TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Intelligentse süsteemi optimaalse käitumise arendamine ja õpetamine golfipallide kogumiseks

Magistritöö

Üliõpilane: Maksim Pristsepov

Üliõpilaskood: 121776IAPM

Juhendaja: lektor Raul Liivrand

Tallinn
2014

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

…………………………………….	………………………………
(kuupäev)	*(allkiri)*

Annotatsioon

Selle töö eesmärk on luua intelligentse süsteemi käitumist golfpallide effektiivse kogumiseks avatud piirkonnas. Peamine tähelepanu on pööratud närvivõrgude kasutamisele käitumise jaoks ja nende õpetamisele geneetilise algoritmi abil. Pakutud lahendused käitumiseks on alalüüsitud ja võrreldatud emuleeritud keskkonnas. Töö sisu mõistmiseks on hea kui lugeja on tuttav närvivõrgudega.

The goal of this work is to create an intelligent system’s behavior for the effective collection of golf balls on open area. The main attention is paid to the neural networks for behavior’s implementation and teaching them with the genetic algorithm. The proposed solutions are analyzed and compared at emulated environment. The work is designed for readership familiar with the neural networks.

Contents

1. Abbreviations and definitions

Environment - implementation of the necessary conditions to compare different types of behavior written in Java. It is a rectangular box with randomly scattered balls, in which robots move and collect balls.

Robot - element in the simulated environment, acting as a system for finding and collecting golf balls. It may use different types of behavior in order to search balls and move within the boundaries of the environment. Also it has a radar before him with a limited vision range and vision angle.

Leader - robot leader with the best fitness of its population, collecting the highest number of balls.

Forward neural network - one of the possible behaviors of the robot for searching balls. It is a neural network with direct signal propagation. A robot with the type of network can only respond to changing environmental conditions, but has no memory.

Recurrent neural network - one of the possible behaviors of the robot for searching balls. It is a neural network with back-propagation signals. A robot with the type of the network has a memory, in other words, the network is able to remember the environment state from the previous step and apply this knowledge to the next step in its favor.

Tree of rules - one of the possible behaviors of the robot for searching balls. In the case of this kind of logic the robot reacts to changing environmental conditions in accordance with the rules described in the form of a tree.

Neuron - the main part of the neural network was established by analogy with biological neurons in the human brain, where artificial neuron operate the role of dendritic inputs and axon is an artificial neuron output. Each neuron can have only one output. Neuron performs the function of the inputs adder and is activated according to the activation function.

Neuron activation function - the function or threshold, exceeding which neuron goes to the active state and sends its signal to other neurons. Output’s power depends on activation function.

Neuron layer - is a set of neurons in a part of the neural network. Usually neuron layer keeps information about a particular feature or characteristic.

Genetic algorithm – is heuristic algorithm for finding the optimal solution by random selection, combination and variations change the desired parameters by methods similar to natural selection in nature.

First population - first set of robots with a randomly generated neuron networks. Afterwards the neuron networks will be crossovered and mutated, which subsequently leads to the identification of the dominant generation features.

Chromosome - the structure in the genetic algorithm, containing the basic parameters of the object (neural network) as a vector, which determine the behavior of the object. Chromosome is carried over the basic operations of the genetic algorithm as crossover and mutation.

Crossover – the process that exchanges chromosomes' genes in the genetic algorithm between two objects in order to create a third individual, that inherits the characteristics of their parents.

Mutate - the process of random changes in some parts of chromosomes in the genetic algorithm. Supports greater diversity of characteristics in the population.

Selection - choice of robot-leaders with the maximum amount of collected balls. The robots can participate in the next generation unchanged.

2. Introduction

All people have their hobbies, for example sports, which includes golf also. Golf field is a grassed area with different height in different areas. There are more than 32,000 golf fields in the world. This one field can reach in size from one to several tens of hectares. You can imagine how many balls miss the hole, but remains lie somewhere in the grass. Searching the missed is a mostly manual labor: divers are looking for balls in the ponds, golf workers are looking balls in shrubs, also on large expanses of fields. At the present high-tech time such a job should be automated.

At this time high resolution glasses were already invented, with which you can view the entire field at a sufficiently great distance. Another device is the program for phones with a good camera that can recognize the image of the ball on the field area of 10 kilometers in less than half a minute. But these methods require human presence, that is a negative factor. It is necessary to develop a fully autonomous system without human intervention. Maintenance costs of such a system to find balls in the open field will be much lower, faster and more efficiently than the manual labor. This work's aim is to develop a similar system that could effectively search balls on the golf field, consider possible solutions, analyze and compare it.

3. Statement of the problem

This work will examine the most effective robot's behaviors for finding golf balls in the open area of the field, using the following solutions, which include a simple description logic, the use of forward neural network, recurrent neural network. Neural networks will be trained using genetic algorithm. Environment will also be created to emulate the proposed solutions in Java in order to compare behaviors' effectiveness between themselves. There will also be a number of tests for these solutions and then will be selected the best behavior of the system that solves the problem based on test results.

This work does not consider the creation of mechanical and electronic components of the proposed system, the recognition of balls in the grass and obstacles.

4. Possible solutions

The main aim of this work is to collect golf balls that are randomly scattered on the field. It was suggested three possible solutions to this problem, two of which belong to a class of neural networks. Later we will choose the best one.

- **Tree of rules -** set of rules in the form of a decision tree. If you have information about the ball in the radar, the robot starts to go right on the ball, without turning. After lifting the ball, it starts to go in a random direction for searching other balls. If another robot is closer to the ball, then it is more likely to raise the ball, so the first robot turns around and goes back in the search mode. This solution was chosen to compare the normal programmed decision tree and neural networks, show the benefits of neural networks.

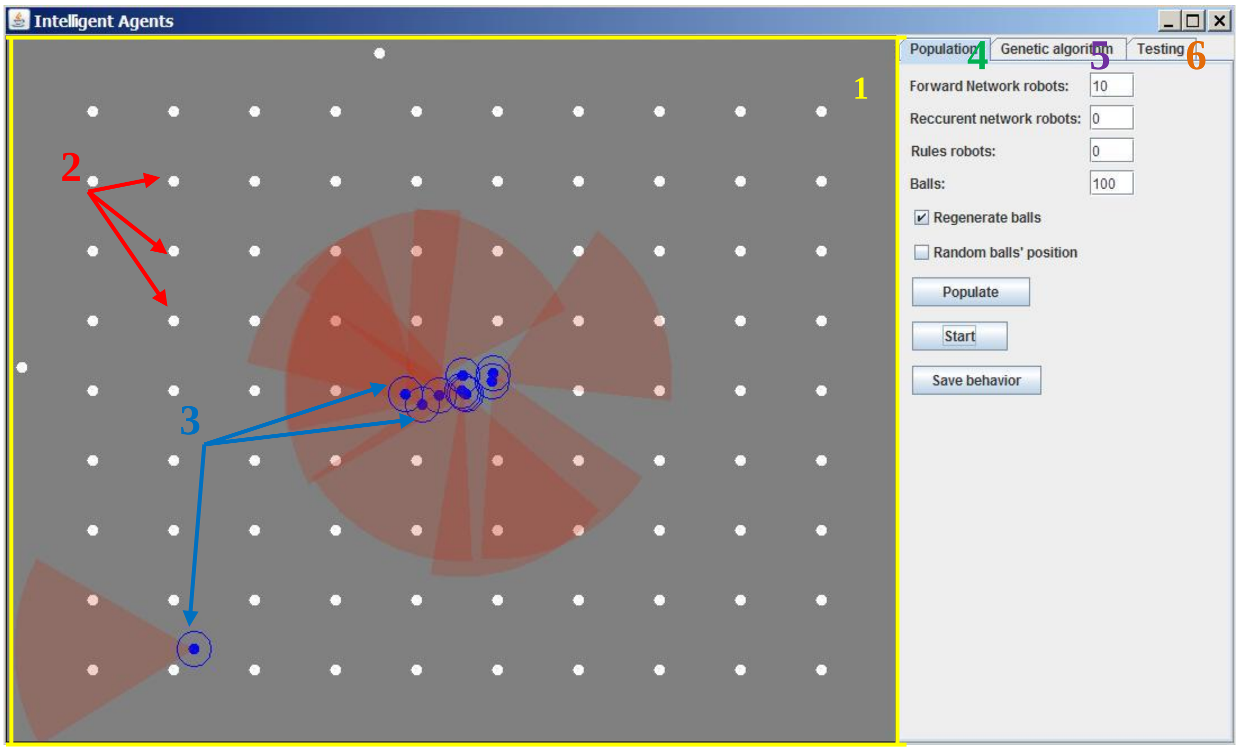
- **Forward neural network -** this method is a neural network without a memory. In other words, the neural network signals go directly from inputs to outputs. This network is deterministic, thus the robot will operate immediately based on the information about balls and robots in vision. In the next step the robot will forget their old state and will operate based on new information.
- **Recurrent neural network** - this method is a neural network with the memory of the previous step of the robot. This network should improve the efficiency of collecting balls by storing information from the previous step.

5. Emulation environment's description

Emulation environment was created with the essential requirements for the comparison of the proposed solutions. The environment is a field scattered with balls (Picture 1), in which the robots can move around and collect balls. Settings menu is created to control the population, testing and learning algorithm (Environment Settings).

5.1.Environment's elements

1. The field, where balls may be randomly scattered, and robots can move around and collect the balls.
2. Golf balls
3. Robot that emulates an automatic system, contains the behavior to move and collect the balls.
4. Population's settings
5. Learning algorithm's settings
6. Menu for testing the proposed solutions



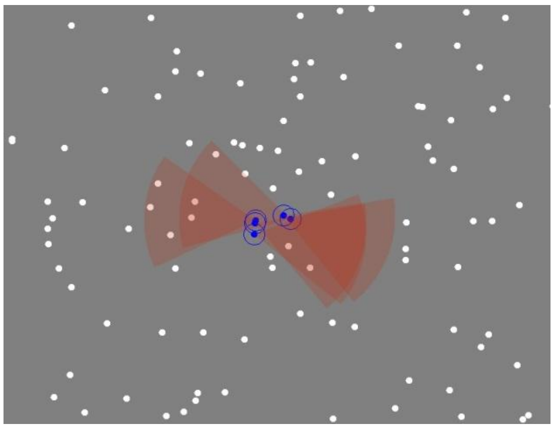
Forward Network robots: 10
Reccurent network robots: 0
Rules robots: 0
Balls: 100
[✓] Regenerate balls
[ ] Random balls' position

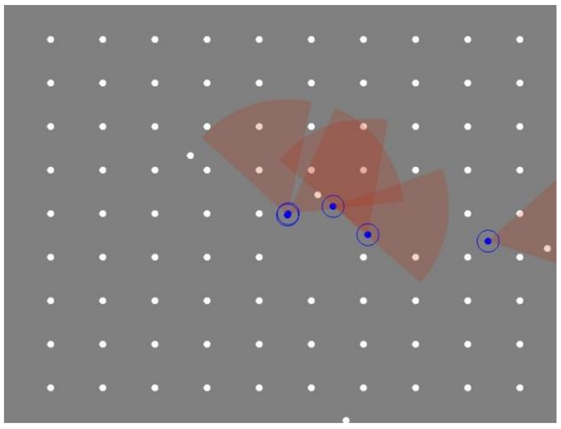
Picture 1 Application's main view

5.2.Environment's requirements

1. The robots can move around the field in any direction.
2. Robots can collect balls.
3. When a robot collects a ball, this one disappears and new ball is created in random location. Thus the same number of balls are on the field at any given time.
4. Field's size can be changed.
5. Field have no bounds. In other words, if the robot reaches the boundary of the field, it comes from the other side. Borders were removed because barriers' recognition by robot is outside the scope of this work.
6. While new population generates, all the robots appear in the center of the field and with the same angle of rotation to ensure that all robots have identical conditions to collect balls.

5.3.Types of golf balls' arrangement


Picture 2 Emulation environment with balls' random position

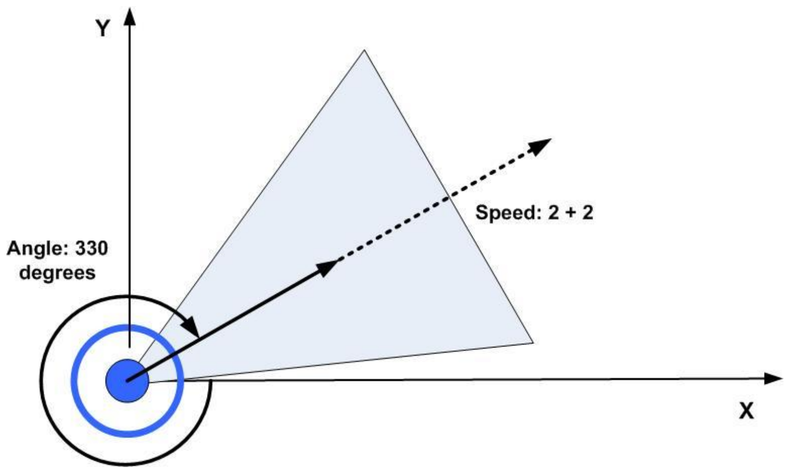

Picture 3 Emulation environment with balls' constant position

When the first population is created, balls can appear in a random location (Picture 2) or be constantly positioned (Picture 3). Constant positioning of balls is mainly used for training so that each robot has the same conditions to collect balls. Otherwise, if balls appear randomly during training, robot-leader appearing in an empty area does not have time to collect the required number of balls and not participate into the next generation.

5.4.Robot's structure

Movement

In order to move robot uses the speed and direction angle. Angle of direction can be in the range from 0 to 360 degrees. The angle starts from the X axis being the first quarter clockwise. The speed of movement of the robot can be changed from 0 to 4 pixels per tick emulation environment.



Picture 4 Robot's movement

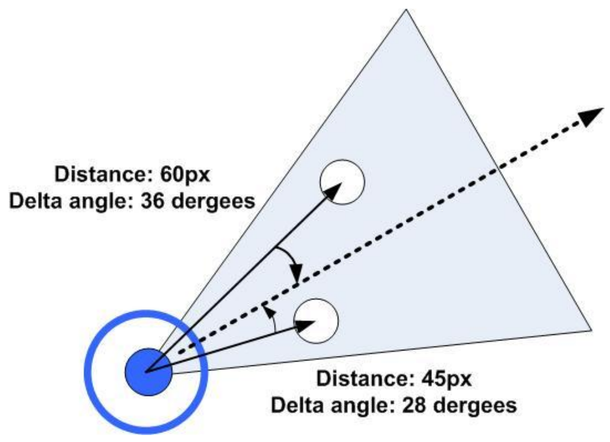
In the picture (Picture 4) the angle of direction of the robot equals 330 degrees. Moving speed was 2 pixels per tick increased to 4.

Obtaining information of the nearest objects

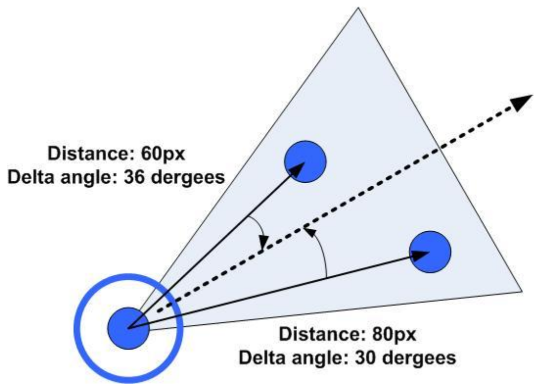
A robot can scan only the environment ahead of yourself on a given distance and a predetermined angle. In other words, it can see everything that is in its radar. The robot understands data such as:

1. Own speed
2. Own angle
3. Number of balls in the radar
4. The closest ball
5. Distance to the closest ball
6. Difference between the robot's angle and the closest ball's angle. (In other words, the angle which the robot must turn to move directly to the ball)
7. Number of robots in the radar
8. The closest robot
9. distance to the closest robot

10. Difference between the robot’s angle and the other closest robot’s angle. (In other words, the angle which the robot must turn to move directly to the other robot)



Picture 5 Nearest balls in vision



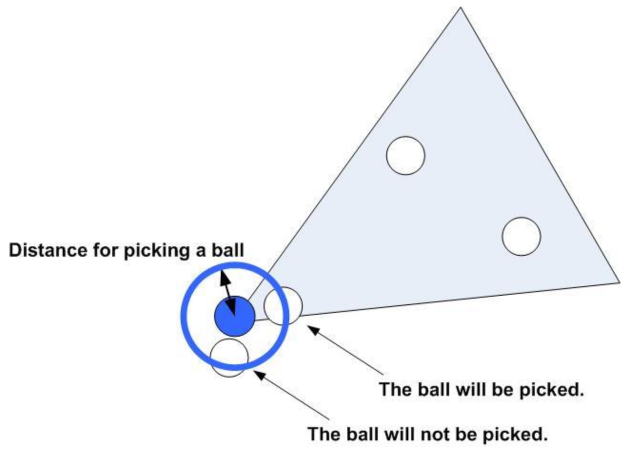
Picture 6 Nearest robots in vision

Thus In the picture (Picture 5) the robot sees two balls in the radar. It understands that the distance to the nearest ball is 45px and the angle is 28 degrees.

Similarly, In the picture (Picture 6) robot sees two robots in the radar. It understands that the distance to the nearest robot is 60px and the angle is 36 degrees. Information about other robots is necessary because, otherwise, the robots will move in the same direction and try to pick the same balls that will decrease the efficiency.

Balls’ collection

A robot can pick the ball if the distance between the ball and the robot is less than 15px and the ball is in the radar. Distance for picking balls is designated by a circle. Those balls that accidentally was in the region but were not detected the robot will not be raised.



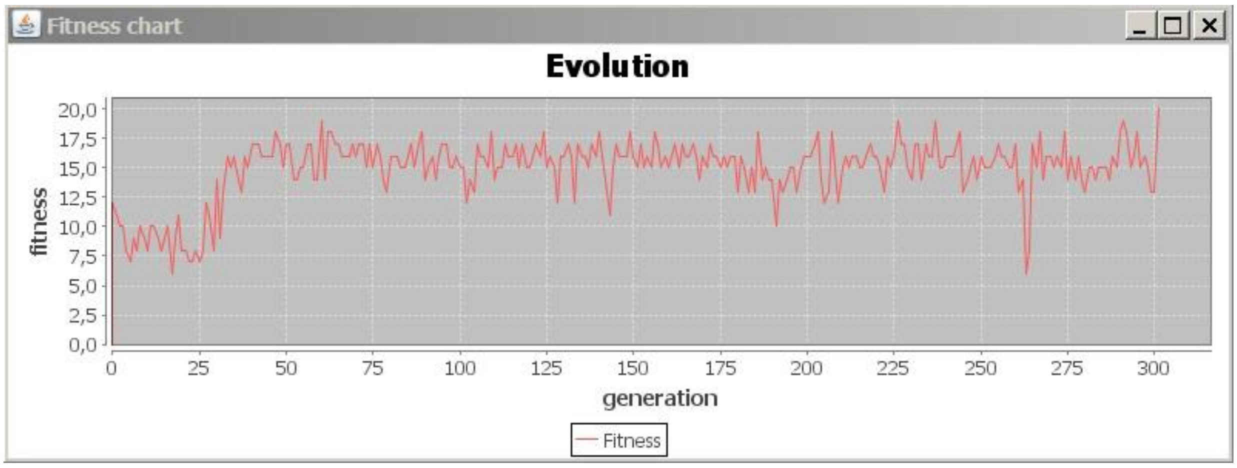
Picture 7 Balls picking

In the picture (Picture 7) the closest ball will be picked, because it is seen by the robot and is picking region.

5.5.Training graphs

In order to to better understand the learning process two graphs were created:

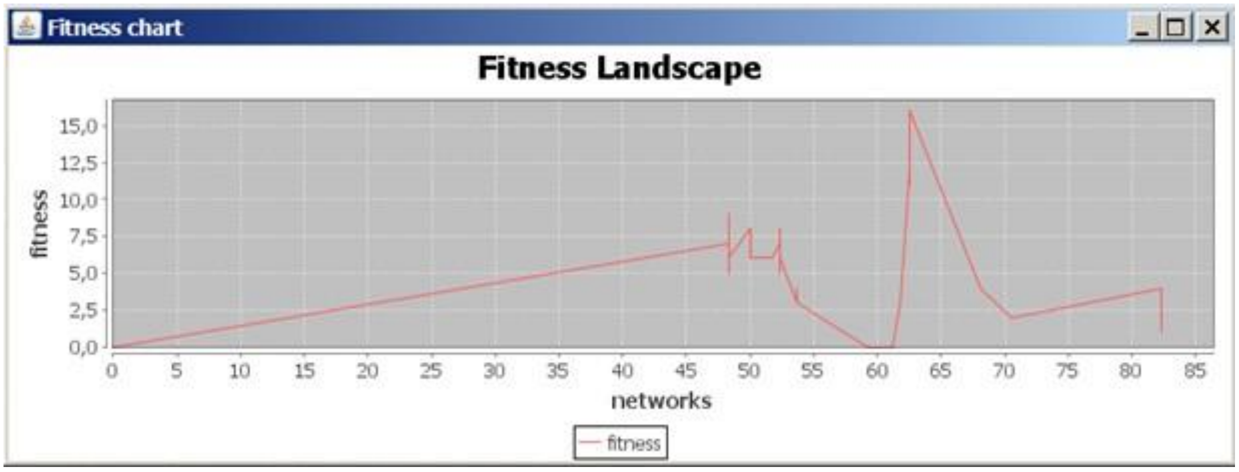
1. **Evolution chart** – the graph shows the process of evolution of the population in the course of learning genetic algorithm. The graph shows the best fitness function value of each population.



Picture 8 Evolution chart

- Y-axis shows the best value of the fitness function.
- X-axis shows the number of generations.

2. **Fitness landscape chart** – the graph shows the relationship between neural networks with the unique structure and fitness.



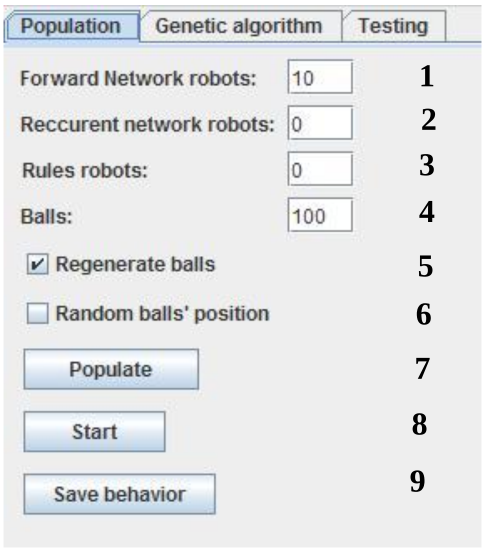
Picture 9 Fitness landscape chart

- Y-axis shows the value of the fitness function (calculated in the number of picked balls).
- X-axis shows neural networks with different unique structures.

5.6.Environment settings

In order to simplify the analysis and comparison of the effectiveness of decisions were made population settings where you can adjust number of balls on the field and the number of robots with different types of behavior to populate. Also genetic algorithm and testing settings were added for finding the optimal training options.

5.6.1. Population settings

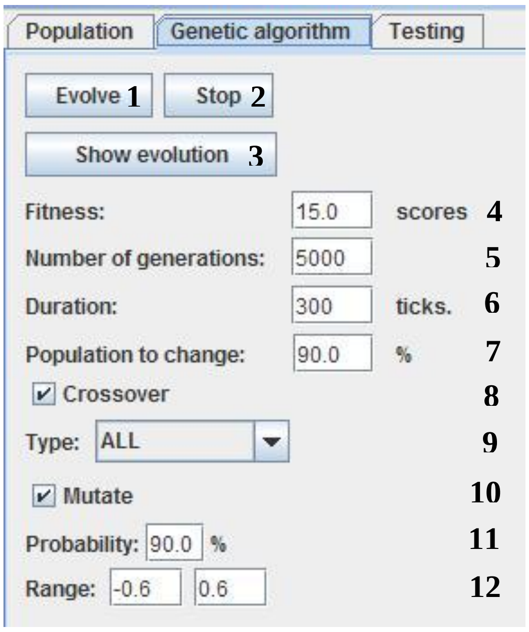

Population | Genetic algorithm | Testing

Forward Network robots: 10 — 1
Reccurent network robots: 0 — 2
Rules robots: 0 — 3
Balls: 100 — 4
[✓] Regenerate balls — 5
[ ] Random balls' position — 6
Populate — 7
Start — 8
Save behavior — 9

Picture 10 Population Settings

1. **Forward network robots:** Number of robots with a forward neural network to create a population
2. **Recurrent network robots:** Number of robots with a recurrent neural network to create a population
3. **Rules robots:** Number robots with a decision tree to create a population
4. **Balls:** Number of golf balls on the field
5. **Regenerate balls:** If checked, the balls will always appear
6. **Random balls' position:** If checked, the balls will occur in a random place, otherwise balls will be placed in the grid.
7. **Populate:** Creates an initial population using the above parameters.
8. **Start/Stop:** Stops or starts the emulation process.
9. **Save behavior:** saves the current behavior (MovementBehavior) robot in a configuration file

5.6.2. Genetic algorithm settings

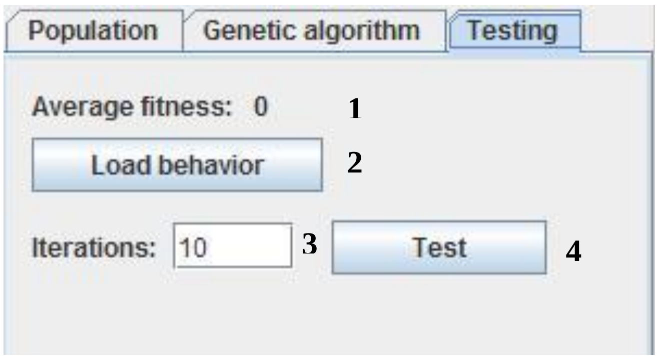


Fitness: 15.0 scores 4
Number of generations: 5000 5
Duration: 300 ticks. 6
Population to change: 90.0 % 7
☑ Crossover 8
Type: ALL 9
☑ Mutate 10
Probability: 90.0 % 11
Range: -0.6 0.6 12

Picture 11 Genetic algorithm settings

1. **Evolve:** Starts training robots using genetic algorithm.
2. **Stop:** Stops learning genetic algorithm.
3. **Show evolution:** Show or hides emulation process of every generation in real time while training.
4. **Fitness:** Sets the threshold of the fitness function, after which the training is considered complete (measured in number of collected balls).
5. **Number of generations**: if the fitness value is not specified then learning will continue until the specified number of generations will be created. In this case, the best robot behavior is chosen from the all generations.
6. **Duration:** Sets the lifetime of each generation (measured in ticks of emulation). In the example robot-leader must collect at least 15 balls in 300 ticks for the successful completion of training.
7. **Population to change:** Sets the amount of robots from the population in percentage that will be changed. For example, if a population contains 10 robots and need to change 80% of the population, then two robot-leaders with a maximum picked balls go into the next generation, the remaining 8 robots will be removed from the population and replaced with mutated individuals.
8. **Crossover:** If checked, the algorithm crosses chromosomes of two robots' neural network in order to create a new one.
9. **Type:** Specifies the type of crossover. Available options: ONE_POINT, TWO_POINTS, UNIFORM, ALL. If you select ALL then the algorithm uses all possible types of crossover.
10. **Mutate:** If checked, the algorithm uses mutation to change neural network's chromosome for training.
11. **Probability:** Sets what percentage of the chromosomes will be mutated in a neural network
12. **Range:** Sets the minimum and maximum amount of chromosome's mutation (interval from -1.0 to 1.0)

5.6.3. Testing settings



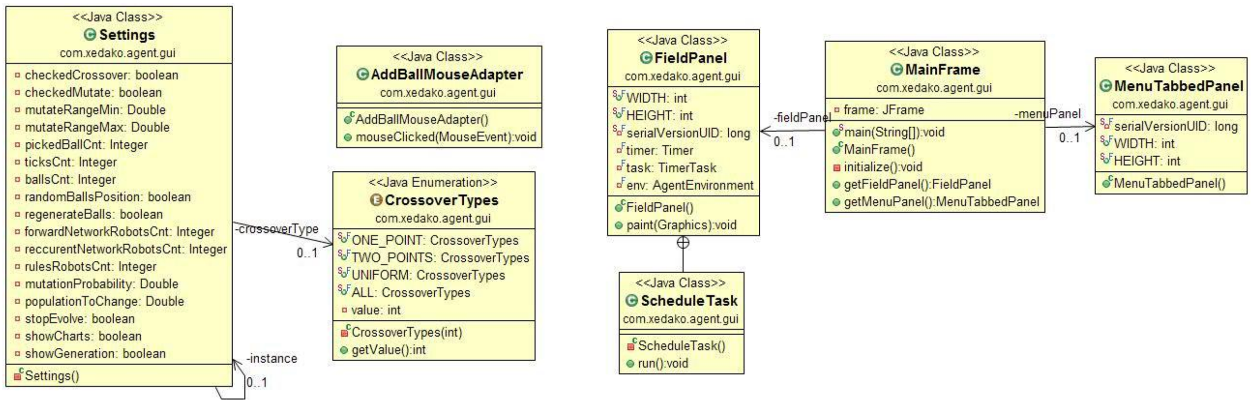
Picture 12 Testing settings

1. **Average fitness:** shows the average fitness value for the robot after the test completed
2. **Load behavior:** loads the robot's behavior (MovementBehavior) from the file and fills the environment with the robots
3. **Iterations :** the number of iterations to test
4. **Test:** runs test with specified number of iterations

6. Application Design

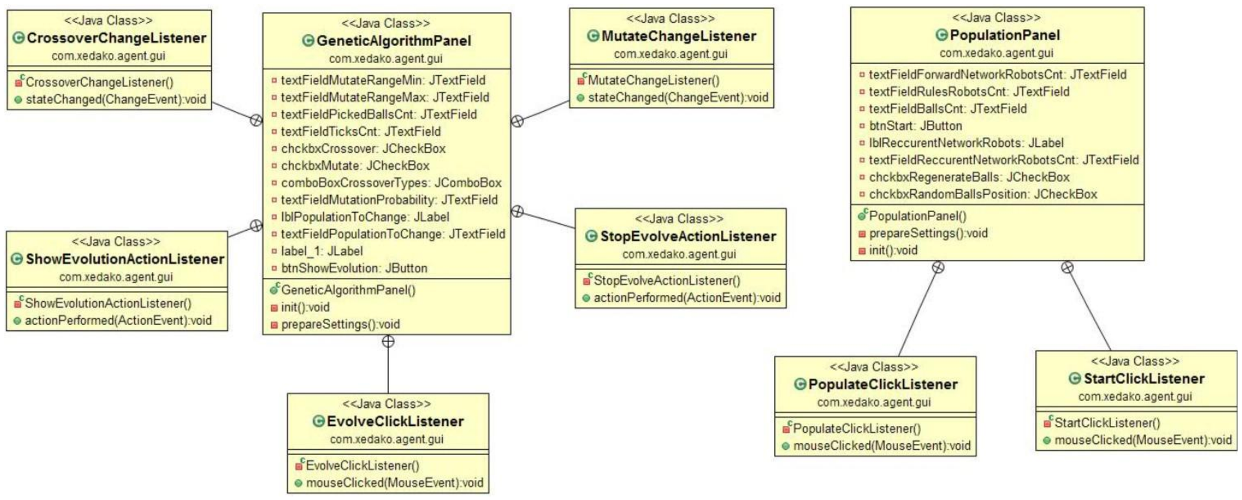
6.1.Graphic user interface

Package **com.xedako.agent.gui** contains classes to display panels and field settings. Also contains classes to control the flow of learning via graphical interface.



Picture 13 Class diagram. Main frame and settings

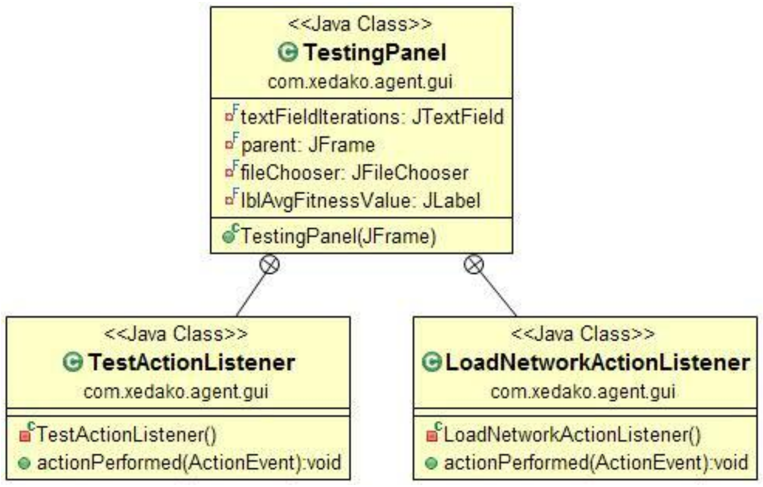
1. **Settings:** contains all the settings that are used in the menus.
2. **CrossoverTypes:** a collection of all kinds of crossover that are used to create a new population
3. **AddBallMouseListener:** lets add additional golf balls on the field during emulation.
4. **MainFrame:** main point for starting the application.
5. **FieldPanel:** contains golf balls, robots. Emulation environment for robots.
6. **ScheduleTask:** runs emulation environment every time and redraws the field. Uses the Timer.
7. **MainTabbedPanel:** contains settings for the population, genetic algorithm and testing, allows to switch between them.



<<Java Class>>
CrossoverChangeListener
com.xedako.agent.gui
CrossoverChangeListener()
stateChanged(ChangeEvent):void
<<Java Class>>
GeneticAlgorithmPanel
com.xedako.agent.gui
textFieldMutateRangeMin: JTextField
textFieldMutateRangeMax: JTextField
textFieldPickedBallsCnt: JTextField
textFieldTicksCnt: JTextField
chckbxCrossover: JCheckBox
chckbxMutate: JCheckBox
comboBoxCrossoverTypes: JComboBox
textFieldMutationProbability: JTextField
lblPopulationToChange: JLabel
textFieldPopulationToChange: JTextField
label_1: JLabel
btnShowEvolution: JButton
GeneticAlgorithmPanel()
init():void
prepareSettings():void
<<Java Class>>
MutateChangeListener
com.xedako.agent.gui
MutateChangeListener()
stateChanged(ChangeEvent):void
<<Java Class>>
PopulationPanel
com.xedako.agent.gui
textFieldForwardNetworkRobotsCnt: JTextField
textFieldRulesRobotsCnt: JTextField
textFieldBallsCnt: JTextField
btnStart: JButton
lblReccurentNetworkRobots: JLabel
textFieldReccurentNetworkRobotsCnt: JTextField
chckbxRegenerateBalls: JCheckBox
chckbxRandomBallsPosition: JCheckBox
PopulationPanel()
prepareSettings():void
init():void
<<Java Class>>
ShowEvolutionActionListener
com.xedako.agent.gui
ShowEvolutionActionListener()
actionPerformed(ActionEvent):void
<<Java Class>>
StopEvolveActionListener
com.xedako.agent.gui
StopEvolveActionListener()
actionPerformed(ActionEvent):void
<<Java Class>>
PopulateClickListener
com.xedako.agent.gui
PopulateClickListener()
mouseClicked(MouseEvent):void
<<Java Class>>
StartClickListener
com.xedako.agent.gui
StartClickListener()
mouseClicked(MouseEvent):void
<<Java Class>>
EvolveClickListener
com.xedako.agent.gui
EvolveClickListener()
mouseClicked(MouseEvent):void

Picture 14 Class diagram. Genetic algorithm panel and population panel

1. **GeneticAlgorithmPanel:** genetic algorithm settings panel
2. **CrossoverChangeListener:** listener for checkbox «Crossover». Determines whether to use the crossover during training
3. **ShowEvolutionActionListener:** listener for the button «Show evolution». Pressing displays the emulation of each generation during training in real-time.
4. **MutateChangeListener:** listener for the button «Mutate». Specifies whether to use the mutation of chromosomes during training.
5. **StopEvolveActionListener:** listener for the button «Stop». Stops and starts training with the best robot emulation from the last generation.
6. **PopulationPanel:** Population settings panel.
7. **PopulateClickListener:** listener for the button «Populate». Pressing fills the field with golf balls, creates robots and places them on the field.
8. **StartClickListener:** listener for the button «Start». Pressing starts or stops environment emulation.



<<Java Class>>
TestingPanel
com.xedako.agent.gui
textFieldIterations: JTextField
parent: JFrame
fileChooser: JFileChooser
lblAvgFitnessValue: JLabel
TestingPanel(JFrame)
<<Java Class>>
TestActionListener
com.xedako.agent.gui
TestActionListener()
actionPerformed(ActionEvent):void
<<Java Class>>
LoadNetworkActionListener
com.xedako.agent.gui
LoadNetworkActionListener()
actionPerformed(ActionEvent):void

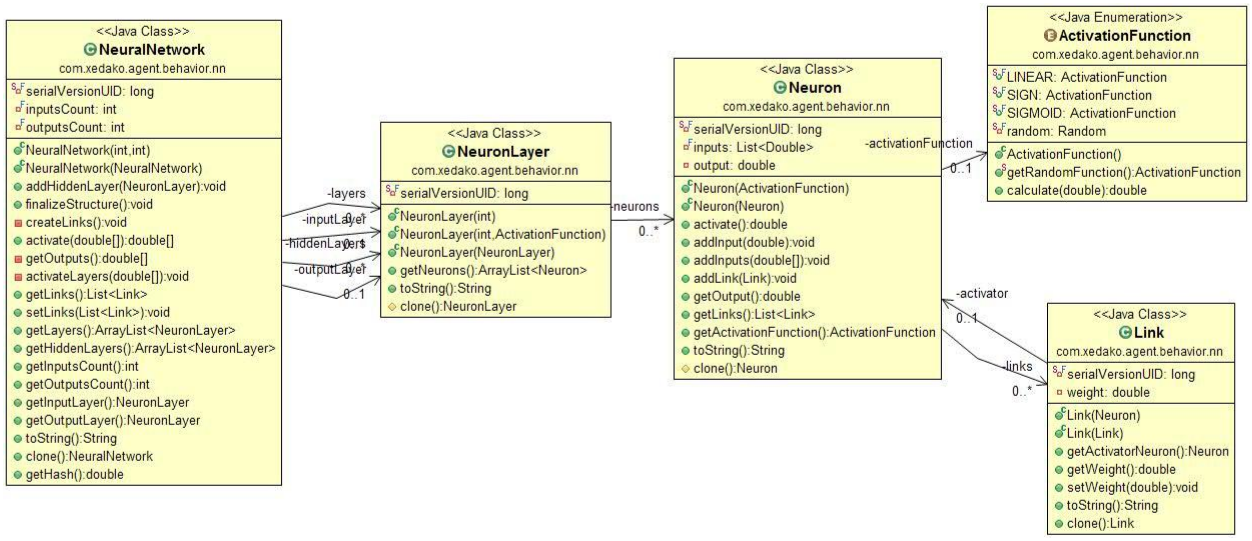
Picture 15 Class diagram. Testing panel

1. **TestingPanel:** Testing settings panel
2. **TestActionListener:** Listener for the button «Test». Pressing runs test with the loaded behavior

3. **LoadNetworkActionListener:** Listener for the button «Load network». Pressing opens the file's browser, where you can select a file to download for testing.

6.1.1. Neural Network

Package **com.xedako.agent.behavior.nn** contains classes for creating neural networks with various configurations.

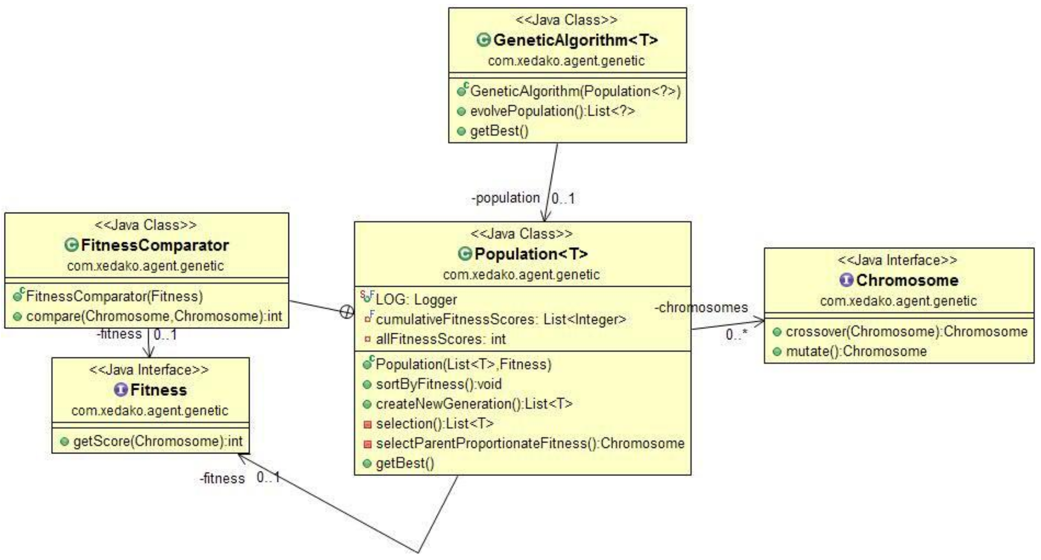


Picture 16 Class diagram. Neural network

1. **NeuralNetwork:** main class for creating a neural network with a given configuration. Accepts the number of inputs and outputs. It is also possible to add hidden neural layers. Method *finalizeStructure* connects neurons between themselves with random weights.
2. **NeuronLayer:** used to create a layer of neurons in a neural network. Creates a predetermined number of neurons with predetermined function.
3. **Neuron:** basic component in the neural network. Receives signals from other neurons. It provides its signal to other neurons based on activation function.
4. **ActivationFunction:** neuron's activation function. Determines the strength of the output signal of the neuron. In this work, the neural network uses linear and sigmoid activation function.
5. **Link:** defines the connection between two neurons. Contains the weight that strengthens or weakens the output signal.

6.1.2. Genetic algorithm

Package **com.xedako.agent.genetic** contains classes for initializing a genetic algorithm.

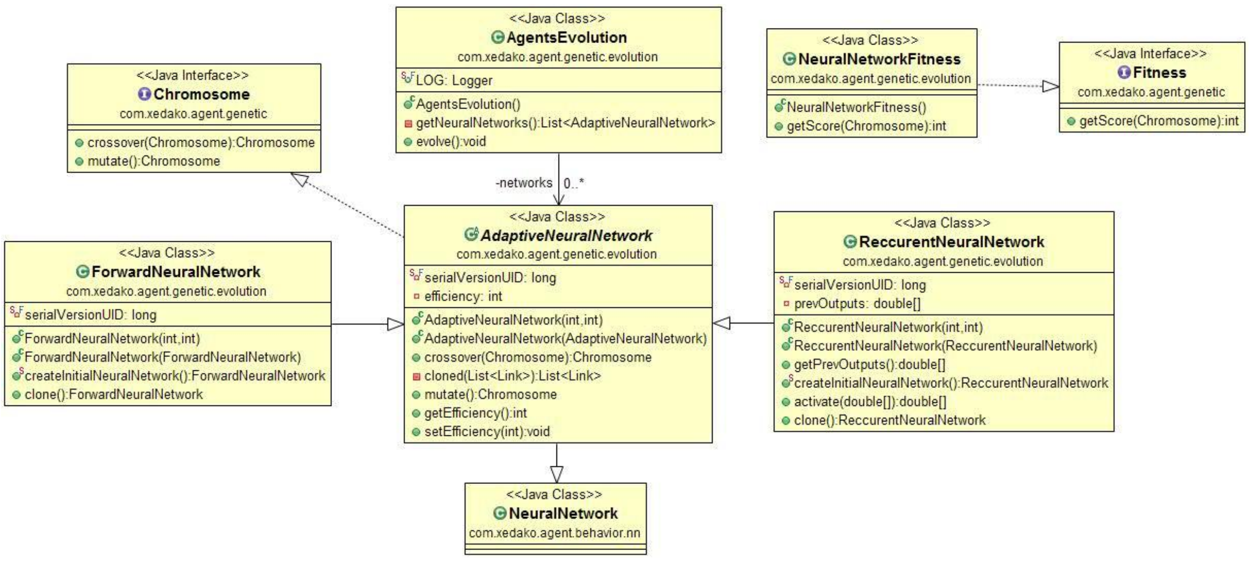


Picture 17 Class Diagram. Genetic algorithm

1. **GeneticAlgorithm:** Used to train the neural network. Starts evolution and returns the best robot-leader at the end.
2. **Population:** Used to store the generation of robots. Sorts generation by the picked balls, selects the best leaders and creates a new generation using the leaders.
3. **Chromosome:** Contains the weights of neural network as a vector. Can mutate and crossover with another chromosome.
4. **Fitness:** interface for fitness function's implementation**.**
5. **FitnessComparator:** Used to sort a population by fitness.

6.1.3. Evolution

Packege **com.xedako.agent.genetic.evolution** contains implementation of genetic algorithm for training neural networks.


<<Java Interface>>
Chromosome
com.xedako.agent.genetic
crossover(Chromosome):Chromosome
mutate():Chromosome
<<Java Class>>
AgentsEvolution
com.xedako.agent.genetic.evolution
LOG: Logger
AgentsEvolution()
getNeuralNetworks():List<AdaptiveNeuralNetwork>
evolve():void
<<Java Class>>
NeuralNetworkFitness
com.xedako.agent.genetic.evolution
NeuralNetworkFitness()
getScore(Chromosome):int
<<Java Interface>>
Fitness
com.xedako.agent.genetic
getScore(Chromosome):int
-networks 0..*
<<Java Class>>
ForwardNeuralNetwork
com.xedako.agent.genetic.evolution
serialVersionUID: long
ForwardNeuralNetwork(int,int)
ForwardNeuralNetwork(ForwardNeuralNetwork)
createInitialNeuralNetwork():ForwardNeuralNetwork
clone():ForwardNeuralNetwork
<<Java Class>>
AdaptiveNeuralNetwork
com.xedako.agent.genetic.evolution
serialVersionUID: long
efficiency: int
AdaptiveNeuralNetwork(int,int)
AdaptiveNeuralNetwork(AdaptiveNeuralNetwork)
crossover(Chromosome):Chromosome
cloned(List<Link>):List<Link>
mutate():Chromosome
getEfficiency():int
setEfficiency(int):void
<<Java Class>>
ReccurentNeuralNetwork
com.xedako.agent.genetic.evolution
serialVersionUID: long
prevOutputs: double[]
ReccurentNeuralNetwork(int,int)
ReccurentNeuralNetwork(ReccurentNeuralNetwork)
getPrevOutputs():double[]
createInitialNeuralNetwork():ReccurentNeuralNetwork
activate(double[]):double[]
clone():ReccurentNeuralNetwork
<<Java Class>>
NeuralNetwork
com.xedako.agent.behavior.nn

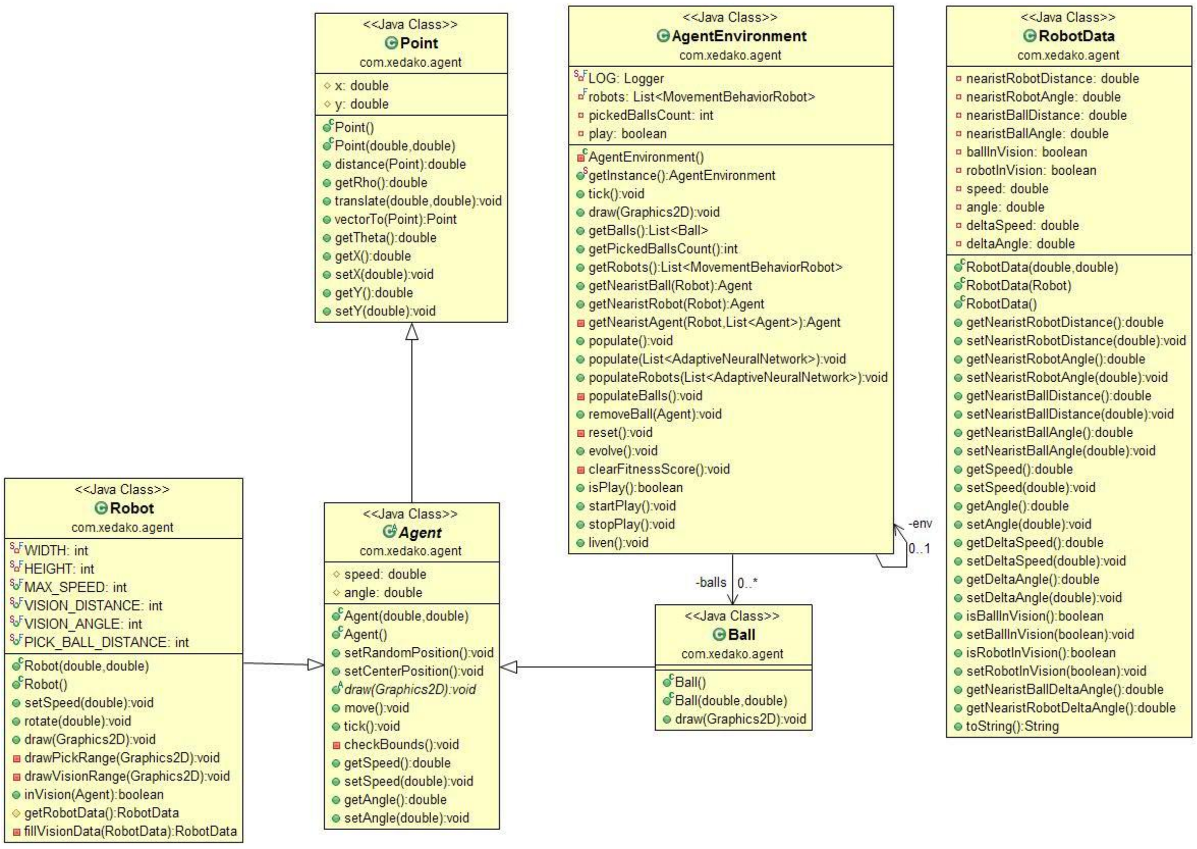
Picture 18 Class diagram. Evolution

1. **AgentsEvolution:** main class for training neural networks using genetic algorithm. Starts training from creating a population and emulating environment as long as the desired number of picked balls will be achieved.

2. **AdaptiveNeuralNetwork:** neural network that can be trained by genetic algorithm. Contains the implementation of crossover and mutation functions for neural network.
3. **ForwardNeuralNetwork:** implementation of a neural network with direct signal propagation
4. **RecurrentNeuralNetwork:** implementation of recurrent neural networks.
5. **NeuralNetworkFitness:** contains the implementation of the fitness function for the neural network

6.1.4. Environment

Package **com.xedako.agent** contains classes for emulation environment, balls' collection, movement of robots.



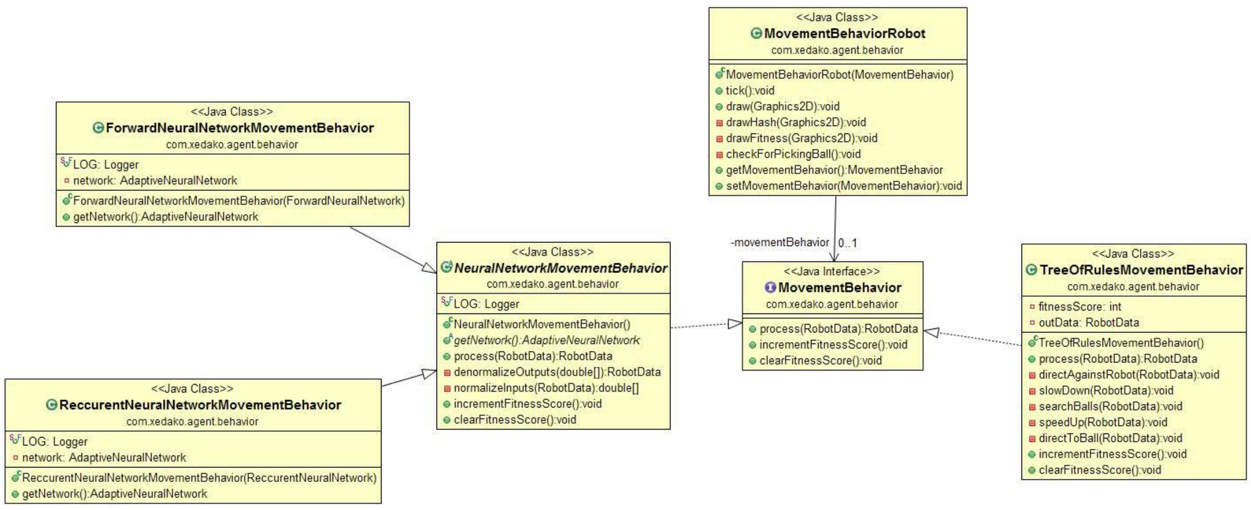
Picture 19 Class diagram. Environment

1. **AgentEnvironment:** The environment's emulation. Contains a list of robots and balls. Provides information about the location of all the objects. Can find the closest objects for the given robot.
2. **Point:** Contains the object coordinates. Also contains functions for manipulating geometric coordinates of the object.
3. **Agent:** Contains speed and direction's angle for a robot. Also contains functions for movement.
4. **Ball:** Represents the golf ball in the simulated environment.
5. **Robot:** Represents the robot to collect golf balls. Contains the limitations: maximum speed, vision angle, vision distance. Also contains rendering logic.
6. **RobotData:** Contains information about the environment, the nearest objects, the parameters for a particular robot. The data is used for the analysis by a specific decision (Forward neural

network, recurrent neural network, tree of rules) and computation speed and rotation's angle for the next step.

6.1.5. Robot's behavior

Package **com.xedako.agent.behavior** contains the implementation of robot behavior.



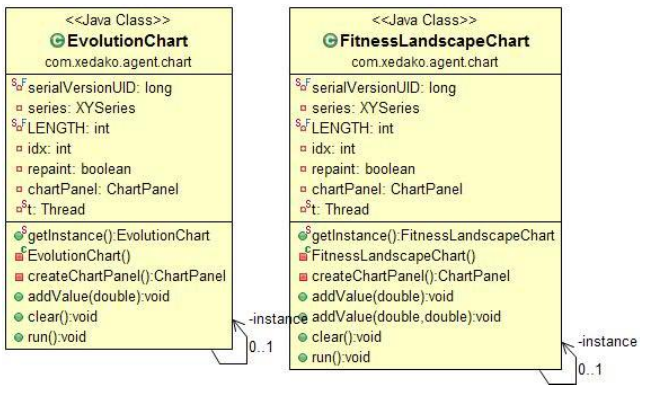
<<Java Class>>
MovementBehaviorRobot
com.xedako.agent.behavior
MovementBehaviorRobot(MovementBehavior)
tick():void
draw(Graphics2D):void
drawHash(Graphics2D):void
drawFitness(Graphics2D):void
checkForPickingBall():void
getMovementBehavior():MovementBehavior
setMovementBehavior(MovementBehavior):void
<<Java Class>>
ForwardNeuralNetworkMovementBehavior
com.xedako.agent.behavior
LOG: Logger
network: AdaptiveNeuralNetwork
ForwardNeuralNetworkMovementBehavior(ForwardNeuralNetwork)
getNetwork():AdaptiveNeuralNetwork
-movementBehavior
0..1
<<Java Class>>
NeuralNetworkMovementBehavior
com.xedako.agent.behavior
LOG: Logger
NeuralNetworkMovementBehavior()
getNetwork():AdaptiveNeuralNetwork
process(RobotData):RobotData
denormalizeOutputs(double[]):RobotData
normalizeInputs(RobotData):double[]
incrementFitnessScore():void
clearFitnessScore():void
<<Java Interface>>
MovementBehavior
com.xedako.agent.behavior
process(RobotData):RobotData
incrementFitnessScore():void
clearFitnessScore():void
<<Java Class>>
TreeOfRulesMovementBehavior
com.xedako.agent.behavior
fitnessScore: int
outData: RobotData
TreeOfRulesMovementBehavior()
process(RobotData):RobotData
directAgainstRobot(RobotData):void
slowDown(RobotData):void
searchBalls(RobotData):void
speedUp(RobotData):void
directToBall(RobotData):void
incrementFitnessScore():void
clearFitnessScore():void
<<Java Class>>
ReccurentNeuralNetworkMovementBehavior
com.xedako.agent.behavior
LOG: Logger
network: AdaptiveNeuralNetwork
ReccurentNeuralNetworkMovementBehavior(ReccurentNeuralNetwork)
getNetwork():AdaptiveNeuralNetwork

Picture 20 Class diagram. Robot's behavior

1. **MovementBehaviorRobot:** Robot with a certain type of behavior.
2. **MovementBehavior:** Interface for robot's behavior.
3. **TreeOfRulesMovementBehavior:** The behavior of the robot using logic in form of a tree.
4. **NeuralNetworkMovementBehavior:** abstract class behavior of the robot with general logic for the use of neural networks.
5. **ForwardNeuralNetworkMovementBehavior:** The behavior of the robot using a neural network with a direct signal propagation
6. **RecurrentNeuralNetworkMovementBehavior:** The behavior of the robot using recurrent neural network.

6.1.6. Chart

Package **com.xedako.agent.chart** contains classes of graphs to analyze the progress of neural networks' training. Charting library was used TFreeJ http://www.jfree.org/jfreechart.



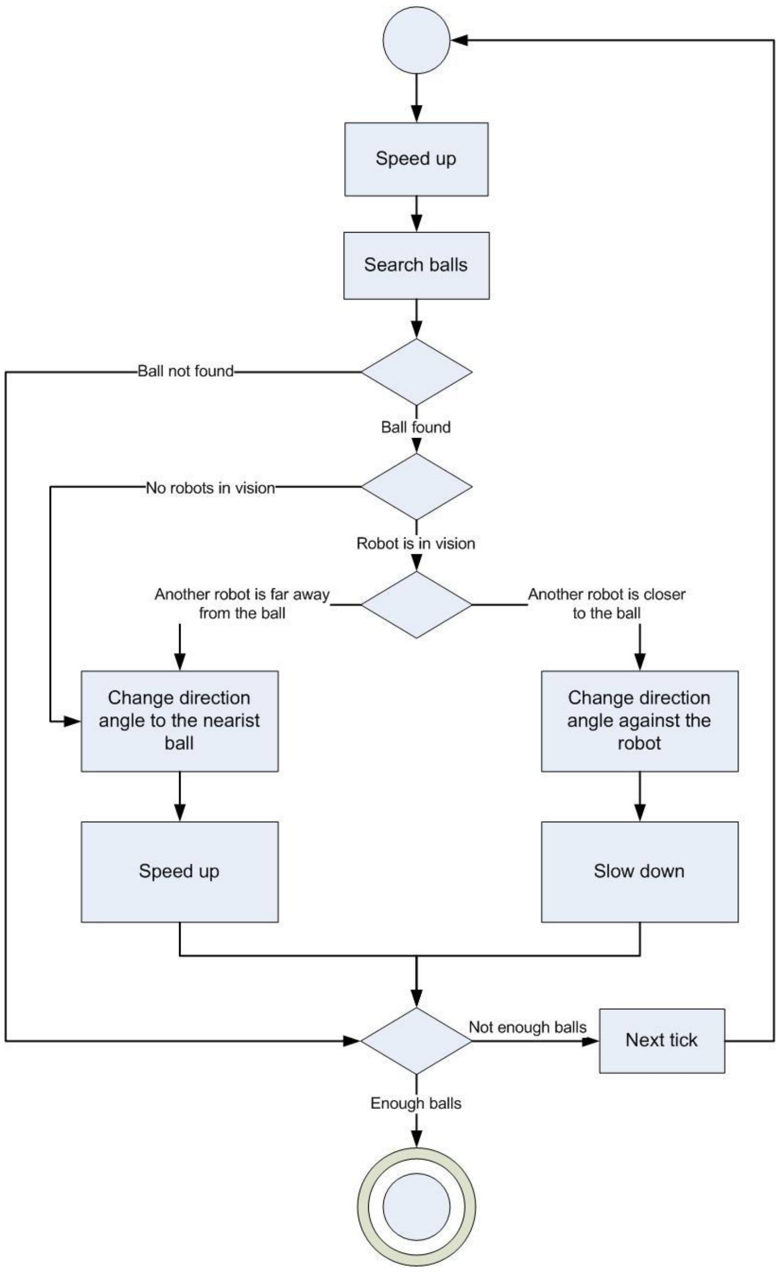
Picture 21 Class diagram. Charts

1. **EvolutionChart:** Chart of the robots' evolution. Shows the best robot-leader fitness in each generation.

2. **FitnessLandscapeChart:** Chart of fitness landscape. Shows relationship between fitness robot-leader's with unique neural network.

7. Solution 1 - Tree of rules

First solution for picking the balls is a programmed logic in the form of a tree. Unlike the neural network, the solution has no need to train. On the other hand the robot will not be able to adapt to changing environment's conditions. It is necessary always to rewrite the logic that is a big drawback of the solution. TreeOfRulesMovementBehavior class implements the solution. The logic of this solution is presented in the following diagram (Picture 22). It is possible to read the description of the decision at paragraphs (14, 15) in (Used materials).


Speed up
Search balls
Ball not found
Ball found
No robots in vision
Robot is in vision
Another robot is far away from the ball
Another robot is closer to the ball
Change direction angle to the nearist ball
Change direction angle against the robot
Speed up
Slow down
Not enough balls
Next tick
Enough balls

Picture 22 Tree of rules algorithm

Description of the algorithm:

1. The initial velocity of the robot is zero, the robot begins to accelerate
2. robot moves with constant velocity in a random direction to search the balls
3. if no balls found, the robot moves into search mode on the next tick
4. If the ball is found and another robots are not in the radar, the robot changes its direction to the ball and accelerates
5. If the ball is found, but the other robot was closer to that ball, the first robot turns, slows down and moves into the search mode on the next tick
6. If the ball is found and the distance between the ball and the second robot is more than the distance between the ball and the first robot, so it has time to pick up the ball first, the robot changes its direction to the ball and accelerates

8. Solution 2 – Forward Neural Network

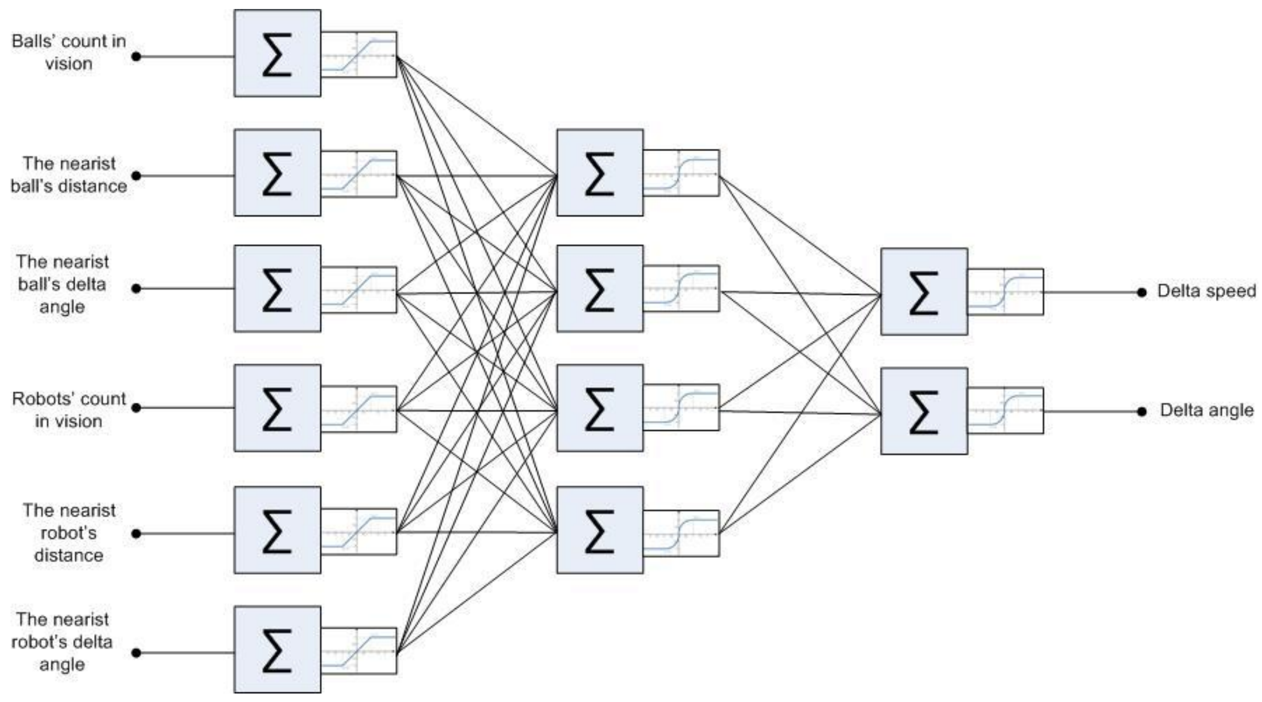
The second solution is implemented using forward neural network. In solution Tree of rules it is necessary to program the robot's behavior and rewrite the logic if environment's conditions was changed. Neural networks are a very effective solution. It is enough to give the neural network information about new environment's conditions, then to teach it with the genetic algorithm, and neural network will create the connections with sufficient weights between the inputs and outputs in order to achieve a performance in a modified environment. ForwardNeuralNetworkMovementBehavior class implements the solution. It is possible to read the description of the decision at paragraphs (11, 12) in (Used materials).

Neural network can be called a software implementation built on the principle of the organization and functions of biological neural chains (human brain). Artificial neural network is a black box, having a number of inputs and outputs, which can set the connections between inputs and outputs. It consists of adders (neurons), linked with connections that can amplify or attenuate incoming signals from other neurons. The neuron can be activated, summing up the signals, and send its output to next neurons. Thus it is possible to indicate which outputs neural network should send for the corresponding inputs. The network can learn and customize the weights of connections between neurons.

8.1. Description of structure

In this work network's structure has been used with the following parameters:

- 6 inputs
- 6 neurons in the first (input) layer with a linear activation function
- 4 neuron in the second (hidden) layer with a sigmoid activation function
- 2 neuron in the third (ouput) layer with sigmoid activation function
- 2 outputs


Balls' count in vision
The nearist ball's distance
The nearist ball's delta angle
Robots' count in vision
The nearist robot's distance
The nearist robot's delta angle
Σ
Delta speed
Delta angle

Picture 23 Forward neural network's structure

In the picture (Picture 23) a forward neural network's structure is shown. "Forward" means that the signals transfer through the network to outputs and are not used at the next activation. This neural network consists of three layers, which contains 6, 4 and 2 neurons respectively. Each layer is connected with the other layer such that each neuron of the first layer has a connection with each neuron of the next layer.

Input layer

First layer plays a role of the input layer. The number of neurons is equal to the number of its inputs. This layer's neurons receive their input and transmit forward, because they have no weights that could strengthen or weaken the incoming signal. Also these neurons use linear activation function, which does not affect the signal. Thus, this layer receives the inputs and sends them to the second inner layer without any modifications.

Hidden layer

Second layer plays the role of the hidden layer. Because of this example uses a lot of input parameters, it was decided to add a hidden layer with 4 neurons. The number of neurons was chosen empirically. The robot's behavior may differ greatly from different combinations of parameters. Hidden layer keeps the features by which the robot will behave more intelligently. This layer's neurons use sigmoid activation function to weaken strong signals and amplify weak signals.

Output layer

Last layer is the output layer. It has a number of neurons, equaled to the number of outputs. Similarly to neurons in the hidden layer, this layer's neurons use sigmoid activation function to weaken the strong signals and amplify weak signals.

Neural network's inputs

1. **Balls' count in vision:** the number of balls that the robot sees in the radar.
2. **The nearest ball's distance:** the distance to the nearest ball
3. **The nearest ball's delta angle:** the angle between the nearest ball and the robot. In other words, it is an angle which the robot must turn to move straight to the ball.
4. **Robots' count in vision:** the number of robots that the robot sees in the radar.
5. **The nearest robot's distance:** the distance to the nearest robot
6. **The nearest robot's delta angle:** the angle between the nearest other robot and the robot. In other words, it is an angle which the robot must turn to move straight to the other robot.

Neural network's outputs

1. **Delta speed**: speed, which must be added to the initial speed. The value can be either positive or negative. Thus the robot can speed up and slow down.
2. **Delta angle:** angle of rotation, which must be added to the initial angle of rotation. The value can be either positive or negative. Thus the robot can rotate both clockwise and counterclockwise.

8.2.Principle of operation of the neural network

1. Input layer's neurons receive input signals and transmit neurons to the inner layer without any modifications.
2. Connections between neurons strengthen or weaken the signals by multiplying weights of connections and signal.
3. Inner layer's neurons summarize all the received signals from the input layer's neurons, which have been modified by weights.
4. Then the sum of signals is passed through the activation function, and altered signal goes to the neurons of the output layer.
5. Neurons in the output layer process the signals similar to the neurons in the inner layer. Signals are multiplied by the corresponding connections' weights and summed by neuron. A sigmoid activation function modifies the sum of the signals.
6. Neurons of the output layer return the final signals.

Principle of operation of the neural network can be represented as a polynomial function where the inputs of the neural network are variables and outputs - function values. The main challenge is to choose the coefficients of multipliers.

In order to create a polynomial function it is necessary to mark the weights of the neural network with variables $\mathbf{w}_{\mathbf{ij,kt}}$, where

- i - number of the layer with the neuron that transmits a signal
- j - number of the layer's neuron that transmits a signal
- k - number of the layer with the neuron that receives a signal
- t - number of the layer's neuron that receives a signal

Thus weight $\mathbf{w}_{\mathbf{11,21}}$ means the weight between the first neuron of the first layer and the first neuron of the second layer.

Denote variables:

- **sigma** - Sigmoid activation function
- **a** - Balls' count in vision
- **b** - The nearest ball's distance
- **c** - The nearest ball's delta angle
- **d** - Robots' count in vision
- **e** - The nearest robot's distance
- **f** - The nearest robot's delta angle

Thus we have:

1. Function polynomial for output **Delta speed**

sigma(

$w_{21,31}$*sigma(a*$w_{11,21}$ + b*$w_{12,21}$ + c*$w_{13,21}$ + d*$w_{14,21}$ + e*$w_{15,21}$ + f*$w_{16,21}$) +

$w_{22,31}$*sigma (a*$w_{11,22}$ + b*$w_{12,22}$ + c*$w_{13,22}$ + d*$w_{14,22}$ + e*$w_{15,22}$ + f*$w_{16,22}$) +

$w_{23,31}$*sigma (a*$w_{11,23}$ + b*$w_{12,23}$ + c*$w_{13,23}$ + d*$w_{14,23}$ + e*$w_{15,23}$ + f*$w_{16,23}$) +

$w_{24,31}$*sigma (a*$w_{11,24}$ + b*$w_{12,24}$ + c*$w_{13,24}$ + d*$w_{14,24}$ + e*$w_{15,24}$ + f*$w_{16,24}$)

)

2. Function polynomial for output **Delta angle**

sigma(

$w_{21,32}$*sigma(a*$w_{11,21}$ + b*$w_{12,21}$ + c*$w_{13,21}$ + d*$w_{14,21}$ + e*$w_{15,21}$ + f*$w_{16,21}$) +

$w_{22,32}$*sigma (a*$w_{11,22}$ + b*$w_{12,22}$ + c*$w_{13,22}$ + d*$w_{14,22}$ + e*$w_{15,22}$ + f*$w_{16,22}$) +

$w_{23,32}$*sigma (a*$w_{11,23}$ + b*$w_{12,23}$ + c*$w_{13,23}$ + d*$w_{14,23}$ + e*$w_{15,23}$ + f*$w_{16,23}$) +

$w_{24,32}$*sigma (a*$w_{11,24}$ + b*$w_{12,24}$ + c*$w_{13,24}$ + d*$w_{14,24}$ + e*$w_{15,24}$ + f*$w_{16,24}$)

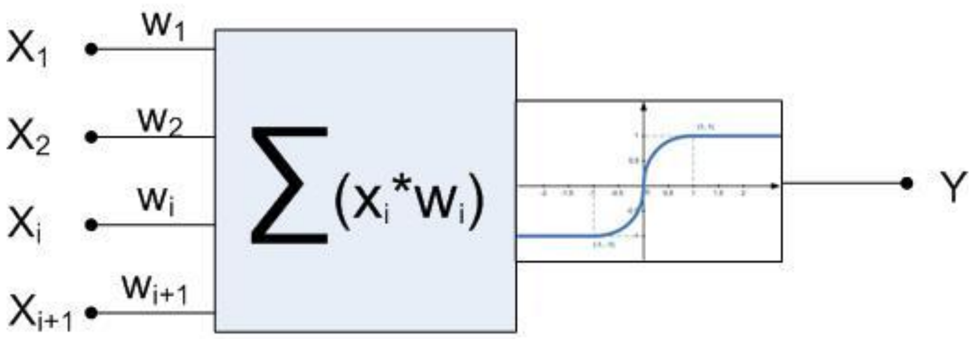
)

So by choosing the weights it is possible to set a completely different connections between inputs and outputs to configure the neural network.

8.3. Neuron

Neuron is a main element in the neural network. Almost all the neural networks use this element. By analogy with biological prototype artificial neuron has one output (axon) and a plurality of inputs (synapses). A signal can come from a neuron's single output to an arbitrary number of inputs of other neurons.

Formally neuron can be divided into three parts: the connections of a neuron, the adder and the activation function.



Picture 24 Neuron's structure

In the picture (Picture 24) the structure of a neuron is shown, where the inputs of the neuron are marked with (x_1, x_2, x_i, x_{i+1}), the weights are marked with (w_1, w_2, w_i, w_{i+1}). Next, the adder computes the sum of inputs and weights. Activation function changes the result and returns the output of the neuron (y). For example, this example uses a sigmoid activation function, which averages output a bit.

Neuron’s connections connect the output of one neuron and the inputs of another neuron. The weight may be positive or negative. Thus, connections with the positive weights amplify an input signal, connections with a negative weights weaken the input signal. In this work the weight range is used (-1, 1), so outputs could also be negative. In this case, the robot will be able to brake, accelerate and turn in a different direction.

Mathematical model

A neuron is a weighted adder, the single output is determined by its inputs and matrix of weights as follow:

$$y = f(u), \qquad u = \sum_{i=0}^{n} (\mathrm{x_i w_i})$$

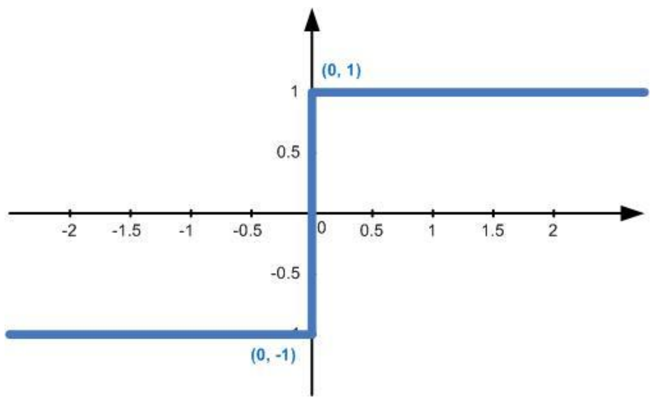
Where $\mathbf{x_i}$ and $\mathbf{w_i}$ — input signals of the neuron and the corresponding weights, respectively. **u** is called the induced local field, and **f (u)**-activation function of the neuron.

8.4.Activation function

Activation function determines the dependence of the signal at the output of the neuron from the weighted sum of the input signals. In this work, the following activation functions were used.

8.4.1. Threshold function activation

Threshold activation function is a difference. Commonly used in digital systems where the task is to classify objects or divide them into groups.



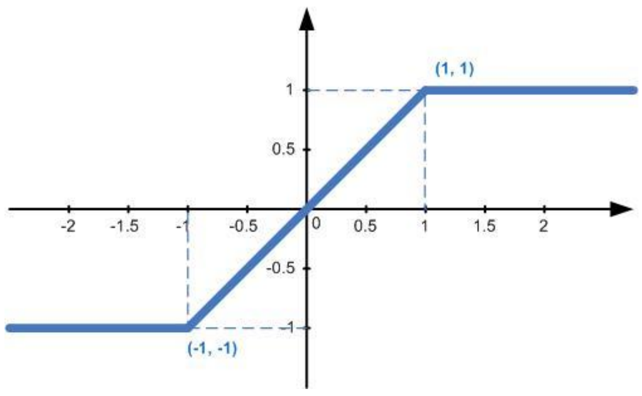
Picture 25 Sign activation function

$$f(x) = \begin{cases} -1, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Until the signal reaches zero, the function will return -1. If the signal is greater than zero, the function will return 1. Thus, the function may return discrete values -1 and 1.

8.4.2. Linear activation function

The function has two linear parts, where the activation function is identically equal to minimum and maximum allowable value. Also the function has an area in which the function is monotonically increasing. This function is used in the first input layer neurons of the neural network, because this function returns the unchanged signal for further processing in a hidden second layer.



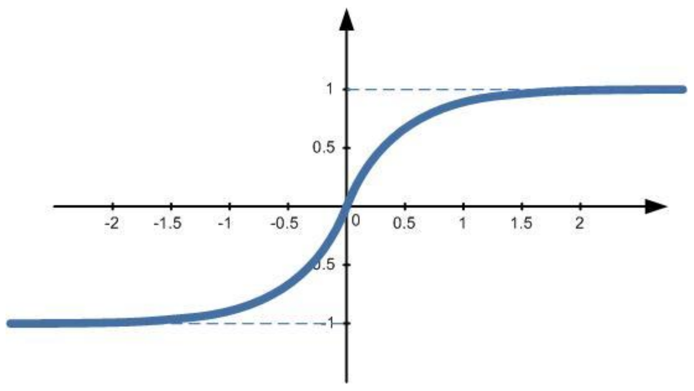
Picture 26 Linear activation function

$$f(x) = \begin{cases} -1, & x < -1 \\ x, -1 \leq x < 1 \\ 1, & x \geq 1 \end{cases}$$

Until the signal reaches 1, the function will return -1. If the signal is in the range of -1 to 1, the function will return the unchanged signal. If the signal is greater than 1, the function will return 1. Thus the function may return the discrete values on plots (-∞, -1) and [1, ∞). Unchanged signals will return at the interval [-1, 1).

8.4.3. Sigmoid activation function

It is monotonically increasing everywhere differentiable S-shaped nonlinear function with saturation. This function can amplify weak signals and weaken strong signals. It is used in the second and third layers in the neural network. The main task of this function is to get rid of noise and smooth signals.



Picture 27 Sigmoid activation function

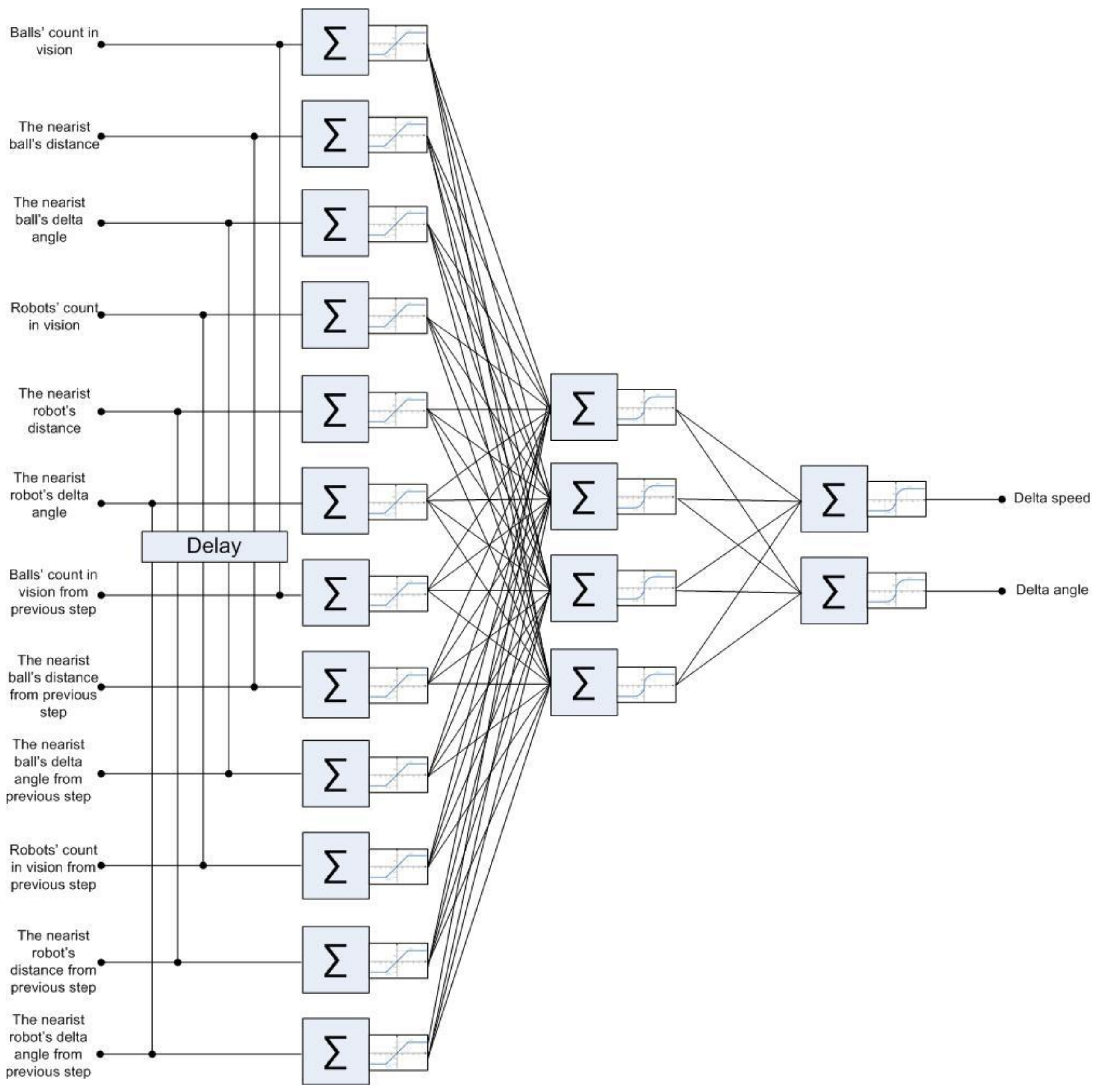
$$f(x) = \frac{2}{1 + e^{-4x}} - 1$$

While input tends to - ∞, output tends to -1. In the same way when input tends to + ∞, the output tends to 1. The graph has S-shaped curve at the section from -1 to 1 and returns values similar to the input signals.

9. Solution 3 - Recurrent neural network

Third decision was made with recurrent neural network. The most difficult type of neural networks, where feedback exists. In this context feedback means that network uses previous inputs at the next iteration. The previous input values are applied to respective inputs at the next activation of the network. RecurrentNeuralNetworkMovementBehavior class implements the solution. It is possible to read the description of the decision at paragraphs (9, 10) in (Used materials).

Presence of feedback allows to store and reproduce the sequence of reactions to one stimulus. This network is able to memorize the previous state and react to it.


Balls' count in vision
The nearist ball's distance
The nearist ball's delta angle
Robots' count in vision
The nearist robot's distance
The nearist robot's delta angle
Balls' count in vision from previous step
The nearist ball's distance from previous step
The nearist ball's delta angle from previous step
Robots' count in vision from previous step
The nearist robot's distance from previous step
The nearist robot's delta angle from previous step
Delay
Delta speed
Delta angle

Picture 28 Recurrent neural network's structure

9.1. Description of the structure

The structure of this network is very similar to the structure of the neural network with direct signal propagation (Forward neural network), so no need to describe the structure again. Unlike the forward neural network that network has the additional inputs from previous iteration.

1. **Balls' count in vision from previous step:** the number of balls in radar in the previous step
2. **The nearest ball's distance from previous step:** distance to the nearest ball in previous step
3. **The nearest ball's delta angle from previous step:** the angle between the ball and the nearest robot. It is an angle the robot has to turn in order to move straight to the ball in the previous step
4. **Robots' count in vision from previous step:** the number of robots in radar in the previous step.
5. **The nearest robot's distance from previous step:** distance to the nearest robot in previous step

6. **The nearest robot's delta angle from previous step:** the angle between the robot and the nearest second robot. It is an angle the robot has to turn in order to move straight to the second robot in the previous step

Principle of operation is similar to the operation of (Forward neural network). However, the last step is distinguished by the fact that the output layer’s neurons return outputs and also transmit outputs to the corresponding inputs for the next activation of the network.

In order to construct polynomial function it is necessary to mark additional variables

1. **g -** Balls’ count in vision from previous step
2. **h -** The nearest ball’s distance from previous step
3. **i -** The nearest ball’s delta angle from previous step
4. **j -** Robots’ count in vision from previous step
5. **k -** The nearest robot’s distance from previous step
6. **l -** The nearest robot’s delta angle from previous step

Thus we have:

1. polynomial function for output **Delta speed**

sigma(

$w_{21,31}$*sigma(a*$w_{11,21}$+b*$w_{12,21}$+c*$w_{13,21}$ + d*$w_{14,21}$ + e*$w_{15,21}$ + f*$w_{16,21}$+g*$w_{17,21}$+h*$w_{18,21}$+i*$w_{19,21}$+j*$w_{20,21}$+k*$w_{21,21}$+l*$w_{22,21}$) +

$w_{22,31}$*sigma (a*$w_{11,22}$ + b*$w_{12,22}$ + c*$w_{13,22}$ + d*$w_{14,22}$ + e*$w_{15,22}$ + f*$w_{16,22}$+g*$w_{17,22}$+h*$w_{18,22}$+i*$w_{19,22}$+j*$w_{20,22}$+k*$w_{21,22}$+l*$w_{22,22}$) +

$w_{23,31}$*sigma (a*$w_{11,23}$ + b*$w_{12,23}$ + c*$w_{13,23}$ + d*$w_{14,23}$ + e*$w_{15,23}$ + f*$w_{16,23}$+g*$w_{17,23}$+h*$w_{18,23}$+i*$w_{19,23}$+j*$w_{20,23}$+k*$w_{21,23}$+l*$w_{22,23}$) +

$w_{24,31}$*sigma (a*$w_{11,24}$ + b*$w_{12,24}$ + c*$w_{13,24}$ + d*$w_{14,24}$ + e*$w_{15,24}$ + f*$w_{16,24}$+g*$w_{17,24}$+h*$w_{18,24}$+i*$w_{19,24}$+j*$w_{20,24}$+k*$w_{21,24}$+l*$w_{22,24}$)

)

2. polynomial function for output **Delta angle**

sigma(

$w_{21,32}$*sigma(a*$w_{11,21}$+b*$w_{12,21}$+c*$w_{13,21}$ + d*$w_{14,21}$ + e*$w_{15,21}$ + f*$w_{16,21}$+g*$w_{17,21}$+h*$w_{18,21}$+i*$w_{19,21}$+j*$w_{20,21}$+k*$w_{21,21}$+l*$w_{22,21}$) +

$w_{22,32}$*sigma (a*$w_{11,22}$ + b*$w_{12,22}$ + c*$w_{13,22}$ + d*$w_{14,22}$ + e*$w_{15,22}$ + f*$w_{16,22}$+g*$w_{17,22}$+h*$w_{18,22}$+i*$w_{19,22}$+j*$w_{20,22}$+k*$w_{21,22}$+l*$w_{22,22}$) +

$w_{23,32}$*sigma (a*$w_{11,23}$ + b*$w_{12,23}$ + c*$w_{13,23}$ + d*$w_{14,23}$ + e*$w_{15,23}$ + f*$w_{16,23}$+g*$w_{17,23}$+h*$w_{18,23}$+i*$w_{19,23}$+j*$w_{20,23}$+k*$w_{21,23}$+l*$w_{22,23}$) +

$w_{24,32}$*sigma (a*$w_{11,24}$ + b*$w_{12,24}$ + c*$w_{13,24}$ + d*$w_{14,24}$ + e*$w_{15,24}$ + f*$w_{16,24}$+g*$w_{17,24}$+h*$w_{18,24}$+i*$w_{19,24}$+j*$w_{20,24}$+k*$w_{21,24}$+l*$w_{22,24}$)

)

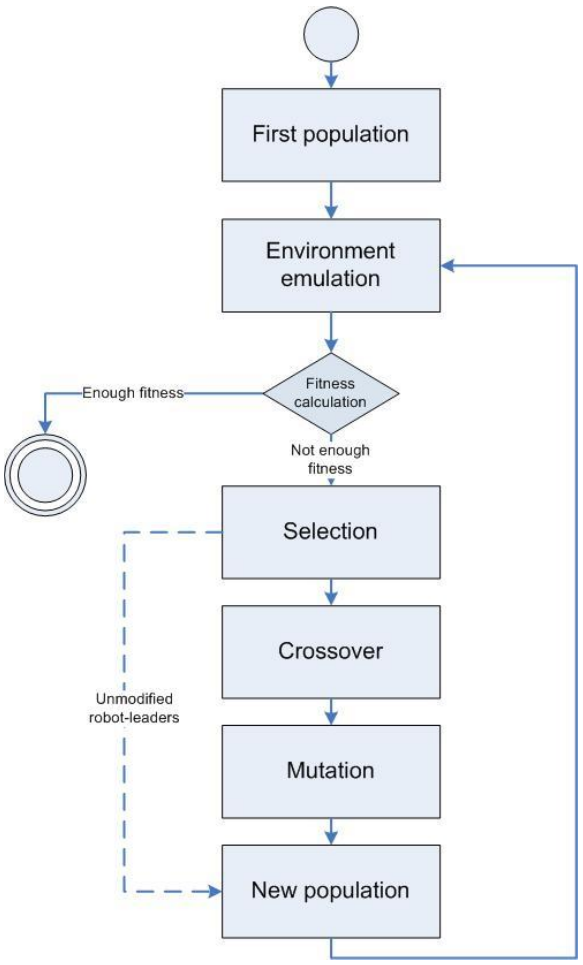
Similarly, by changing the weights, it is possible to set a completely different connections between inputs and outputs and configure the network properly.

10. Genetic algorithm

Genetic algorithm was implemented and used for training the neural networks. This is an heuristic algorithm for finding the optimal solution via random selection, combining and changing variations of the parameters by methods similar to natural selection in nature.

10.1. Description of the algorithm

This algorithm can be divided into 7 main processes shown in the chart (Picture 29): first population, selection, crossover, mutation, new population, environment population, fitness function.



Picture 29 Genetic algorithm

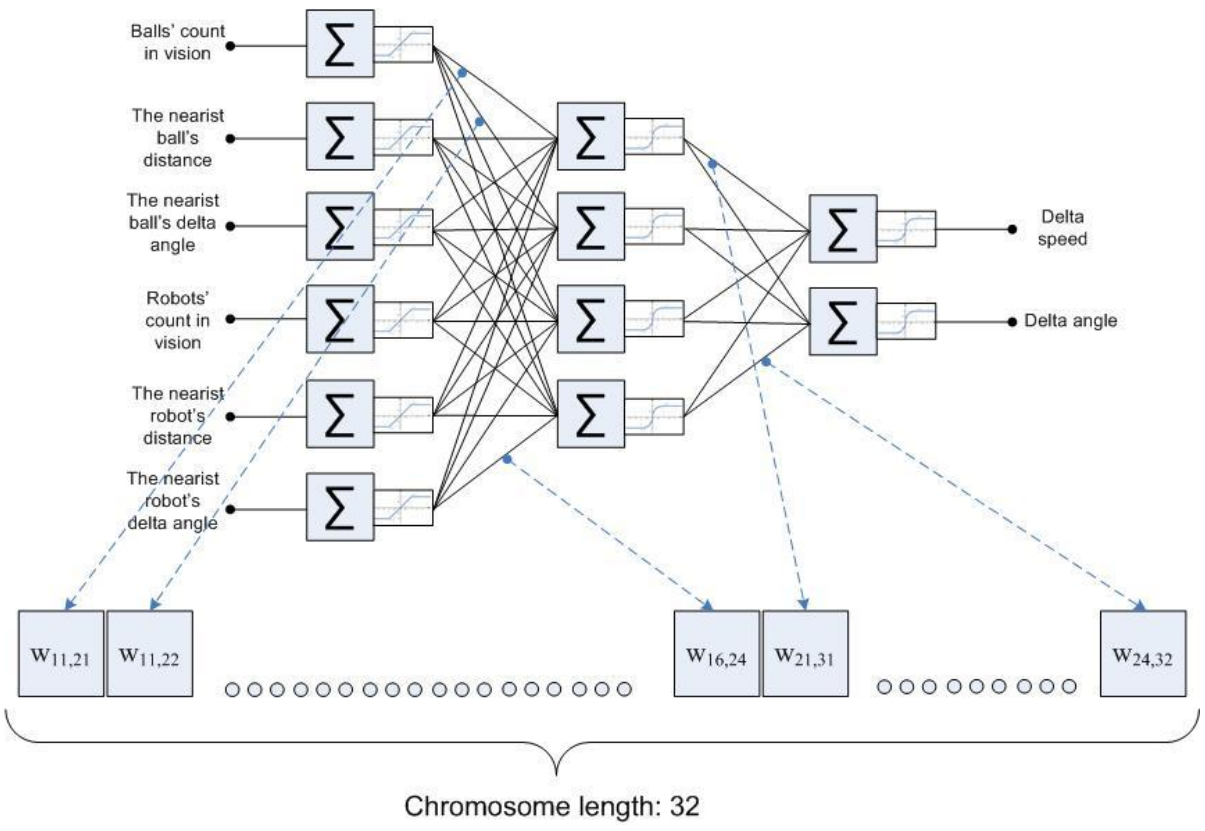
1. **First population:** at this stage robots will be added into the environment added with a given neural network (forward or recurrent neural network), whose weights are generated randomly. At the time all the robots usually rotate in one place, move randomly, in a

word, they are not trained.

2. **Environment emulation:** At this stage emulation starts. Balls are scattered at an equal distance from each other, and robots are created with one of the two neural networks. The environment will be launched for specified number of ticks in settings.
3. **Fitness calculation:** Each robot has to collect balls after emulating. Fitness to the environment for each robot equals to the number of collected balls. If the fitness is greater or equal than the target fitness value in settings, the evolution ends.
4. **Selection:** All the robots will be sorted by the value of fitness. At this stage the specified percentage of the population will be moved to the new population without any modifications in order to save old population's features.
5. **Crossover:** a neural network can be represented as a vector of weights (chromosome). At this stage, two neural networks (parents) will be randomly selected, and their chromosomes are crossed for creating a new neural network (child).
6. **Mutation:** crossover interchanges parents' chromosome and cannot generate chromosomes with very different characteristics. At this stage, all the child chromosomes will be mutated in order to get different features.
7. **New population:** at this stage a new population is filled with crossed and mutated chromosomes in order to calculate new fitness for each robot in environment.

10.2. Chromosome representation

Genetic algorithm is used to train the neural network. Neural network can be represented in a convenient form that can be easily changed. That form is a chromosome - a vector of bits or numbers which characterize the structure of the network. In the scope of the work, the network is presented in the form of the weights' vector, as shown in the picture (Picture 30).



Picture 30 Chromosome representation

The chromosome contains 32 weights for forward neural network and 56 for recurrent neural network. By replacing and changing the weights it is possible to set the network structure, by which robots will start to collect balls.

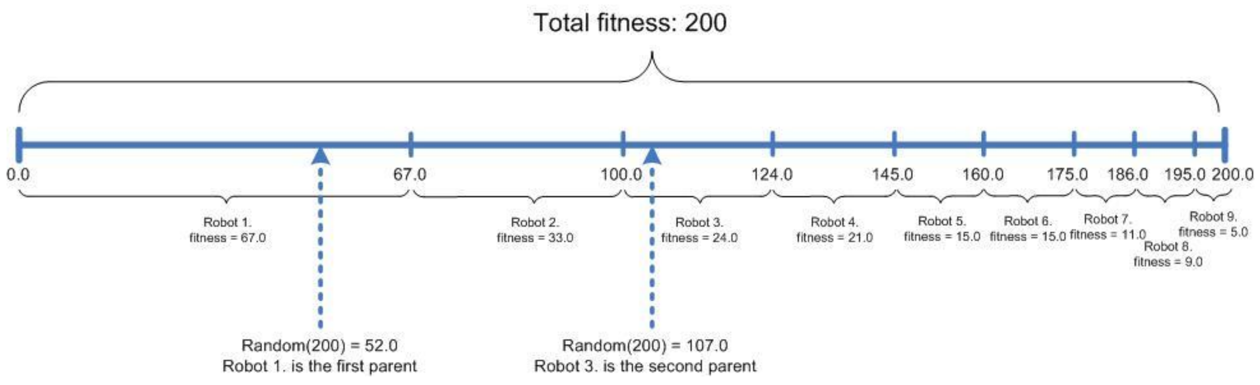
10.3. Creating population

As mentioned above, the initial population is created with random weights, in other words a random set of genes. It is necessary to create the next population, using crossover and mutation only for robots whose neural networks are configured more proper (robots that collect more balls). (Roulette wheel selection) algorithm is used to select the robots from the population.

All the populations must have the same number of individuals (robots). However, the population may not contain too few individuals. By analogy with the evolution in nature, if the population is too small, it may not contain all the necessary features of individuals, where one individual of them could collect all the best features for achieving the highest efficiency. In this work it was decided to use 10 robots for each population.

10.3.1. Roulette wheel selection

This algorithm allows select a robot from a population in proportion to their fitness value. Always two robots (parents) are selected whose chromosomes are used to create a new robot (child). In other words, robots, whose fitness value is higher than others (robots that collected more balls) will be more likely to become a parent.



Picture 31 Roulette wheel selection

In the picture (Picture 31) the algorithm is presented for selecting robot-parents. As can be seen the population has 9 robots. After the emulation each robot has collected balls. This number of balls is the value of the fitness. Robots have been sorted according to the value of fitness in ascending order. For example, the first robot's fitness value is 67 (it collected 67 balls). Then all the fitness values are summed and total fitness is 200. On the chart it is possible to notice that there are cumulative fitness (67.0, 100.0, 124.0, etc.) after each robot, which includes previous robots' fitness value.

Roulette wheel selection's algorithm

```
Chromosome selectParentProportionateFitness()
Collections.sort(chromosomes, new FitnessComparator(fitness));
allFitnessScores = 0;
cumulativeFitnessScores.clear();
cumulativeFitnessScores.add(allFitnessScores);
```

```
for (Chromosome c : chromosomes) {
    double score = fitness.getScore(c);
    allFitnessScores += score;
    cumulativeFitnessScores.add(allFitnessScores);
}
```

```
int probability = (int) (Math.random() * allFitnessScores);
for (int i = 1; i < cumulativeFitnessScores.size(); i++) {
  if (cumulativeFitnessScores.get(i - 1) <= probability
  && cumulativeFitnessScores.get(i) >= probability) {
       return chromosomes.get(i - 1);
  }
}
return ListUtil.getLast(chromosomes);
```

All chromosomes are sorted in ascending order by their fitness value. The list `cumulativeFitnessScores` contains the cumulative value of fitness, the first value is 0. Next this list is filled with values in cycle. Each chromosome's efficiency is calculated and recorded in this list as the cumulative value. Next, a random number (`probability`) is calculated from the total fitness. In the next cycle the range is being found, where `probability` would be greater than the first number and less than the second. if the range is found, the function returns a chromosome belonging to the range, otherwise it returns the chromosome with the smallest value of fitness.

Thus, the chromosome has a probability of becoming a parent in proportion to the value of fitness.

10.4. Selection

At the selection's stage a certain percentage of individuals will be selected in population and transferred to the new population. The remaining individuals, that cannot be transferred to the new population due to their small fitness, will be removed. Selection will be processed before the crossover and mutation, because the next population must use all chromosomes from previous one in order to save as many features as possible. Number of robots, that will be removed, marked as a percentage in settings (**Population to change**). For example, if the population has 10 robots and population to change is 80%, then only two robot-leaders will be "alive" and can be to new population. The other 8 robots will be removed after creating a new population. Thus robots' survival depends on its fitness value.

10.4.1. Fitness function

Fitness function is a measure that shows how useful the specific structure of the network for picking balls. Fitness value is calculated after starting the emulation and equals the number of collected balls.

Because of the complexity of the system it is difficult to make a formula which was represented a smooth ascending function that clearly shows the effectiveness of all networks' structure. So the next function is simple enough and was chosen as a fitness function.

Fitness = picked balls' count

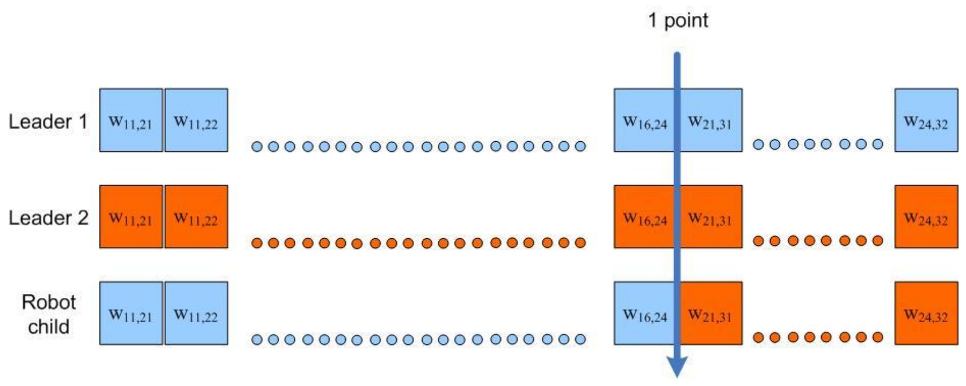
The curve must increase during the evolution at which robots could develop every time. In this case evolution passes quickly enough.

10.5. Crossover

Crossover – a genetic operator used to vary the robots from one population to another. It is similar to biological reproduction which is based on genetic algorithm. Using the algorithm (Roulette wheel selection) it is possible to select two robot-parents and to create a robot-child with the help of their chromosomes. There are three methods using in order to create a new chromosome: one-point crossover, two-points crossover, uniformly distributed crossover.

10.5.1. One-point crossover

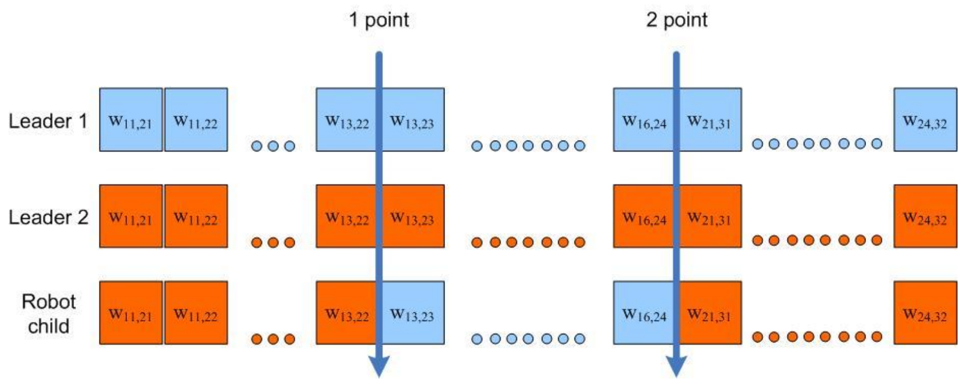
Two parents' chromosomes are selected, which are divided in half at a random location. It takes a random part of the first chromosome and a random part of the second chromosome for creating a new one.



Picture 32 One-point crossover

10.5.2. Two-points crossover

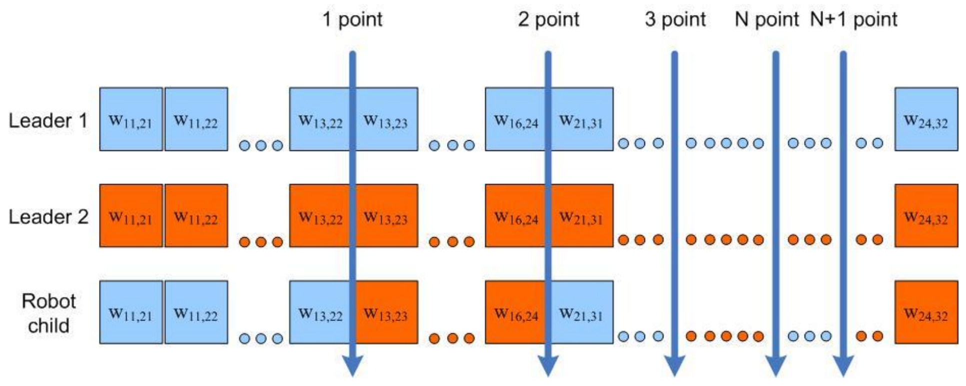
It is similar to the algorithm (one-point crossover), but this algorithm selects two points for dividing parents' chromosomes. Likewise, a new random chromosome takes part from one of the two parents' chromosome.



Picture 33 Two-points crossover

10.5.3. Uniformly distributed crossover

This algorithm uses a fixed index for blending chromosomes. Unlike previous algorithms, this crossover allows mix the genes not only by segment, but also by individual genes. Thus new chromosome has more differences.



Picture 34 Uniformly distributed crossover

Uniformly distributed crossover's algorithm

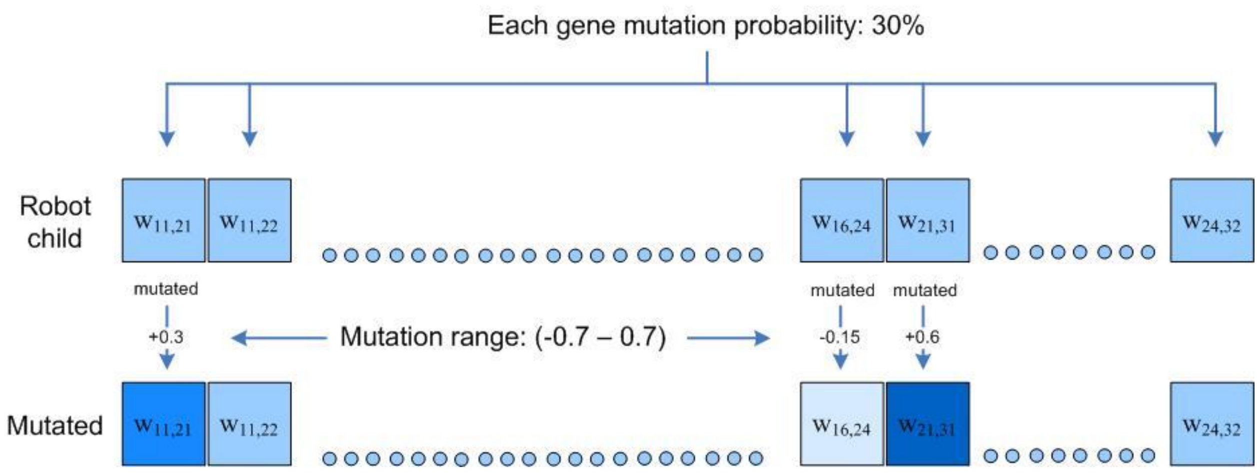
```
List<T> uniformCrossover(List<T> list, List<T> list2, int
crossCount) {
    List<T> crossed = new ArrayList<T>();
    int cumulIdx = 0;
    int crossPoint1Idx = 0;
    int crossPoint2Idx = 0;
    boolean firstList = true;
    int segmentAvgSize = list.size() / crossCount;
    for (int i = 0; i <= crossCount; i++) {
        crossPoint2Idx = (int) ((i != crossCount)
            ? cumulIdx + MathUtil.random(1, segmentAvgSize -1)
            : list.size());
        cumulIdx += segmentAvgSize;

        if (firstList) {
            crossed.addAll(list.subList(crossPoint1Idx,
            crossPoint2Idx));
        } else {
            crossed.addAll(list2.subList(crossPoint1Idx,
             crossPoint2Idx));
        }
            firstList = !firstList;
            crossPoint1Idx = crossPoint2Idx;
        }
        return crossed;
    }
```

The function takes two lists with the same size and the number of points indicating how many parts chromosomes will be divided to. The number of points is a random variable. Next, it is necessary to find the first and the second point of each segment. Firstly, the average length of the segment (`segmentAvgSize`) must be found. Secondly, the first point (`crossPoint1Idx`) omits 0. The variable (`cumulIdx`) denotes the point, where offset will be added to the side by a random amount. Thus, by adding the random offset it is possible to find the second point (`crossPoint2Idx`). After that, the list (`crossed`) copies chromosome' segment from the first or second parent, where the first point of the segment is (`crossPoint1Idx`) and second point is (crossPoint2Idx). In the final step, the parent is changed to copy segment from. The first point becomes the second, the new second point will be found.

10.6. Mutation

Mutation is a genetic operator used to vary robots' neural networks from one population to another. It is analogous to biological mutation, which randomly change some genes. Using the algorithm (Roulette wheel selection) two robot-parents are selected, and it is possible to create a robot-child with the help of their chromosomes. Crossover cannot always guarantee high diversity. Mutation makes huge changes in the chromosome, which can quickly lead to a global maximum of the fitness function.



Picture 35 Mutation

This operator is characterized by the probability and the range of mutation. In the picture (Picture 32) the probability of each gene's mutation is 30%. The range of mutation varies (from -0.7 to 0.7). These values can be set in the settings of learning (Probability) and (Range), respectively.

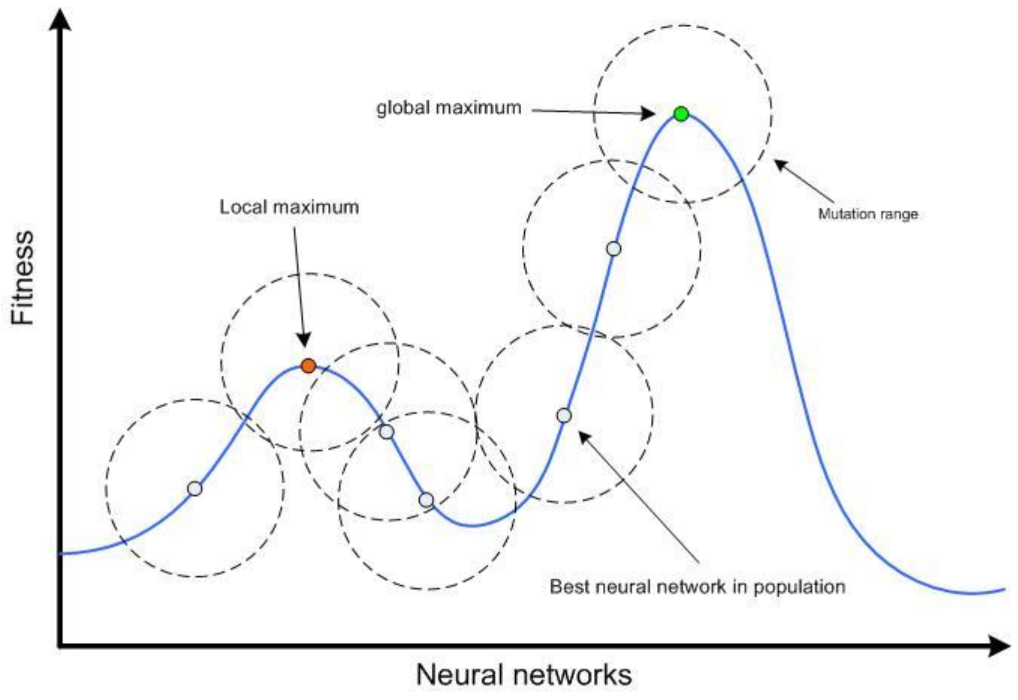
Mutation algorithm

```
Chromosome mutate() {
    AdaptiveNeuralNetwork cloned = clone();
    for (int i = 0; i < links.size(); i++) {
        if (MathUtil.random(0, 100) <= mutationProbability) {
            Link link = links.get(i);
            double mutated = link.getWeight()
            + MathUtil.random(mutateRangeMin, mutateRangeMax);
            link.setWeight(MathUtil.toRange(mutated, -1, 1));
        }
    }
return cloned;
```

The function clones all the genes (weights) of the chromosome. Next in a loop the probability of mutation is calculated for each gene. If the probability is higher than the target value in the settings (mutationProbability), then the weight is changed by a random value from the range (mutateRangeMin, mutateRangeMax). Finally the function returns the mutated chromosome.

10.7. Fitness landscape

Fitness landscape (another name is adaptive landscape) - is a graph where the axis X - all possible network's structures, the axis Y is a fitness value. The graph is used to visualize the dependence between all possible genotypes (networks with different structures) and the values of fitness. Very similar genotypes are close on the chart, while different genotypes are far away from each other.



Picture 36 Fitness landscape. Local and global maximum

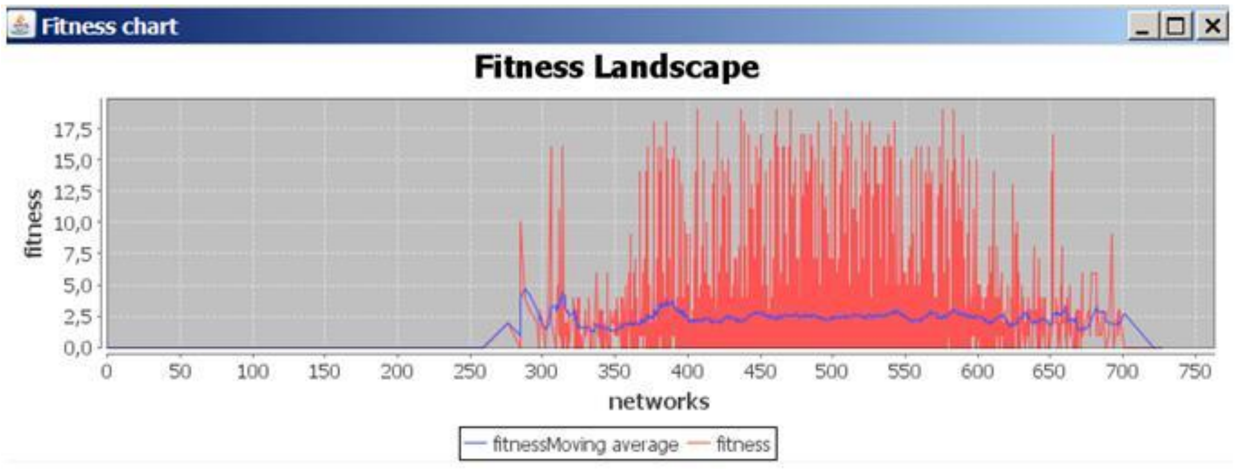
It is very important to analyze this graph because it is very important to avoid local minimum problem. During the evolution of the population robot's network with the best fitness is located on the plot. If the robot is at the foot of a local maximum and the mutation rate is small, then the genetic algorithm will find a neural network located on the top of a small hill, next mutations will generate similar networks, the best of which will still be at the local maximum. Algorithm finishes and returns the best neural network from local maximum. In order to avoid it, the range of mutation should be large enough to create a network that would be above the local maximum. Only in this case, the genetic algorithm will find the best of all possible networks.

In order to distinguish the networks with different structures and interpret it in a unique numeric value, the following algorithm was used

```
double getHash() {
     List<Link> links = getLinks();
     double hash = 0;
     for (int i = 0; i < links.size(); i++) {
          hash += i * (links.get(i).getWeight() + 1);
     }
     return hash;
}
```

This function takes all the weights of the neural network and multiplies it by weight's serial number in a loop. By the way it is possible to identify the location of a neural network in the graph by obtained unique numeric hash.

In order to display the fitness landscape 5000 populations were created, where robot-leaders were selected from and shown on the graph.



Picture 37 Fitness landscape

In the picture (Picture 37) the fitness landscape is shown for 5000 neural networks with different structure. Red line shows the fitness value for each neural network. Blue graph is a moving average. It is possible to notice that the landscape is very noisy. Moving average is almost straight. It says that the similar neural networks on the graph can give different fitness values. According to this graph it is possible to find a neural network with sufficient good fitness value even with a very small range of mutation and creating small number of populations.

11. Testing solutions

In the scope of this work there were three solutions for the robot's behavior for pickings balls.

1. Tree of rules - the tree contains the rules by which the robot will collect balls.
2. Forward Neural Network - neural network with no memory, only reacting to current events.
3. Recurrent Neural Network - neural network with memory that can also take into account the events from the previous step.

In order to compare these solutions it is necessary to make tests with various conditions of environment to get average fitness value. There are 4 types of tests for each of the solutions (12 tests in total), and the most effective solution will be chosen based on the results.

11.1. Types of tests

1. There is an environment with a large number of robots and a small number of balls (20 robots and 10 balls). The test should show how robots behave in a high competition.
2. There is an environment with a small number of robots and a small number of balls (5 robots and 10 balls). The test should show how robots can effectively seek balls scattered at great distances.
3. There is an environment with a large number of robots and a large number of balls (20 robots and 100 balls). The test should show the behavior of robots in normal environmental

conditions when balls and robots are almost simultaneously in robot’s radar.

4. There is an environment with a small number of robots and a large number of balls (5 robots and 100 balls). The test should show how robots will react to large accumulations of balls, and how effectively they will collect balls.

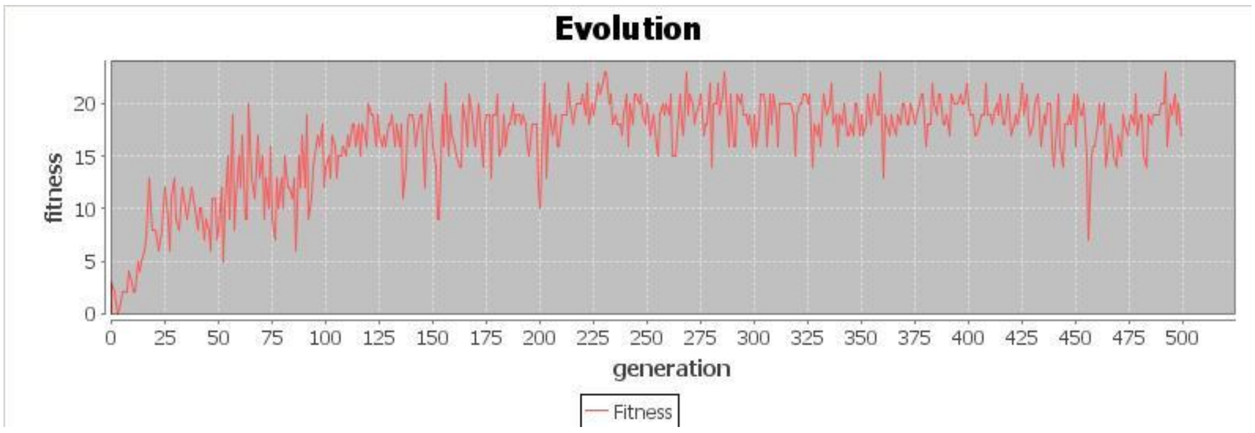
11.2. Conditions for tests

1. The emulation will be run 500 times for each of solution.
2. 500 generations will be used to train neural networks, where the best one will be selected for participating in tests.
3. Emulation’s duration is 300 ticks.
4. In order to train neural networks balls are spaced constantly in the field.
5. Balls are scattered randomly in the field for testing.
6. During training the number of balls will be restored.
7. Robots will appear in the center of the field, the initial rotation angle is 0 degrees.
8. Forward neural network must have three layers with 6, 4 and 2 neurons respectively.
9. Recurrent neural network must have three layers of 12, 4 and 2 neurons respectively.
10. The following settings will be used for learning the neural networks
    1. Forward neural network: 10
    2. Recurrent neural network: 10
    3. Balls: 100
    4. Fitness: 0.0
    5. Number of generations: 5000
    6. Duration: 300 ticks
    7. Population to change: 90.0%
    8. Crossover type: ALL
    9. Mutation probability: 90%
    10. Mutation range: -0.6 – 0.6
11. Field width is 800px and height is 600px
12. the following robot’s settings will be used for each test
    1. maximum speed - 4px/tick
    2. Vision range – 160px
    3. Vision angle – 160 degrees
    4. distance to pick a ball – 15px

11.3. Neural networks’ training charts

In order to test the neural network, they must be trained.

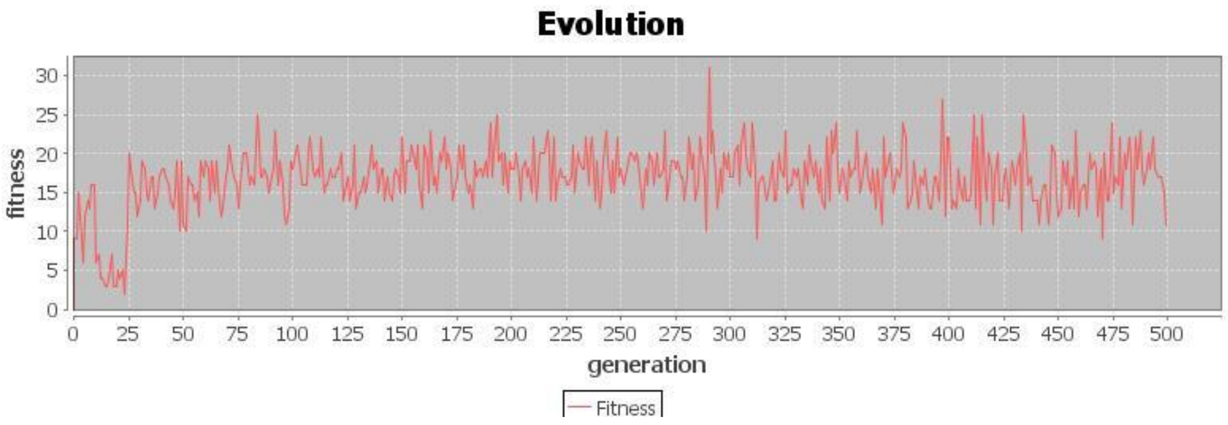
Forward neural network’s training chart



Picture 38 Forward neural network evolution in 500 generations.

Forward neural network is trained for 500 generations (Picture 39) Network with the best configuration was able to collect 24 balls for 300 ticks.

Recurrent neural network's training chart



Picture 39 Recurrent neural network evolution in 500 generations.

Recurrent neural network is trained for 500 generations (Picture 39). Network with the best configuration was able to collect 32 balls for 300 ticks.

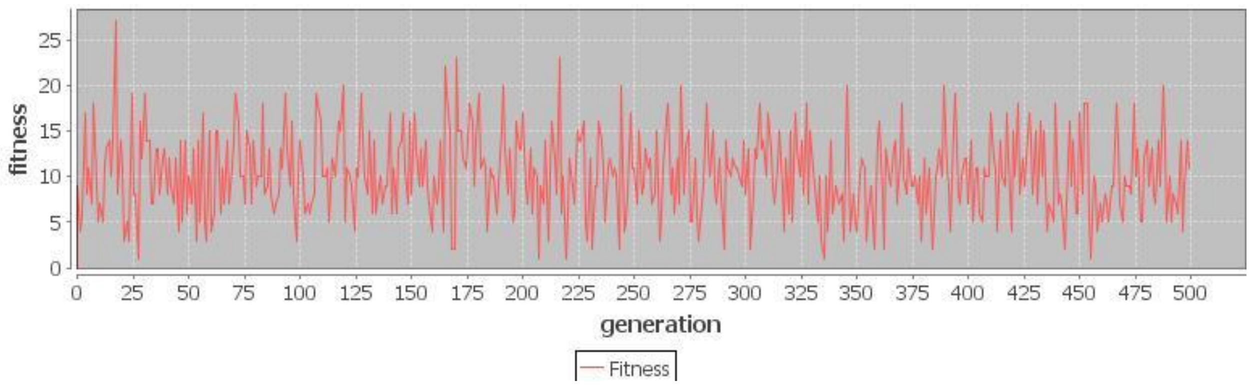
11.4. Tests of Tree of rules

Test 1

Test parameters

- Number of balls: 100
- Number of robots: 20

Results



Picture 40 Testing chart for Tree of rules with 100 balls and 20 robots

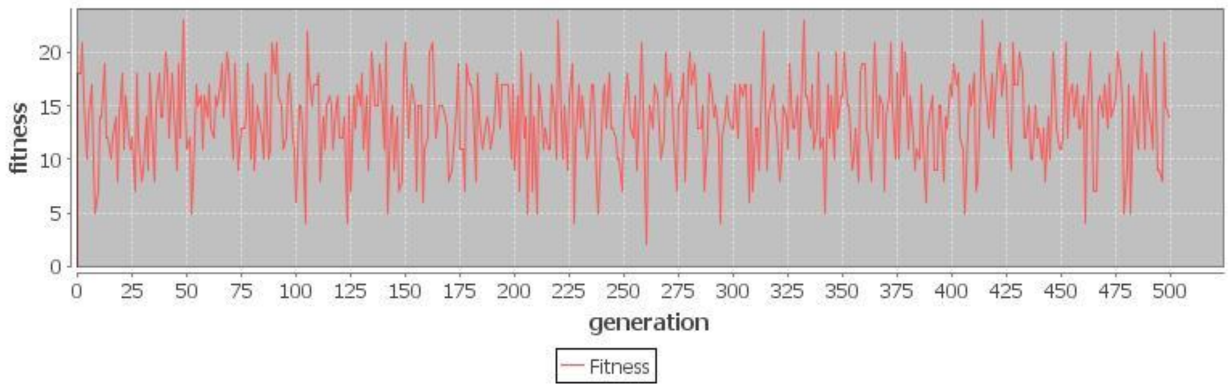
Average fitness: 10.282

Test 2

Test parameters

- Number of balls: 100
- Number of robots: 5

Results



Picture 41 Testing chart for Tree of rules with 100 balls and 5 robots

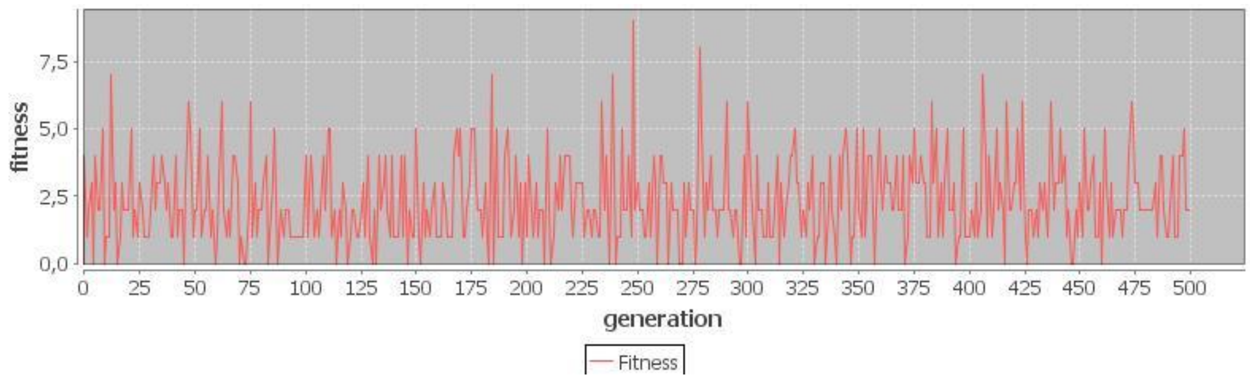
Average fitness: 13.868

Test 3

Test parameters

- Number of balls: 10
- Number of robots: 20

Results



Picture 42 Testing chart for Tree of rules with 10 balls and 20 robots

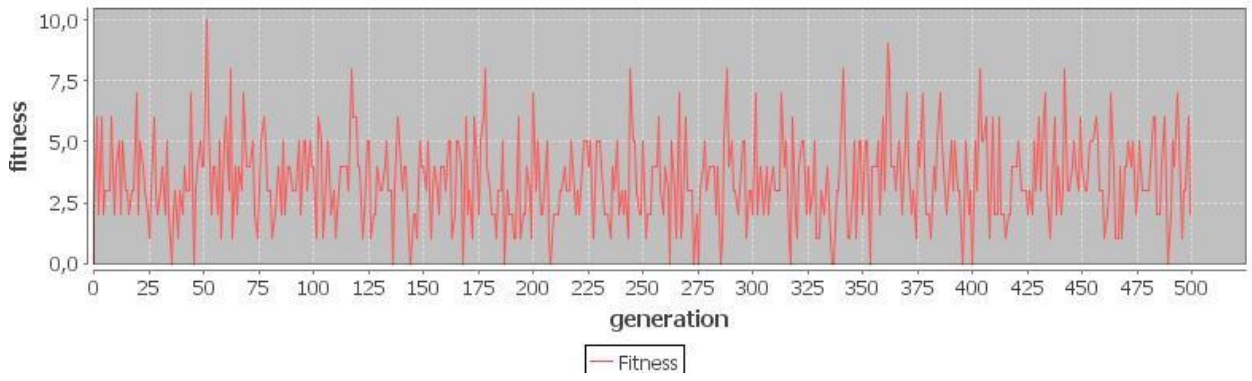
Average fitness: 2.322

Test 4

Test parameters

- Number of balls: 10
- Number of robots: 5

Results



Picture 43 Testing chart for Tree of rules with 10 balls and 5 robots

Average fitness: 3.428

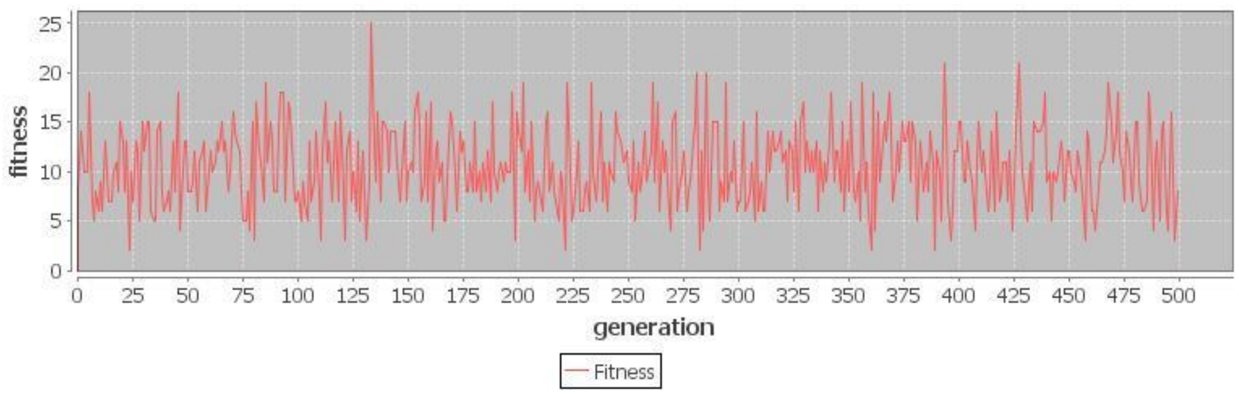
11.5. Tests of Forward neural network

Test 5

Test parameters

- Number of balls: 100
- Number of robots: 20

Results



Picture 44 Testing chart for forward neural network with 100 balls and 20 robots.

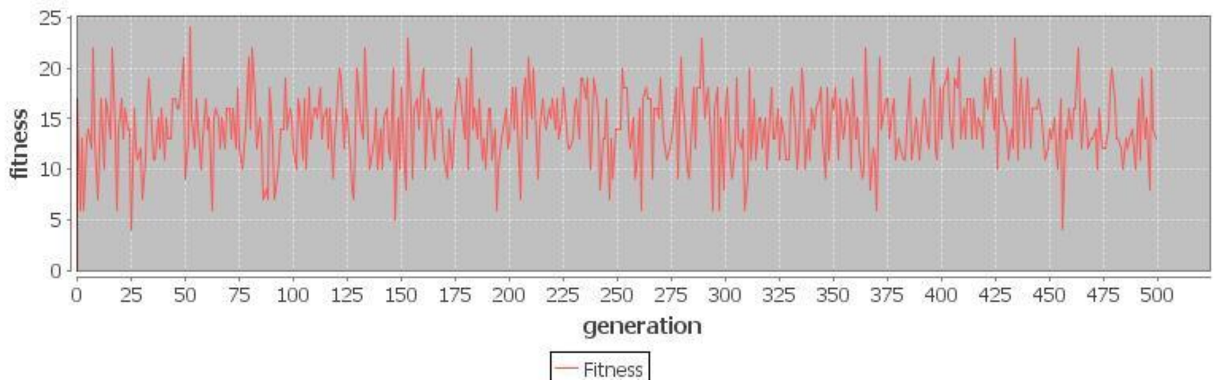
Average fitness: 10.54

Test 6

Test parameters

- Number of balls: 100
- Number of robots: 5

Results



Picture 45 Testing chart for forward neural network with 100 balls and 5 robots.

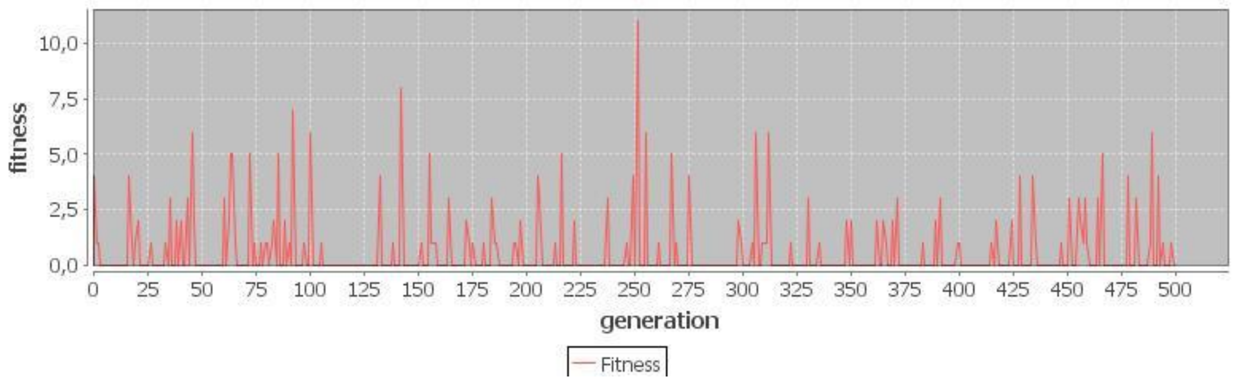
Average fitness: 14.162

Test 7

Test parameters

- Number of balls: 10
- Number of robots: 20

Results



Picture 46 Testing chart for forward neural network with 10 balls and 20 robots.

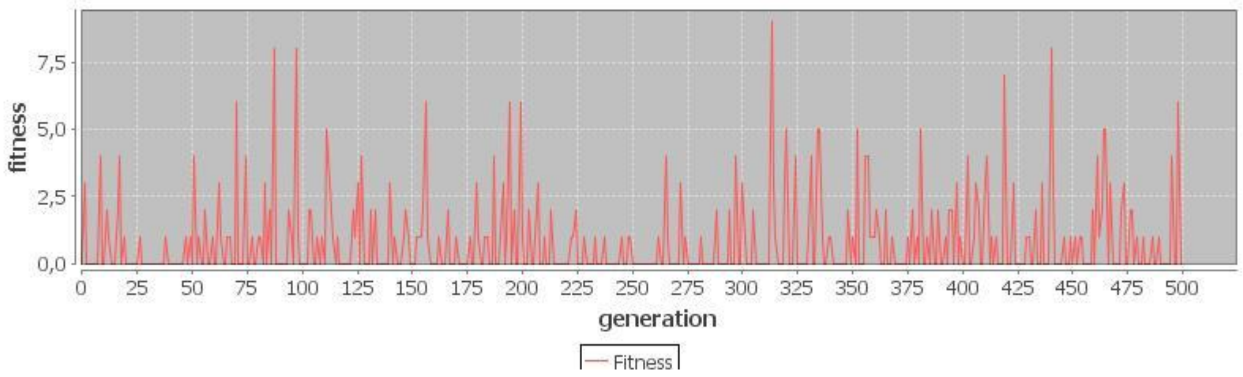
Average fitness: 0.554

Test 8

Test parameters

- Number of balls: 10
- Number of robots: 5

Results



Picture 47 Testing chart for forward neural network with 10 balls and 5 robots.

Average fitness: 0.786

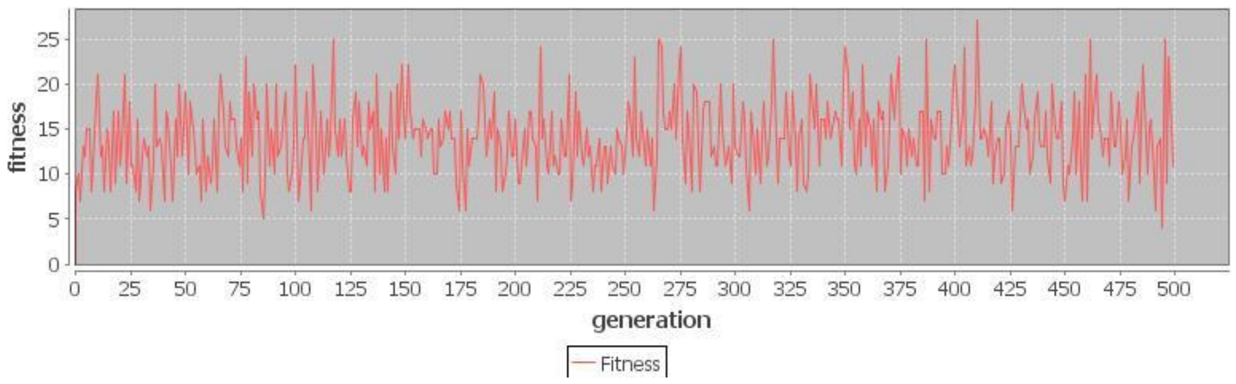
11.6. Tests of Recurrent neural network

Test 9

Test parameters

- Number of balls: 100
- Number of robots: 20

Results



Picture 48 Testing chart for recurrent neural network with 100 balls and 20 robots.

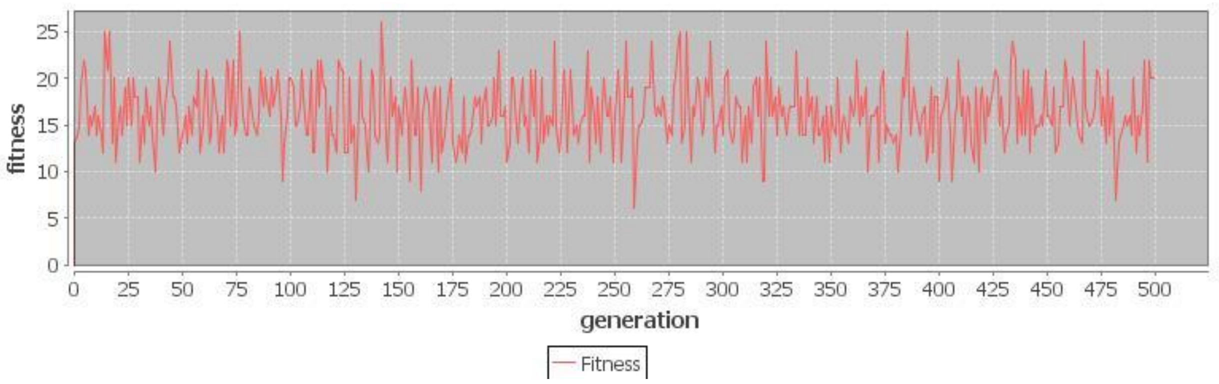
Average fitness: 13.972

Test 10

Test parameters

- Number of balls: 100
- Number of robots: 5

Results



Picture 49 Testing chart for recurrent neural network with 100 balls and 5 robots.

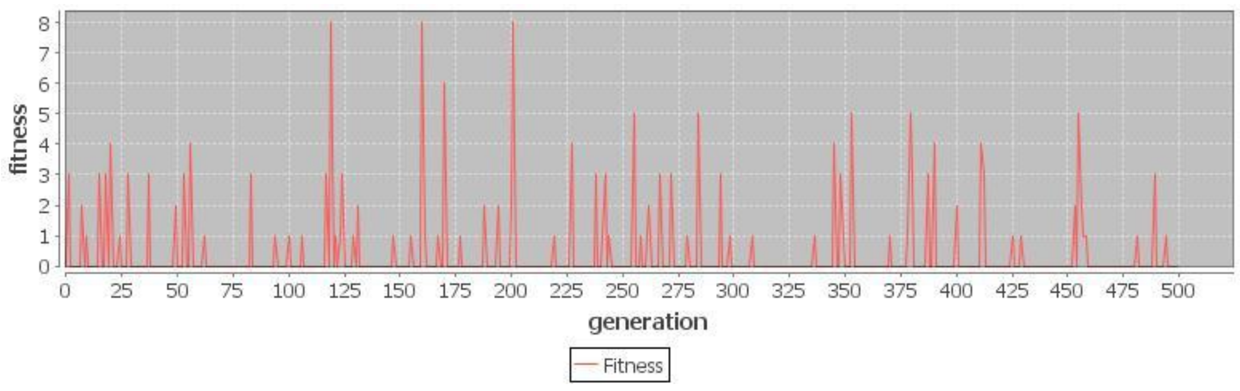
Average fitness: 16.336

Test 11

Test parameters

- Number of balls: 10
- Number of robots: 20

Results



Picture 50 Testing chart for recurrent neural network with 10 balls and 20 robots.

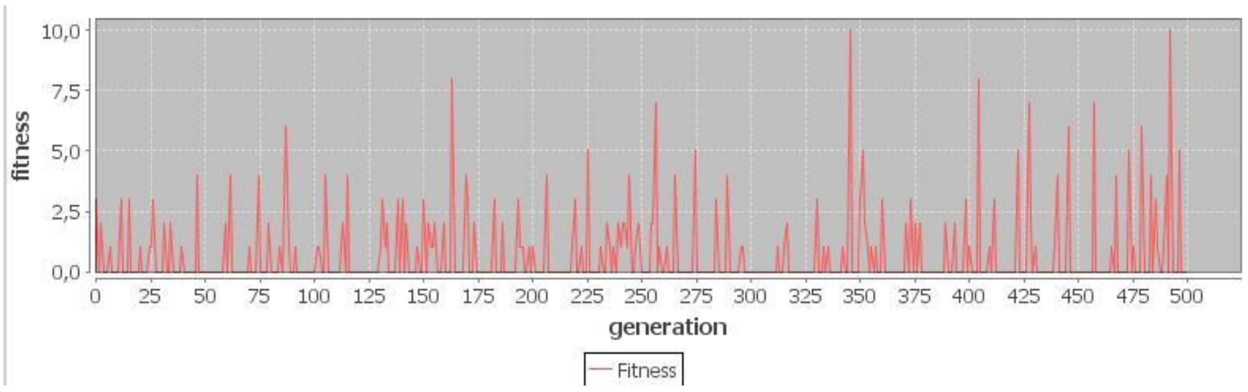
Average fitness: 0.376

Test 12

Test parameters

- Number of balls: 10
- Number of robots: 5

Results



Picture 51 Testing chart for recurrent neural network with 10 balls and 5 robots.

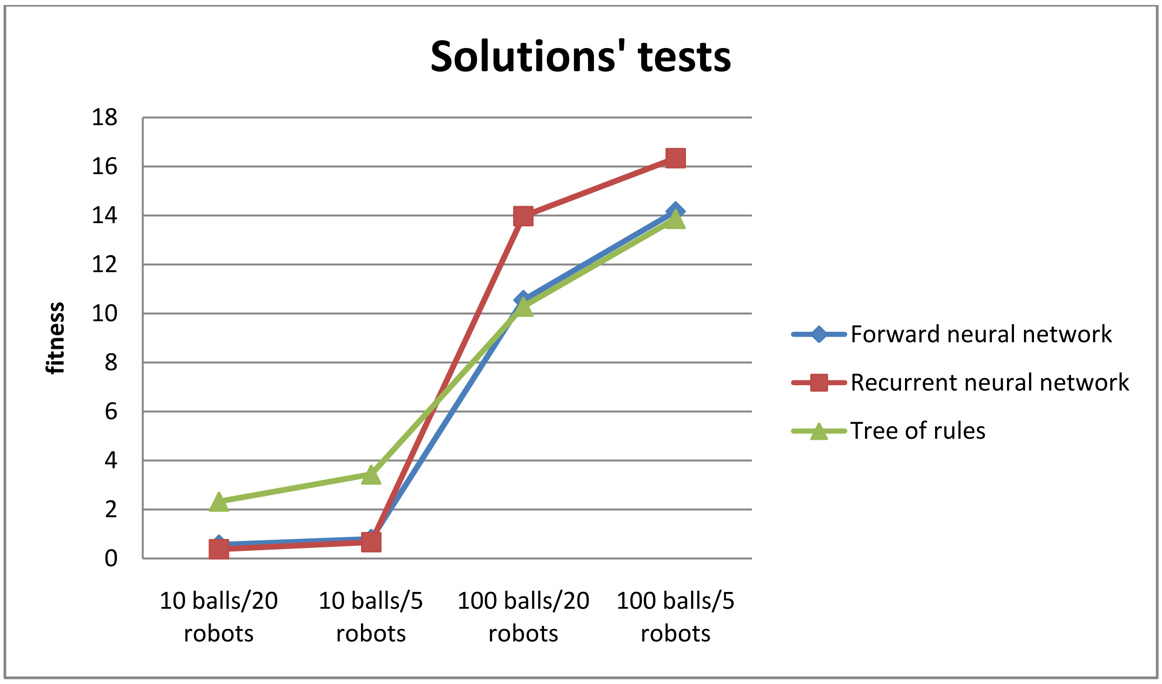
Average fitness: 0.66

11.7. Test results

Test results table

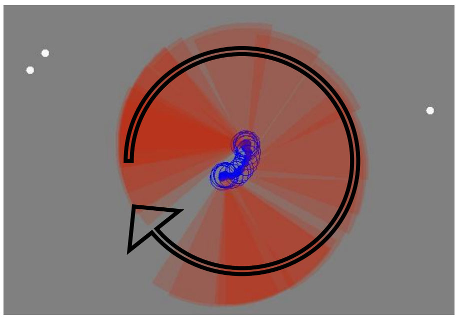
Balls	Robots	Forward neural network	Recurrent neural network	Tree of rules
10	20	0.554	0.376	2.322
10	5	0.786	0.66	3.428
100	20	10.54	13.972	10.282
100	5	14.162	16.336	13.868

Graph of test results



12. Analysis and selection of the best solution

After testing it is possible to see that both neural networks showed almost identical results while emulation with few balls. In both tests with 10 balls robot with trained forward neural network collected average 0.554 and 0.786 balls per 300 ticks. Robot with trained recurrent neural network collected 0.376 and 0.66 balls per 300 ticks. In this case, simple description logic tree of rules is much better than the neural networks. A robot with behavior based on tree of rules collected 2,322 and 3,428 balls. Analysis showed that the neural networks cannot search the balls and move in a random direction when there are no balls in the radar.



Picture 52 Inefficient behavior for searching balls

They rotate at a single location and scan a certain area (Picture 52). The reason for this behavior is that forward and recurrent neural networks' inputs omit zero, i.e. inputs: balls' count, the nearest ball's distance, the nearest ball's delta angle, robot' count, the nearest robot's distance, the nearest robot's delta angle inputs are 0. Thus, while the calculating the polynomial function for output delta angle, all the weights are multiplied by zero inputs, as a result the output is 0 too.

Polynomial function for output **Delta angle**

sigma(

$w_{21,32}$*sigma(0*$w_{11,21}$ + 0*$w_{12,21}$ + 0*$w_{13,21}$ + 0*$w_{14,21}$ + 0*$w_{15,21}$ + 0*$w_{16,21}$) +

$w_{22,32}$*sigma (0*$w_{11,22}$ + 0*$w_{12,22}$ + 0*$w_{13,22}$ + 0*$w_{14,22}$ + 0*$w_{15,22}$ + 0*$w_{16,22}$) +

$w_{23,32}$*sigma (0*$w_{11,23}$ + 0*$w_{12,23}$ + 0*$w_{13,23}$ + 0*$w_{14,23}$ + 0*$w_{15,23}$ + 0*$w_{16,23}$) +

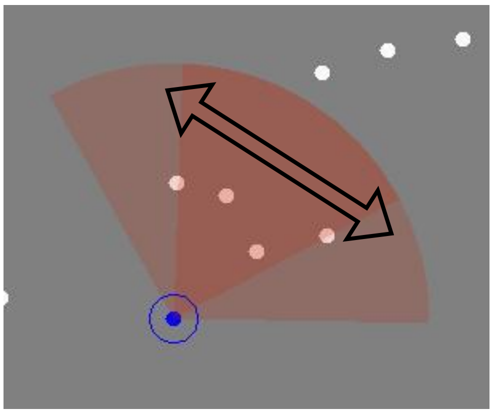
$w_{24,32}$*sigma (0*$w_{11,24}$ + 0*$w_{12,24}$ + 0*$w_{13,24}$ + 0*$w_{14,24}$ + 0*$w_{15,24}$ + 0*$w_{16,24}$)

) = sigma($w_{21,32}$*sigma(0) + $w_{22,32}$*sigma (0) + $w_{23,32}$*sigma (0) + $w_{24,32}$*sigma (0)) = sigma(0) = 0

In this case, the direction's angle of the robot cannot be changed. The robot's speed has the same result, so speed cannot be also changed.

In case of Tree of rules robot moves with constant velocity in a random direction to search the ball, so it can scan bigger area and therefore collect more balls.

However Recurrent neural network showed excellent fitness for tests with a lot of balls. In tests with 100 balls, robot with behavior based on this network has collected an average of 13,972 and 16,336 balls for 300 ticks, which is far superior comparing to the results of the other two solutions. Analysis showed that feedback made some randomness in the behavior of the robot.



Picture 53 Efficient behavior for search balls

When robot moves, it constantly rotates from side to side (Picture 53) and therefore covers a large area, and is able to scan and collect large piles of balls at once.

Thus, it is possible to conclude that these neural networks are not ideal. It is necessary to find other ways of presenting information or restructure the network. Neural networks are not able to look for balls on the empty field that described simple logic (tree of rules) can do perfectly. However, genetic algorithm found good configuration for recurrent neural network, which has surpassed the results of the forward neural network and tree of rules. Therefore, in the scope of the work, I would suggest using recurrent neural network to solve the problem. Perhaps it is necessary to change the values of the inputs or replace zero values with negative numbers. In favor of the neural network the fact exists that the genetic algorithm can always find the most versatile solution for a clearly defined problem, even such solution that is very difficult or almost impossible to describe by simple logic.

13. Summary

In the scope of this work the most effective solutions were implemented for finding golf balls on open area of the field using the following programming approaches, like a simple description logic (Tree of Rules), the use of Forward neural network and Recurrent neural network. Also the neural networks were trained using genetic algorithm. The environment was also created in Java in order to simulate the proposed solutions to compare behaviors, analyze the effectiveness of each behavior. A series of tests were passed that showed the effectiveness, suitability of each solution for different environmental conditions. According to tests' results the advantages and disadvantages of neural networks were detected comparing to the simple description logic. In this work recurrent neural network was chosen from the proposed solutions as the most effective solution of the problem (collecting balls) because in case of very difficult environmental conditions it is not always possible to write an effective logic that would have been better than trained neural network.

I would like to develop the direction of neural networks in the future to solve this problem of finding balls on the golf fields. Also I would like to train neural network to recognize the pattern of balls in the grass and detect the obstacles (trees and ponds).

14. Kokkuvõte

Selles töös kõige tõhusamad lahendused on rakendatud golfpallide leidmiseks avatud piirkonnas, kasutades järgmist programmeerimist lahendust, nagu kirjeldatud loogika (Tree of rules), Forward ja Recurrent närvivõrkude kasutamine. Mõlemad närvivõrgud olid õpetatud geneetilise algoritmi abil. Keskkond oli loodud Javaga selleks, et simuleerida pakutud lahendusi, võrrelda käitumist ja analüüsida iga käitumise effektiivsust. Testid olid tehtud, mis näitasid iga käitumise eelised ja puudused erinevates keskkonna tingimustes. Testide tulemustest närvivõrkude plussid ja minused olid tuvastatud võrreldes kirjeldatud loogikaga. Selles töös Recurrent närvivõrk on valitud nagu kõige tõhusam lahendus antud probleemi lahendamiseks (pallide kogumiseks), sest ei ole võimalik või on väga raske kirjutada effektiivsust loogikat keeruliste keskkonnatingimuste juhul, et see lahendus oleks parem kui närvivõrgu.
Tulevikus tahaksin arendada selles suunas närvivõrku golfpallide leidmiseks piirkondades. Samuti tahaksin treenida närvivõrku tunnustada palli mustrit muru sees ja avastada takistusi piirkondades (puud ja tiigid).

15. Used materials

1. Artificial intelligence. [WWW] http://en.wikipedia.org/wiki/Artificial_intelligence.
2. Artificial neural network [WWW] http://en.wikipedia.org/wiki/Artificial_neural_network
3. Нейронные сети [WWW] http://www.aiportal.ru/articles/neural-networks/neural-networks.html
4. Модель нейрона [WWW] http://www.aiportal.ru/articles/neural-networks/model-neuron.html
5. Многослойные нейронные сети [WWW] http://www.aiportal.ru/articles/neural-networks/multilayer-networks.html
6. Классификация нейронных сетей [WWW] http://www.aiportal.ru/articles/neural-networks/classification.html

7. Обучение нейронной сети [WWW] http://www.aiportal.ru/articles/neural-networks/learning-neunet.html
8. Intelligent agent [WWW] http://en.wikipedia.org/wiki/Intelligent_agent
9. Recurrent neural network [WWW] http://en.wikipedia.org/wiki/Recurrent_neural_network
10. Herbert Jaeger Fraunhofer Institute for Autonomous Intelligent Systems (AIS) since 2003: International University Breme: A tutorial on trainingre current neuralnet works ,covering BPPT, RTRL ,EKF and the" echo state network" approach [WWW] http://minds.jacobs-university.de/sites/default/files/uploads/papers/ESNTutorialRev.pdf
11. Feedforward neural network [WWW] http://en.wikipedia.org/wiki/Feedforward_neural_networks
12. Simon Haykin: Feedforward neural network: An introduction [WWW] http://media.wiley.com/product_data/excerpt/19/04713491/0471349119.pdf
13. Prof. Leslie Smith Centre for Cognitive and Computational Neuroscience Department of Computing and Mathematics University of Stirling: An Introduction to Neural Networks. [WWW] http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html#algs
14. Decision tree [WWW] http://en.wikipedia.org/wiki/Decision_tree
15. Деревья принятия решений: [WWW] http://logic.pdmi.ras.ru/~sergey/teaching/ml/notes-01-dectrees.pdf
16. Neural Networks [WWW] http://www.emilstefanov.net/Projects/NeuralNetworks.aspx
17. Genetic algorithm [WWW] http://en.wikipedia.org/wiki/Genetic_algorithm
18. Mutation (genetic algorithm) [WWW] http://en.wikipedia.org/wiki/Mutation_%28genetic_algorithm%29
19. Selection (genetic algorithm) [WWW] http://en.wikipedia.org/wiki/Selection_%28genetic_algorithm%29
20. Fitness proportionate selection [WWW] http://en.wikipedia.org/wiki/Fitness_proportionate_selection
21. Генетический алгоритм. Просто о сложном [WWW] http://habrahabr.ru/post/128704/
22. Stuart J. Russell, Peter Norvig: Artificial Intelligence: A Modern Approach, 1995 [WWW] http://www.amazon.com/Artificial-Intelligence-A-Modern-Approach/dp/0131038052

16. Notes