Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3027

Design and Implementation of a Framework for Online Evolution of Robotic Behaviour

Michael Mattes

Course of Study:	Software Engineering
Examiner:	Prof. Dr. rer. nat. habil. Paul Levi
Supervisor:	Florian Schlachter
Commenced:	15.12.2009
Completed:	15.06.2009
CR-Classification:	D.2.2

The framework is written in C++ as a plug-in to the existing symbricator simulation environment that is an extension of delta3D simulation engine. The simulation environment provides models of existing robots complete with sensors and actuators, a mechanism to add controllers to the robots and arenas to deploy them to.
The robot controllers use the sensor inputs to compute activation values for the actuators. The computation of the activation values for each robot is done by artificial neural networks. Generating those neural networks is the main purpose of the framework. Generally they are created by an evolutionary algorithms and are evolved over a period of time to show the desired robotic behavior.
The graphviz C API is used to save resulting nets to disk, to visualize them or to load existing nets from disk to use them as a starting point for evolution or to observe the behavior of robots controlled by them. One of the main problems using evolutionary algorithms is evaluating the behavior shown by the robotic. This is complicated further by the principle of online evolution, which means only data available to the robot can be used for evaluation. Fitness functions assign a score to the behavior shown by the robot which is used in the process of evolution. To alleviate the problems of evaluating robotic behavior a set of fitness functions suitable for standard scenarios like wall following and food gathering will be provided by the framework. They can either be used to test and compare new evolutionary algorithms or as an example and starting point for customized fitness functions.
After finishing evolution within the simulation environment, the controller networks must be transferred to real robots where they must adapt themselves to the changes in their environment. To help users with this task all necessary framework classes like classes for neural networks, evolutionary algorithms as well as robot controllers are written in such a way that permits using them within the simulation environment as well as within real hardware.

Contents

List of Figures

List of Tables

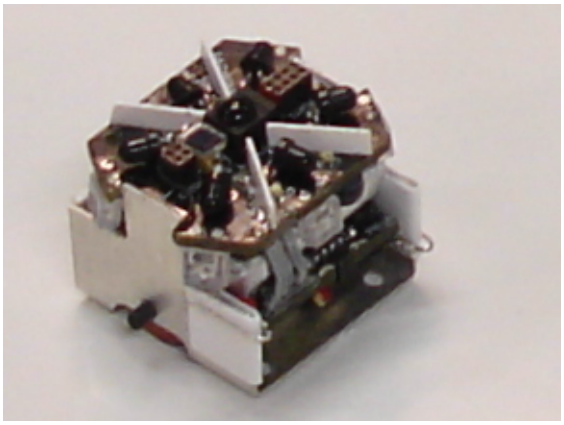
1. Introduction

1.1. Symbrion

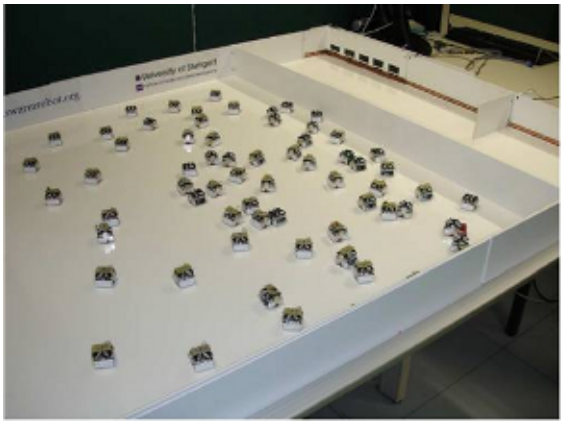
This thesis was undertaken in the symbrion project[sym10], a project sponsored by the Europian Union. Among the goals are research of super-large-scale swarms fulfilling tasks, self-assembling into symbiotic multi-robot organisms as required by fulfillment of tasks. The robots shall adapt to changing environments and be autonomous of human intervention. Short-term and long-term evolution is both a key to these aims as it is a subject of research itself.
Figure 1.1 shows robots in the different stages of organization beginning with a single robot, robots as a large swarm and as a multi-robot organism at last.

1.2. Structure of this Document

In the second chapter we start with some means to control robots, namely Decision Trees, Finite State Machines and Artificial Neural Networks. In the next chapter the three paradigms of machine learning and samples learning methods for each paradigm are presented. Chapter 4 then introduces Evolutionary Algorithms in general and the three algorithms (NEAT, GNARL and EANT) implemented by the framework. Chapter 5 starts with an overview over Delta3D, the simulation engine used, then continues with the simulation and finally presents the implemented framework. Chapter 6 contains some experimental results and Chapter 7 some conclusions and an outlook to future work.



(a) single robot



(b) swarm of robots

(c) multi-robot organism

(d) image from simulation

Figure 1.1.: Images show robots used in the symbrion project in different stages of organization. From a) single robot, to b) swarm and finally c) multi-robot organism. The last picture d) shows a simulation of those robots. Images taken from [sym10]

2. Robot control

For robots to fulfill any task he needs some way to make decisions based on his situtaion. Sensors are used to perceive the situation and to obtain data decisions are based on. This chapter shows some ways a robot can make the necessary decisions.

2.1. Decision Trees

Decision Trees[RN03] are a simple machine learning algorithm for making decisions. Their main advantages are that humans can easily understand how those decisions are reached and that they are easy to implement.
Input to decision trees is a *situation* described by several *attributes*, either discrete or continuous. Their output is also either discrete or continouous. Learning a discrete output is called *classification* and learning a continuous output is called *regression*.
Decision Trees reach a decision by testing an attribute at a node starting at the root node and then following an edge standing for the result of the test. Leaf nodes contain the output of the tree.

2.1.1. Example

In this example one has to decide whether to wait for a table in a restaurant[1] . Attributes describing the situation are

- Alternate - alernatives nearby? (yes/no)
- Bar - bar area in restaurant? (yes/no)
- Fri/Sat - is it friday or saturday? (yes/no)
- Hungry - are we hungry? (yes/no)
- Patrons - other guests? (none/some/full)
- Price - price class (cheap, moderate, expensive)
- Raining - is it raining? (yes/no)
- Reservation - do we have a reservation? (yes/no)

[1] Example taken from [RN03].

- Type - type of restaurant? (chinese/french/italian/korean)
- WaitEstimate - how long do we have to wait? (<10/10-30/30-60/>60)

Note that some attributes (price, type) are not necessary to reach a decision. Figure 2.1 shows the decision tree for this example.

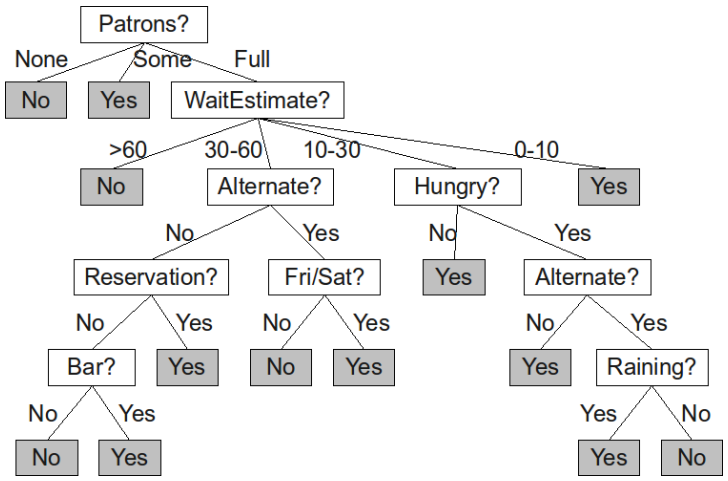


Figure 2.1.: Decision Tree for restaurant example. Taken from [RN03].

2.1.2. Constructing Decision Trees

Decision Trees can be induced by a recursive divide and conquer algorithm using a set of training data[CCF+08]. Each entry in the set consists of attribute values and a class associated to the entry.
An attribute is selected as the root node by the algorithm and for every value that attribute can take a branch is made. Every branch divides the training set into subsets.
Down every branch another attribute is selected for the node and a branch is made if the remaining entries in this subset belong to more than one class. This is repeated until all leaf nodes of the tree unambiguously identify a single class.
One selection criteria that can be used is *information gain*. Information gain is calculated using the *information value* or *entropy* of attributes:

$$entropy(p_1, p_2..., p_n) = -p_1 \log p_1 - p_2 \log p_2... - p_n \log p_n \tag{2.1}$$

where $p_1, p_2..., p_n$ are the fractions of each entry down a branch to the total of entries of this attribute. The logarithm is expressed in base 2. Information gain (for attribute X_n) is then:

$$gain(X_n) = entropy(X_n) - \sum_i entropy(X_i) \tag{2.2}$$

where $\sum_i entropy(X_i)$ is the sum of the entropy of all child nodes of X_n. Several enhancement to this criteria exist including using *gain ratio* instead of information gain, methods for dealing with noise, missing data, numerical attributes and many more.

2.1.3. Use of Regression Trees in Artifical Intelligence

A common problem in AI is navigation of a robot through difficult terrain affecting movement. Regression Trees can be used to predict the effect of terrain on navigation actions[DGF00].
A robot is tasked to travel from Point A to Point B over a given terrain as fast as possible. He has to plan his movement and therefore has to know how the terrain affects his speed and his turning rate. Figure 2.2 shows an example of a terrain. The

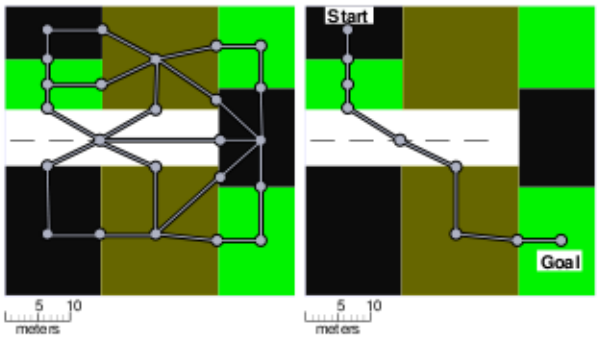


Figure 2.2.: Terrain for navigation challenge. Darker colours denote terrain rougher terrain. On the left are all possible paths and on the right is the path taken. Image taken from [DGF00].

approach used consists of three components: Exploration, Regression tree induction and Action models. During exploration experience is gathered how different terrain affects movement. This is used during regression to induce the trees for every type of action (e.g. moving forward, turing left or turning right). The action models are then generated by querying the trees for the mean error of each action on the different terrain classes. Figure 2.3 shows a tree outputing error for moving forwards.

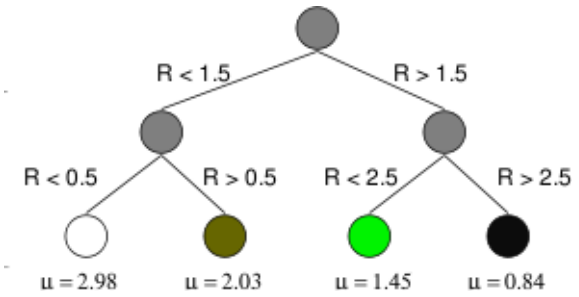


Figure 2.3.: Decision Tree induced for moving forward. μ is the predicted error for that action and R the level of roughness ranging from 0 to 3. Image taken from [DGF00].

2.2. Finite State Machine

Finite State Machines (FSM) are a common tool for artificial intelligence in computer games and robotics. The basic model out of theoretical computer sciences is a determinstc FSM used for accepting languages. For controlling agents or robots accepting a language is not enough, therefore *Moore machines* or *Mealy machines* are used, both introducing output functions.
The formal definition[Bro] of both types of machines is very similar, the difference is in the output function: in Moore machines it is associated with states and in Mealy machines with transisitions.

Definition 1 *A machine M is defined by a sextuple*

$$M = (S, s_0, \Sigma, \Delta, \delta, \lambda),$$

consisting of

S: *finite, non-empty set of states*

s_0: *start state* ($s_0 \in S$)

Σ: *finite input alphabet*

Δ: *finite output alphabet*

δ: *transition function* ($\delta : S \times \Sigma \to S$)

λ: *output function (Moore* $\lambda : S \to \Delta$, *Mealy* $\lambda : S \times \Sigma \to \Delta$)

Tools like Matlab Simulink[2] allow mixing both types of machines and in addition events like state entering or leaving are defined and actions can be associated with

[2]http://www.mathworks.com/products/stateflow/

those events.
FSMs are often designed by hand, defining all states and transitions. But there also exist methods using reinforcement learning or evolutioniary algorithms. There are two ways FSMs get their input data in control tasks. Either by polling the FSM's data sources (sensors) or event-driven, pushing event data to the FSM.

2.2.1. Example

A swarm of mini robots explores a turbine avoiding obstacles (walls and other robots) searching for blades and then inspecting found blades[CM06]. A robot is only allowed to leave a blade at its tip. Figure 2.5 shows a FSM for the robots high the and figure 2.4 shows the environment of the robots, simulated as well as real.
Optionally, robots can serve as a beacon for other robots by remaining at the tip of a blade for some time. Another robot seeing a robot at the tip of a blade knows that blade was already inspected and can avoid it, thus saving time.

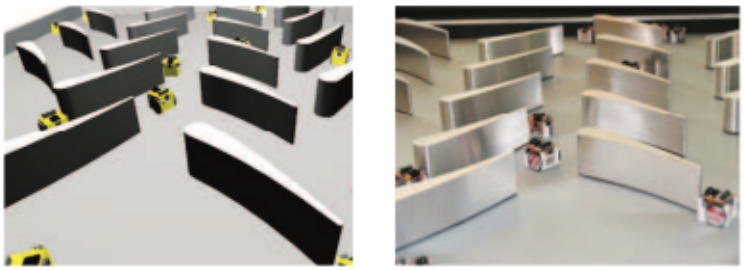


Figure 2.4.: Left: turbine set-up in simulator. Right: real turbine set-up. Image taken from [CM06].

2.3. Artificial Neural Networks

Artificial Neural Networks (ANNs) are a simplified model of human or animal brains. Units called *neurons* are interconnected to compute one or more output values out of several input values. Connection strengths or *weights* as they are called are the most important parameters of an ANN.

2.3.1. Perceptron

Perceptrons[NF00] (figure 2.7) are the simplest of all ANNs first presented by McCulloch and Pits in 1943. They constist only of a single neuron and its inputs.
A Neuron has a set of input x_i, associated weights w_i, a function Σ summating

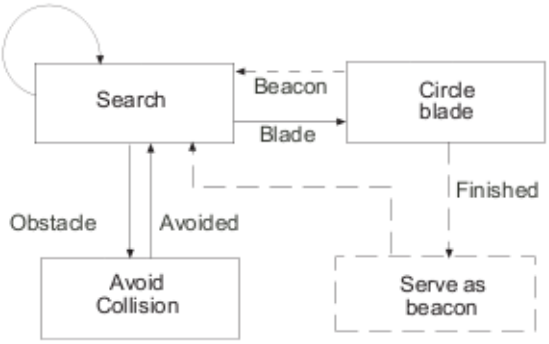


Figure 2.5.: FSM controlling high-level behaviour of robots. Image taken from [CM06].

the weighted inputs, a threshold value θ and an activation function Φ, usually a sigmoid function, computing the output y of the neuron. The output is computed using equation (2.3).

$$net = \sum_{i=1}^{n} w_i x_i$$
$$y = \Phi(net - \theta) \tag{2.3}$$

Activation Functions

Figure 2.6 shows some commonly used activation functions for neurons. Of those functions the logistic function is used most often because its derivate is easily calculated (2.4). The derivate is used in numerous learning functions (see chapter 3).

$$y' = k * y * (1 - y) \tag{2.4}$$

From now on, the logistic function will be used as activation function for all neurons.

Problem with Threshold Values

A problem with threshold values is that they cannot be trained by learning algorithms. Therefore an input unit with a constant input of 1, called bias unit, will replace the threshold value δ. Unlike threshold values weights of connections between bias units and neurons can be trained by learning algorithms. Equation 2.5 shows input and weight vector of a perceptron with n inputs where a bias unit

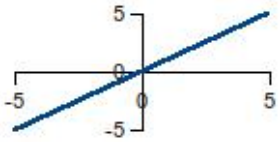
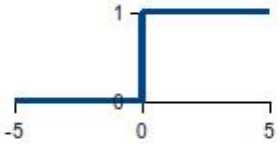
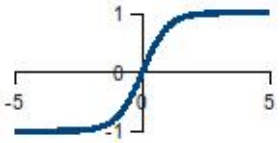
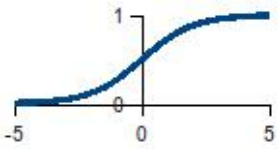
Identity	$y = net$	
Binary Function	$y = \begin{cases} 1, \text{ if } net > 0 \\ 0, \text{ else} \end{cases}$	
Tangens hyperbolicus	$y = \tanh(net)$	
Logistic function	$\frac{1}{1+e^{-k*net}}, k = 1$	

Figure 2.6.: Common activation functions for perceptrons.

replaces its threshold value.

$$\vec{w} = (w_1, w_2, ..., w_{n-1}, w_n, -\theta)$$
$$\vec{x} = (x_1, x_2, ..., x_{n-1}, x_n, 1) \tag{2.5}$$

The new input vector is called *extended input vector*.

Shortcomings of perceptrons

A perceptron divides the input space into two halves. Each half corresponds to a class the input can belong to. Correct classification is only possible if both classes can be seperated by a hyperplane. That problem is referred to as *linear seperability*[NF00].

Definition 2 *Two sets of points A and B in an n-dimensional space are called linearly separable if n + 1 real numbers $w_1, ..., w_{n+1}$ exist, such that every point $(x_1, x_2, ..., x_n) \in A$ satisfies $\sum_{i=1}^{n} w_i x_i \geq w_{n+1}$ and every point $(x_1, x_2, ..., x_n) \in B$ satisfies $\sum_{i=1}^{n} w_i x_i < w_{n+1}$.*

As we can see in figure 2.8 the logical functions *AND* and *OR* are perfectly seperable but their is no way to do that for *XOR*.

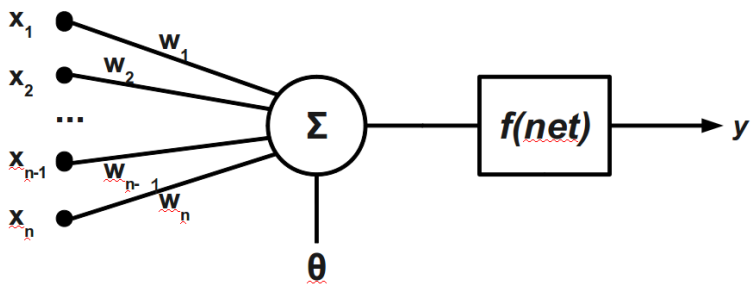
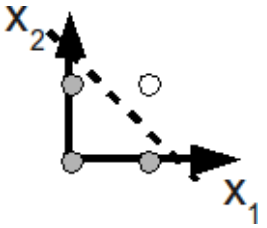


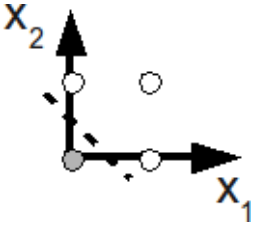
Figure 2.7.: Perceptron

x_1	x_2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

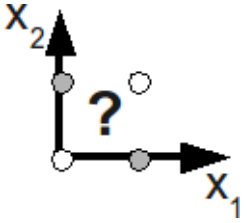
(a) Function overview



(b) And



(c) Or



(d) Xor

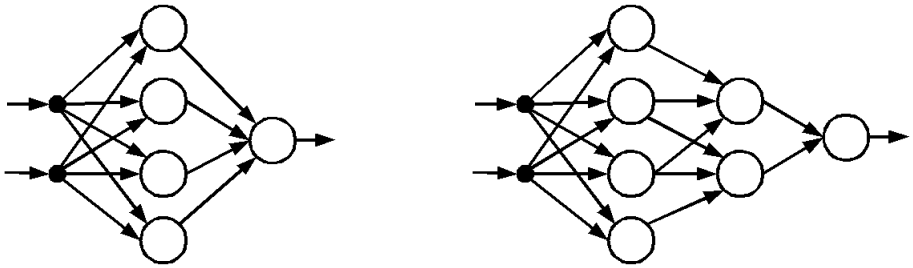
Figure 2.8.: Perceptrons and linear seperability.

2.3.2. Multi Layer Perceptron

Multi Layer Perceptrons (MLP)[NF00], also called feed forward networks, aim to solve this by connecting several layers of neurons to calculate the output. Every neuron in layer $L_i, i \in 0..n$ is connected to one or more neurons in *layers* $L_j, j > i \wedge j \in 0..m$ where m is the total number of layers L in the network. Layer L_0 is called the input layer, Layer L_m is called output layer, all other layers are called hidden layers. Neurons in these layer are respectively called input neurons, hidden neurons and output neurons.

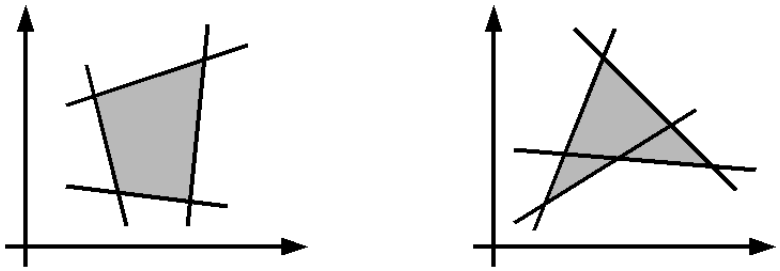
Definition 3 *w_{ij} is the j-th component of neuron i's weight vector and W is an $n \times n$ Matrix of weights w_{ij}.*

A MLP with four layers of neurons (one input, two hidden and one output) is enough to classify any input. Figure 2.9 shows two MLPs with one repsectively



(a) MPL with one hidden layer.

(b) MLP with two hidden layers.



(c) Region formed by MLP 2.9a.

(d) Region formed by MLP 2.9b.

Figure 2.9.: Two MLPs with one respectivly two hidden layers and the regions in the input space formed by them.

two hidden layers and the regions in space classified by those MLPs.

2.3.3. Recurrent Neural Networks

The output of MLPs only depends on the current pattern presented to them. They have no memory and cannot detect time-depended features in the input patterns. In Recurrent Neural Networks[NF00] incoming connections to neurons are therefore not restricted to lower layers, neurons are allowed to have connection from the same or upper layers and even from itself. The output of neuron i in timestep t is then:

$$y_i^t = \Phi\left(\sum_i w_{ij}x^t - j + \sum_k c_{ik}y_k^{t-1}\right) \tag{2.6}$$

y_k^{t-1} is the output of a neuron in the same or in an upper layer from the previous time step and c_{ik} is the weight of this connection.

2.3.4. Advantages of Neural Nets

ANNs have several advantages[NF00] for robot control making them the preferred choice in evolutionary robotics. Some of them are:

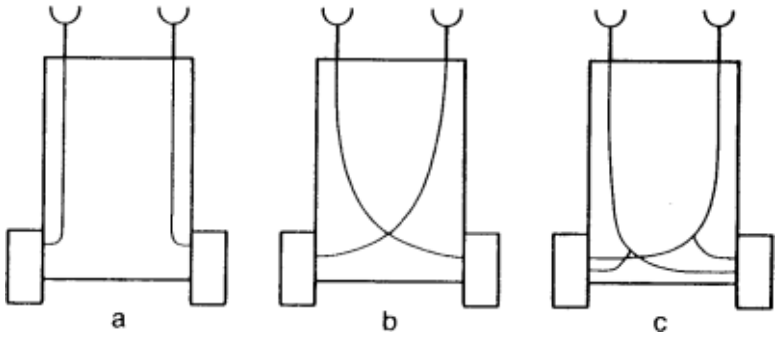


Figure 2.10.: Types of Braitenberg vehicles. Images from [Bra84].

- Inputs (sensors) are directly mapped to outputs (actuators) and they can handle continous sensor input and, depending on the chosen activation fucntion, provide either discrete or continous output.
- Sensor data is often affected by noise. The output of neural networks is not strongly affected by this since input to neurons is the weighted sum of several sources and a single noisy input contributes only some of the data for a neuron.
- Small changes of a network parameters (weights) only result in small changes of network behaviour. Thus search space of neural networks for learning and evolutionary algorithms is smooth.
- As neural networks are a simplified model of human or animal brains they are somewhat biologically plausible making them an interesting choice for replicating natural behaviour.

2.3.5. Examples Uses of Artificial Neural Nets

Braitenberg Vehicle

ANNs can be used to implement Braitenberg vehicles. In Braitenberg vehicles[Bra84] inputs are directly connected to two Outputs (left motor and right motor). They are able too show several behaviours, like following or avoiding light sources, depending on sensor used and how the sensors are connected to the motor outputs. Figure 2.10 shows the three different types of vehicles presented by Braitenberg. All of them show a different behaviour

a) following behaviour,

b) avoidance behaviour and

c) more complex behaviour depending on connection strength

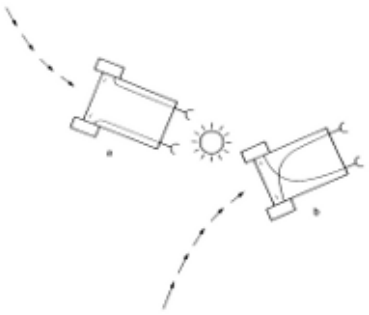


Figure 2.11.: Braitenberg vehicles of type a) and b) with light sensors. Image from [Bra84].

Figure 2.11 shows the behaviour of Braitenberg vehicles of type a) and b) with light sensors.

Extended Braitenberg Architectures

In Extended Braitenberg Architectures[LS95] several Braitenberg vehicles are combined to form a self-sufficient agent on basis of a Khepera robot where each Braitenberg vehicle is implemented as a simple ANN.
The robot is tasked to bring several pegs in an arena back to its home basis, but its energy is limited so it has to recharge now and then in a seperate recharging station. It is outfitted with several sensors (infrared, ambient light, floor and battery level) allowing it to sense the pegs, its home basis and the recharging station. Its behaviour is determined by interaction of all processes respectively Braitenberg vehicles each adding up to the value of the control architecture's output.

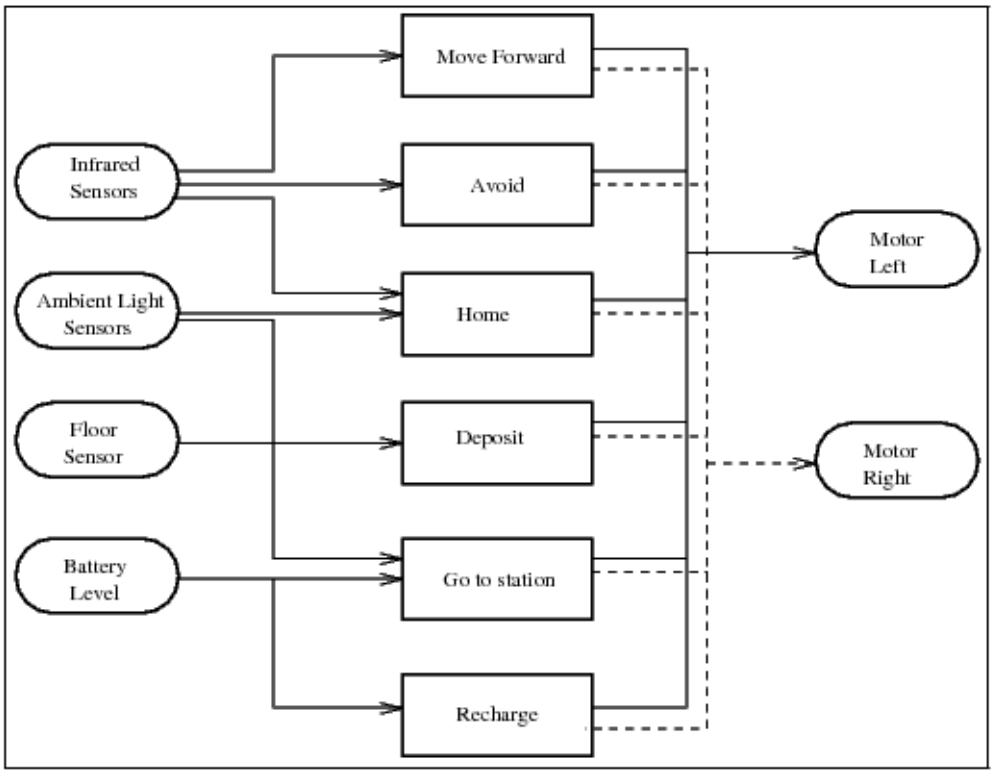


Figure 2.12.: Control architecture of an extended Braitenberg vehicle. Taken from [LS95].

3. Machine learning

The three classes of machine learning are:

- Supervised learning is learning from external supervision and given sets of training data.
- Unsupervised learning is learning without a supervisor and without an external evaluation of the output.
- Reinforcement learning, a subclass of supervised learning, is learning by interaction with the environment and getting feedback in form of rewards.

All learning algorithms are concerned with updating connection weights. Therefor they compute a modifcation Δw_{ij}^t and add that value to the old weight w_{ij}^{t-1} to calculate the new weight w_{ij}^t

$$w_{ij}^t = w_{ij}^{t-1} + \Delta w_{ij}^t \tag{3.1}$$

Often Δw_{ij}^t is modified by a learning rate η to prevent wide oscillations of the weight parameters

$$w_{ij}^t = w_{ij}^{t-1} + \eta \Delta w_{ij}^t \qquad 0 < \eta \leq 1 \tag{3.2}$$

3.1. Supervised learning

Learning algorithms following the supervised learning paradigm minimize the error between the output y computed by the network for a given input pattern x and a teaching pattern t out of training pattern μ by modifying the network's connection weights. The complete training set consists of M pairs of input and teaching patterns. The *error function* is defined as

$$E = \frac{1}{2} \sum_{\mu=0}^{M-1} \sum_{i=0}^{N-1} \left(t_i^\mu - y_i^\mu \right)^2 \tag{3.3}$$

where N is the number of output units of the network.

3.1.1. The Delta Rule

The *Delta Rule* is a learning method for one-layered networks. It minimizes equation 3.3. First the output of neuron j for input vector x of pattern μ,

$$y_i^\mu = \Phi\left(\sum_{j=0}^{n-1} w_{ij} x_j\right) \tag{3.4}$$

is computed. After obtaining the output of neuron i the *delta error* for this neuron can be determined for pattern μ:

$$\delta_i^\mu = \dot{\Phi}\left(\sum_{j=0}^{n-1} w_{ij} x_j\right)\left(t_i^\mu - y_i^\mu\right). \tag{3.5}$$

$\dot{\Phi}$ is the derivate of the neuron's activation function Φ. Afer determining the delta error

$$\Delta w_{ij}^\mu = \delta_i^\mu x_j^\mu \tag{3.6}$$

is added to all connection weights.

3.1.2. Backpropagation of error

Backpropagation of error is also known as *generalized delta rule*. It extends the delta rule to apply to networks with one or more hidden layers. A random pattern μ out of the training set is presented to the net and the steps

1. computing the activation values of all neurons in the net,
2. determining the delta error of output neurons
3. backpropagating the delta error to hidden neurons,
4. and updating connection weights

are executed for each pattern until the Total Sum Squared Error,

$$TSS = \frac{1}{M}\frac{1}{N}\sum_{\mu=0}^{M-1}\sum_{i=0}^{N-1}\left(t_i^\mu - y_i^\mu\right)^2 \tag{3.7}$$

falls below a satisfactory value. Activation values are computed as illustrated in section 2.3.2. The delta error for output units is determined with equation 3.5. The delta error for hidden neuron j is obtained by backpropagating the delta error of output neurons to the hidden neurons:

$$\delta_j^\mu = \dot{\Phi}\left(\sum_{k=0}^{n-1} w_{jk} x_k^\mu\right)\sum_{i=0}^{m-1} w_{ij}\delta_i^\mu \tag{3.8}$$

where n is the size of the input vector x of neuron j, m is the number of neurons neuron j is connected to, w_{ij} is the weight associated with that connection and δ_i^μ is the delta error at that neuron. The connection weights w_{ij} are then updated by

$$\Delta w_{ij}^\mu = \eta(\delta_i^\mu x_j^\mu) \tag{3.9}$$

where η is a learning rate smaller than 1.

3.1.3. Backpropagation through time

Normal backpropagation of error doesn't work for recurrent neural nets. *Backpropagation through time* is a training algorithm that does work for recurrent neural nets. Its basic idea is converting recurrent neural nets to feed-forward neural nets by *unfolding* them. Unfolding means that for a finite number of timesteps $1, 2, ..., T$ the network is copied and the recurrent connections are not drawn to the neurons of this timestep but instead to the corresponding neurons of the previous time step. Each timestep t input $x(t)$ is fed to the network and its the state of all units $o_1(t), ...o_n(t)$ and the derivate of all activation functions $o_1'(t), ...o_n'(t)$ is recorded. In timestep $t = 0$ the state of all neurons is considered to be zero. Error for backpropagation, measured at end of each timestep, is then

$$e^{(t)} = \left(o^{(t)} - y^{(t)}\right)^T. \tag{3.10}$$

The error $e_i^{(t)}$ can be set to zero for hidden units. Problems arise from the fact that each weigth w_{ij} is copied for each timestep and thus weights are not unique anymore. To remedy this a virtual unit with a value of one is placed in each connection replacing weights. This unit is then multiplied with the weight during compuation of values. Now weights are unique again and backpropagation can be performed as usual.
Backpropagation starts at timestep T going down through all timesteps. Delta error at timestep T is

$$\delta^{(T)} = D^{(T)} e^{(T)} \tag{3.11}$$

where $D^{(t)}$ is a $n \times n$ matrix where the i-th diagonal component is $o_i'(t)$. The error at timestep $i = 0, ..., T-1$ is

$$\delta^{(i)} = D^{(i)} \left(e^{(i)} + W\delta^{(i+1)}\right). \tag{3.12}$$

3.2. Unsupervised learning

Unsupervised learning methods do not update connection weights based on an error function. They rely on input patterns only. Connection weights are updated in a self-organizing process.

3.2.1. Hebb's Rule

Hebb's Rule[NF00] updates connection weights if both connected neurons are simultaneously active. The basic formula is

$$\Delta w_{ij} = \eta x_i y_j \tag{3.13}$$

where x_i are y_i are the activation values of the connection's source i and its target j and η is the learning rate.
The Hebb Rule in its basic form can only strenghten weights and not decrease them. An activation function that allows negative output or decreasing the connection weight if only one of the neurons is active can remedy that problem.

3.3. Reinforcement learning

Often it is impractical or even impossible to get training data needed to train an agent. Then supervised learning algorithms can't be used for learning. Reinforcement learning [LCC+02] does not depend on external training data to train an agent. Here an agent learns from interaction with his environment.
He is rewarded for actions that are useful in his current situation but he doesn't know beforehand which actions that will be. Therefore he has to *explore* the set of possible actions to seek those that give rewards. On the other hand, he has to *exploit* actions he already knows to gather as much reward as possible.
There is always a trade-off between *exploration* and *explotation*. Both have to be balanced for an agent to succeed in his given task.

3.3.1. Discrete-time TD(λ)

Discrete-time TD(λ) is an algorithm that can be used to calculate weight updates for a neural network. It uses back-propagation to calculatea gradients for weight updates (see chapter 3.1.2). The algorithm is shown in figure 3.1.

$\vec{w} \leftarrow$ an arbitrary initial value
$\vec{e} \leftarrow \vec{0}$ {dimension of $\vec{e}$ = dimension of $\vec{w}$}
for each episode **do**
  $x \leftarrow$ random initial state
  while not end of episode **do**
    $x' \leftarrow \nu(x, \pi(x))$
    $\delta \leftarrow r(x, \pi(x) + \gamma V_{\vec{w}}(x) - V_{\vec{w}}(x$
    $\vec{e} \leftarrow \lambda\gamma\vec{e} + \partial V_{\vec{w}}(x)/\partial\vec{w}$
    $\vec{w} \leftarrow \vec{w} + \eta\delta\vec{e}$
    $x \leftarrow x'$
  end while
end for

- S is the state space $S \subset \mathbb{R}^p$ where $\vec{x}$ is a vector of p real-valued variables defining the state of the system.
- U is the control space $U \subset \mathbb{R}^q$ where $\vec{u}$ is a vector of q real-valued variables called *action*. The Agent influences the system through actions.
- π is a *strategy* or *policy*. $\pi : S \rightarrow U$ mapping states to actions. Applying a policy from starting state x_0 leads to a sequence of of states $x_{t0}, x_{t1}, x_{t2}, ...$ called *trajectory*.
- $\nu(x, u)$ is a function determining the next state x for the system when action u is taken.
- $r(x, u)$ is a *reward function* mapping state-action pairs to real numbers.
- $V_{\vec{w}}(x)$ is a *value function* cumulating reward over a trajectory and γ is a *discount factor* ($\gamma \in \lfloor 0, 1 \rfloor$) causing convergence:
- δ is the *temporal difference error* (or TD error) for state-value prediction.
- $\vec{e}$ is called *eligibility trace*. Eligility traces mark network parameters that have to be updated when a TD error occurs and λ is a *decaying parameter* for eligility traces.

Figure 3.1.: Discrete-time TD(λ) algorithm.

4. Evolutionary Algorithms

4.1. Introduction to Evolutionary Algorithms

Evolutionary algorithms (EAs) are kind of reinforcement learning algorithms[KS05]. EAs and reinforcement learning are similar in that they both evaluate the performance of agents within their environment using a reward function (*fitness function*) to direct their search. A huge advantage of EAs is that they require comparatively little domain knowledge[SM]. Evolution of artificial neural networks is called *Neuroevolution*and algorithms that evolve both weights and topologies of networks are called *Topology and Weight Evolving Neural Networks. (TWEANNs)* [SM] . In *Evolutionary Programming* only mutation is used to change weights (*parametric mutation*) and topology (*structural mutation*). *Genetic Algorithms* also use *crossover*[SM].
EAs work by evaluating a *population* of solutions and then selecting a number of those solutions as *parents* for the next *generation* of solutions[NF00]. This is repeated until a stopping criterion like a predetermined fitness value is met.

4.1.1. Encoding

Characteristics of an individual solution(*phenotype*) are encoded as a string of parameters (*genotype*)[SM]. Two basic encoding schemes can be differentiated:

- *direct encoding* - genotype specifies all connections and neurons of a phenotype
- *indirect encoding* - genotype contains rules to construct a phenotype

Encoding is an important property of EAs since it directly affects the solutions that can be found by an EA (*representation problem*). A simple direct encoding would be a binary encoding for a network with a fixed number of neurons where each positive bit signifies a connection between two neurons. Am example for such an encoding can be found in figure 4.1.

4.1.2. Selection

Several mechanism for selecting the parents of a new generation exist[NF00]. In *roulette wheel selection* the fitness value f_i of a whole generation of N individuals is

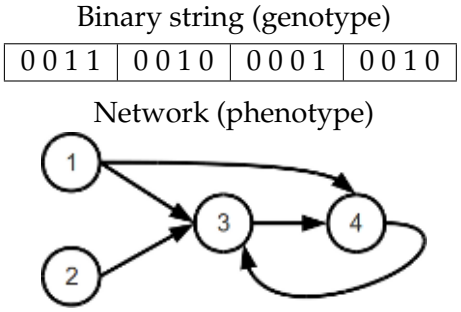


Figure 4.1.: Example for a simple binary encoding of a network.

summed up and each individual i is assigned a selection propability equal to

$$p_i = \frac{f_i}{\sum_i^N f_i}. \tag{4.1}$$

Then N indivuals are randomly selected using their assigned propabilites. Problems occur if all individuals have a similar fitness value or if few individuals dominate their generation. A way to avoid these problems is *rank based selection*. Indidivuals are sorted according to their fitness and then are assigned a propability corresponding to their rank. *Truncation selection* also sorts indivuals but it selects the M best individuals of their generation who create O offspring where $N = M \times O$. In *tournament based selection* two individuals are randomly drawn from the population. A random number between 0 and 1 decides which one is chosen for the next generation with a higher propability for the individual with higher fitness.
Often the best individuals are preserved for the next generation (*elitism*). This is mostly done in tournament based selection but also in other selection mechanisms.

4.1.3. Mutation

Mutation allows an EA to find new solutions. Different algorithms use different mutation operators. Operators can be classified as either structural or parametric. Parametric mutations change the network parameters or weights. Structural operators modify the topology of a network by adding or deleting connection and adding or deleting neurons. Not all algorithms use all of those operators.

4.1.4. Crossover

In crossover parts of the string encoding of two solutions is swapped to create a new solution. Swapping can occur at a single location and is then called *one point crossover* or in *multi point crossover* swapping occurs at more than one location in the string[NF00]. Figure 4.2 shows a one point crossover using a simple binary encoding.

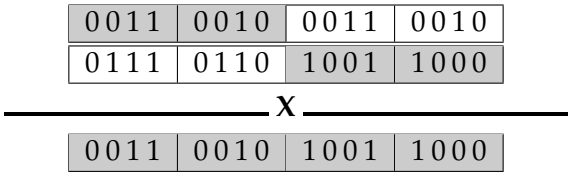


Figure 4.2.: Example for one point crossover with a simple binary encoding of a network.

4.1.5. Design of Fitness Function

The fitness function is integral for EAs to work. They reward a solution for accomplishing the set tasks and thereby allow selection. A common problem is that for complex tasks all solutions of an early generation have low or even zero fitness values. This prevents EAs from selecting promising solutions and the search for solutions becomes directionless. This is called *bootstrap problem* [NF00].
Several solutions to this problem have been proposed including *co-evolution* where two or more *species* compete against each other, decomposing the task into subtasks and their fulfillment rewarding or using *incremental evolution*, i.e. starting with a simplified version of the task and increasing complexity more and more.

4.1.6. Re-evaluation

Sometimes it becomes important to re-evaluate networks one or more times[BHE09]. That is the case if the fitness function is noisy and therefore its results are not entirely dependable. Overall fitness then can be averaged and allows a more reliable evaluation of a networks. Another case were re-evaluation is reasonable is when several networks are evaluated on a single robot one after another. A bad solution may drive the robot into a corner where it is stuck for some time and thereby wasting evaluation time of the next solution.

4.2. GeNeralized Acquisition of Recurrent Links (GNARL)

GNARL[ASP94] is an evolutionary programming algorithm for evolving artificial neural networks with recurrent links. There is no distinction between genotype and phenotype, mutation and selection are directly used on the networks.
Networks evolved with GNARL must satisfy the following requirements at all times:

R_1: no links *to* an input neuron,

R_2: no links *from* an output neuron,

R_3: and at most one link between neurons.

4.2.1. Initialization of GNARL

The algorithm starts with a population of random nets forming the first generation. Users of GNARL must supply a number of parameters for these initial networks:

- number of input nodes (m_{in})
- number of output nodes (m_{out})
- minimum and maximum number of hidden nodes (h_{min}, h_{max})
- minimum and maximum number of links (l_{min}, l_{max})

The number of hidden nodes is then randomly choosen from $[h_{min}, h_{max}]$ as is the number of links from $[l_{min}, l_{max}]$. Starting and ending neurons of links are randomly selected from all neurons in the network, but must satisfy R1-R3.

4.2.2. Evolution with GNARL

Each generation a fitness function is used to evaluate all networks in the population. The best performing fifty percent are choosen as parents for the next generation. Offspring comprising the next generation are mutated copies of their parents. Severity of all mutations depends on the temperature T of the parent network η, where f is the fitness function used to evaluate the networks and f_{max} is the maximum value of the fitness function:

$$T(\eta) = 1 - \frac{f(\eta)}{f_{max}} \tag{4.2}$$

This temperature is further modified by a uniform random variable $U(0,1)$ over the interval $[0,1]$ forming the *instantaneous temperature* $\widehat{T}$:

$$\widehat{T}(\eta) = U(0,1)T(\eta) \tag{4.3}$$

Networks are subjected to two types of mutations: *parametric mutations* and *structural mutations*. Parametric mutations alter the network weights according to the formula:

$$w = w + N(0, \alpha\widehat{T}(\eta)) \quad \forall w \in \eta \tag{4.4}$$

$N(\mu, \sigma^2)$ is a gaussian random variable and α is a proportionality constant. There are four types of structural mutations: adding/removing links and adding/removing neurons. The number of structural modifications for each type is determined by:

$$\Delta_{min} + \lfloor U(0,1)\widehat{T}(\eta)(\Delta_{max} - \Delta_{min}) \rfloor \tag{4.5}$$

Δ_{min} and Δ_{max} are upper and lower bounds that must be specified for each type of structural mutation. The weights of new links are set to zero and new neurons are added without links as not to perturb network functionality. The resulting sequence of mutations (structural and parametric) each offspring is subjected to is:

1. removing neurons,
2. removing links,
3. modifying link weights,
4. adding links,
5. and adding neurons.

4.3. NeuroEvolution of Augmenting Topologies (NEAT)

NEAT[SM] is a genetic algorithm able to evolve neural nets with recurrent links. It uses crossover, parametric mutation and structural mutation to find a solution. Speciation is used to protect innovation from an inital drop in fitness. Similar genomes are grouped into species and share their fitness. Historical markings are introduced and used in speciation to determine which species a genome belongs to and in crossover to align matching genes.
NEAT tries to minimize solutions. For this, starting genomes are minimal, i.e. only input and output nodes and every input node is connected to every output node.

4.3.1. Genetic encoding

Direct encoding is used to encode neural nets. A genome encoding nets consists of two types of genes, *connection genes* and *node genes*. Node genes represent neurons in the net, but the algorithm mainly works with connection genes who represent the net's links. Connection genes contain several pieces of information: the two nodes they link, the link's weight, an enable bit and an innovation number which is used during crossover. Figure 4.3 shows an example of a genotype and the corresponding phenotype.

4.3.2. Historical markings and crossover

Whenever a connection gene is added to any genome a global counter is incremented and its value is assigned to the gene as its innovation number. To prevent an inflation of every mutation adding a new connection is compared to all previous mutations adding links. If it adds the same genes the same innovation number is assigned.

Genotype

Node 1 Input	Node 2 Input	Node 3 Output	Node 4 Output	Node 5 Hidden

Node Genes

In 1 Out 3 Weight 0.35 Disabled Innovation 1	In 1 Out 4 Weight 0.5 Enabled Innovation 2	In 2 Out 3 Weight -0.2 Enabled Innovation 3	In 2 Out 4 Weight 0.66 Enabled Innovation 5	In 1 Out 5 Weight -0.13 Enabled Innovation 8	In 5 Out 3 Weight 1.2 Enabled Innovation 9	In 3 Out 3 Weight 0.4 Enabled Innovation 13

Connection Genes

Phenotype

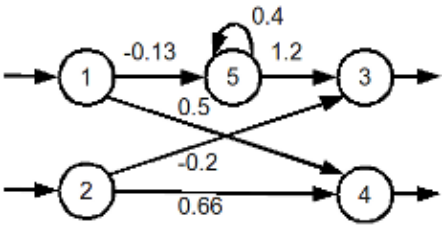


Figure 4.3.: Genotype to phenotype mapping. Note that the connection gene from Node 1 to Node 3 is disabled and therefore does not appear in the genotype.

During crossover genes are aligned according to their innovation number. Connection genes with identical innovation number are inherited with their weights averaged or randomly selected from one of the parents. Genes which innovation number does not match with a gene of the other parents are called *disjoint genes*, genes with an innovation number higher than any other of the other parent's genes are called *excess genes*. Both disjoint and excess genes are inherited from the parent with the higher fitness. If they have the same fitness value they are inherited randomly. There's a small chance for disabled genes to become reenabled. Figure 4.4 shows an example of a crossover.

4.3.3. Speciation

Species consists of genomes who are similar to each other. Similarity is determined by a distance function (equation 4.6) that calculates the distance between two genomes i and j. Its value must be lower than a compatibility threshold δ_t for a genome representing the species.

$$\delta(i,j) = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \bar{W} \tag{4.6}$$

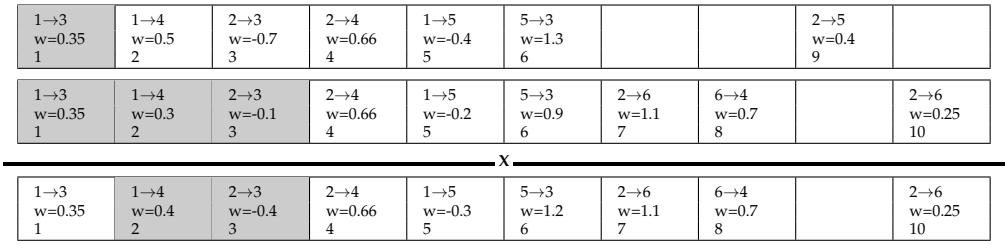


Figure 4.4.: Alignment of two genomes during crossover. Genes with innovation number 7, 8 and 9 are disjoint genes, gene with innovation number 10 is an excess gene. Genes with a gray background are disabled. In this example the second genome has a higher fitness value. The first gene was reenabled and connection weights were averaged during crossover.

E is the number of excess genes, D is the number of disjoint genes, W the average weight difference of matching genes, coefficients c_1, c_2 and c_3 are user-supplied parameters.
Species are represented by a random member of the previous generation. Distance to all species is measured in sequence for all genomes of the current generation until a species is found where distance below δ_t is found. If no species is found for a genome it starts a new species.
Members of a species share their fitness values, sanctioning large populations and thus forcing population to remain relativly small. This practice is called *explicit fitness sharing*. Adjusted fitness value f'_i for genome i is calculated as follows:

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(i,j)} \tag{4.7}$$

Where $sh(i,j)$ is the sharing function:

$$sh(i,j) = \begin{cases} 1 & \text{if } \delta(i,j) < \delta_t \\ 0 & \text{else} \end{cases} . \tag{4.8}$$

4.3.4. Evolution with NEAT

After evaluating every genome each species is allocated a number of offspring. The number of offspring n of species k is:

$$n_k = \frac{\bar{F}_k}{\bar{F}_{tot}} |P|. \tag{4.9}$$

$\bar{F}_k$ is the average fitness valueof species k, $\bar{F}_{tot}$ is the sum of all average fitness values $\sum \bar{F}_k$ and $|P|$ is the population size.

The new generation of a species consists of its best performing member, called *species champion*, and n_k-1 offsprings. The offspring's parents are randomly selected out of the best perfoming n_k members of the previous generation.
After crossover, parametric and structural mutation is applied to the offspring. Parametric mutation modifies connection weights with a fixed propability by an uniform random number that's upper and lower bounds have to be specified by the user.
There are two types of structural mutations: *add node* and *add connection*. Add connection adds a connection between two unconnected nodes. Add node adds a node and two connections by disabling an existing connection. The source of the first connection is the source of the disabled connection, its target is the new node and its weight is set to one. The source of the second connection is the new node, its target is the target of the disabled node and its weight is set to the weight of the disabled connection.

4.4. Evolutionary Acquisition of Neural Topologies (EANT)

4.4.1. Genetic encoding

EANT[KS05] encodes neural networks in a tree-based linear genome. There are basically three types of genes in the genome: input genes representing input neurons, neuron genes representing hidden or output neurons and jumper genes representing forward or recurrent connections.
Neuron genes encode outgoing connection strength, their depth and number of inputs to the neuron, jumper genes encode id of their source, their connection strength and wether they are feedforward or recurrent and input genes encode connection strength and id of associated input neuron. In addition every gene has an associated learning rate used for parametric mutation. Figure 4.5 shows the encoding for the neural net also used in figure 4.3 and the stepwise interpretion of the genome.
One feature of the genetic encoding used by EANT is that the phenotype can be evaluated without decoding it by using a first in last out stack.

4.4.2. Initialization of EANT

The are two different methods to initialize EANT: the *grow* and the *full* method. Both have in common that they use no jumper genes.

Grow Method

A random number of input nodes and neuron nodes is assigned to each neuron node startig from the output neurons until arriving at a given maximum depth.

Genotype

Neuron 3 Weight - Depth 0	Input 2 Weight -0.2	Neuron 5 Weight 1.2 Depth 1	Jumper 5 Weight 0.4 Recurrent	Input 1 Weight -0.2	Neuron 4 Weight - Depth 0	Input 1 Weight 0.5	Input 2 Weight 0.66

Phenotype

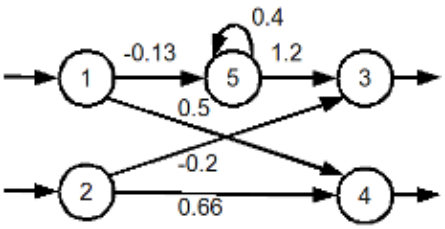


Figure 4.5.: Genome encoding a neural net with EANT and corresponding net.

The genome must be edited afterwards because neuron nodes can have multiple input genes for the same network input. E.g. one could simply select one of those input genes, add the connection strenghts of the others and then remove them.

Full Method

Nodes are added to the genome until a given depth is reached. Then all nodes are connected to a fixed number of inputs. The result is a symmetric tree structure for each output neuron.

4.4.3. Evolution with EANT

Starting with a population either initialized by the growth or full method for N generations only *exploitation* is done, i.e. parametric mutation. After every generation the best μ create λ offspring.
After those N generations have passed an *exploration* step follows, i.e. structural mutation. The best half of the population and remaining members younger than M generation are selected and make up the starting population of the next exploitation phase.

Structural Mutation

There are three types of structural mutations in EANT:

- adding a recurrent or forward jumper gene,
- removing a recurrent or forward jumper gene
- adding a subnetwork.

Structural mutation apply only to neuron genes with a mutation propability of p_m. If a neuron gene is going to be mutated a strcutural mutation is randomly selected. Subnetworks are neurons that are connected to approximatly 50% of the network's inputs.

Parametric Mutation

Parametric Mutation pertubes the weight of each gene in the genome. To get all weight updates, the parameters τ and τ' have to be calulated first:

$$\tau' = \frac{1}{\sqrt{2n}} \tag{4.10}$$

and

$$\tau = \frac{1}{\sqrt{2\sqrt{n}}} \, . \tag{4.11}$$

With that the adjusted learning rate σ_i' can be calculated:

$$\sigma_i' = \sigma_i e^{\tau' N(0,1) + \tau N(0,1)} \, . \tag{4.12}$$

If this adjusted learning rate is lower than a threshold e_0 it is set to the threshold value:

$$\sigma_i' < \epsilon_0 \Rightarrow \sigma_i' = \epsilon_0 \, . \tag{4.13}$$

Finally the weight update can be caluclated:

$$w_i' = w_i + \sigma_i' N_i(0,1) \qquad 0 \leq i < n \, . \tag{4.14}$$

$N(0,1)$ is a gaussian distribution of zero mean and unity standard devation and n is the number of genes in the genome.

5. Evolutionary Framework

The evolutionary framework is a shared library written in C++ used by the symbricator simulation. The symbricator simulation itself is an application that uses Delta3D as simulation engine.

5.1. Delta3D

Delta3D is an open-source game and simulation engine integrating a host of other open-source projects including Open Scene Graph (osg), a 3D graphics toolkit, and Open Dynamics Engine (ODE), a physics engine. It provides a C++ API that can be used on Microsoft Windows with Visual Studio and on Linux with gcc.

5.1.1. Important Classes

This is a small selection of classes from the Delta3D API used or derived from in either the simulation or the plugin.

GameApplication

`GameManager` is the central class controlling the application. One important aspect of the class used in the plugin is that it can be used to add or remove drawables from the scene.

Object

An object is a drawable that can be added to the scene with physical properties. The `GlobalFoodController` that will be presented later uses `Object` to create its food objects.

GameManager

`GameManager` manages actors and components. Its singleton instance can be used e.g. to find actors and components and send messages.

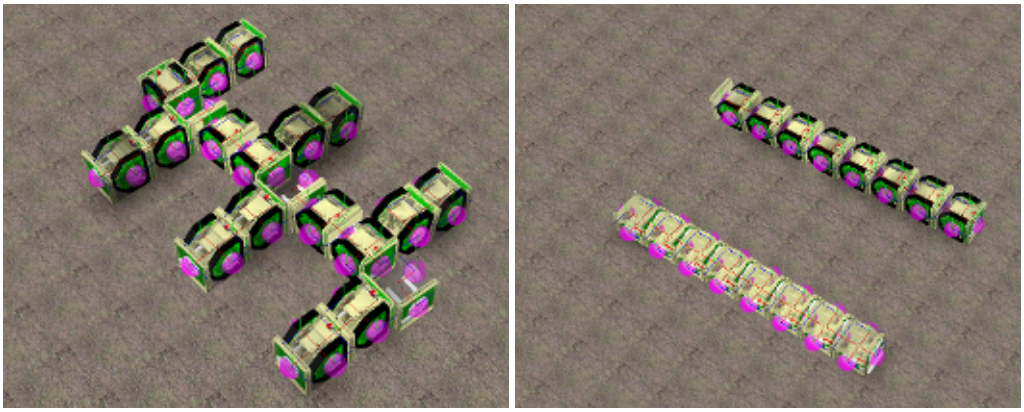


Figure 5.1.: Two screenshots from the simulation using sample scenarios.

GMComponent

Classes deriving from this class can be added to the game manager as a component. Their `ProcessMessage()` method is called whenever a game event occurs. The most important game event is the tick, the basic unit of time in the simulation.

5.2. Simulation

The Simulation is a shared library written in C++ and dependend on Delta3D. It is loaded by a Delta3d executable and contains an entry point where all initialization is done.
It provides preconfigured robot models with sensors, actuators and communication channels as wells as an optional gui to interact with a running simulation. Experiments can be configured with xml files called scenario files that allow to configure robots, sensors, controllers and the map to be used as well as to load external code via a plugin mechanism.
Also included are a number of sample controllers and scenarios using them. Figure 5.1 shows two screenshots from sample scenarios.

5.2.1. Important Classes

This section provides an overview over those classes important for writing plugins or extending existing ones.

SimulationEntryPoint

The `OnStartup()` method of `SimulationEntryPoint` is the first method to be called from the simulation library. It is responsible for all initialization work of the

simulation including calls to parse the config files, registering components with the `GameManager` and loading plugins.

SimulationActorsRegistry

`SimulationActorsRegistry` registers all defined actor types (robots) with the `GameManager`. Its method `addRobotToGM()` is used to add robots to the `GameManager`.

SimulationArenaComponent

`SimulationArenaComponent` is the component used to control an experiment. It provides access to robots in the scene, allows adding or removing of robots and controllers even during an experiment. In addition the scene can be reset, a new scene file can be loaded or the scene can be cleared of all robots and controllers.

GetConfig

`GetConfig` is used to parse the xml scene files containing the experiment that. This includes robot actors, controllers and plugins. The parsed data is put into vectors and is then accessible from the outside via getters.

PluginEntryPointBase

`PluginEntryPointBase` is an abstract interface containing a method `StartPlugin()`. When a plugin is loaded `SimulationEntryPoint` looks for a child class of PluginEntryPointBase and calls its `StartPlugin()` method. This allows adding of controllers to the `GameManager`, adding objects to the scene or executing any other initializing code required by the plugin.

ControllerComponentBase

`ControllerComponentBase` is the base clase for all robot controllers. All controllers defined in the scene file must be derived from this class. It provides methods to read and write the robots position and orientation in the scene, to access the robot's sensors, actuators and connectors and to communicate with other robots in the scene. All parameters defined in a scene file for a controller (see figure 5.2) are placed in a hashmap where the keys are the names of the parameters.

5.2.2. Loading Plugins

There are two ways to get a plugin loaded by the simulation. First is adding a `<Plugin>` tag to the scene file. A child of `PluginEntryPointBase` must be present in the plugin library. Second is adding a child of `ControllerComponentBase`

1. `<Plugin name="libann.so" path="/home/example/lib/>`
2.
```
<Controller
      name="myController"
      type="Plugin:/home/example/lib/libann.so"
      robotToControl="r0" parameter="">
    <Param name="x" value="y"/>
    ...
  </Controller>
```

Figure 5.2.: Example for defining plugin components, plugin controllers and parameters for controllers in a scene file.

as controller of a robot. The `name` attribute of the `Controller` tag hast to start with 'Plugin' for `Controllers` to recognize it has to load the controller from a plugin library. The attribute must also contain path and name of the plugin library. For an example see figure 5.2.

5.3. Framework

The framework is a shared library loaded by the simulation's plugin mechanism. Its purpose is to assist in creating experiments using artificial neural networks and evolutionary algorithms. Most of the experiments will be within the simulation but experiments without can also be imagined.
The most important parts are classes for artifical neural networks, allowing to create feedforward and recurrent artificial neural networks and classes for evolutionary algorithms who evolve artificial neural networks. Both can be used without the simulation as the pole cart experiments will show. Controller classes provide integration with the simulation.
As an example for a fitness function food object can be added to the simulation. Gathering them increases fitness of a controller.

5.3.1. Artifical Neural Networks Classes

This section presents the classes for artificial neural networks. They can be used independently of other framework classes. `RTNN` is the main class, `NNInput` and `NNOutput` are used to load and save nets, `Neuron`, `Link`, and `ActivationFunctions` are mostly for internal use. Figure 5.3 shows a class diagramm for `RTNN`, `Neuron`, `Link` and `ActivationFunctions`.

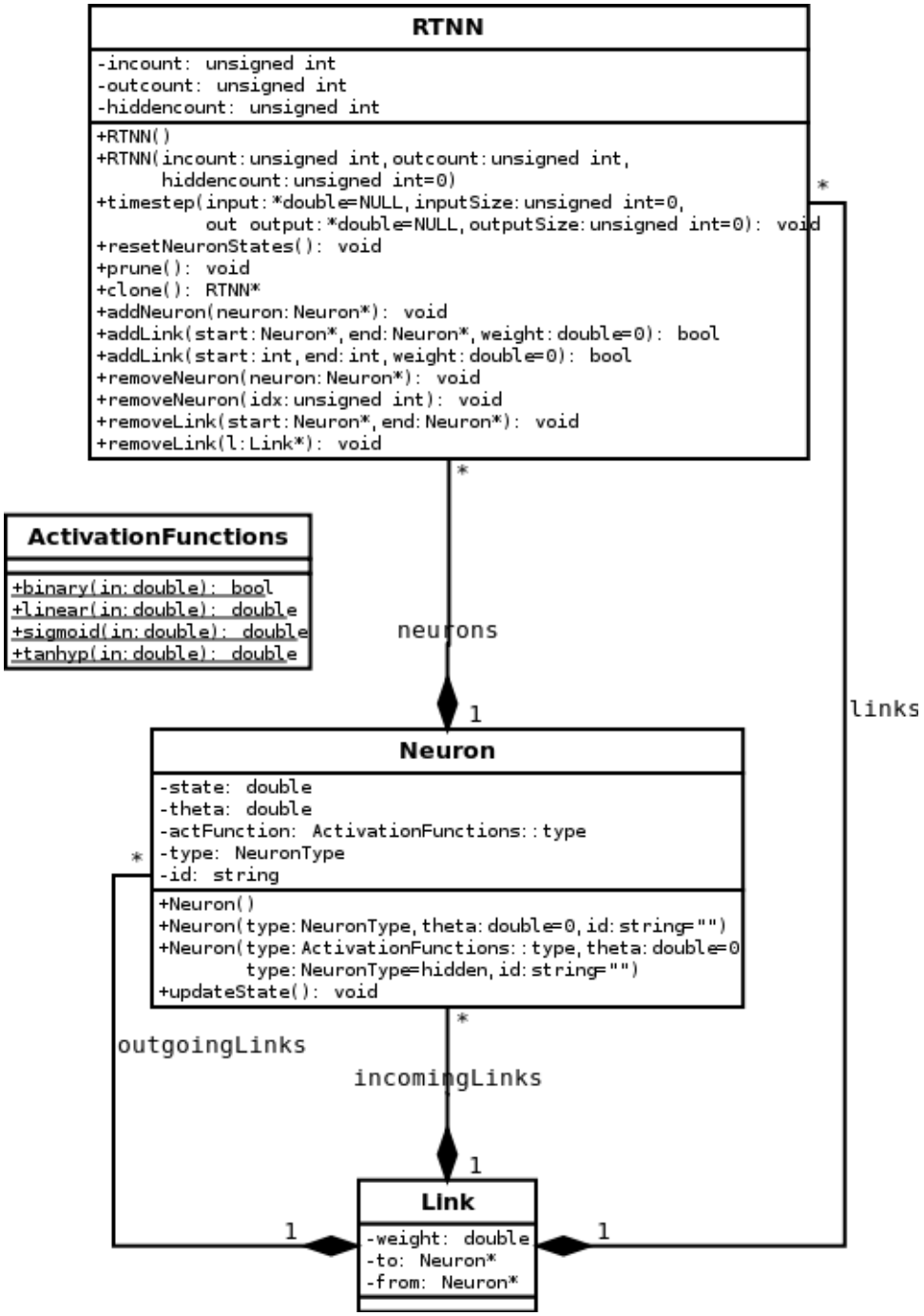


Figure 5.3.: UML Class Diagramm for Neural Networks (without input and output classes).

RTNN

This is the main class for neural networks in the framework. It is named after *recurrent time neural networks*. Each timestep the state of a neuron is propagated to other neurons connected that are connected to it. That means in a 3-layer MLP it takes three timesteps for the input to affect the output.
All neurons are stored in a single vector, starting with input neurons, then output neurons and at last hidden neurons. That means the index i of the first hidden neuron is $i = incount + outcount$.

Attributes:

- `incount`: Number of input neurons.
- `outcount`: Number of output neurons.
- `hiddencount`: Number of hidden neurons.

Methods:

- `timestep()`: If `input` is not `NULL` sets the state of input neurons to the values in `input` and calls `updateState()` for each neuron. Then after calling `flushState()` for each neuron writes the states of output neurons to `output`.
- `prune()`: Some EAs evolute nets where neurons without a path from an input or to an output neuron. That may affect performance of the network and those neurons are not necessary for computation of the networks output. Therefore those neurons can be removed after evolution (but not during).
  This method removes all hidden neurons and their links from the net that do not belong to a path from an input to an output neuron. First calculates the transitive closure of the network with the Warshall algorithm. Then checks if a path from an input neuron and to an output neuron exists. If not the neuron is deleted.
- `clone()`: Creates a clone of the neural network. All neurons and links are replicated and added to a new network that is then returned.
- `resetNeuronStates()`: Sets the state of all neurons to 0. This is important in evolution so networks do not have a memory of their past during a new evaluation of the network.

Neuron

`Neuron` is the computing unit of a network. There are three types of neurons to distinguish. Input neurons whose state is set by outside input, output neurons whose state is read by `RTNN` and used as the networks output and hidden units that are just computing units with no additional function.

Attributes:

- `state`: State of neuron after last state update.
- `theta`: Threshold value of neuron.
- `actFunction`: Function Pointer to one of the methods in `ActivationFunctions`. Used to calculate the neuron's states.
- `type`: Type of neuron as defined in an enumeration (input, output, hidden)
- `id`: Id or name of neuron, used mainly for printing and logging neurons.

Methods:

- `updateState()`: Calculates a new state for its neuron. This state is written into a temporary state attribute until `flushState()` is called. This is done by `RTNN` when the new states of all neurons is calculated.

Link

`Link` represents a connection between two neurons. Start and end of the connection can be the same neuron.

Attributes:

- `weight`: Weight parameter for connection.
- `to`: Ending neuron of connection.
- `from`: Starting neuron of connection.

ActivationFunctions

`ActivationFunctions` contains the activation functions for neurons as static methods. There is also a `typedef` that allows `Neuron` to define an attribute for a function pointer to one of the methods of this class. Table 2.6 shows the graphs of those functions.

Methods:

- `binary()`: Returns 1 if input is greater than 0.
- `linear()`: Output is same as input.
- `sigmoid()`: Logistic function with $k = 1$.
- `tanhyp()`: Hyperbolic tangent.

NNOutput

`NNOutput` is used to save nets as text files or visualize them as images. It uses the Graphviz library[1] for laying out networks as graphs and then writing them to disk.

Methods:

- `generateGraph()`: Creates a graphviz graph structure out of a network. There is a subgraph for each type of neuron to ensure that after layouting input neurons and output neurons are drawn close to each other.
- `outputToFile()`: Writes a graph to a file. If format is specified as "dot" writes a textfile, for "png" an image file is generated. Uses the dot algorithm for layouting. Layouting is optional.
- `outputToBuffer()`: Writes a graph to a buffer.

NNInput

`NNInput` reads networks saved by `NNOutput`.

Methods:

- `readFromFile()`: Opens a dot file and creates a `RTNN` from the information found in the file.

5.3.2. Evolutionary Algorithms Classes

These classes contain the evolutionary algorithms implemented by the framework. They can be used together with the network classes without other classes of the framework. Figure 5.4 shows the most important classes.
Currently implemented are GNARL (see section), a variant of GNARL, EANT (see section 4.4) and a variant of NEAT (see section 4.3).

[1] for further information please see www.graphviz.org

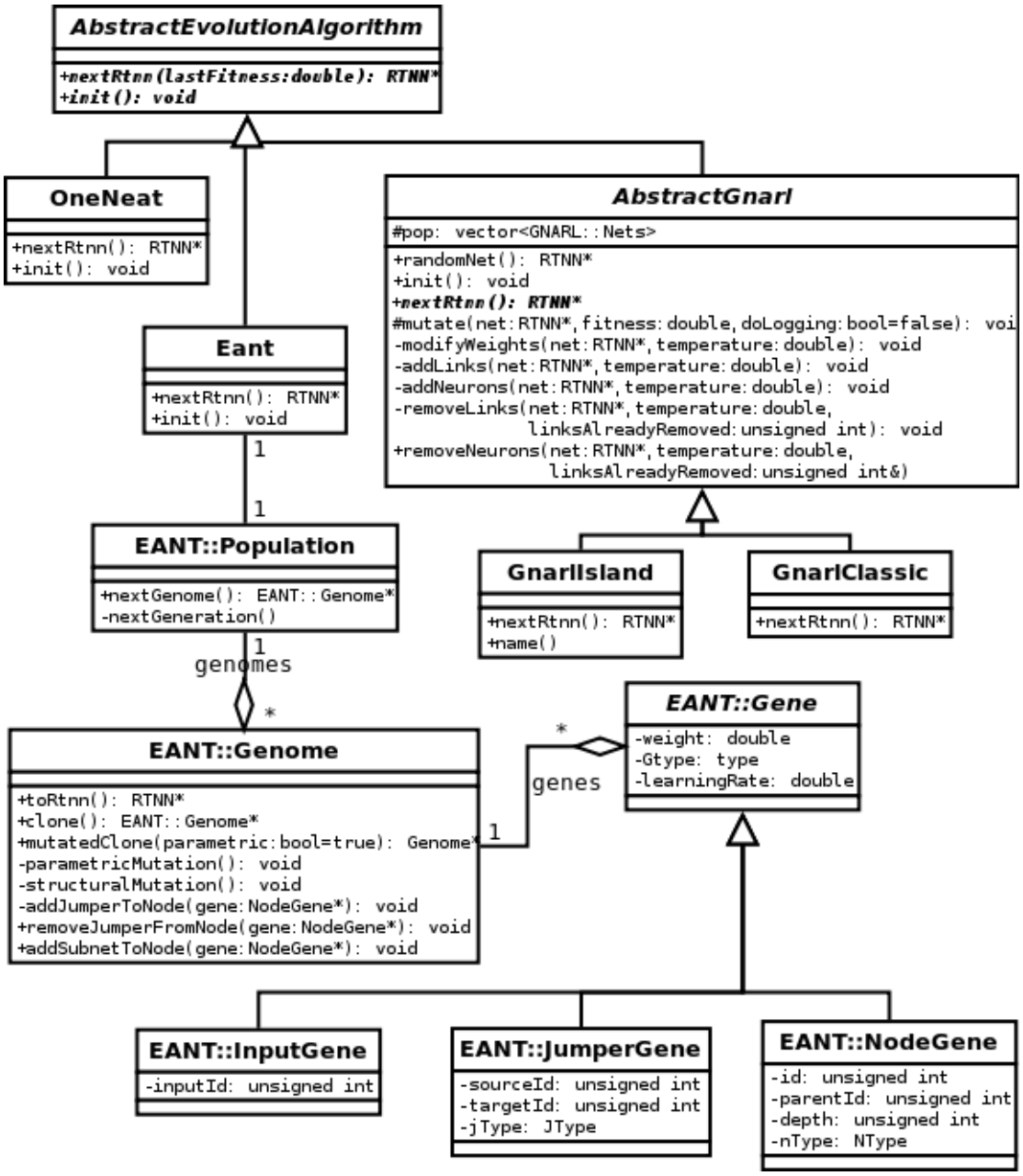


Figure 5.4.: UML Class Diagramm for Evolutionary Algorithms (without classes for OneNeat).

AbstractEvolutionAlgorithm

`AbstractEvolutionAlgorithm` is the abstract base class used by all implementation of the framework's evolutionary algorithms. Defines some attributes and setters and getters for them that are common for all algorithms like population size, number of input and output neurons and the activation function that will be used for hidden and output neurons.

Methods:

- `nextRtnn()`: Returns the next network that's to be evaluated by the framework. Parameter is the fitness of the last network that was evaluated.
- `init()`: **Must** be called before using an algorithm. Initializes the algorithm's population.

AbstractGnarl

`AbstractGnarl` is the abstract base class of `GnarlClassic` and `GnarlIsland` and a child of `AbstractEvolutionAlgorithm`. Both algorithms use the same mutations and initialize networks the same way. The difference is in the way they choose the next network to be evaluated. The methods in this class all work as described by the GNARL algorithm.

Attributes:

- `pop`: Population of networks. Type is a struct with two fields: a `RTNN` and a double value for the fitness it gathered during its last evaluation.

Methods:

- `randomNet()`: Creates a random network for use in the GNARL algorithm. See section 4.2.1.
- `mutate()`: Mutates a network in the following order:
  1. removing neurons,
  2. removing links,
  3. modifying connection weights,
  4. adding links,
  5. and adding neurons.

  Removing is done before adding so no links are added to neurons that will be removed. As removing neurons also removes links associated with them

removing links is done afterwards so that the number of links removed before can be taken into account. Adding neurons comes after adding links so that no new links are added to new neurons as stated in the GNARL algorithm. The number of structural modifications for each type of modification is determined by equation 4.5.

- `modifyWeights()`: Modifies connection weights (see equation 4.4).
- `addLink()`: Adds a number of connections between pairs of unconnected neurons. Start and end of connection can be the same neuron. Initial weight of connection is zero.
- `addNeuron()`: Adds a number of unconnected neurons to the net.
- `removeLink()`: Removes a number of connections from the network. The number of connections that were removed by `removeNeuron()` is subtracted from this number.
- `removeNeuron()`: Removes a number of neurons and all their incoming and outgoing connections from the network.

GnarlClassic

`GnarlClassic` is an implementation of Gnarl as decribed in section 5.3.2.

Methods:

- `nextRtnn()`: Returns the next network that's to be evaluated. If all networks of a generation were evaluated the next generation is created by cloning and mutating the selected parents. The next network is then the first network of the new generation.

GnarlIsland

`GnarlIsland` is a modification of `GnarlClassic` replacing its selection scheme. Instead of selecting each generation, selection is done after each evaluation.

Methods:

- `nextRtnn()`: Returns the first member of the population. After evaluation, the network is mutated and placed at the end of population. The network with lowest fitness (if it exists) is deleted and a mutated clone of the evaluated network is also placed at the end of the population.

Eant

`Eant` is used by the framework to access the EANT algorithm (see section 4.4). A genome of EANT could be interpreted as a tree-based program [**?**] but for use by the framework instead it is converted to a `RTNN`.

Methods:

- `nextRtnn()`: Returns the current genome as a `RTNN`.
- `init()`: Creates a new `EANT::Population` object with the parameters for evolution.

EANT::Population

`EANT::Population` contains and manages all genomes used by the EANT algorithm.

Methods:

- `nextGenome()`: Returns the genome that is to be evaluated next. If all genomes of a generation were evaluated calls `nextGeneration()` and returns the first genome of the next generation instead.
- `nextGeneration()`: Creates the next generation of genomes. Which mutations are used depends on the phase the algorithm is in. During *explotation* parametric mutations are used, during *exploration* structural mutations are used.

EANT::Genome

`EANT::Genome` represents a single member of EANTs population. It provides methods for mutating, cloning and conversion to a `RTNN`.
Contains a vector of pointers to `NodeGene` in addition to the vector of pointers to Genes. The index in the node vector is the unique Id of `NodeGene`. This simplifies a lot of operations of this algorithm because shallow copies can be used for genes and knowing the id of a node gene is enough to gain access to it.

Methods:

- `toRtnn()`: Converts this genome to a `RTNN`. First creates an input neuron for each input of the network and then calls a utility method `parseNeuron` that recursively creates a `Neuron` for each `NodeGene` and the links implied by the node genes. Afterwards traverses the genes in reverse order. All encountered Genes are put on a stack. If a `NodeGene` is encountered a number of Genes

is removed from the stack equal to its number of inputs before putting it on the stack. For `InputGene` removed from the stack a connection from the corresponding input neuron is added and for each jumper gene a connection between the corresponding neurons is added.

- `clone()`: Creates and returns a deep copy of the genome (for all pointers new objects are created).
- `mutatedClone()`: Creates a clone, mutates and then returns it. Type of mutation depends on a boolean parameter `parametric`. If true uses `parametricMutation()` and else uses `structuralMutation()`.
- `parametricMutation()`: Applies parametric mutation as described by equation 4.14 to the genome.
- `structuralMutation()`: For each node gene in the genome a random number between 0 and 1 is generated. If it is smaller than a predefined threshold randomely selects on of the structural mutations and applies it to the gene.
- `addJumperToNode()`: Adds a jumper connection between a `NodeGene` and another randomely selected `NodeGene` the first one is not yet connected to.
- `removeJumperFromNode()`: Removes a random jumper connection from a `NodeGene`.
- `addSubnetToNode()`: Adds a `NodeGene` to a `NodeGene`. For each network input a random number is generated and if it is lower than a threshold an `InputGene` is added to the new `NodeGene`.

EANT::Gene

`EANT::Gene` is an abstract base class for genes in the EANT algorithm.

Attributes:

- `weight`: Weight of conncection implied by this gene.
- `gType`: Type of gene (node, jumper, input).
- `learningRate`: Learning rate for this gene.

EANT::NodeGene

`EANT::NodeGene` is both a node gene and a connection to the parent of the node gene.

Attributes:

- `id`: Id of node gene and index in the genome's vector of node genes.
- `parentId`: Id of parent node gene.
- `depth`: Depth in the tree structure of the genome.
- `nType`: Type of neuron (hidden or output).

EANT::JumperGene

`EANT::JumperGene` represents a recurrent or forward connection between two node genes.

Attributes:

- `sourceId`: Id the connection's source node gene.
- `targetId`: Id the connection's target node gene.
- `jType`: Type of connection (forward or recurrent).

EANT::InputGene

`EANT::InputGene` is a connection from an input of the network to a node gene.

Attributes:

- `inputId`: Specifies which input this gene corresponds to.

OneNeat

`OneNeat` is an interface class to an implementation of the ONE-Neat algortihm[oT10]. The main differences to NEAT (see section 4.3) are:

- In NEAT, Genomes consists of Genes. ONE-Neat introduces another layer between genome and genes. A genome now consists of chromosomes that contain genes. Figure 5.5 shows a genome and its mapping to a phenotype. Chromosomes change the way mutations work. They have evolvable modifiers that affect mutation of genes within them. Also their is a new mutation

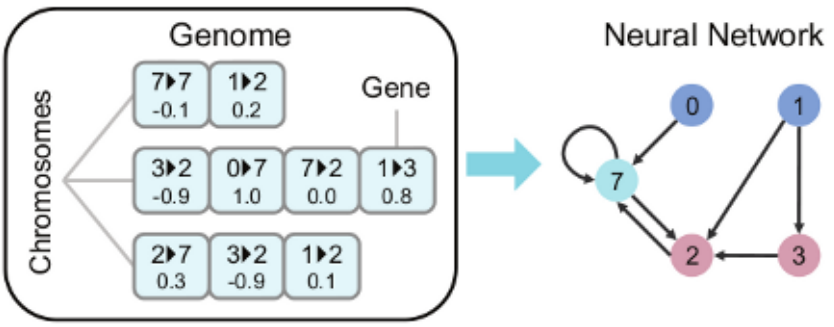


Figure 5.5.: ONE-Neat genome and mapping to phenotype[oT10].

that moves genes between chromosomes. Recombination works differently for genes on a single chromosome than that of genes on different chromosomes.

- ONE-Neat does not use innovation numbers to check if a mutation already occured. Instead it uses the source and target id of neurons. New neurons are assigned a random number as id and id is inheritable. This increases the likelihood that two neurons with the same id are indeed the same innovation.

5.3.3. Controller Classes

These classes control the behaviour of the robot using the classes presented before.

RobotController

`RobotController` is a child of `ControllerComponentBase` receiving an event message every tick of the simulation. It is the main object for evolution and evaluation on a robot.

Attributes:

- `driver`: Object responsible for steering the robot.
- `ticksPerDriver`: Number of ticks a single driver or net is evaluated.
- `stopEvolutionFitness`: If a controller's fitness exceeds this value, evolution is stopped.
- `stopEvolutionGeneration`: If the population's generation exceeds this value, evolution is stopped.

Methods:

- `initialize()`: Uses the parameters from the parameter hashmap to initialize this class and `driver` and to retrieve and initialize an object of the used evolutionary algorithm with `EvolutionAlgorithmFactory::getEvolutionAlgorithm()`.
- `ProcessMessage()`: On a tick message passes control of the robot to `driver`. If `ticksPerDriver` ticks have elapsed calls `nextRtnn()` of the evolutionary algorithm object. If a robot gathered `stopEvolutionFitness` or more fitness, reevaluates instead if it was set in the scene file.
  Robots are reset to the starting position in the scene file if it was set in the scene file.
- `RobotFoundFood()`: This method is called by `GlobalFoodController` if a robot found food. Finding food is currently the only built-in way for a robot to gain fitness.

AnnDriver

`AnnDriver` is responsible for steering the robot using an `RTNN`. It reads sensor values, possibly scaling and shifting them before feeding them into the controller network. After a single step of the network reads the network's outputs, also possibly scaling and shifting them and then feeds them into the robots actuators. If users want to add VirtualSensors or otherwise change the robots behaviour they should change this class.

Methods:

- `drive()`: Method called by `RobotController`. Steering is done here.

VirtualSensors

`VirtualSensors` is a container for static methods working on the output of the robot's sensors. Much preprocessing is possible, from counting pixels to edge detection or even object identification. Currently there is only a method for counting pixels belonging to food objects in images from a camera sensor.

Methods:

- `SenseFood()`: Counts purple pixels. Those most propably belong to food objects. Maps the number of pixels to a specified range (standard is [0,1]).

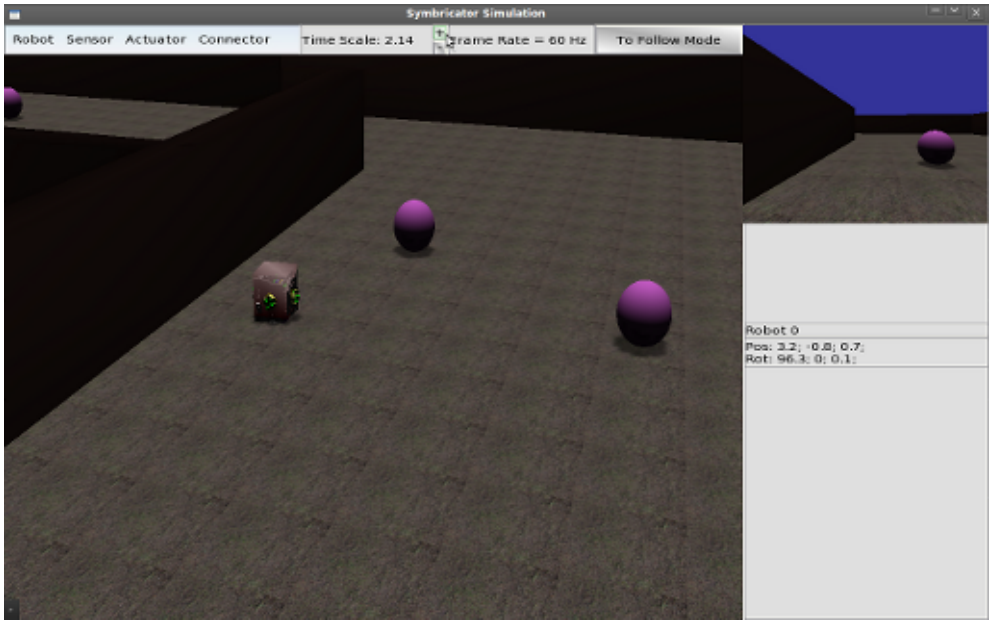


Figure 5.6.: Screenshot of the simulation with a robot and some food objects.

5.3.4. Component Classes

Component classes are classes that affect the global behaviour of an experiment, not just a single robot. At the moment there is a food controller and a class for adding components to the game manager.

StartPlugin

The plugin library defines a function `CreatePluginEntryPoint()` that is called by the simulation to load the plugin. It returns a singleton instance of `StartPlugin`.

Methods:

- `StartPlugin()`: This method is called when the plugin is loaded. Initialization code can be placed here. At the moment adds a `GlobalFoodController` to the game manager.

GlobalFoodController

`GlobalFoodController` is a child of `dtGame::GMComponent` and adds violett balls to the scene that can be gathered by robots as food. Figure 5.6 shows two 'balls of food'. The dimension of the map currently used are those of 'HoverMap'.

Methods:

- `GlobalFoodController()`: Initializes a number of random locations where food will be placed.
- `ProcessMessage()`: Checks if any robot is near enough to a food object. If yes, calls the robots's controller's `RobotFoundFood()` method, removes the food object and adds a new one to a random location.

6. Experiments and Results

This chapters contains evaluations of the evolutionary algorithms. They were tested with the double pole balancing benchmark, a standard benchmark for controllers. No results for test within the simulation are available because it takes much too long to get comparable results. Each evaluation within the simulation would take several seconds to minutes, each experiment was repeated thirty times and there were hundreds of experiments.
Table 6.1 shows the parameters that were used in most of the experiments. The number of input and output directly result from the Pole Cart requirements, population size and activation function are choosen intuitively. Every experiment regardless of other parameters was repeated 30 times.

population size	250
activation function	linear
input nodes	3
output nodes	1

Table 6.1.: Common parameters for all evolutionary algorithms in Pole Cart experiments.

6.1. Double Pole Balancing Without Velocities

The double pole balancing is a standard benchmark for controllers[KS05]. The task is to uphold both poles by applying force to the cart they are hinged on. The only information available to the controller is the offset from the center x, the pole angles θ_1 and θ_2. The output of the controller is taken as the force F that is applied to the cart.
The task fails if offset from the center is to great or the poles fall down. If the controller manages to uphold the poles for 100,000 cycles he succeeds.

6.1.1. EANT

There were several batches of experiments with EANT. All took the parameters from table 6.2 expect were something else is mentioned. An explanation of those parameters can be found in section 4.4.

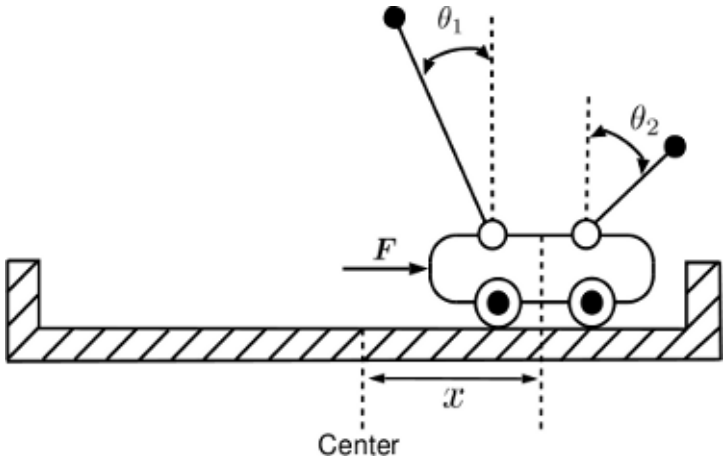


Figure 6.1.: Image showing double pole balancing problem. F is the force applied to the cart, θ_1 and θ_2 are the angles of the poles and x is the offset from the center. [KS05]

M	80
N	20
μ	10
λ	20
inital connection propability	1.0
ϵ	0.1
mutation propability	0.5
global learning rate	0.5

Table 6.2.: Parameters for experiments with EANT.

Varying Starting Hidden Nodes

In the first batch the number of hidden nodes at the start of evolution was varied from 0 to 9. Table 6.3 shows the result of these experiments. Figure 6.2 visualizes these results.
As we can see EANT starting with only one hidden nodes performs best in this comparison. A propable explanation would be that one hidden node is close to the number of hidden nodes required by this task and the search space is less complex than for more hidden nodes (remember hidden nodes all come with connections to inputs and their parents).
One interesting aspect is that the first structural mutation occurs after the $\mu * \lambda * N = 4000 - th$ generation. This signifies that the task requires at least one hidden node for even the best evaluation with zero starting nodes takes more evaluation than this number and that also explains the high number of evaluations for zero

hidden nodes at start. But we can also learn that this task requires no additional connection between nodes expect those that exist at the beginning because experiments with one or more hidden nodes at the beginning find a solution without structural mutation.

hidden nodes	min	max	mean	median
0	4815	73055	14910	9843
1	1616	5081	3111	2917
2	1974	8731	3678	3215
3	1818	12754	4297	3690
4	1611	11338	4251	3567
5	1804	12418	5195	5317
6	1821	9329	4527	4483
7	1629	17291	5487	5124
8	2284	12172	5742	5250
9	675	10640	4694	4595

Table 6.3.: EANT experiments with various number of hidden nodes.

Scaling the Network's Output

In contrast to the logistic function that is limited to an output in $[0, 1]$ or the hyperbolic tangent that is limited to one in $[-1, 1]$ the linear function is not limited. In theory weights of connections to the output neuron could scale the output as required. But of course they have to learn to do it. In these experiments the effect of scaling the output was viewed. The experiments were done for scalings of 1, 2.5, 4, 5.5, 7, 8.5 and 10. Each scaling was used for 0 ... 9 hidden nodes. Table 6.4 shows the results for one hidden node, the other results can be found in appendix A.1.1. Figure 6.3 shows the results for one hidden node as a graph.

A scaling of 1.0 clearly performs worse than the other scalings while a scaling of 4.5 seems to perform best. The median of a scaling of 4.5 is slightly higher than greater scalings but minimum and maximum evaluations are closer to the mean or median.
The weight of initial connections is in [-1, 1] and the input is scaled to the same range. As we have one hidden node that is connected to the output at the beginning, the output can't be greater than 3 for starting nets, and that number is highly impropable (assuming all inputs and all links have a value of 1). The actual output would be much closer to zero as weights center around zero. Learning rate limits the growth of connection weights and thereby is slowing down the algorithm to find the right scaling. Scaling the network output helps because it reduces the total

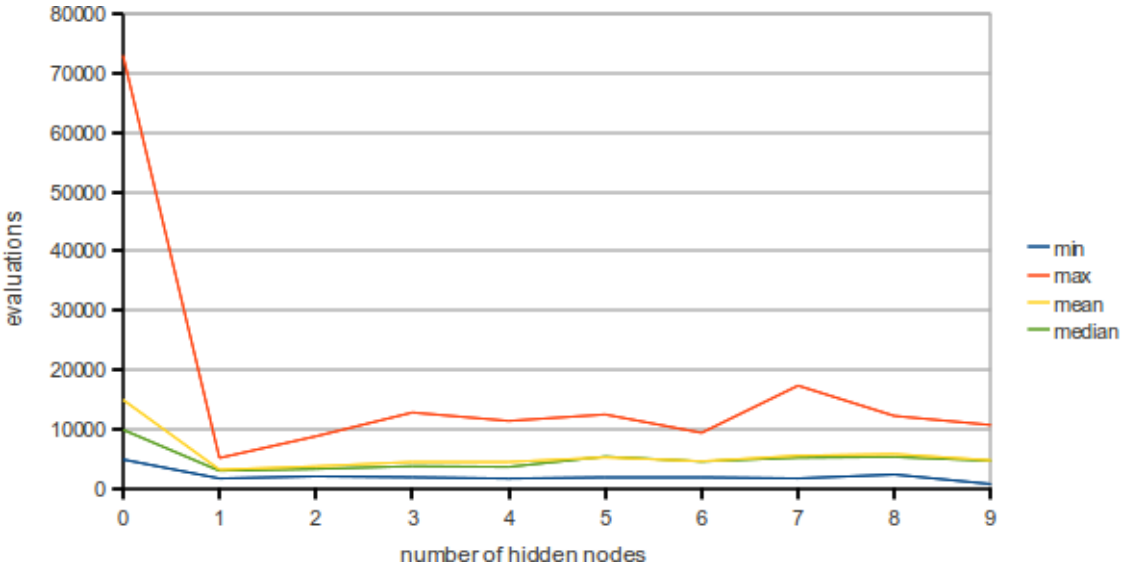


Figure 6.2.: Graph with number of evaluations over number of hidden nodes for EANT.

value of adaptions to weights needed to be done by the algorithm and the algorithm just has to find the right proportionality between weights.
This setup was repeated with hyperbolic tangent as activation function. The difference between the results for both activation functions are negligible. The table with the results can be found in A.1.2.

scaling	min	max	average	median
1	1773	5978	3167	2870
2.5	815	2413	1428	1448
4	576	2370	1040	1015
5.5	248	1213	794	821
7	416	1611	772	707
8.5	436	1877	851	794
10	228	1433	638	618

Table 6.4.: Results for EANT with one hidden nodes and varying scaling of network output.

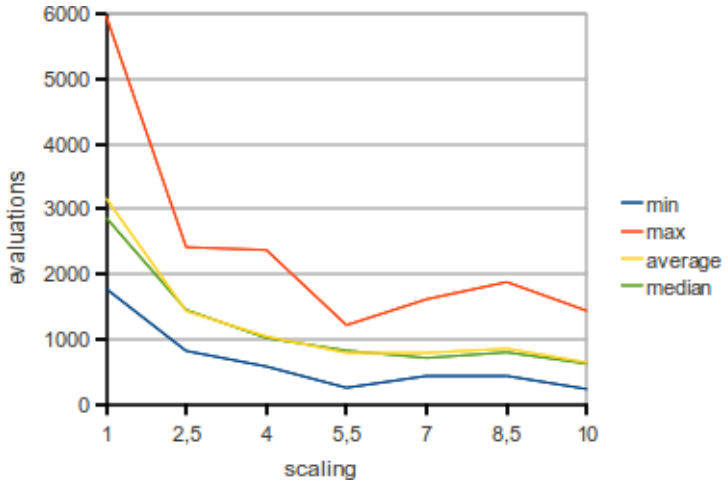


Figure 6.3.: Graph with number of evaluations over scaling of output for EANT.

Varying μ and λ

In this batch of experiments μ and λ were varied. Figure 6.5 shows a matrix of the medians. This matrix is visualized in figure 6.4 and 6.5.
The whole data for this batch can be found in A.1. The total population size in this batch was set to $\mu * \lambda + 30$ and scaling was again 4.5.
The results of this batch were a bit unexpected. Expected was that the performance of larger populations would be better than that of smaller populations. Instead performance worsens with growing μ for each tested λ and vice versa. The relation of $\mu * \lambda$ to population size is not very likely to be an explanation cause most algorithms are finished well before N generations and population size is irrelevant before the N-th generation.
There could be a high propability that a starting net with one hidden node is close to a solution, with a large population size evolution would then waste time on useless networks and on evolving several good solutions. That could also the high maximum value for $\mu = 2$ and $\lambda = 2$. For such a small population there is then also a propability that no starting net exist that is close to a solution.
The reason for lower λ's to perform better than higher λ's could be variance in the population. Lower λ means that a higher proportion of the population create less offspring keeping more solutions that are not as good in their current state but may perform much better after some adaptions. With a higher λ solutions that perform well in the current generation suppress solutions that might be better later on.
This leads to the conclusion that there has to be a balance between μ and λ and that in general *lambda* should be rather low. If this is also true for μ remains unclear, it might depend on the propability that there is a network close to a solution in the starting population.

		μ				
		5	10	15	20	25
λ	2	416	472	532	661	731
	10	408	749	937	979	1301
	18	511	920	1184	1476	1836
	26	680	1072	1471	1771	2033
	32	814	1269	1646	2309	2612

Table 6.5.: Matrix showing the median of evaluations with varying μ and λ.

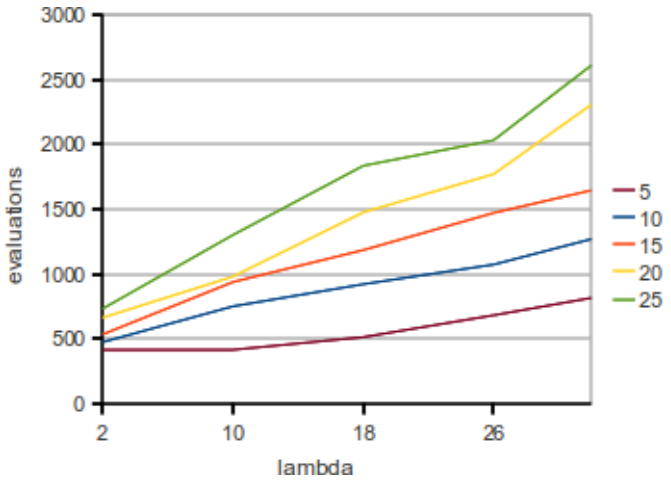


Figure 6.4.: Visualizing performance of EANT over μ for various λ's.

6.1.2. GNARL

Table 6.6 shows that were used in most of the experiments with both the classic GNARL and the island variant.

Varying Minimum and Maximum Number of Hidden Nodes and Links

Tables 6.7 and 6.8 show the best and worst three results for the GNARL algorithms with varying minimum and maximum number of hidden nodes and links. The complete tables can be found in appendix A.2.1 and A.2.2. There are a number of things that can be learned from analyzing the results.
First, GNARL, in both variants, is a highly random algorithm, minimum and maximum number of evolutions are far apart and there are almost always outliers that are far of from the median.
And second, GNARL is destructive. The experiments that performed best (marked gray in the table) had a cushion of excess links and hidden nodes at the beginning,

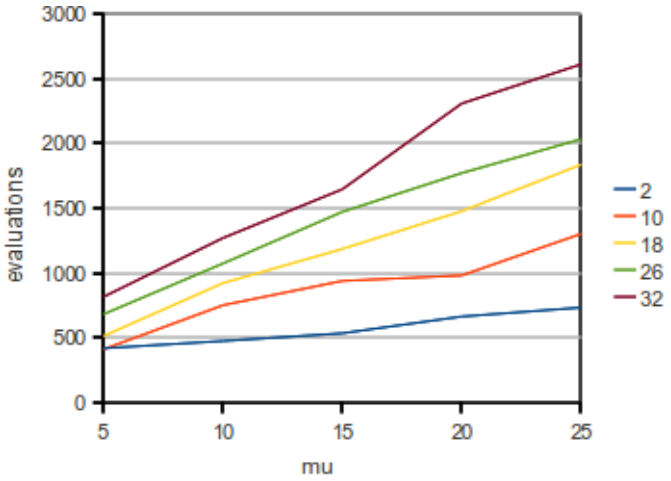


Figure 6.5.: Visualizing performance of EANT over λ for various μ's.

MinLinksRemove	1
MaxLinksRemove	5
MinLinksAdd	1
MaxLinksAdd	5
MinNeuronsRemove	1
MaxNeuronsRemove	3
MinNeuronsAdd	1
MaxNeuronsAdd	3
LikelihoodParam	0.2
Alpha	1
MaxFitness	100000

Table 6.6.: Parameters for experiments with GNARL.

far more than needed for a solution.
Interestingly, the island variant outperforms its classic predecessor by a large margin. Both the median and the maximum number of evaluations are clearly lower in Island GNARL. As the only difference between both algorihtms is the scheme for selecting networks that will be replaced the scheme must be the reason for the difference in performance. It seems that the more agressive scheme works better for GNARL.

minimum hidden nodes	maximum hidden nodes	minimum links	maximum links	min	max	mean	median
3	6	7	14	250	52249	8511	407
4	8	8	16	259	10739	961	440
2	4	6	12	261	55389	16280	9293
1	2	1	2	1518	49246	25974	27567
1	2	1	2	3220	61500	27216	27747
3	6	0	0	1591	43751	24580	28248

Table 6.7.: Best and worst three results for Classic GNARL.

minimum hidden nodes	maximum hidden nodes	minimum links	maximum links	min	max	mean	median
3	6	7	14	250	52249	8511	407
4	8	8	16	259	10739	961	440
2	4	6	12	261	55389	16280	9293
1	2	1	2	1518	49246	25974	27567
1	2	1	2	3220	61500	27216	27747
3	6	0	0	1591	43751	24580	28248

Table 6.8.: Best and worst three results for Island GNARL.

Varying Population Size

In this batch of experiments the population size was varied from 30 to 300. The minimum number of hidden nodes was set to 2, the maximum number to 4 and the minimum number of links was set to 6, the maximum number to 12. The tables with the results can be found in appendix A.2.3.
Two interesting observations can be derived from those results. The population size does not really affect the performance of GNARL in this context and in every experiment at least one run found a solution in the second generation. This can be only be explained by a high propability that there is a good solution in the initial population.

6.1.3. ONE-Neat

The implementation of ONE-Neat has two parameteres that can be set from the outside, a boolean meta-mutation and an integer chromosome count. Experiments with a chromosome count of one and meta mutation of false yielded no result. All experiments stopped at the same fitness value (2933). The experiments were re-

peated with meta-mutation turned on and a chromosome count from 2 to 5. Table 6.9 shows the results.
The best result was obtained for experiments with a chromosome count of three. Presumably because this strikes the right balance between the effects of meta mutation and the increase of the search space that comes with additional chromosomes (more chromosomes, more parameters to optimize).

chromosome count	**min**	**max**	**mean**	**median**
2	4669	42627	15910	11617
3	4815	32289	13597	9903
4	4810	67014	13725	9335
5	4856	86173	19529	12420

Table 6.9.: Results of experiments with ONE-Neat.

7. Conclusion and Perspectives

With the framework it is now easier to create experiments in neuroevolution using delta3D and the symbrion simulation. It provides the three algorithms EANT, ONE-Neat and GNARL including a variant of plus the implentation for neural networks to Also the simulation was extended to set custom parameters for controllers and outfit robots with arbitrary sensors. Together this allows creation of experiments with higher complexity than before and moves time and effort from implementing infrastructure to implementing the experiments. Still there is a lot of work that could be done. That includes

- machine learning algorithms for neural networks for comparison with evolutionary algorithms and combining evolution and life-learning,
- further evolutionary algorithms and
- a mechanism to parse the arena and add random objects to it for on the fly creation of mazes, adding obstacles or any other way to change the environment e.g. to test generalization of a solution.

even Also there is much that can be done regarding the evolutionary algorithms. Finding fitting parameters for them was not easy since it was badly documented in most of the papers. An early conclusion that can be drawn from the experiments is that the number of hidden nodes (and connections) at the beginning of evolution has a huge impact on performance. It would be interesting if general rules for chosing this number exist. Some thought should also be put into how to use the networks output. Clearly just feeding the output into the robot's actuators doesn't work. They should either be scaled into the range accepted by the actuator or maybe such a scaling should even be evolved alongside the neural network.
In the evaluation of evolution it showed that GNARL is a highly random algorithm. Owing to this randomness it is sometimes a lot faster and sometimes a lot slower than EANT or ONE-Neat, although it seems that the new variant reduces this effect by a huge margin. For more complex tasks EANT is most propably better suited since it is less random and therefore the effort used to find a solution stays a lot more predictable and within bounds.
For the future a benchmark that can only be solved by a larger network (at least ten or twenty neurons) should be devolped. Only such a benchmark will give a truly reliable answer to which algorithm performs best. The usefulness of a benchmark like polecart is limited since random starting nets can come close to solving the

problem posed by it.
A comparison of the selection scheme used by Island GNARL to the more traditional schemes like those employed by GNARL or EANT would be interesting. Does it only improve performance in that special case or does it work better than the other schemes in general?
There are a many more questions that need answers and the hope of the author is that his work helps a little in finding answers to them.

A. Results of Experiments

A.1. EANT

A.1.1. Varying Hidden Nodes and Scaling

scaling	hidden nodes	min	max	mean	median
1	0	4822	66168	13919	12842
	1	1773	5978	3167	2870
	2	1955	10547	3748	3051
	3	1426	9333	3835	3559
	4	1828	12763	5996	5552
	5	1662	10544	3948	3433
	6	1862	15283	5523	4814
	7	1124	11744	5005	4517
	8	1582	11512	4662	4421
	9	1707	11438	4937	4269
2.5	0	4629	46110	9252	6166
	1	815	2413	1428	1448
	2	728	2557	1468	1419
	3	816	4049	1680	1477
	4	815	2699	1561	1550
	5	959	5244	2097	1778
	6	465	8016	2118	1851
	7	1092	6309	1984	1690
	8	627	6214	2179	1858
	9	810	4870	2686	2375
4	0	4359	29262	8430	5277
	1	576	2370	1040	1015
	2	417	1861	1003	942
	3	591	2804	1263	1134
	4	544	3481	1363	1236
	5	758	2786	1323	1249
	6	625	2962	1411	1250
	7	516	3790	1572	1337
	8	421	3663	1750	1534
	9	667	4110	1829	1601

scaling	hidden nodes	min	max	mean	median
5.5	0	4217	33416	10838	8633
	1	248	1213	794	821
	2	284	1316	905	922
	3	225	1835	985	975
	4	281	2444	1122	1004
	5	469	4648	1350	1218
	6	500	2992	1331	1263
	7	741	3245	1432	1288
	8	227	2688	1172	1108
	9	298	4018	1476	1325
7	0	4069	24819	7759	6816
	1	416	1611	772	707
	2	235	1449	760	676
	3	241	2033	828	759
	4	214	2557	1006	934
	5	433	2812	1112	956
	6	366	2068	1112	1104
	7	445	2091	1178	1073
	8	410	3363	1204	1017
	9	442	3472	1443	1337
8.5	0	4130	21054	6460	4781
	1	436	1877	851	794
	2	258	1655	807	768
	3	225	1913	931	861
	4	423	2610	1081	982
	5	244	2988	1026	838
	6	219	2101	974	860
	7	433	3326	1097	1021
	8	287	4396	1231	1075
	9	348	4223	1169	1014
10	0	4116	16529	6507	4755
	1	228	1433	638	618
	2	222	1637	777	678
	3	471	1845	913	866
	4	431	1613	891	858
	5	443	2144	1037	842
	6	291	2854	1129	1046
	7	283	2592	1109	1027
	8	335	3422	1241	1017
	9	246	4056	1251	1119

A.1.2. Varying Hidden Nodes and Scaling using Hyperbolic Tanget as Activation Function

scaling	hidden nodes	min	max	mean	median
1	0	4628	46239	12567	9735
	1	1813	6619	3635	3519
	2	1045	6508	3682	3427
	3	1877	8577	4292	4237
	4	2024	8830	4714	4854
	5	1460	10616	4160	4125
	6	1653	13673	5703	5091
	7	1848	13424	4993	4339
	8	1220	10806	4938	4561
	9	1873	9748	5062	4577
2.5	0	4451	64639	11707	8876
	1	428	2700	1422	1363
	2	875	2447	1544	1453
	3	611	2315	1576	1632
	4	560	3612	1754	1702
	5	633	6613	2056	1670
	6	768	4231	1817	1778
	7	857	4690	2253	1985
	8	865	6248	2334	1967
	9	1055	6988	2586	2378
4	0	4299	27212	9278	4925
	1	619	1476	1046	1037
	2	429	1426	948	914
	3	615	1887	1081	1059
	4	639	1825	1059	963
	5	413	4915	1393	1161
	6	506	3136	1412	1304
	7	651	4321	1686	1439
	8	614	3113	1525	1378
	9	611	3312	1611	1363

scaling	hidden nodes	min	max	mean	median
5.5	0	4211	25017	9604	8445
	1	442	1387	880	868
	2	475	3211	1078	983
	3	361	1777	1015	903
	4	513	2467	1129	1022
	5	414	2110	1050	1011
	6	547	2746	1194	1058
	7	451	2855	1347	1180
	8	411	3707	1338	1246
	9	643	2885	1437	1422
7	0	4129	33766	9447	4990
	1	214	1431	762	734
	2	417	1950	914	849
	3	390	3686	1005	823
	4	240	1956	1025	979
	5	434	1947	898	799
	6	451	2219	1177	1023
	7	443	4280	1142	973
	8	288	2685	1288	1331
	9	285	3382	1135	1056
8.5	0	4213	28167	8246	5026
	1	241	1436	671	639
	2	308	1673	801	726
	3	226	2037	933	883
	4	211	1694	849	790
	5	293	2130	974	884
	6	411	2220	970	866
	7	425	2458	1089	946
	8	246	2016	973	921
	9	412	3163	1239	1089
10	0	4117	20611	6771	4665
	1	58	1613	671	640
	2	289	1506	746	807
	3	210	2420	795	779
	4	441	4411	1129	964
	5	358	2231	928	819
	6	327	2139	868	856
	7	460	2456	1178	1073
	8	269	2761	1198	1035
	9	405	1939	973	889

A.1.3. Varying μ and λ

μ	λ	population size	min	max	mean	median
5	2	30	119	4566	695	416
	10	70	171	1505	506	408
	18	110	220	1153	547	511
	26	150	413	2043	748	680
	32	180	346	1700	835	814
10	2	40	211	1499	488	472
	10	120	311	1238	698	749
	18	200	261	1451	936	920
	26	280	356	1605	1008	1072
	32	340	699	2400	1278	1269
15	2	50	226	1315	598	532
	10	170	652	1537	973	937
	18	290	608	2207	1183	1184
	26	410	1065	2100	1524	1471
	32	500	1071	2442	1731	1646
20	2	60	384	1954	770	661
	10	220	221	1788	1029	979
	18	380	396	1955	1374	1476
	26	540	1220	3247	1897	1771
	32	660	1393	3478	2337	2309
25	2	70	328	1492	767	731
	10	270	591	2078	1317	1301
	18	470	983	2775	1827	1836
	26	670	764	3619	2154	2033
	32	820	1011	3700	2592	2612

A.2. GNARL

A.2.1. Varying Starting Hidden Nodes and Links for Classic GNARL

minimum hidden nodes	maximum hidden nodes	minimum links	maximum links	min	max	mean	median
0	0	0	0	2771	60250	21730	18476
		1	2	2274	105750	32123	25868
		2	4	1635	92210	24781	17400
		4	8	1372	276004	32405	23722
1	2	0	0	1591	50499	20437	20559
		1	2	1356	75250	24189	19814
		2	4	1723	75998	24673	18664
		5	10	850	71494	24406	22388
2	4	0	0	2330	62400	26056	24371
		1	2	3220	61500	27216	27747
		2	4	1267	125250	27478	20969
		3	6	477	103501	28381	25657
		6	12	261	55389	16280	9293
3	6	0	0	1591	43751	24580	28248
		1	2	1518	49246	25974	27567
		2	4	530	80513	21451	15588
		3	6	350	175751	31346	17641
		7	14	250	52249	8511	407
4	8	0	0	2253	69748	21896	17687
		2	4	601	65854	25751	25867
		4	8	288	56250	19921	13809
		8	16	259	10739	961	440

A.2.2. Varying Starting Hidden Nodes and Links for Island GNARL

minimum hidden nodes	maximum hidden nodes	minimum links	maximum links	min	max	mean	median
0	0	0	0	1811	43433	15077	10613
		1	2	2219	44677	14939	11601
		2	4	4477	44746	15171	11026
		4	8	2489	43481	13316	11015
1	2	0	0	1966	47307	15470	9656
		1	2	448	29848	10121	7701
		2	4	252	45250	11947	8405
		5	10	318	44120	9467	7596
2	4	0	0	1900	55137	15319	10189
		1	2	1635	94074	13308	8208
		2	4	256	39792	10304	5866
		3	6	354	29490	5535	1314
		6	12	258	16415	1160	486
3	6	0	0	887	34735	10150	7675
		1	2	730	39876	10775	8253
		2	4	253	33093	7502	3074
		3	6	402	26808	7519	2861
		7	14	250	714	391	335
4	8	0	0	1422	42105	13137	9794
		2	4	285	93985	17244	10484
		4	8	253	36011	4035	703
		8	16	256	2367	462	325

A.2.3. Varying Population Size for Classic GNARL

population size	min	max	mean	median
30	36	78357	17213	12199
60	70	53046	13328	11908
90	92	45399	13534	13326
120	123	67649	14723	9958
150	175	56397	15253	10601
180	201	39062	15969	18024
210	225	62996	16153	13566
240	251	45596	9696	3710
270	273	39958	10459	3250
300	300	62996	11468	791

Bibliography

[ASP94] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5:54–65, 1994.

[BHE09] N. Bredeche, E. Haasdijk, and A.E. Eiben. On-line, on-board evolution of robot controllers. *Artificial Evolution*, 2009.

[Bra84] Valentino Braitenberg. *Vehicles: Experiments in synthetic psychology*. The MIT Press, 1984.

[Bro] Bob Brown. State machines.

[CCF+08] Soumen Chakrabarti, Earl Cox, Eibe Frank, Ralf Hartmut Gting, Jiawei Han, Xia Jiang, Micheline Kamber, Sam S. Lightstone, Thomas P. Nadeau, Richard E Neapolitan, Dorian Pyle, Mamdouh Refaat, Markus Schneider, Toby J. Teorey, and Ian H. Witten. *Data Mining: Know It All*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[CM06] Nikolaus Correll and Alcherio Martinoli. Towards optimal control of self-organized robotic inspection systems. 2006.

[DGF00] Natasha Balac Daniel, Daniel M. Gaines, and Doug Fisher. Using regression trees to learn action models. In *IEEE Systems, Man and Cybernetics Conference*, 2000.

[KS05] Y. Kassahun and G. Sommer. Evolution of neural networks through incremental acquisition of neural structures. Technical Report Number 0508, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Juni 2005.

[LCC+02] Docteur De L'inpg, Specialite Sciences Cognitives, RÃ©mi Coulom, M. Jean, Della Dora President, M. Kenji, M. Manuel, M. Stephane, Jorrand Directeur, Mme Mirta, and B. Gordon Examinateur. Reinforcement learning using neural networks, with applications to motor control, 2002.

[LS95] Dimitrios Lambrinos and Christian Scheier. Extended braitenberg architectures. Technical report, University of Zurich, 1995.

[NF00] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics : The Biology, Intelligence, and Technology of Self-Organizing Machines*. Cambridge, EUA : Massachusetts Institute of Technology, 2000.

[oT10] University of Tuebingen. Deliverable sd4.5: Prototype of virtual robot sexuality, February 2010.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence a modern approach*. Prentice Hall, Upper Saddle River, N.J., 2nd international edition edition, 2003.

[Roj96] Raul Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, 1996.

[SM] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:2002.

[sym10] Symbrion homepage (www.symbrion.org), 2010.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Michael Mattes)