the *LBest*

architecture and the population size. If a general rule is found that connects those two parameters that would simplify the search of optimal parameters in future implementations of the algorithm.

Another area of further work would be to test how far the algorithm can be pushed by training it against increasingly sophisticated and successful robots from the Robocode rankings. One potential problem with this line of research would be that some of the robots use peculiar approaches to winning, such as ramming into their enemies, which would not fit with how I set up the robot using the neural network in this project. Even then, focusing only on robots that fit in with the setup of this research would provide enough information about the full capabilities of this approach.

Testing the algorithm against more sophisticated robots would probably require an increase of network size or the number of iterations. Because of the parallelization issues discussed in Section 4.1.2 however, that would be prohibitively time-consuming. In order to get around Robocode's interfacing issues, I designed a client-server system to take care of assigning neural networks to robots, keeping track of their score and reporting them to the PSO. However, while coding the server I discovered a bug in one of the Robocode functions crucial for this system and so I was unable to implement the server. The bug was reported and has been fixed, but because of time constraints I was unable to finish the implementation of the server [12]. When this happens, however, it'll be possible to also implement a Hall-of-Fame approach, similar to what I have done with Connect Four.

Bibliography

[1] Victor Allis. A knowledge-based approach of connect-four. Masters Thesis, 1988.

[2] Alex Champandard. *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. New Riders, 2004.

[3] Connect Four example. http://en.wikipedia.org/wiki/File:Connect_four_game.svg.

[4] Encog machine learning framework. http://www.heatonresearch.com/encog.

[5] Cornelis Franken. PSO-based coevolutionary game learning. Masters Thesis, 2004.

[6] Jiri Grim, Petr Somol, and Pavel Pudil. Probabilistic neural network playing and learning tic-tac-toe. *Pattern Recognition Letters*, 2005.

[7] James Kennedy, Russell Eberhart, and Yuhui Shi. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[8] Peter Kissman. Symbolic search in planning and general game playing. Abstract for ICAPS 2010 Doctoral Consortium.

[9] Klaus Meffert. JGAP robocode. http://jgap.sourceforge.net/doc/robocode/robocode.html.

[10] Nils Nilsson. *Artificial Intelligence : a New Synthesis*. Morgan Kaufmann Publishers, 1998.

[11] PSO implementation in the Encog Machine Learning Library. https://github.com/encog/encog-java-core/blob/master/src/main/java/org/encog/neural/networks/training/pso/NeuralPSO.java.

[12] Robocode bug report. http://sourceforge.net/p/robocode/bugs/351/.

[13] Robocode/scoring. http://robowiki.net/wiki/Robocode/Scoring.

[14] John Tromp. John's connect four playground. http://homepages.cwi.nl/∼tromp/c4/c4.html.

Appendix A

Connect Four Data

Epochs:	200	300	400	500	600	700
Score:	30.6%	44.2%	39.6%	44.1%	28.6%	41.5%

Table A.1: Varying iterations for network: 42-11-1; LBest 5; Vmax: 0.2; Inertia: 0.9; Population: 15

Inertia:	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Score:	53.7%	44.4%	53.6%	54.3%	58.2%	48.7%	29.4%	42%	32%	38.5%

Table A.2: Varying inertia for network: 42-11-1; LBest 5; Vmax = 0.2; Population: 15; Epochs: 200

Inertia:	Score:
0.05	47.8%
0.1	61.7%
0.15	53%
0.2	30.6%
0.25	33.4%
0.3	29%
0.4	33.8%
0.5	27.7%
0.6	27.8%
0.7	16.5%
0.8	22.7%
0.9	22.2%
1	22.3%
1.1	25.4%

Table A.3: Varying Vmax for network: 42-11-1; LBest 5; Inertia: 0.9; Population: 15; Epochs: 200

Network:	42-11-1	42-21-1	42-42-1	42-60-1	42-84-1
Score:	52.7%	60.4%	53.1%	55.6%	67.7%

Table A.4: Varying networks: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 15; Epochs: 300

Population:	15	20	25	30	35	40	45	50	55	60
Score:	55.2%	58.7%	54.8%	57.8%	56.9%	68.1%	70.5%	57.1%	64.7%	61.4%

Table A.5: Varying population size for network: 42-84-1; LBest 5; Vmax = 0.1; Inertia: 0.5; Epochs: 50

Neighborhood Size:	Score:
3	62.5%
5	70.5%
7	64.3%
9	57.9%
11	54.2%
13	54.2%
15	58.8%
17	62.2%
19	64.9%
21	57.7%
23	67.2%
25	60.4%
27	52.2%

Table A.6: Varying neighborhood size for network: 42-84-1; LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 45; Epochs: 50

Appendix B

Robocode Data

Epochs \ Network	7-14-1	7-28-1	7-50-1
25	1408.2 / 1309.72		
50	1498.87 / 1328.96	1357.07 / 1439.25	1336.8 / 1366.13
75	1520.5 / 1316.14	1648.24 / 1318.6	1574.48 / 1353.76
100	1657.08 / 1262.78		
125	1729.49 / 1274.49		

Table B.1: Simulation results (EvoBot / Control) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30

Network \ Epochs	25	50	75	100
8-28-1	592.26 / 598.97	584.57 / 586.67	590.24 / 595.33	
8-50-1	576.8 / 603.37	587.08 / 586.72	582.53 / 593.3	
8-75-1		581.31 / 598.21	588.41 / 597.17	585.46 / 599.98
8-100-1		577.56 / 598.37	587.14 / 585.78	583.93 / 585.31
8-125-1		545.23 / 587.87	551.79 / 587.02	577.18 / 593.29
8-150-1		560.19 / 588.67	560.36 / 582.19	585.13 / 591.88

Table B.2: Simulation results (EvoBot / Crazy) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30

Epochs \ Network	7-7-1	7-14-1	7-28-1
15	578.96 / 593.41	588.42 / 573.36	589.94 / 574.19
25	581.4 / 587.97	591.69 / 581.74	595.61 / 561.71
50	588.09 / 583.38	586.02 / 565.11	598.42 / 572.15
75	595.72 / 575.09	600.5 / 567.59	591.13 / 578.81

Table B.3: Simulation results (EvoBot / Crazy) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30

Epochs \ Network	7-7-1	7-14-1
15	1189.28 / 910.22	1185.17 / 931.64
25	1186.26 / 890.59	1189.59 / 888.51
50	1181.89 / 903.27	1191.24 / 905.13
75	1187.43 / 904.5	1180.15 / 923.1

Table B.4: Simulation results (EvoBot / VelociRobot) with settings: LBest 5; Vmax: 0.1; Inertia: 0.5; Population: 30

Appendix C

Robocode Testing Robots

C.1 Stationary Bots

Code for the main loop and the event handler for detecting robots of the stationary `ControlBot`:

```
public void run(){

        while(true){

                setTurnRight(30);
                waitFor(new TurnCompleteCondition(this));

        }

}

public void onScannedRobot(ScannedRobotEvent e){

        fire(1);

        scan();
}
```

The same functions in the robot that uses the neural network are shown below. Note that the main loop is the same as the code above, while the `onScannedRobot()` function feeds information to the neural network and performs a turn according to its output.

```
BasicNetwork n;
double[] input = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
double[] output = new double[1];

public void run(){

        while(true){

                setTurnRight(30);
                waitFor(new TurnCompleteCondition(this));

        }

}

public void onScannedRobot(ScannedRobotEvent e){

        fire(1);

        //save previous values
        //previous bearing:
        input[1] = input[0];
        //previous distance:
        input[3] = input[2];
        //previous energy:
        input[5] = input[4];

        //get bearing, distance, energy, velocity
        input[0] = e.getBearingRadians();
        input[2] = e.getDistance()/100;
        input[4] = e.getEnergy()/100;
        input[6] = e.getVelocity();

        n.compute(input, output);

        turnRight((output[0]-0.5)*10);

        scan();
}
```

C.2 Moving Bots

Code excerpt from the Robocode sample robot `Crazy`. The functions show how the robot moves around the battlefield and how the boolean `movingForward` is used.

```
/*******************************************************************************
 * Copyright (c) 2001-2012 Mathew A. Nelson and Robocode contributors
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://robocode.sourceforge.net/license/epl-v10.html
 *
 * Contributors:
 *     Mathew A. Nelson
 *     - Initial implementation
 *     Flemming N. Larsen
 *     - Maintainance
 *******************************************************************************/
boolean movingForward;

public void run() {
        while (true) {

                setAhead(40000);
                movingForward = true;

                setTurnRight(90);

                waitFor(new TurnCompleteCondition(this));

                setTurnLeft(180);

                waitFor(new TurnCompleteCondition(this));

                setTurnRight(180);

                waitFor(new TurnCompleteCondition(this));

        }
}
```

```
public void onHitWall(HitWallEvent e) {
        // Bounce off!
        reverseDirection();
}

public void reverseDirection() {
        if (movingForward) {
                setBack(40000);
                movingForward = false;
        } else {
                setAhead(40000);
                movingForward = true;
        }
}

public void onScannedRobot(ScannedRobotEvent e) {
        fire(1);
}

public void onHitRobot(HitRobotEvent e) {
        // If we're moving the other robot, reverse!
        if (e.isMyFault()) {
                reverseDirection();
        }
}
```

`EvoBot` shares all of its code with `Crazy` bot with the exception of the `onScannedRobot()` event handler. The code for that function is almost the same as the code shown in Section C.1, the sole difference being how the turning is handled:

```
        if(movingForward)
                turnRight((output[0]-0.5)*10);
        else
                turnRight(-1*(output[0]-0.5)*10);
```

C.3 Dodging Bots

Below is the code for `VelociRobot` that shows how it moves around the battlefield. Note the event handler `onHitBullet()` that implements the dodging functionality for the robot.

EvoBot shares most of its code with VelociRobot with the exception of the onScannedRobot() which is the same as the one in the previous section.

```
/*******************************************************************************
 * Copyright (c) 2001-2012 Mathew A. Nelson and Robocode contributors
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://robocode.sourceforge.net/license/epl-v10.html
 *
 * Contributors:
 *     Joshua Galecki
 *     - Initial implementation
 *******************************************************************************/
int turnCounter;

public void run() {

        turnCounter = 0;
        setGunRotationRate(15);

        while (true) {
                if (turnCounter % 64 == 0) {
                        // Straighten out, if we were hit by a bullet and are turning
                        setTurnRate(0);
                        // Go forward with a velocity of 4
                        setVelocityRate(4);
                }
                if (turnCounter % 64 == 32) {
                        // Go backwards, faster
                        setVelocityRate(-6);
                }
                turnCounter++;
                execute();
        }
}

public void onHitByBullet(HitByBulletEvent e) {
        // Turn to confuse the other robot
        setTurnRate(5);
}
```

```
public void onHitWall(HitWallEvent e) {
        // Move away from the wall
        setVelocityRate(-1 * getVelocityRate());
}
```