NTNU
Norwegian University of
Science and Technology

Adaptive Robotics

Dag Henrik Fjær
Kjeld Karim Berg Massali

Master of Science in Computer Science
Submission date: June 2009
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Robots are commonplace in society; they perform all sorts of jobs. However, most are hard-wired to perform a few fixed tasks and have little or no ability to adapt to changing or unforeseen circumstances. Artificial Intelligence (AI) enters the picture when robots must be capable of autonomous behavior in unpredictable environments. In this project, the student must utilize AI machine-learning techniques to produce a robot with adaptive capabilities. Possible techniques include reinforcement learning (RL), artificial neural networks (ANNs), evolutionary algorithms (EAs), or combinations of these, such as evolving neural networks. A few e-puck robots are available in our laboratory, and these should be sufficient for building adaptive robots. Students who desire more advanced equipment will have to arrange for it themselves.

Assignment given: 15. January 2009
Supervisor: Keith Downing, IDI

Abstract

This report explores continuous-time recurrent neural networks (CTRNNs) and their utility in the field of adaptive robotics. The networks herein are evolved in a simulated environment and evaluated on a real robot. The evolved CTRNNs are presented with simple cognitive tasks and the results are analyzed in detail.

1 Preface

The task was given by the Department of Computer and Information Science (IDI) at NTNU, and more specifically our supervisor, Professor Keith Downing.

1.1 Problem definition

> Robots are commonplace in society; they perform all sorts of jobs. However, most are hard-wired to perform a few fixed tasks and have little or no ability to adapt to changing or unforeseen circumstances. Artificial Intelligence (AI) enters the picture when robots must be capable of autonomous behavior in unpredictable environments. In this project, the student must utilize AI machine-learning techniques to produce a robot with adaptive capabilities. Possible techniques include reinforcement learning (RL), artificial neural networks (ANNs), evolutionary algorithms (EAs), or combinations of these, such as evolving neural networks. A few E-puck robots are available in our laboratory, and these should be sufficient for building adaptive robots. Students who desire more advanced equipment will have to arrange for it themselves.

In addition to our work with adaptive techniques and solving minimally cognitive tasks, we would like to give a simple introduction to using the E-puck robots.

We would like to thank Professor Keith Downing for being a great source of inspiration, and for always making himself available. Thanks to Professor Randall D. Beer for explaining some of the key concepts, and for discovering several flaws in our approach. We would also like to thank Associate Professor Gunnar Tufte for providing us with new and interesting perspectives. Last, we wish to thank PhD Candidate Boye Annfelt Høverstad for all the work he did in order to get us started with the clustering.

Contents

List of Figures

List of Tables

2 Introduction

What are robots? There are no good, formal definitions of robots. In ancient Greece, around 2500 years ago, they developed some of the first machines that were able to perform actions in their environment. In the 15th century, Leonardo da Vinci created machines that were able to move around and perform simple tasks. However, it was not until 1948 that autonomous machines were able to perceive their environment [35]. If you take perception into account, we can consider these the first real robots, and they were created by William Grey Walter. The robots were called Elmer and Elsie, and both of them had three wheels that allowed them to navigate their environment. They were capable of photo taxis, which means that they could move towards a source of light, and therefore make their way to a charging station when they ran low on battery power. In one of his experiments, Walter put a light in front of one of the robots and watched as it observed itself in a mirror. "It began flickering, twittering and jigging like a clumsy narcissus", he wrote [35]. Walter also argued that this was evidence of self-awareness.

Evolutionary robotics is an imitation of natural evolution applied to robots. We can think of a robot as an organism. These organisms have not been designed, and they are the result of an evolutionary process.

Why would we want evolve robots? It is very difficult to design complicated behavior by hand. With many traditional approaches, one can easily run into certain difficulties. We will continue to give a brief example of one of these problem scenarios, and later we will return to it and look at the details. With direct programming, we can enable robots to perform various tasks, ranging from simple to complicated. However, as the complexity grows, they become almost impossible to maintain. Among the various traditional approaches is Brooks' Subsumption Architecture [1], which decomposes behaviors into simple modules. Each module is decomposed into a set of layers, and each individual layer implements a specific goal. Each layer's goal subsumes that of the underlying layers, e.g. the decision to move forward by the eat-food layer takes into account the decision of the lowest obstacle-avoidance layer. As the number of layers increases, the goals begin to interfere with each other, making the action selection completely random. This is a common problem for most direct programming approaches concerning autonomy - at some point they become very difficult to maintain.

Another problem with the direct programming approach is that its search space is limited to that of your own mind, whereas an adaptive approach is typically evolved, making the search space much larger. Real Darwinian evolution is not driven by any goals, but rather the resources and threats imposed by the environment. If the world is inhabited by dangerous carnivores, legs that enable an individual to escape by running away are likely to be beneficial. We are able to create the resources and threats in our artificial world such that we can guide the evolutionary process towards our goals. This in turn provides us with a selection of individuals that are likely to possess the traits we are looking for.

This brings us to the next question: How do these individuals determine what do to next, i.e. what defines their behavior? The model that encompasses our

action-selection and behavior is dynamical neural networks, more specifically Continuous-Time Recurrent Neural Networks (CTRNNs). We would like to examine how we can benefit from these networks, and whether they provide better solutions than the feed-forward network from our prestudy.

An interesting dilemma is that, in this day and age, we have extremely powerful hardware; state of the art computers are quickly catching up to human performance, in terms of processing power. Yet, many of the simplest cognitive tasks remain unsolvable for a computer, even for super computers. It is clear that processing power alone is not enough to make intelligent machines. If we assume that we have the processing power needed for cognitive tasks, what then, makes us unable to create machines that can solve cognitive tasks like the ones animals so trivially solve every day? The solution to that question probably lies in how we program our machines. Computers are only very fast calculators, the real cleverness of computers are found in the software, rather than hardware. All that the computer really gives us is a sandbox in which we can mold the behavior of the computer. Faster computers allows for more complex molds.

Since the computer is programmable, we can create our own universe. This enables us to model a replica of the real world in which we can play around in. In this simplified version of the world we can perform experiments on simulated robots. So far it all sounds good, but there is a catch. Simulated robots are no use if they only work in the toy universe on the simulator. For some experiments, using only the simulator is enough, but we want to develop robots that work in the real world too. And with more complex robots, this poses a serious challenge. The model of the real world inside the simulator will never be completely accurate when compared to the real world. Real world scenarios are extremely complex. Light, friction, noise and other phenomena found in the real world are truly random events. The robots have to be developed with that in mind, in order to tackle the real world randomness.

One way of trying to create robots that can have the possibility to solve cognitive tasks is to look at biology. Animals solve challenging tasks all the time. Anticipating predators, sensing dangers or space travel are all examples of tasks performed by non-human and human animals. All these tasks are solved by means of the animals' brains. A fully working animal brain is a huge network of interconnected cells and the topology is very complicated. However, the individual cells are very simple. These brains have evolved through Darwinian evolution over billions of years. The most powerful brains can be found in humans, but it seems that all brains are based on the same technology, namely neuroscience. Artificial brains, also known as artificial neural networks which are based on this brain theory started to become popular in the 1960s [13]. Advancements in brain theory will no doubt lead to advancements in brain inspired robot controllers.

Darwinian evolution (survival of the fittest) is currently the only model known to man, that is able to produce the level of adaptivity and intelligence of animals. These brains cope with the complexity of life and can quite accurately predict the future. This level of adaptivity is far beyond anything humans have ever made. Evolution as a developmental tool for robot controllers is therefore a very interesting subject to explore further.

A recent paper [27] reports of some problems with adapting robot controllers developed in a simulator on to the real world. Proper understanding of why simulator developed controllers struggle to cope with the real world, can lead to better robot controllers. In order to create robots that can work well in real life scenarios we need to improve our understanding of how to develop robot controllers: What technologies are suited, and what developmental tools can be used? When developing robot controllers in a simulated environment, how can we move the controller onto the real world? The controller itself has to be inherently adaptive in order to cope with the change of environments.

We are going to look at different possible models for creating robot controllers and further explore these in empirical experiments.

Many machine learning approaches require training examples as well as correct answers. In the case of robotics, it is sometimes hard to know a priori what decisions to make. An answer to the problem is to rate the actions taken by a robot, and use those the results to determine what to do next. An incremental search can take the form of an evolutionary process inspired by Darwinian evolution. What we evolve must not suffer the same limitations as direct programming approaches do, e.g. conflicting goals in selection mechanisms.

In the prestudy, simple feed-forward networks were evolved and applied to a set of problems, including one in which a robot was rewarded for finding its way out of a maze. Through these experiments, certain limitations became apparent, e.g. an encounter with the perceptual aliasing problem [28]. The perceptual aliasing problem occurs when two inputs are near identical, but the desired outputs are different. This is described further in the next section.

It could seem that our behavior space was too simple to achieve the behavior we were looking for. This continued effort tries to solve some of the problems, including the perceptual aliasing problem, through the use of dynamical neural networks. With these networks, we can take a few steps further in our pursuit of adaptive behavior. Real biological systems are often unpredictable, and there is a large variety of behaviors even within the same species. By evolving simple models of biological frameworks, we hope to discover some of the same diversity.

Intelligence is often measured in terms of behavior, though some argue that the ability to make predictions is a better measure [29]. After all, being able to predict changes in the environment is an important part of adaptivity. The dynamical neural networks used herein provide a form of prediction, and though we evaluate them based on their performance, the predictive ability indirectly determines which networks get selected and propagated into future generations.

We should also point out that it is not in our interest to create a black box that does the job for us, but rather a system that is more fault-tolerant and adaptive than what we are able to engineer by hand.

Throughout this report we would like to:

- Tackle the perceptual aliasing problem
- Handle the real world problem
- Further investigate evolutionary robotics

During the prestudy, the perceptual aliasing problem prevented us from finding efficient solutions. The feed-forward networks that we were using didn't seem capable of producing anything but very simple behavior. This is one of the problems that we would like to address by implementing a network structure with fewer constraints. See section 5.2.

The real world problem, in this context, describes the issues that might occur when transferring a solution that is evolved in a simulated environment onto the real world robot. These issues indicate differences between the simulated world and the real world, such as lighting conditions, sensors, friction and the batteries' effect on the robot. When using the simple network from the prestudy, the real world problem was not much of an issue. However, with the more complicated networks herein, this becomes a far more challenging task. This part of the task is important since it indicates whether this is a viable approach to solving real world problems.

In a general sense, we would like to continue investigating evolutionary robotics and some of its applications and limitations.

2.1 Report outline

This report is organized into the following chapters:

(2) Background. This chapter provides some background concerning adaptive robotics as well as a description of some of the existing technologies and approaches in the field. We try to get some concept of their shortcomings, and what we can learn from them.

(3) Design. This chapter contains a detailed description of the system that was developed for this report.

(4) Experiments. This chapter explains and analyses the results of the experiments herein, and what can be learnt from them.

(5) Conclusion.

3 Background

This is the continuation of a prestudy concerning adaptive robotics. In the prestudy, simple feed-forward networks were evolved and applied to a set of problems, including one in which a robot was rewarded for finding its way out of a maze. Through these experiments, certain limitations became apparent, e.g. an encounter with the perceptual aliasing problem [28]. The perceptual aliasing problem occurs when two inputs are near identical, but the desired outputs are different. Consider a robot that tries to find its way out of a maze: it is likely that it will face the wall several times (similar perceptual input, but at different times), but it is not desirable that it always reacts the same way. This specific problem is illustrated in figure 33 on page 56.

This section will try explore some of the state-of-the-art technologies and mention some of the more traditional approaches for robot controllers. In short we will have a look at known methods for robot controller design.

The following section is not meant to be a full list of possible ways to model a robot controller, but rather those we have been looking into in our exploratory research prestudy. We will also mention some important and relevant concepts in the fields of machine learning and AI.

Robot control defines the relation between the sensors and actuators in a robotics system. The four most commonly used approaches to robot control today are [22]:

(1) Deliberative control. This indicates that the robot uses all its stored internal states and sensor input in order to plan the next action. Typically it searches through all the available choices and picks the one that it considers to be most rewarding. If the robot has to respond quickly, this might not be a good approach. Also, if the robot finds itself in a dynamic environment, its actions will be less optimal if it spends too long thinking (due to the fact that the environment will be different by the time it acts).

(2) Reactive control. This type of control works well for rapidly changing environments. Note that this "simple" form of control has no memory or learning. Basically the robot only acts directly on the sensor input that it is currently receiving. In some ways this resembles instinctive behavior (i.e. not thinking, but reacting).

(3) Hybrid control. A combination of the deliberative- and reactive control approaches. This system usually uses a control system that decides when inputs require planning or whether they can be acted upon directly. An important part of the control system is solving conflicts between the two parts. It is common that the reactive parts have very high priority in the system, such as "avoid obstacles", while parts that require planning are often more abstract and less important, e.g. "explore the environment".

(4) Behavior-based control. Systems that use behavior-based control have a set of different parts (often layers) that are encoded as behaviors. There is no centralized control as with hybrid systems, but a network of behaviors that can communicate by sending inputs and outputs to each other. Some of these systems are explained in detail in the next section.

3.1 Emergence in Artificial Intelligence (AI) systems

Emergence is a complex and abstract phenomena. Emergence is important for many of the technologies described below and for life as we know it. A formal definition of emergence is difficult to formulate, but the following one tries to capture the essence of it.

> An unpredictable quality arises as a result of collaboration between entities, or the organization of entities in a particular way.

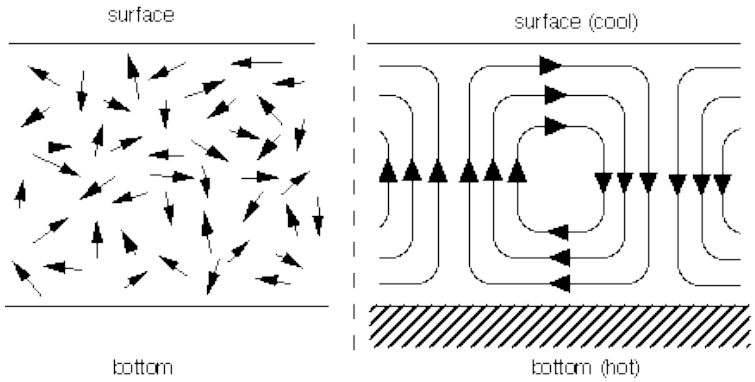

Figure 1: Benard Rolls

It should also be noted that emergence relies heavily on local interactions. Emergence is often the result of a self-organizing system. Local interactions in the micro world are seen as self-organizing systems in the macro world. An example taken from thermodynamics, is Benard Rolls, as seen on figure 1[12]. If energy

in form of heat is applied from the bottom, convection cells will appear and begin to rotate. This is an example of an emergent phenomena.

Self-organizing systems are known to be very fault-tolerant (resilient). This is because they are a product of the cooperation of simpler modules and thrive on randomness.

Emergence is as previously mentioned a complex phenomena. It is therefore very hard to design systems that have emergent properties. Designing a system that has the desired emergent property and not just any emergent property, is even harder. However, there are techniques that can be used to achieve this, such as evolutionary algorithms.

In our system, emergent behavior is a result of the evolved dynamics in the Continuous-Time Recurrent Neural Network.

3.2 Behavior-based

Control systems for adaptive autonomous robots have traditionally been designed using a "divide and conquer" strategy. This implies that the problem at hand can be divided into smaller sub-tasks. Behavior-based robotics typically have a number of simple behaviors and a control mechanism that determines the strength of each behavior and selects the optimal one at the given time.

3.2.1 Subsumption architecture

One example of the behavior-based approach was introduced by Brooks and it is known as the Subsumption Architecture [1]. The idea behind this architecture is to avoid any symbolic representation or symbolic reasoning. Behavior modules are then organized such that they are task-oriented. Each module can be thought of as an individual action function where inputs are mapped onto an action. Control systems are then built up of many of these action functions where more abstract behaviors are on top of more basic ones. An example would be to have obstacle-avoidance low in the hierarchy while eat-food would be high in the hierarchy as shown in figure 2 on the following page. It is important to mention that multiple actions can "fire" at the same time. Therefore actions at lower positions in the hierarchy can inhibit actions that are present higher up, ie. lower level actions have higher priority.

The complexity level of the subsumption architecture is no greater than $O(n^2)$, where n is the largest number of either behaviors (actions) or the number of percepts. This computational simplicity is one of the strengths of the subsumption architecture.

Reactive approaches have several advantages such as: simplicity, speed and robustness against failure. But the strengths they carry with them are also their inherited limitations. Since reactive agents only rely on local information they are forced to take the short-term view. Building a purely-reactive controller such that it can learn from experience or exercise complex behaviors such as adaptivity is hard and complex to understand.

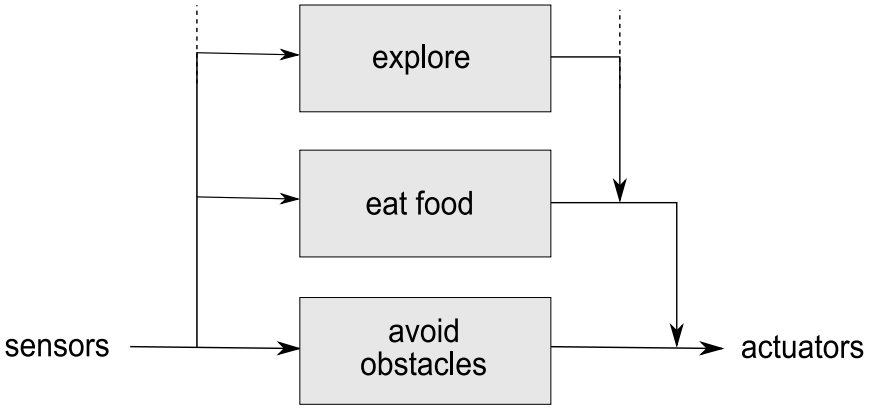

Figure 2: Subsumption architecture

3.2.2 Maes' Agent Network Architecture

Maes' activation networks use dynamic action selection. They try to avoid using modules, but instead focus on tasks. These can be viewed as a set of competence modules, which are shown in figure 3. They have pre- and post conditions and an activation level that specifies the relevance of the module in a given situation. The higher the activation level, the greater chance the module will affect the behavior. The different competence modules are connected with successor links, predecessor links and conflictor links. The competence modules compete to control the agent's behavior.

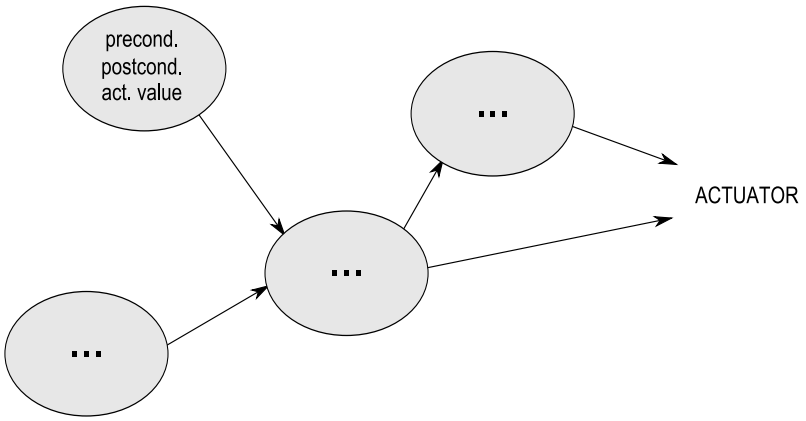

Figure 3: Maes' Agent Network Architecture

There are similarities between these networks and neural networks, such as the activation level and connectivity. However, since these competence modules are described in declarative terms, they are easier to understand. They are also more constrained by their very precise design, giving little room for emergent

representation of behavior.

3.2.3 InterRap

Just like in TouringMachines, InteRRAP consists of three control layers. The lowest layer (behavior) deals with reactive behavior, the middle layer (planning) deals with proactive planning such that the agent can reach its goals and the layer on top (cooperation) handles social interactions. Each layer has its own knowledge base which is the representation of the world that is relevant to the particular layer.

In InteRRap, layers communicate with each other in order to process input and select action output functions. The communication is done with either bottom-up activation or top-down execution. Bottom-up activation happens when a low-level layer is incapable of solving a problem, and it therefore passes the control to a higher-level layer. Top-down execution refers to a situation where a high-level layer utilizes functionality from a low-level layer. Each layer implements three general functions, which are situation recognition, goal activation and scheduling.

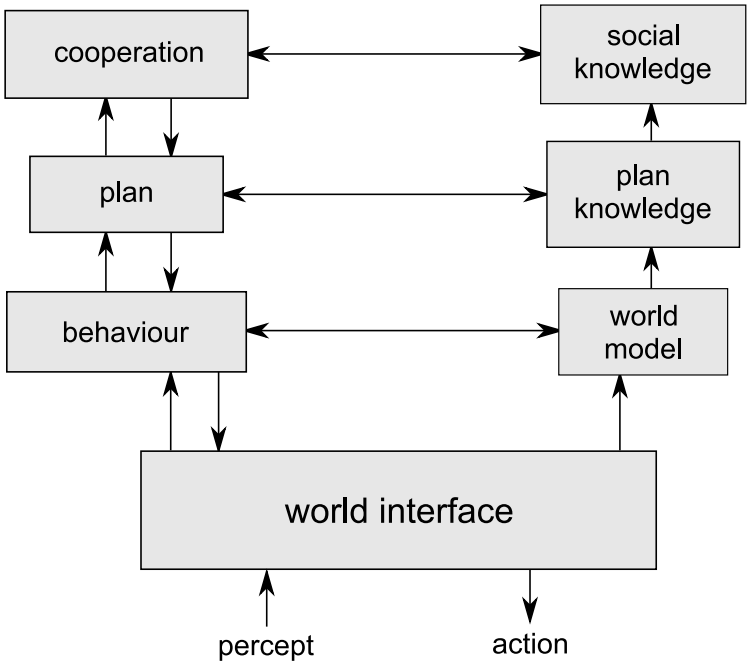

Figure 4: InteRRAP architecture

3.2.4 Case-based reasoning(CBR)

Creating behavior-based controllers based on CBR is reasonable since a priori knowledge from the designer is needed in behavior-based controllers.

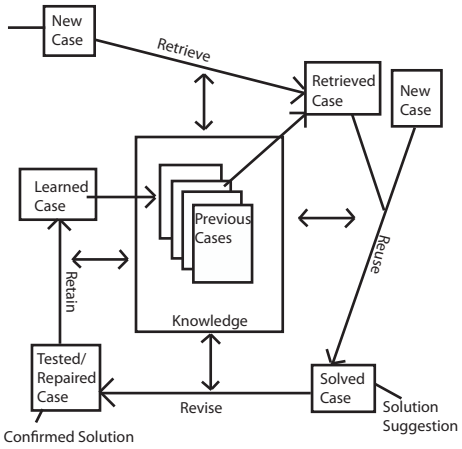

Figure 5: CBR cycle

CBR is a technique where knowledge is stored and looked up as cases are being presented for the system. New cases are matched against old cases and then stored for later usage in the knowledge base illustrated in figure 5. This knowledge base is a sort of memory, where solutions are previous cases (experience) stored in the knowledge base [23]. The method has its roots from the work on understanding reasoning as a form of explanation driven process.

A paper from Georgia Institute of Technology presents an approach to robot navigation using spatio-temporal CBR. Here they developed a hybrid-architecture (reactive, proactive) using a reactive schema-based system coupled with a proactive planning system. Their CBR module supported navigational states of type "GOTO" which were used for goal-directed navigation. This was done by extracting behavioral parameters from the CBR module depending on the sensory data, and then creating a vector, which in turn worked on the actuators.

Using the CBR approach, the team at Georgia Institute of Technology produced successful results both in simulation and on real robotics. One limitation that they report is that cases could only be learned but never forgotten. This posed a problem in cases where training in the simulator could not reflect all the characteristics of the real environment[24].

3.2.5 TouringMachines

TouringMachines [1] is a hybrid architecture, which has reactive and proactive properties. Hybrid(reactive-proactive) systems try to address some of the limitations that purely-reactive agents pose. In TouringMachines this is done by a horizontal layer based architecture. This allows for subsystem separation for the reactive part and a proactive part.

As seen on figure 6 on the next page there are four layers. The idea behind this architecture is that each layer produces action suggestions in different ways. The reactive layer instantly produces action suggestions. This layer reacts much like

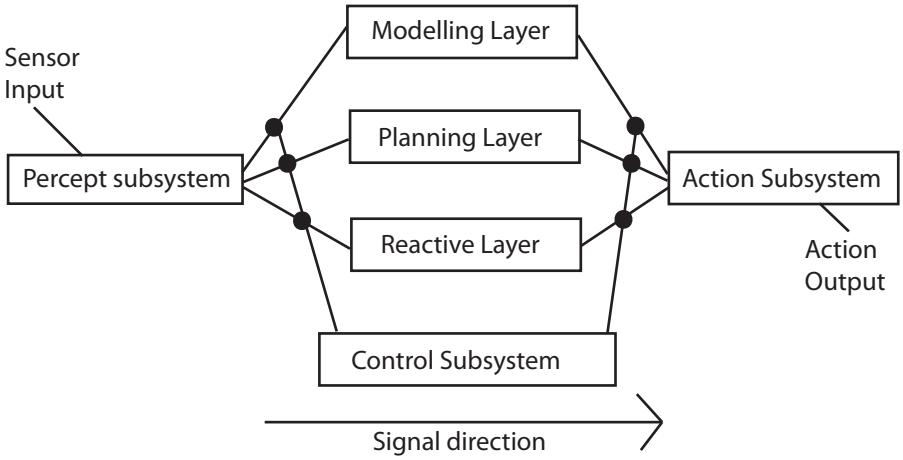

Figure 6: Model of a horizontal TouringMachine [1] architecture

Brooks subsumption architecture would, by mapping sensory input into actuator output. A simple name for this layer would be the reaction layer.

The proactive part of TouringMachines is situated in the planning layer. This layer is responsible for the overall performance. Planning is done through the use of schemes, however, it does not generate plans from scratch. Rather, it uses skeleton schemes that are placed in a hierarchical manner and generates a plan according to the situation at hand. The planning layer essentially searches for schemes that match the goal. Another name for this layer could be the solution layer.

The modelling layer contains the world model. Conflicts can then be discovered in the modelling layer and new goals to solve these are then generated. These are then posted down to the planning layer. The planner then has to find a schema that will solve the conflict.

The control system is responsible for choosing which of the above layers that is supposed to have control over the robot at any given time. This essentially mean that the control system can inhibit a layer in such a way that inputs would not reach that layer.

The decomposition of the reactive layer and proactive layer makes this architecture appealing due to its simplicity. However, in order for the robot to behave coherently you must have a centralized mechanism that controls which of the layers are to be in control of the robot. This is problematic in practice and the architecture has many of the same problematic characteristics of behavior-based ones. Real adaptivity and learning is still too complex and hard to achieve.

3.3 The evolutionary approach

Evolutionary robotics is a term which started to emerge in the early 1990s [10]. It relies on genetics and Darwinian selection (survival of the fittest) by sexual or asexual reproduction. Evolutionary search techniques for computational purposes date back to the 1950s, but were not used for developing controllers for robotics until the 1990s.

This process has many names: evolutionary computation, evolutionary algorithms or genetic programming. But they all cherish the same core values, illustrated by figure 7 and figure 8 on the next page.

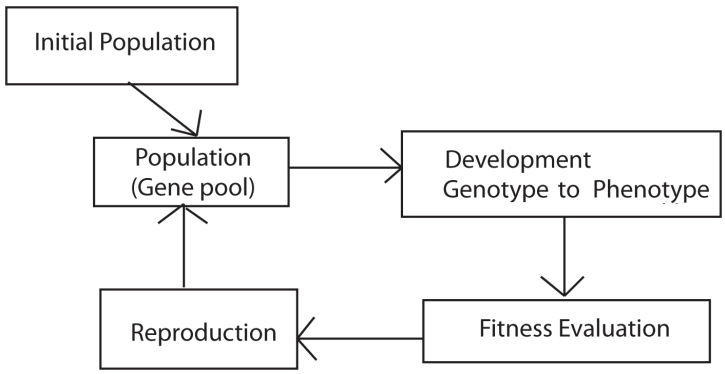

Figure 7: Overview of an EA

In this approach there is no need to break down the task into smaller subtasks. By starting with a random population of controller setups, evolution allows individuals that perform well to reproduce and propagate their genes into future generations. This way a variety of controllers get tested and those with desirable properties can emerge. This is where artificial evolution usually differs from evolution in nature. Evolution in nature is open-ended where as artificial evolution often is not open-ended. Open-ended evolution means that there is no a priori goal for evolution.

Gene representations in nature are based on the DNA alphabet consisting of A, T, C, G as building blocks. Gene representations in artificial evolution vary from bits(0,1) to real numbers. Choosing a good genotype representation is important so that a small change in the genotype reflects a small change in the phenotype, and other way around. Direct mapping from genotype to phenotype is commonly used in artificial evolution, but methods for development exist.

Reproduction can either be done sexually or asexually. In sexual reproduction a type of crossover mechanism is used, while asexual production is similar to replication. Both types exist in nature as well but with very low error rate. In artificial evolution this error rate is much greater, typically ranging from 0.01% up to 100%. This error is called mutation. Mutation is a small change in the genotype. Mutation can be seen as exploitation in the fitness landscape while crossover resembles exploration.

The term fitness is commonly used when talking about evolution. Fitness comes

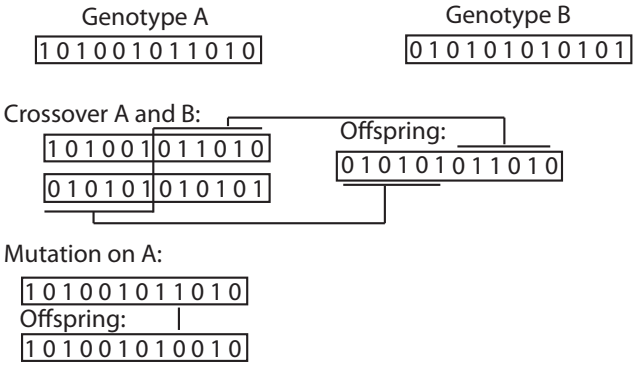

Figure 8: Crossover and mutation

from biology where the true meaning is how well a certain phenotype propagates its genes. High fitness doesn't necessarily mean that the phenotype is stronger or bigger, but rather how well it can reproduce. This is where we can observe how well a certain gene affects the fitness of a phenotype. Genes that make the phenotype exceptionally good at reproduction can be said to become immortal [19].

Since evolution is a search technique, it is important to know about genotype and phenotype search spaces. Evolution searches cumulatively and incrementally through the search spaces.

Varieties exists, such as co-evolution of predator and prey. This can lead to progressively more complex evolutionary processes which are not possible with just one population.

Evolution can be allowed to tap into just about anything in the control system of a robot. However, without development one needs to specify everything in the genotype. This can result in a huge genotype. From a neural network perspective it is common to allow evolution to work on the weights within a neural network, the topology or the Hebbian learning rules for each neuron.

The evolutionary approach is very appealing for a number of reasons. First and foremost it is a good way of achieving emergent behavior. Evolution incrementally closes in on a near-optimal solution, or at least tries to. Evolution is very successful in nature and it is the only process that is known to create complex systems such as the human brain.

Most evolutionary algorithms are not as true to nature as one would wish. A common problem in evolutionary computation is that the phenotype is given a fitness after a full run. This means that even if the phenotype did some horrible mistakes during the fitness evaluation but got lucky, or that the fitness function did not notice the mistakes, it will often be able to reproduce.

3.3.1 Gaussian mutation for Genetic Algorithms

In nature we find that mutation is the exception and crossover is the rule. In other words, most animals reproduce sexually, where a male fertilizes a female and the genes are combined in order to produce an offspring. Mutation of genes happen by an accident during the copying process, and is considered to be a very rare phenomena. That being said, there exists animals that only reproduce asexually, but mutation is still very rare.[19]

When looking at genetic algorithms evolving neural networks, crossover of genotypes often does not make sense. This is due to the fact that one "concept" found in one part of a neural network won't make sense when set into a different neural network. Using asexual reproduction with mutation can give much smoother travel in the genotype search space when evolving neural networks.

In most genetic algorithms mutation rates are much higher than in nature. The classic way of performing mutation is the bit-flip mutation illustrated in the previous chapter. Unless you manipulate more than one gene, the movement in the genotype search space is limited to one dimension.

An alternative approach when using a real valued genotype representation is to use Gaussian mutation. Creating offspring using Gaussian mutation consists of adding a random value from a Gaussian (see figure 25 on page 45) distribution to each element of an genotype's (parent) vector. This leads to movement in genotype search space in any n dimensions. A way of looking this is to picture a sphere around the parent location, and movement can occur in any direction from this location, only limited by the amount of dimensions. Since we potentially change more than one gene, the movement can occur in multiple dimensions. And as a consequence of this, steps in search-space can occur that would be impossible with bit-flip mutation. This kind of property can be desirable, and is not obtainable unless you allow bit-flip mutation to mutate all the genes. The creation of a bit-flip mutation mechanism, which can flip all bits in such a way that we get smooth transitions in genotype search space, is difficult. This makes Gauss-mutation a good alternative when using real numbers to represent the genotype.

Figure 9 illustrates how Gauss-mutation potentially mutates vector based genotypes. It should be noted, that most uses of Gaussian mutation has a mean set to 0 and a very low variance. This would imply that on average there should be no change (or at least, very small) to the genotype.

0.100	0.400	0.700	-0.200	⇒	0.080	0.401	0.710	-0.190

Figure 9: Gaussian mutation

Although Gaussian mutation possibly is less biologically plausible, it remains a useful tool for GAs. And in [26], experiments show that Gaussian mutation can outperform bit-flip mutation.

3.3.2 Evolutionary growth of complexity

The problem of classical engineering approaches is closely linked to the complexity of the machine that is being engineered. In fact, if we look at classical engineering, it is clear that if we want to build a machine with a certain complexity level we need an even more complex machine to build it. This simply means that we go from complex to less complex. If we then follow the engineering trail backward, we end up with our own brains. Thus, using the classical engineering approach we will never be able to produce something more complex than ourselves.

This is where evolution steps in as a possible solution to the problem mentioned above. Evolution goes the other way around. Our theory and understanding of evolution shows us that complexity can grow with time. In nature this would be the same as going from a single-celled organism to a multi cellular organism [20].

3.3.3 Baldwin effect

The Baldwin Effect (ontogenic evolution) is an organism's ability to adapt to its environment during its lifetime (phenotypic plasticity). The most common case of phenotypic plasticity is perhaps the ability to learn. Among other examples is the increase of strength through exercise or tanning in order to better endure exposure to sunlight.

There are two phases of the Baldwin Effect. First, the phenotypic plasticity allows an organism to adapt to a mutation which is only partially successful. If this mutation increases the fitness of the individual, it is likely that it will propagate. However, phenotypic plasticity requires a lot of energy and time, such as the training that is necessary for increased muscular strength. This is where evolution comes into play. It is capable of finding alternatives that can replace the phenotypic plasticity, making it instinctive. This must not be confused with Lamarckism, where the genotype is modified based on the phenotype. Note that in the Baldwin Effect, the genotype is never modified directly.

An example provided by Terrence Deacon [14]:

First phase:

An earthquake closes off a traditional migratory route. Migratory animals change their route and end up in a colder location.

Second phase:

Some of the animals are better able to withstand the cold. Natural selection favors those animals and the population adapts to their new circumstances.

This example is illustrated in figure 10 on the following page.

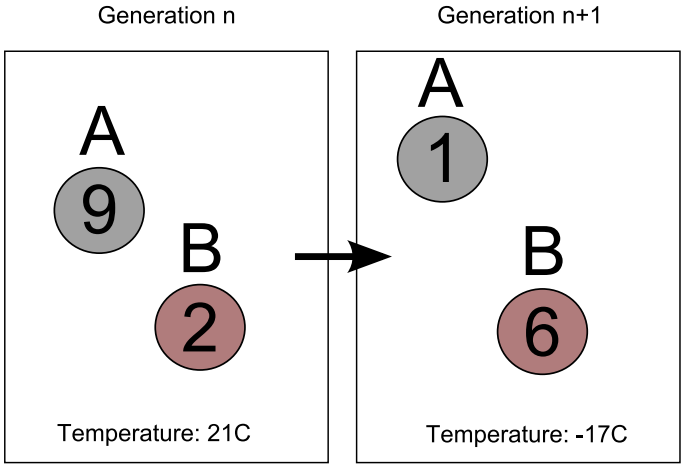

Figure 10: The Baldwin Effect, showing two individuals and the effect that the environment has on their fitness.

3.4 Artificial development

Artificial development is a sub-field of evolutionary computation, and it is motivated by the lack of scalability, robustness and non-trivial structures as a result of direct encoding. The development occurs in the mapping from genotype to phenotype. It is also worth to note that development is a continuous process that does not stop once the organism is completely mature. This enables adaption to changing environments and self-repair.

How does development solve the problems concerning scalability, robustness and non-trivial structures?

3.4.1 Scalability

As the search space grows exponentially with the size of the genotype, it could be beneficial to use indirect encoding from genotype to phenotype. Development uses a genetically encoded growth program in recursive steps.

3.4.2 Robustness

Biological organisms ability to regenerate is an inspiration for many groups of researchers. Self-repair sometimes appears as an emergent property of artificial development.

3.4.3 Structures

When we evolve e.g. robot controllers, we typically end up with complex structures. Through the application of development, simpler structures can appear by exploiting environmental aspects.

3.4.4 Indirect encoding

This indirect encoding is usually based on biological development which can be divided into four simple steps. A zygote (fertilized egg) develops into a multi-cellular organism through these steps:

Pattern formation. Cells organize in different regions according to their type, in order to become distinct parts (body segments).

Morphogenesis. Some of the cells change shape (expand/contract) exerting a force on other cells. This causes formation of general shape in the organism.

Cell differentiation. Cells become structurally and functionally different from each other.

Growth. In this step, the organism is enlarged. This is called by cell-divisions and expansions. Programmed cell death can help generate fingers or toes with sheets of tissue.

These steps have analogous computational processes such as re-writing, iteration and time. Artificial development has been applied to several types of computational problems, e.g. evolution of robot controllers [36] and electronic circuit design [37].

3.5 Neural networks

Neural networks stem from the human knowledge about how the biological brain works[18]. Computer science had largely abandoned neural networks in the late 1970s, but research continued in other fields. Physicists like John Hopfield analyzed their storage and optimization properties.

With the invention of the back-propagation learning algorithm in the mid 1980s the field of neural networks started to blossom once again. Inter-connected networks of neurons make up a highly parallel computation machine. Feed-forward architectures commonly propagate signals through the network using sigmoidal or hyperbolic tangent transfer functions or a step function, these are found in table 1 on the next page.

Neural networks make use of connection strengths (named weights) between the neurons in order to classify inputs into output signals. Learning the correct weights is therefore crucial for the network to work properly for most cases. These models are called connectionist models of intelligent systems[4]. As a small definition: “Instead, connectionists suggest that intelligence is to be understood as the result of the transmission of activation levels in large networks of densely interconnected simple units”[3].

Function	Definition	Range
Step	$\begin{cases} 0 : x < 0 \\ 1 : x \geq 0 \end{cases}$	[0, 1]
Sigmoidal	$\frac{1}{1-e^{-x}}$	(0,1)
Hyperbolic	$\tanh x$	(-1,1)

Table 1: Transfer functions

Neural networks have been applied to a wide variety of problems with great success. Despite researchers questioning the "validity" of the connectionist approach, the current view is that connectionist and symbolic models are complementary to each other and not competitors.

3.5.1 Self-organizing maps

T. Kohonen is recognized as the father of self-organizing maps(SOM). SOMs are a form of neural network where the structure of the nodes can be said to carry the information. Through an unsupervised learning process cells on a lattice tend to become self-organized.

Transfer functions are normally not necessary for SOMs, as it is the structure of the network that is of importance and not the signal propagation.

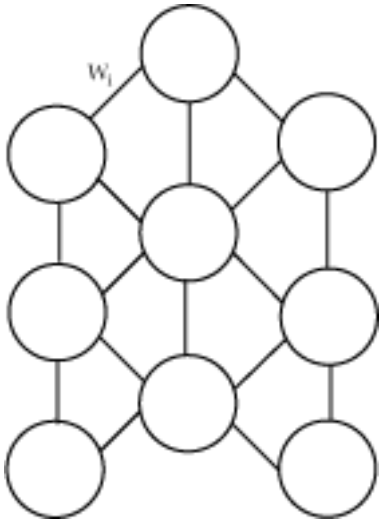

Figure 11: Two-dimensional Self-organizing map

An initial setup of the grid is typically a random configuration or in some cases, it contains some a priori knowledge which can make learning easier or faster. The learning is usually based on moving cells, and making their neighbours follow. This way the grid becomes an elastic moving grid which then can be trained to fit a certain problem. In a simple form, training can adjust the weights (connections) of a cell to its closest neighbors by a decreased adaption gain. This makes one change to a cell propagate to neighboring cells. Variations

of this basic method exists, and has reported decent results on e.g. cortial map development [5].

Self-organizing maps can be used to create maps of the world around a robot. One example of map creation for robots using self-organizing maps is described in a paper written at the University of Athens[7].

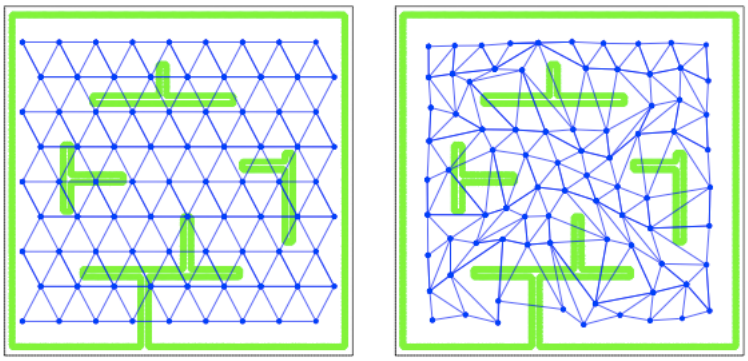

Figure 12: Self-organizing map shown before and after learning [7]

From figure 12, the initial state is an untrained grid. After learning it gets stretched so it is possible to determine where there are walls and where there is open space. They then used an A* algorithm [8] to find paths through the environment. The results they report were satisfactory during simulations. However, there were no reported tests on actual robots.

Self-organizing maps are hard to analyze mathematically, but extensive testing has shown that the idea is plausible. Choosing the correct amount of nodes for the environment can have a huge impact on performance. Therefore it can arguably be hard to apply a map technique like the one described above to environments that are dynamic.

3.5.2 Hopfield nets

The Hopfield model [13] was an important milestone for the development of artificial neural networks and was proposed by John Hopfield in the early 1980s. The model showed how a group of simple processing units can have complex computational power and behavior.

The Hopfield network is a bidirectional auto-associative network with memory, and it can recall a saved pattern when presented as input in the form of a noisy version of that pattern. A Hopfield net has a single layer of processing units where every unit is connected to all the others, but not back to itself. In addition, every node has an external input I. Unlike most neural networks, this model treats all the neurons as both input and output units.

An important issue that must be taken into consideration when designing neural networks is that of global synchronization. A solution to this problem, is the

assumption that the activation of a neuron takes some units of time. The network can be arranged such that it takes delay and time into consideration. A network that makes no such assumptions behaves like a stochastic dynamical system. In other words, units are updated one at a time, and in random order.

The sum of inputs to a neuron can be calculated as

$$net_j = \sum_{i=1}^{n} s_i w_{ij}$$

where s_i is the state of unit i. Each unit gets its state updated according to the signum function, i.e. it will be +1 if $net_j > 0$ and -1 if $net_j < 0$. If the net input is zero, the state doesn't change.

The network processes input the following way: Input is given in the form of a vector containing the initial states of all the neurons. A random unit is selected and updated based on weighted input received from all the other neurons. Another unit is selected and the operation repeats itself. When none of the units change state when updated, the network has converged. A simple Hopfield net is displayed in figure 13.

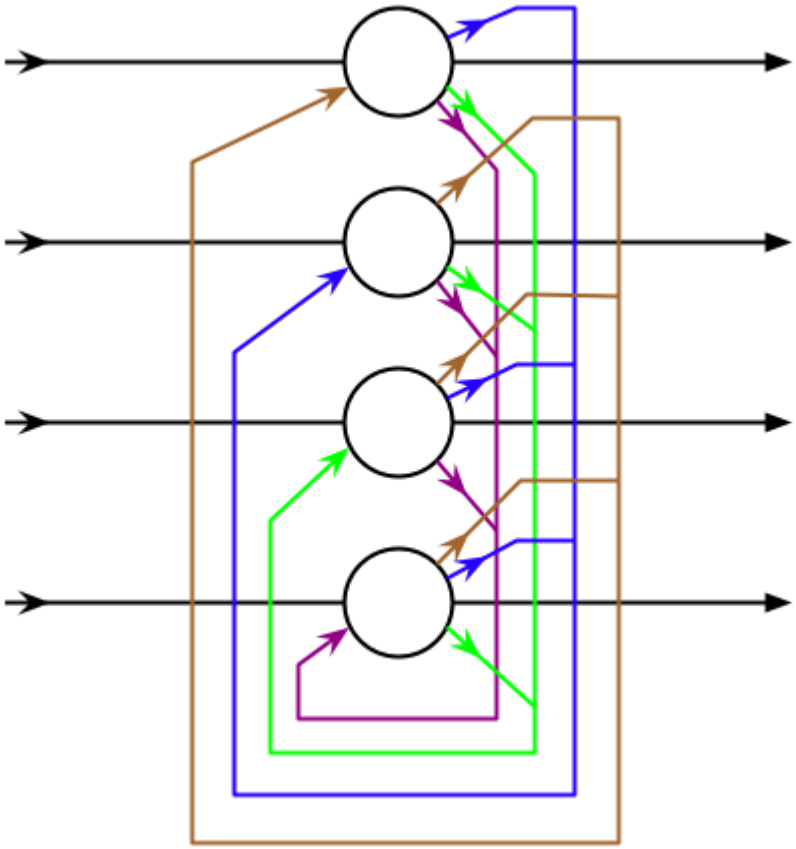

Figure 13: Hopfield net

Description	Update rule
Classic Hebbian	$If : B \uparrow \wedge A \uparrow \rightarrow w \uparrow$
Anti-hebbian	$If : A \uparrow \wedge B \uparrow \rightarrow w \downarrow$
General Hebbian	$If : A = B \rightarrow w \uparrow If : A \, opposite \, (B) \rightarrow w \downarrow$

Table 2: Hebbian learning

3.5.3 Hebbian learning

Early models described by Holland is based on a concept of "fire together, wire together". The name Hebbian learning comes from the Canadian psychologist Donald Hebb who reasoned about how the human brain learned through response and stimulus.

This is a type of unsupervised learning where weights are strengthened when two connected neurons fire at the same time. Varieties exists, classic Hebbian, anti Hebbian and general Hebbian learning[6]. To illustrate how some of these update rules work consider table 2.

A and B are two connected nodes and w is the strength of the connection between them. The $\uparrow$ means that the neuron is firing and w$\uparrow$ means that the connection weight is strengthened and w$\downarrow$ means that the connection is dampened.

Hebbian learning algorithms are very appealing due to their computational simplicity. In addition the learning algorithm is very robust against noisy data. But since there is very little control over what is going on during training, the algorithm has a tendency to make the network fire on all inputs once it has been trained. This happens in particular when using the classic Hebbian update rule.

Allowing each neuron to have an individual update rule helps the network to resist the above mentioned problem. But this in turn poses a new problem. Determining which update rule to give to which neuron is hard. Evolution(described in more detail below) can be allowed to tap into this, but this makes Hebbian learning more computationally expensive.

3.5.4 Back-propagation learning

Back-propagation learning is also known as supervised error based learning. The amount of work required in finding the correct set of weights for a neural network increases substantially when the network grows in size and connectivity. Back-propagation is a numerical gradient decent method for finding these weights. Back-propagation is a supervised learning algorithm and arguably one of the most successful methods for finding a good set of weights [9]. The comeback of neural networks can likely be credited to back-propagation's good results and wide variety of applications, particularly in pattern recognition.

The idea behind back-propagation is to minimize the error between the desired outputs and the actual outputs. In a feed-forward neural network without hidden layers the problem is trivially solved by the delta rule [10]. However, solving problems without hidden layers poses a serious problem as they are unable to

A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Table 3: XOR function

solve problems that are not linearly separable. One example of a non-linear separable problem is XOR, see table 3.

The natural solution is to add hidden layers to the neural network. In order to compute the error contribution made by the hidden layers, the delta rule has to be extended into the generalized delta rule. After sweeping forward, the delta rule can find an error. The error contribution found by the delta rule is then transmitted backwards into the hidden layers using the same weights. This is the reason behind the name back-propagation. We want to change the weights of the network to minimize the output error. This method can also be applied to recurrent neural networks.

A continuous transfer function is needed, since the back-propagation algorithm relies on derivatives. The step function (see table 1 on page 25) cannot be used.

When using the sigmoid function in a network where all outputs at layer n are connected to all inputs in layer $n+1$, a typical use of back-propagation would be [11]:

1) Sweep the signal forward in the network using the sigmoidal function.

$y_i = \frac{1}{1-e^{-x}}$

2) Update weights. t = desired output at node i while o = actual output at node i.

$\triangle w_{ij} = \eta \sum_{d \in D} (t_{id} - o_{id}) o_{id} (1 - o_{id}) x_{jd}$

Adding hidden layers gives the network more expressiveness, however it should be noted that having too large networks can end in poor results due to over-fitting. It can be showed that neural network architectures using back-propagation can learn any arbitrary mapping from inputs to outputs. The need of knowing the correct output for every input is a huge drawback. When working with robotics in a dynamical environment, knowing the correct output for every input(percept) is very hard, if not impossible.

3.5.5 Continuous-Time Recurrent Neural Networks

In recent robotics papers [25] the use of more complex networks is emerging. These networks, as introduced by R. Beer are called Continuous Time-Recurrent Neural Networks and are biologically plausible in respect to their properties [17]. This is one of the technologies used in this report, and we will get back to the specific details of our setup. For now, here's a general outline:

These types of networks use the following state equation[16]:

Variable	Meaning
y	State of each neuron
τ	Time constant
w_{ij}	Strength of connection between the i'th and the j'th neuron.
g	gain
θ	bias term
I	External input

Table 4: Variables in a CTRNN state equation

$$\tau_i \dot{y}_i = -y_i + \sum_{j=1}^{N} w_{ij}\sigma(g_j(y_j + \theta_j)) + I_i, i = 1, ..., N$$

where σ is the sigmoidal activation function:

$\sigma(x) = \frac{1}{1-e^{-x}}$

and the remaining variables are explained in table 4.

Normal feed forward Neural networks lack the ability to remember and therefore only support reactive behavior. Continuous-time recurrent neural networks allow the network to remember from previous experiences. Remembering from previous experience allows the robot to become proactive and anticipate future events [17]. CTRNNs are arguably one of the simplest ways of achieving this type of behavior. Using evolution to evolve the weights seems to work well with CTRNNs [16]. A growing interest is being shown by the robotics community.

The implementation used in this report is described in detail in the design section.

3.6 Reinforcement learning

Problems that involve learning through trial and error interactions with the environment are referred to as reinforcement learning problems. One can search through a space of behaviors in order to find one that behaves well in the environment. Through the use of statistical tools it is possible to estimate the utility of taking specific actions. If the problem is modelled with care, some reinforcement learning algorithms can converge to the global optimum. This is the ideal behavior that maximises the reward.

With reinforcement learning there is no need for a human expert in the actual domain, nor any need for hand-crafted and complex sets of rules as with expert systems. Among the examples in [15] was the simple game of tic-tac-toe. This isn't a complicated game, but it is difficult to train an agent using traditional techniques since most of them require a complete specification of their opponent. With reinforcement learning, the problem could be solved without a priori knowledge of the opponent. However, it would have to play many games before it could behave like a skilled player.

In order to play well with reinforcement learning, one would have to set up a table containing every possible state of the game. Note that for certain problems

this would be extremely exhaustive in terms of memory. The problem could be reduced by using value approximation techniques such as neural networks or decision trees, but that would typically produce less accurate value estimations. Each state in the table would display the chance of winning from that position. If we consider two values in the table, where value S gives a higher chance of winning than T, we would say that S is better than T. Assuming that we play with Xs and our opponent with Os, the states with three Xs in a row give us a probability of 1 to win. States with three Os in a row give a probability of 0 to win, and all the other states give a probability of 0.5 of winning.

The next step is to play several games with the opponent. For each move, we would examine all the possible moves by looking at their values in the table, and then we would usually select the most rewarding one (greedy behavior). Sometimes however, we would select a random move instead of the best one, since this leads to better exploration of the solution space. When we get to a new state, we change its value such that it better reflects our chances of winning. The update is done in the following way: When we move from state A to state B, we read the value of state B and adjust the value of A such that it comes closer to the value of B. The rate of change depends on a predetermined step size α. The update function is

$$V(s) \leftarrow V(s) + \alpha[r_{ss'} + \gamma V(s') - V(s)]$$

where s is the state before the greedy move and s' is the state after the greedy move. $r_{ss'}$ is the reward or penalty in going from s to s' and γ is a discount factor.

This approach differs significantly from an evolutionary solution. An evolutionary approach would use a fixed solution candidate and evaluate it only after a full game. What happens during the game would be ignored, and the fittest solution would reward all the moves taken on its way to the local- or global optimum. Both of the approaches search the solution space, but learning a value function benefits from the information that is available while the game is being played.

Above we described a lot of different technologies that exist, along with some problems that we have run into. These technologies can be used by themselves, but a more interesting approach would be to combine a number of them and to extend them for robotics. Jesper Blynel and Dario Floreano [27] used CTRNNs and evolution to see if reinforcement learning-like abilities could be evolved, and if they could be extended to robotics. Their results on the T-Maze task show that this is possible and that the "learning" resembles a type of learning found in biological systems.

We want to explore adaptive behavior and how this can be achieved. We also want to look at how nature has achieved it, since we find adaptive behavior in nature. Nature is proof-of-concept for a biology inspired approach.

4 Design

4.1 Robot

In our experiments, we are using an E-puck robot named WALL-E. It has eight infrared sensors, a camera, three microphones and an accelerometer. In our current experiments, we will only be using the infrared sensors and the wheels.

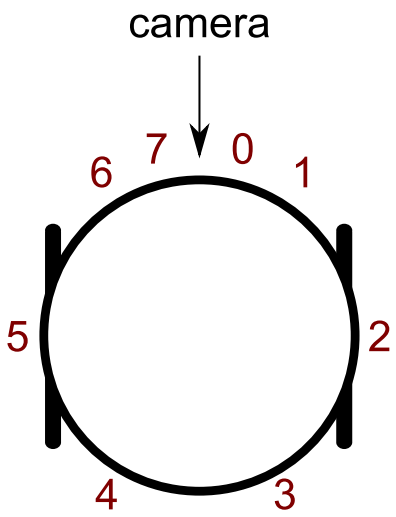

Figure 14: Proximity sensors.

The robot can be controlled either by uploading data to its memory, or by sending/receiving instructions over Bluetooth. We have chosen the latter approach, such that we can perform offline computations (in any language of choice). The robot can only understand machine code compiled with a C/C++ compiler that supports the dsPic chipset, whereas any language that supports Bluetooth can communicate with it otherwise.

We have chosen the Python programming language for this purpose. Python's syntax closely resembles pseudo code, something that should make it easy for others to get started with the E-puck irregardless of what programming background or paradigm they come from.

Those of you looking into this report with the purpose of getting started with the E-puck, see Appendix A.

4.1.1 Reality problem

It is common to run into some difficulties when running synaptic weights that have been evolved in a simulator on a real robot. When the neural network is subject to Darwinian evolution, the evolutionary process evolves solutions that are good in that specific environment. This means that evolution finds solutions that are good in the simulator. When this solution then is applied to the robot,

Figure 15: WALL-E, the robot

the solution might be unable to cope with the physical variations found in the real world. Variations which the simulator can't simulate, or variations that are very hard to simulate accurately. Realistic physics, Newtonian physics, is computationally very expensive, and very hard to completely simulate realistically. A completely realistic simulation which reflects the real world is hard and possibly impossible to create. Recent papers on CTRNNs also describe encounters with this reality problem [27].

In our particular case, friction against objects and timing of the internal dynamics of the CTRNN is subject to the reality problem. One problem with friction in the simulator is that friction varies a lot in the real world. While running on the real robots some parts of the walls might have just a slightly rougher surface, which in turn will make the robot turn slower, or not turn at all. While evolving the CTRNN solution the friction is more predictable, and evolution might have exploited this during simulation. The timing of the CTRNN internal dynamics is another issue which can provoke the reality problem. With our current genetic algorithm, we speed up time in order to test phenotypes faster. This makes it hard to control how often and at which time we update the internal state of the CTRNN. Sensory input which is being read by the CTRNN also changes with a tiny amount from the simulator to the robot, and in reality (on the robot) from one run to another due to lighting conditions.

From the above described problem, it is possible to see that an evolved solution that doesn't touch the wall much is a good counter measure against the problem with friction . However, wall touching is only part of the problem (removes friction from our reality problem "equation"). A good solution has to, in general

be robust enough to withstand a lot of noise. Adding noise to the simulator has been somewhat successful in other experiments [25].

4.2 Environment

Part of the criteria for this task, was the ability to adapt in a changing environment. For this purpose we created a table and a simulated environment with the following specifications:

Length: 150 cm
Width: 120 cm
Height: 9.5 cm
Floor color: S5000N 9929(dark gray)
Wall color: S1500N 9915(light gray)

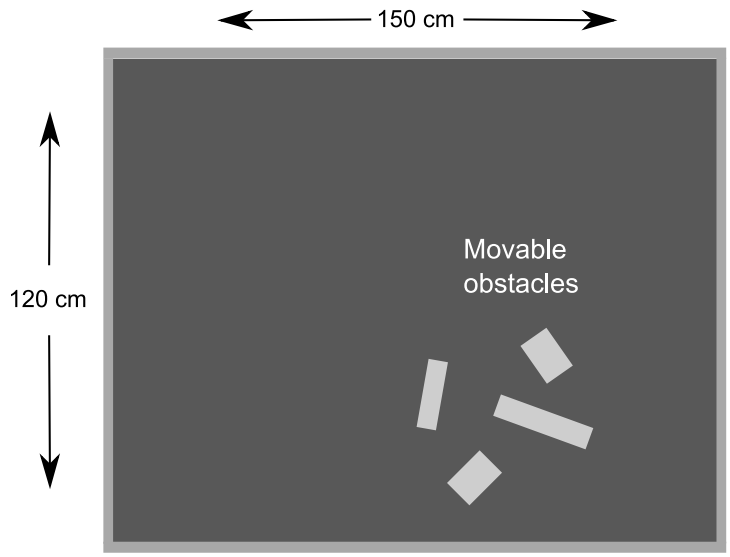

Figure 16: Simulated environment (table)

Since further (or other) experiments might use cameras, the floor is painted quite dark, while the walls are light, such that it can distinguish between them.

We have put some movable obstacles in the environment, such that we can easily set up mazes and paths. See figure 16 and figure 17 on the next page.

4.3 Continuous-time recurrent neural network

A CTRNN is a dynamical neural network composed of a system of differential equations. Each differential equation approximates the state of a single neuron in the network:

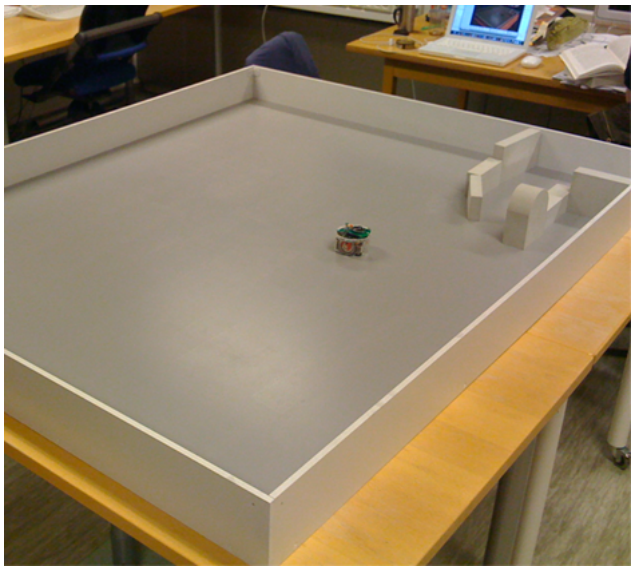

Figure 17: Physical environment (table)

$$\dot{y} = \frac{1}{\tau_i}(-y_i + \sum_{j=1}^{N} w_{ji}\sigma(y_j + \theta_j) + I)$$

where y is the state of each neuron, τ is its time constant (learning rate), y_i is the state of the post-synaptic neuron, w_{ji} is the synaptic strength (weight) from neuron j to neuron i, y_j is the pre-synaptic neuron, θ_j is a bias term and I is a constant external input (e.g. sensor input). σ is the logistic activation function $1/(1 - e^{-x})$.

Basically, for each neuron it does the following: Calculate the sum of incoming connections $w\sigma(y_j + \theta_j)$, subtract the current output y_i, add the constant external input I and finally multiply the result with $1/\tau_i$.

Real neurons operate in continuous time. In order to produce behaviours that rely on continuous time (on a computer), we have to discretize time by integrating neurons over small time steps. The accuracy of this approximation depends on the size of the time steps: small time steps give accurate approximations while large time steps result in less accurate approximations.. The general recommendation is to keep the time step smaller than the smallest time constant by a factor of ten [30].

These differential equations can be thought of as artificial neurons. When connected, they are reminiscent of how neurons in a real brain are wired, i.e. they form a dynamical neural network. The state of each neuron can be thought of as a nerve cell's mean membrane potential, and $\sigma(x)$ is associated with its short-term average firing frequency [17].

4.3.1 Dynamical properties

CTRNNs are typical dynamical systems in the sense that they have an arbitrary number of variables that vary over time, depending on the values of these same variables. These networks can be thought of as a type of neural networks, but they are really systems of differential equations.

In dynamical systems, effects are not proportional to their causes, i.e. small changes can have large effects and great changes can have small effects. The dynamics in CTRNNs can be understood through the circular relation between the system's differential equations: a single differential equation (neuron) affects the other components (neurons), and these components in turn affect the first differential equation. In other words: any change in the first component gets feedback to itself, through its effect on the other components.

The feedback is considered positive if it amplifies the recipient's initial state, i.e. if a differential equation's change in one direction receives feedback that takes place in the same direction. If the feedback given to a differential equation works in the opposite direction, it is considered negative. These feedbacks have opposite effects: negative feedback stabilizes the system towards an equilibrium (stable state), whereas positive feedback accelerates it towards chaos.

In order to examine the dynamical properties of a CTRNN, it can be useful visualize it in a phase plane. A phase plane displays plots of trajectories in the state space. This can indicate the presence of an attractor, a repellor or a limit cycle for our given parameter values. The model can also tell us something about the stability or instability of the system. Using Randall Beer's Dynamica software, which is an extension package for Mathematica, we can easily produce a phase portrait for a 2-neuron CTRNN circuit.

The blue dots in the portrait (figure 18 on the following page) represent stable equilibrium points, i.e. points where the competing forces are balanced. The green dots are semi-stable equilibrium points. Note that these points occur where the trajectories intersect (nullclines). These trajectories are solution curves, and for every point in the plane, precisely one trajectory passes through it.

These equilibrium points occur where $dx/dt = 0$. We look at the stability of these points because we want to find out what happens to x when we move away from them. The stable points will return to 0, whereas unstable points will make x grow in time and move away from the equilibrium point. The semi-stable points can have a periodic behaviour that oscillates between stable and unstable.

Unstable- and semi-stable equilibrium points can cause difficulties when moving a phenotype (CTRNN) from the simulated environment and onto the real environment because of significant changes in the parameter space, differences in the sensors and the fact that the battery level on the real robot influences the sensory readings. This can be remedied to some extent by evolving with imitated noise in the simulated environment in order to produce a more robust solution.

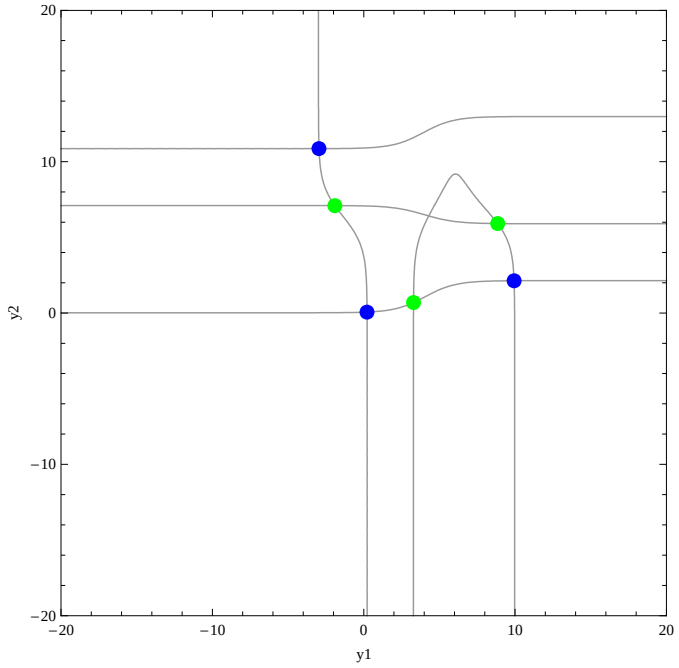

Figure 18: Phase portrait of 2-neuron circuit, where stable equilibrium points are denoted by blue dots and semi-stable points are shown as green dots.

4.3.2 Variables

The state equation for a single neuron is:

$$\dot{y} = \frac{1}{\tau}(-y + w\sigma(y + \theta) + I)$$

The neuron state equation can be simplified by setting $\tau = 1$ and $I = 0$. If we assume that the neuron has no self-connection, the state equation can be simplified to:

$$\dot{y} = -y_i \cdot s$$

where s is a small integration time-step for discretizing time, i.e. giving an approximation of continuous time. We can easily iterate through the time-steps: Let's say that we have $y(t_0) = 20$ and the time-step $s = 0.01$. We find that the change in y with respect to time is $\dot{y} = -y(t_0) \cdot s = -20 \cdot 0.01 = -0.2$. This tells us that $y(t_1) = 20 - 0.2 = 19.8$.

For the next timestep, we get $\dot{y} = -y(t_1) \cdot s = 19.8 \cdot 0.01 = 0.198$ which gives $y(t_2) = 19.8 - 0.198 = 19.602$ etc. We can see that $\dot{y}$ declines along with y_i until y_i reaches 0.

In the figure below, we can see this behaviour over time with starting values 1 and -1.

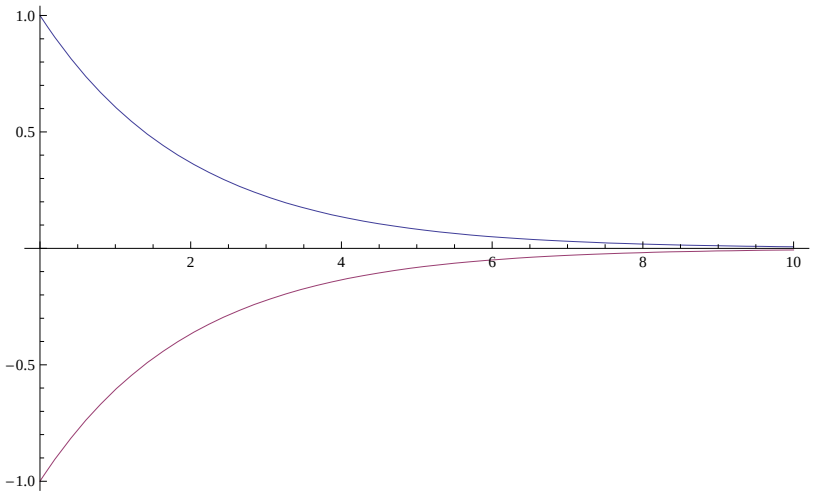

Figure 19: Simplified behaviour (CTRNN state equation)

Time constants

The next variable that is subject to examination is the time constant τ. This variable affects the decay time of the system. If τ is small, it will decay quickly, allowing the system to reach an equilibrium state in little time, as a response to changes in the environment. In this case, its output is a smooth decay curve.

If the constant is large, the decay happens slowly and provides a form of memory as the new state is affected by the previous one [31]. With a small constant, the output becomes less accurate, i.e. we integrate over large time steps. Note that it is important to keep the size of the time step s smaller than τ to avoid instability in the output. The effect that τ imposes on the output is illustrated in figure 20 on the next page.

Sensory input

The output eventually converges to the sensory input, I. In the simplified equation without any connections, the output decays to I with the time constant τ.

Figure 21 on page 40 shows three different neurons with their own respective sensor inputs. The first neuron (blue) converges to $I = 0.1$ with a small time constant, i.e. it decays quickly. The second neuron (red) converges to $I = 0.5$ with a large time constant (slow decay). The last neuron (yellow) converges to $I = 1.4$ with a small time constant.

Logistic function

The logistic function gives values between 0 and 1 where negative inputs result in output values below 0.5 and positive inputs yield outputs higher than 0.5. Note that the logistic function's output converges towards the extremes, such that all values below -4 will stay at approximately 0 and values above +4 will stay at approximately 1. In theory, the difference between the logistic function and transfer functions such as the hyperbolic tangent, is only a mapping. In practice,

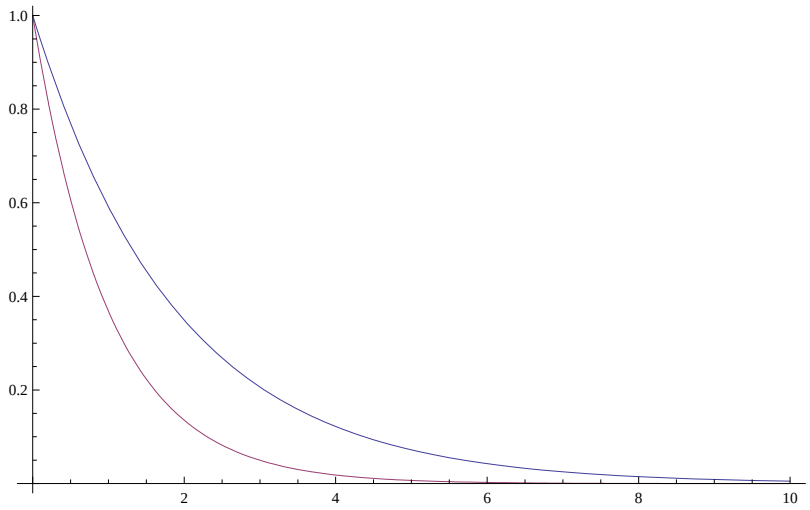

Figure 20: τ affects the decay. The blue line shows the impact that a large τ (0.95) has on the decay and the red line shows the same for a smaller τ (0.5).

it might be possible to discover more interesting dynamics with other transfer functions. Consider that the volume in the parameter space that generates CTRNNs with phase portraits with only stable point attractors might be larger when using the logistic function than when using the hyperbolic tangent function [32].

Weight term

The weights are multipliers that determine the impact of each node's output to the network. The weights (along with the topology) are essential components that help tailor the network for a particular purpose. The weights both amplify and inhibit neurons in order to create a desired behaviour. This is explained in detail in the section concerning emergence.

Bias term

The bias term, θ, is passed to the logistic function. This term affects the output of the logistic function by saturating it towards zero or the weight value, depending on whether it is positive or negative. In other words, we can use this term to bias the output of the logistic function into a desired section, which is typically where the output changes with the input.

Figure 23 on page 42 illustrates the logistic function σ with three different biases: -5, 0 and 5. From the graph we see how θ can be used to bias the logistic function's output into a specific section.

4.3.3 Visualization

In order to debug and ensure the correctness of our phenotypes, we wrote a stand-alone module that can visualize them. This allows us to check whether our genetic algorithm and artificial neural network are producing what we expect

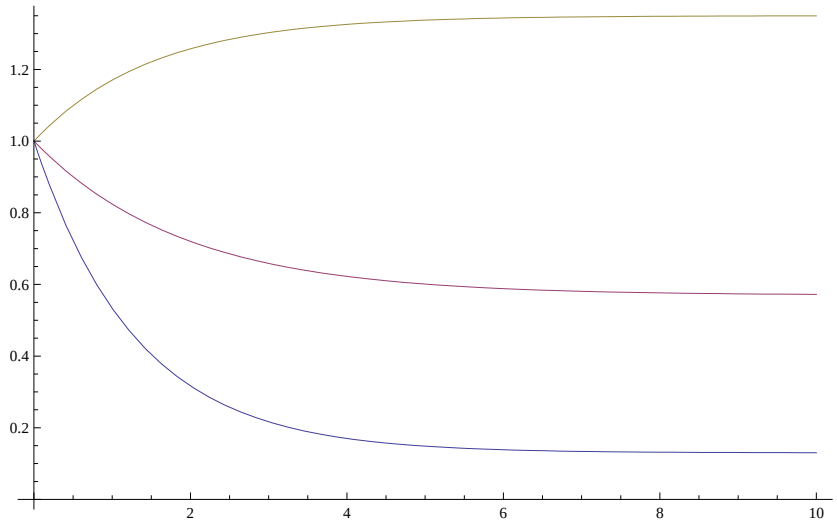

Figure 21: The effect of sensor input

them to. When a phenotype is being evaluated the artificial neural network is displayed as shown in figure 24 on page 43. The thickness of the weighted graphs indicate the weights' strength. This tells us something about how a neuron that fires can excite or inhibit other neurons in the next layer. The visualization also tells us whether or not the neurons are connected the way we intended. This tool currently only works with the initial stages of development where you can examine the flow through the neural network from input- to output neurons. In other words, it does not provide any useful information when used with recurrent networks.

4.4 Genetic Algorithm

The CTRNNs herein are evolved with an extrinsic genetic algorithm that uses the principles of Darwinian evolution in order to evolve synaptic weights. Our genetic algorithm does the following:

(1) Evolve a set of random synaptic weights (genotypes) that we refer to as our genotype pool. Each genotype has a specific length, which is the number of synapses that we want to evolve. Each gene resides within a given range. The number of genotypes in this pool is called the genotype population.

(2) Every genotype can develop into a phenotype through a simulated biological cell development process: pattern formation, morphogenesis, cell differentiation and growth. This part is under development, and instead it performs a simple copying process. This is the segment where we could eventually enable development.

(3) Each phenotype is evaluated according to our criteria (which is unsupervised). In our specific context, the phenotypes are tested in a simulated environment (Breve) and assigned a fitness value after a full run has been completed.

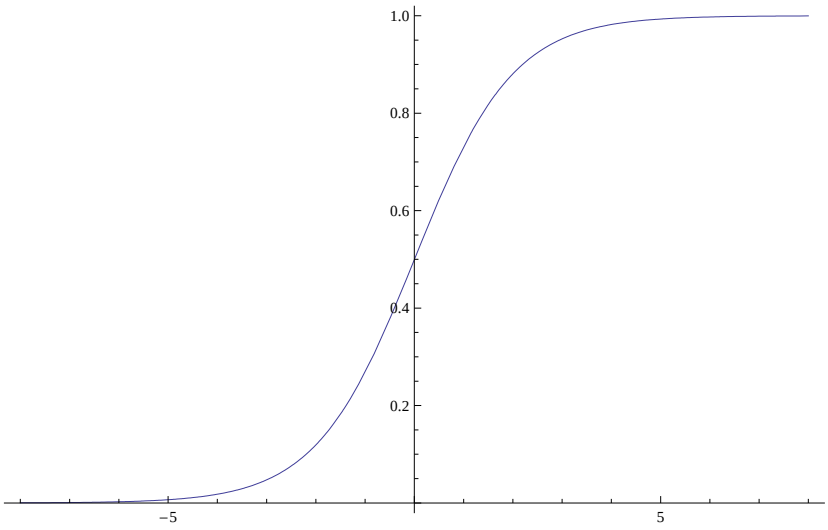

Figure 22: The logistic function σ

A fitness value is assigned to every genotype relative to how well its phenotype performed. The fitness functions used will be described in further detail below.

(4) Different selection mechanisms are being used in our experiments. Due to our varied fitness values we are using both tournament selection and rank selection. The reason we rely on probability in this step is because we want to avoid premature convergence. In nature, the best wolf sometimes falls off a cliff, making more room for alternative paths in the solution space. We also wanted to see which results the different selection mechanisms gave.

Previous version: The parents are likely to produce an offspring (recombination), but there is also a small chance that the best of the two gets a walkover into the next generation (replication). In the cases where the parents recombine, the offspring inherits part of the genes from one parent, and the rest from the other. This typically results in exploration of the solution space. There is also a chance that mutation can occur in the offspring, a small tweak that can result in novel features as well as exploitation of the solution space. This phase is repeated until there are enough candidates to fill up the next generation with the specified population.

New version: Purely Gaussian mutation.

(5) Return to step number two and continue working with the latest pool of individuals. Each cycle (complete genotype pool) is called a generation.

4.4.1 Genotype representation

There are a number of known methods for genotype representation. Both bit and real valued representations is widely used. For our GA we have chosen to use a real valued representation. We have chosen the latter, mainly because it

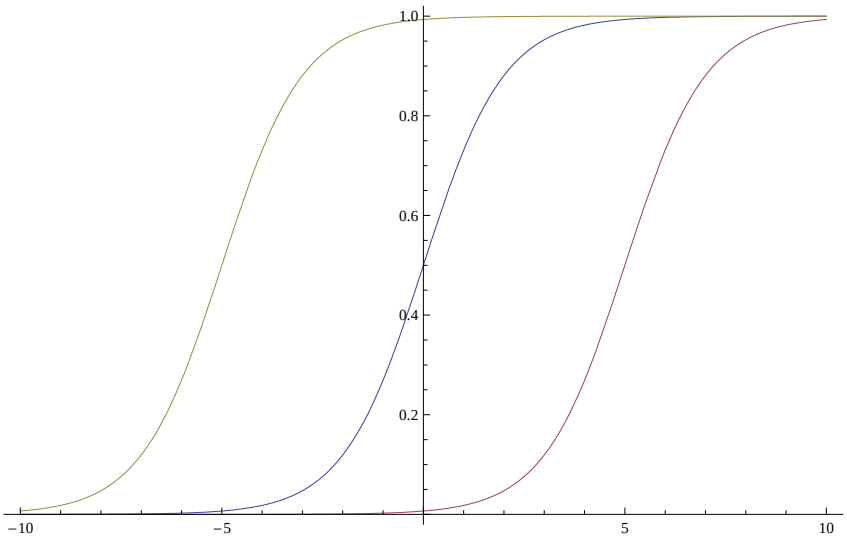

Figure 23: The effect of θ on the sigmoid-function. The bias values shown here, from left to right, are -5, 0 and 5.

goes well with Gaussian mutation. In addition to this we do not need to convert from bits to floating point numbers when we convert the genotype to phenotype.

Each genotype is built up of the CTRNNs time-constants, bias and connection weights. The initial construction of the genotype is set up as if the CTRNN was fully connected. A total genotype is simply a list of neurons. Each neuron is built-up of a time-constant, bias and weights. The list of the neurons weights are the weights that go from neuron j to neuron i(w_{ji}).

This type of genotype scales quadratically with the number of neurons of the network. Since each neuron has one time-constant and one bias, the size of the genotype is then $n^2 + 2n$ where n is the number of neurons in the network.

In our experiments we never allow the network to be completely fully connected. This would imply that we could remove some of the unused weights, but we quickly found out that we would not gain much(if any) performance gain by making the genotype smaller. Our GAs bottleneck is situated in the fitness evaluation of phenotypes, and not in genotype handling. Instead we set the unused weights w_{ji} to 0. A minor and positive consequence of not removing any weights, is that we can now determine the number of neurons by using the quadratic equation $n^2 + 2n = 0$.

4.4.2 Fitness functions

Arguably the most important feature of any genetic algorithm is the fitness function one choose to use. In Evolutionary Robotics [10] Nolfi and Floreano state that if the fitness function becomes too restrictive the overall possibilities for emergence are reduced. So we try to use fitness functions that do not specify much, but rewards general behaviors instead.

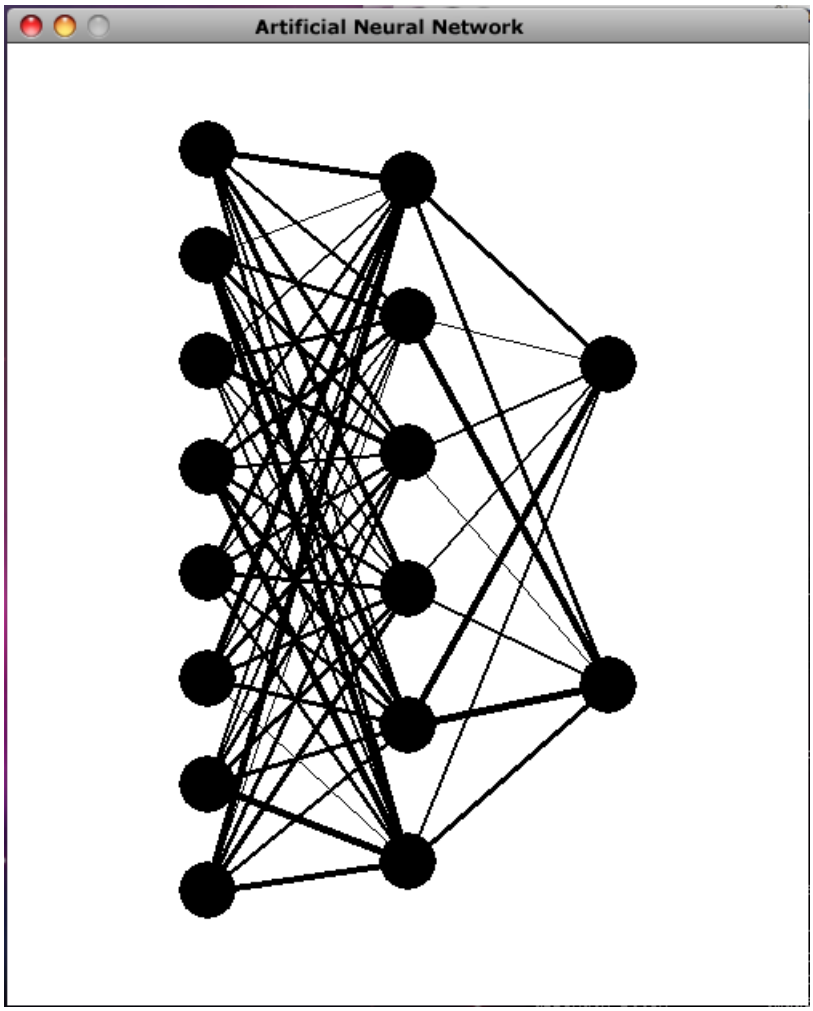

Figure 24: Visualization of EANN

Earlier results were obtained using this fitness function: $f = aW - bS$, where a and b are constants, W being the sum of wheel speeds counted each time cycle and S being the total sum of inputs counted each time cycle. The idea behind this fitness function is to reward movement and penalize sensor input. Simply put: high speed is good, being close to walls is not.

The fitness function f = aW - bS is very sensitive to a and b. If they are scaled such that a >> b, the network will often converge to solutions that drive full speed ahead while facing the wall. When a << b, the network tends to converge to solutions that move in circles, typically with just one of the engines moving forward such that it receives little sensor input, but does not score high on motor speed either.

Due to the problems described above concerning scaling we experimented with different variations of the fitness function. One of these variations was a fitness

function which added Φ each time cycle. This variation is taken from Evolutionary Robotics[10].

$\Phi = V(1 - i)$
$0 \leq V \leq 1$
$0 \leq V \leq 1$

Where V is the sum of the absolute values of each wheel speed (right wheel + left wheel) and i is the sensor reading of the most active sensor (ie. the one which is the closest to an object). Note that we rescaled our wheel speeds to be between -0.5 and 0.5, where -0.5 is full speed backwards and 0.5 is full speed forward.

$V = \frac{wheel_1 + wheel_2}{2}$
$i = max(sensors)$

The general idea behind this fitness function is the same as the first one, rewarding high wheel speed and penalizing sensor input. However, they both seem to give cyclic results. Therefore we also tried variation where one included the following component: $(1 - \sqrt{\delta v})$, where δv is the difference between the two wheels. This component rewards wheel speeds that are similar. Now rewriting the above function to $\Phi = V(1 - \sqrt{\delta v})(1 - i)$. This function now captures the general idea of what we want to see in our robot controller. First, it rewards high wheel speeds. Second, it rewards similar wheel speeds on both wheels, which rewards movement in straight lines. It also penalizes sensor input, such that staying close to walls is bad.

While the above fitness functions measure how well the robot is doing, it only measures how well actuators are doing and only relies on local variables that are known to the robot at any given time. However, it does not capture a general behavior. These fitness functions basically check "how fast does the actuator move" instead of seeing how well a solution is, from a more biological point of view. This and input from fellow researchers inspired us to write a fitness function that checks how good a total solution is. In order to do this we decided to base fitness on distance travelled. The main problem that had earlier steered us away from such a solution was the reality problem. In the simulator such a task is trivial, in the real world it is much less trivial. In the simulator one can get the coordinates of the robots location very accurately, this however, is not possible in real life. One solution to this is to integrate the wheel speeds, but this leads to another problem when friction fails! Since the robot could be running into a wall and spin the wheels at max speed and gain good fitness.

Even though the above problems do exist we decided to go through with the fitness function. We ended up with a Euclidean distance measure as a fitness function. It should be noted that Euclidean distance might not be a good way of solving mazes, since the robot then is forced to move a lot back and forth. Moving back and forth would give zero fitness. However, in our simple maze euclidean distance is a good measure, since the optimal solution is not backwards at all, but rather a steady increase in fitness. The closer the robot gets to the exit in our maze, the more fitness it will get, and should it be able to get out it will increase its fitness even more.

Euclidean distance fitness:
$S_p = (P_x, P_y)\ E_p = (Q_x, Q_y)\ ED = \sqrt{(P_x - Q_X)^2 + (P_y - Q_y)^2}$

Where S_p is the starting point, E_p is the end point and ED being the euclidean distance between the two points. ED is also the number we used as fitness for the phenotypes.

As a final variation of measuring fitness we combine the fitness produced by the ones described above. We can the scale the individual values up or down in order to see if we are able to get interesting behaviours. In the case of a combination of fitness functions we will describe how the combination is computed in detail. Since all the possible variations isn't going to be described in this section. We will document the configuration per experiment. A full description of the possible variations would take up too much space, and they would not be relevant unless used.

4.4.3 Crossover and mutation

Previous work had the most successful individuals using a 70% mutation rate and 30% crossover rate. This however, is not the case when we start testing on CTRNNs. Due to the dynamic nature of CTRNNs, crossover simply did not give smooth jumps in phenotype search space. Mutation seems to be the preferred way of evolving neural networks, specially highly dynamic ones such as CTRNNs.

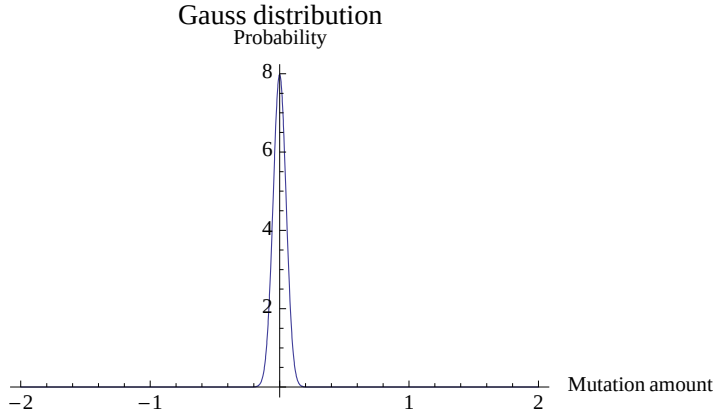

Figure 25: Gauss distribution with mean = 0 and variance = 0.05

We also experimented with different ways of performing mutation. Previously we had only used the classic "change one gene" approach. This type of mutation is commonly known as bit-flip mutation. One problem with the bit-flip approach, is the way it moves in genotype search space. The movement using classic mutation happens in one dimension only, unless you start manipulating multiple genes, which is what Gauss-mutation does. In all our experiments, Gauss-mutation is used.

4.4.4 Number scales

Early experiments lead to stationary circling behavior, no matter which evolutionary properties we tried. We then learned about the importance of proper scaling of all the numbers that is used with the CTRNN. Proper scaling of numbers is of great importance in order to get a system that behaves the way one should expect. Unless we encode the outputs from the network, the robot will be unable to move. The same principle applies for the sensors that are being set as input to the network. If one is using the wrong ranges the neural network can be totally blinded by the number and not respond to sensor changes. Interpolation is used on the sensors and wheel speeds such that their ranges are correct when used to set wheel speed, calculate fitness and as input to the neural network.

Sensors on the robot and in the simulator are scaled between 0 (no walls in sight) to 3000 (sensor practically touching the wall). This range will not work well with the logistic transfer function $\sigma = 1/(1 - e^x)$, showing in figure 26. The logistic transfer function is active when $-5 <= x <= 5$, and converge towards 0 when $x <= -5$ and converge towards 1 when $x >= 5$. In order to properly inject sensory information into the input neurons we set up the system to interpolate the sensor range [0, 3000] to the range [-10, 10]. This step is important, if we did not scale the number properly we would only see changes in output activation values from the input neurons when the sensor values were very small, ie. when the sensors are just beginning to see an obstacle.

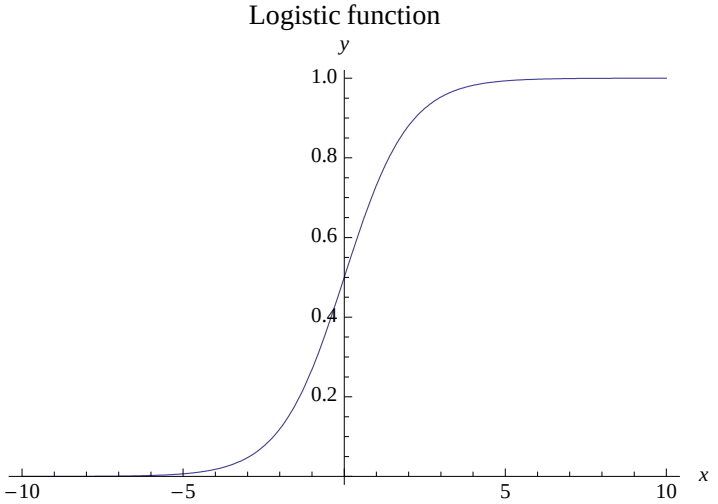

Figure 26: Logistic transfer function

Further use of interpolation is used in order to extract the wheel speed for each wheel from the neural network. The output from the neural network is as shown on figure 26 between [0,1]. These values are interpolated into the range [-1000,1000].

The fitness function $\Phi = V(1-\sqrt{\delta v})(1-i)$ also requires proper scaling of input i and wheel speed V. The first component V is the sum of the wheels. In order to

make the sum of the wheels make sense in this setting the original wheel speed range [-1000,1000](where -1000 is max speed backward and 1000 is max speed forward) is interpolated to [-0.5, 0.5]. Input i is interpolated from the range [0, 3000] to [0,1]. A thorough analysis of why these ranges are needed can be found in [10, p. 73].

Since the simulator was originally tuned to the E-puck robots, we used the same scales as we used for testing on the actual e-puck robots.

4.4.5 Thread problems

One of the major changes that were done compared to our earlier results was that the GA and simulator were running on a cluster in order to speed up tests. This gave us a whole set of new problems. Our simulator was coded thread-wise using one thread for the simulator (Newtonian physics, environment setup and sensor handling, etc) while the controller was running in a thread alone. Each node on the cluster is a 3GHz single-core machine. Our earlier machines where dual-core machines. One problem that this posed is that fitness evaluation varied a lot, and were not consistent. This where due to the nature of our fitness functions, which gave higher fitness with more cycles. On a dual-core architecture it seemed to have less of an effect than on the single-core nodes on the cluster, which it seemed to give huge differences. We suspect that this lead our GA to favour solutions that were computationally less expensive, and it still might do that.

The behaviour of CTRNNs also vary with how often you update and change the internal state, in our case using the Euler integration method. Using threads on different architectures(multi-core and single-core) and on different operating systems which has different schedulers can result in very different behaviours depending on the setup of the system. This happens because of the amount of time the thread is allowed to run. On a system where the thread is allowed to run practically in parallel with the simulator, it would naturally check the sensors, and update the internal state of the CTRNN more often. This resulted in very shifting behaviour depending on how often the thread was allowed to run. And in turn made us go for a non-threaded solution.

4.5 Simulation

Simulations were carried out in the Breve simulator. This allowed us to carry out both rendered and non-rendered simulations. We observed the robot running for each configuration in a rendered version to make sure that the initial setup performed as expected. The non-rendered simulation made it possible to run larger population sizes and increased amount of generations at much greater speed.

We created our own E-puck robot within the Breve simulator. It is a modified version of a simple Braitenberg robot, set up with eight sensors placed approximately at the same position as seen on the real robot. These sensors were tuned to somewhat match those on the E-pucks. The simulator environment was set up to match the real world table size and robot size.

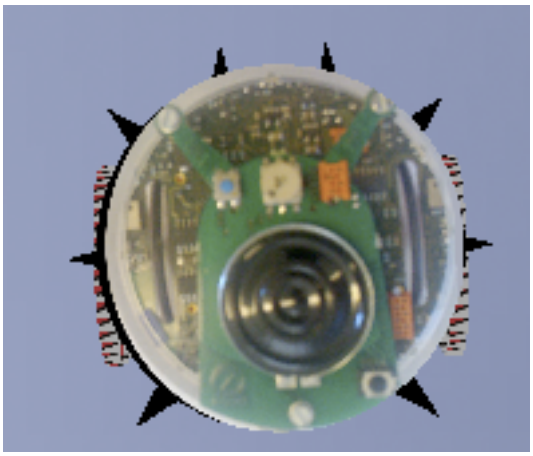

Figure 27: Breve simulator and the e-puck robot.

Weights found during evolution in the simulator were easily placed onto the real world robots for further evolution and tuning to the real world sensory mechanics. This was necessary because the weights found in the simulator is trained against simulated sensors. Sensors in the simulator are not affected by varied amounts of light or quality in the electronics. We therefore set the system up so we could evolve basic weights in the simulator to avoid the bootstrapping problem and then continue evolution on the real world robot. This would allow evolution to further improve the weights.

Running simulations became really time consuming. We experienced that fitness evaluation was the real time sink. Each phenotype took somewhere between 25 to 30 seconds in the simulator. This simulated approximately 80 seconds in real time.

This would mean that a population size of 50 over 30 generations would take in excess of 10 hours of simulation time. The solution to this problem was either to look into the code that Breve uses (and attempt to optimize this) to simulate or get access to a cluster and create a distributed version of the GA. We decided to go for the distributed version as our program turned out to be "easily" distributable.

If we assume the same simulation time per phenotype on a distributed architecture, using 50 nodes on a cluster the simulation time would be drastically reduced. We hope to be able to simulate a run with the same population size (50) and number of generations (30) in less than 30 minutes as our calculations predict. Some overhead is to be expected, however it would be insignificant.

After writing a distributed version of the evolutionary algorithm we confirmed our predictions.

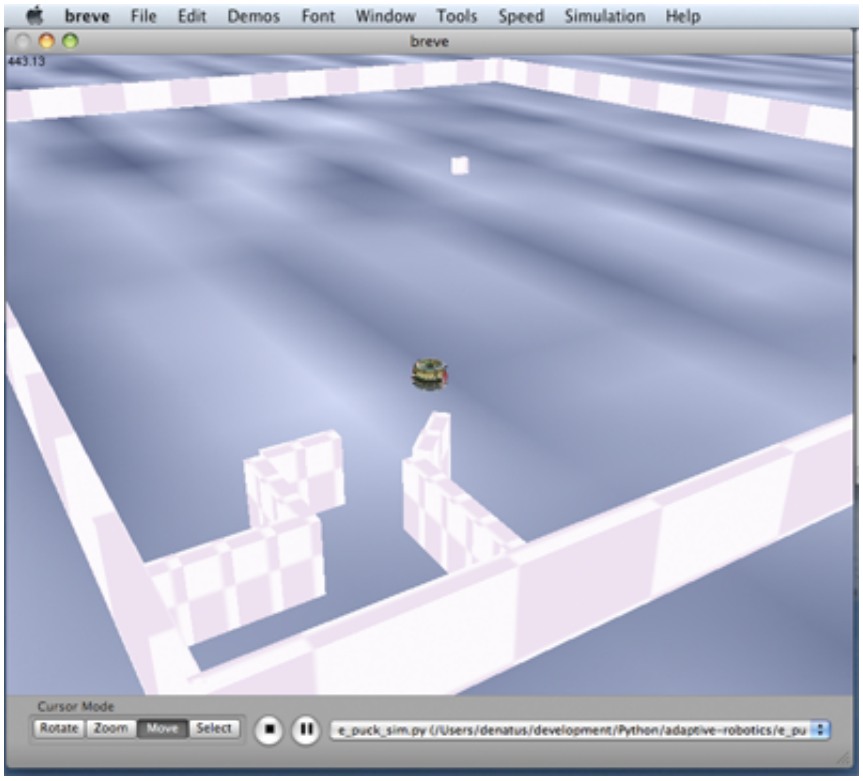

Figure 28: Visual run in the Breve simulator.

4.6 Cluster

In all our experiments we are executing an evolutionary search in a simulator. This process is computationally expensive, and would simply take too much time unless we were able to write a distributed evolutionary algorithm. By evaluating fitness in parallel we are able to drastically reduce the amount of time the evolutionary algorithm uses.

Our distributed version consist of a distributor program(master node) that distributes each fitness evaluation(slave nodes) onto the cluster nodes as shown in figure 29 on the next page.

In order to minimize the amount of time it takes to run the evolutionary algorithm from start to end, we set the size of the genotype pool to match the amount of nodes we use on the cluster. As an example to make this clearer: if we use 30 nodes on the cluster, we set the size of the genotype pool to be any number which is dividable by 30 and that will use all 30 nodes as much as possible, such as 30, 60 or 90.

By using a distributed model for our clustering, we are able to drastically reduce the amount of time the evolutionary algorithm takes to complete. For instance, if we use 30 nodes on the cluster and a population size of 60, and the fitness evaluation of one phenotype takes 1 min. Each generation would then take

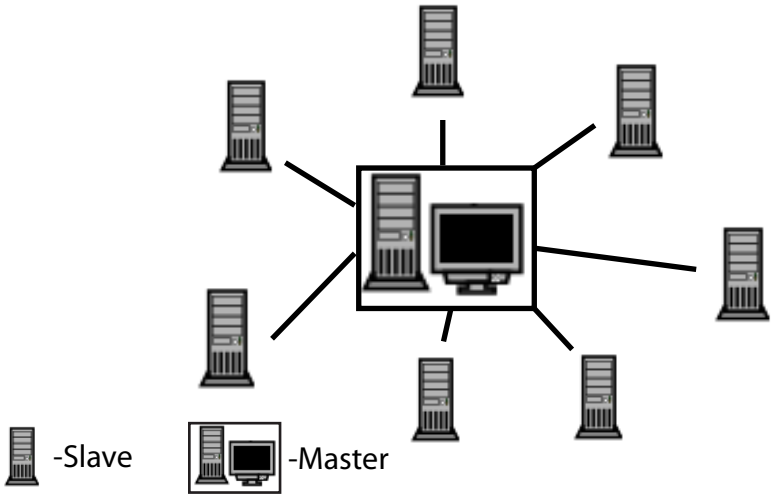

Figure 29: Distributed simulation

2 minutes. In a non distributed evolutionary algorithm, the same generation would take 60 minutes.

The model we use is called a master-slave model. This model has been been criticized by recent papers [33], but it seems to be the best solution we could find to distribute our genetic algorithm.

4.7 Evolution pipeline

This section explains how we conduct our search for the fittest phenotypes. The first step is the evolution in the simulated environment, and the second step is evolution on the robot.

The evolutionary process in the simulated environment begins with the genetic algorithm producing a set of random phenotypes containing synaptic weights, time constants and bias values. Each complete phenotype is transformed into a fully-fledged CTRNN by filling its attributes into a bare bones dynamical neural network.

Complete and ready for testing, the CTRNNs are passed onto the cluster's master node that distributes them onto a number of slave nodes running our simulated environment. The phenotypes (CTRNNs) are evaluated and assigned a fitness value according to their performance. Once all the phenotypes have been evaluated, the master node communicates the results back to the genetic algorithm. This loop, displayed in figure 30 on the following page, continues for a specified number of generations.

A summary of the procedure could be described as follows:

(1) Genetic algorithm: Produce synaptic weights, time constants and bias values and pass these to the CTRNN.

(2) CTRNN: Insert phenotypes into bare bones dynamical network and provide testable phenotypes for the cluster.

(3) Cluster: Distribute phenotypes to slave nodes. Wait for phenotypes to be evaluated (in parallel) and communicate the results back to the genetic algorithm.

(4) Repeat steps (1) to (4) for a given number of generations.

See illustration in figure 30 on the next page.

Evolving from scratch (random weights) on the robot would take an unimaginable amount of time and effort. However, the fresh phenotypes coming from the simulator are rarely well-adapted for real-life environments. Despite our effort to mimic the real world as closely as possible in the simulated environment, the weights rarely work without some continued evolution. This worked directly with the initial feed-forward neural network, but not so well with the CTRNNs. This problem is described in detail in the section concerning the "Reality problem (3.1.1)."

In this second stage of evolution (now on the physical robot), the last generation of phenotypes from the simulation phase is used for initial weights, time constants and biases. The genetic algorithm provides these weights to the CTRNN which in turn passes the CTRNN phenotypes to the robot controller. Note that the robot controller application resides outside, but communicates with the actual robot using Bluetooth. The phenotypes are evaluated and assigned a fitness according to their performance in the real-life environment. These fitness values are passed from the controller and back to the genetic algorithm. This loop, which is shown in figure 31 on the following page, can continue to run for

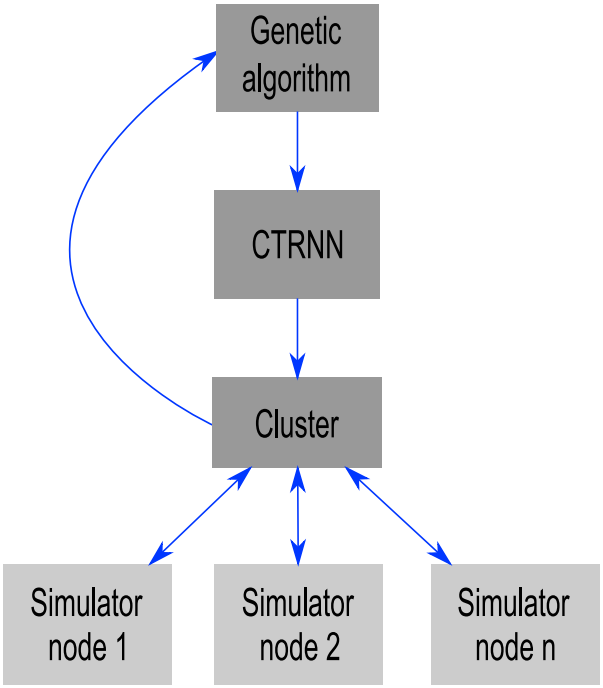

Figure 30: Run on simulated robot

several generations. In this case, however, it is rarely run for more than a few generations.

The general outline of this procedure is as follows:

(1) Genetic algorithm: Obtain the previously evolved genotypes and pass them to the CTRNN.

(2) CTRNN: Insert phenotypes into bare bones dynamical network and provide testable phenotypes to the robot controller.

(3) Robot controller: Communicate with physical robot using bluetooth, i.e. read sensory input, integrate neurons and pass motor speeds (actions) back to the robot. Evaluate fitness and pass it back to the genetic algorithm.

(4) Repeat steps (1) to (4) for a given number of generations.

See illustration in figure 31 on the next page.

4.7.1 Software

The Python Programming Language. All our code is written in Python. http://www.python.org

The Breve Simulation Environment. 3D simulation environment for ALife. http://www.spiderland.org/

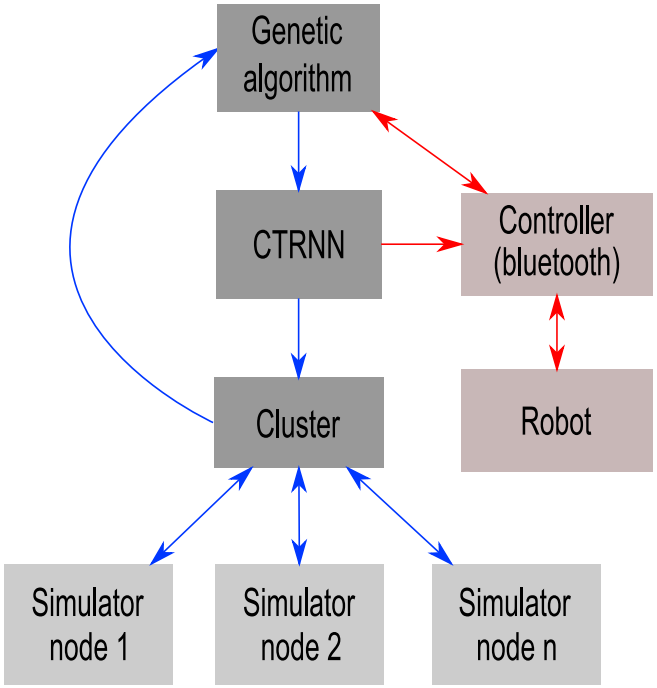

Figure 31: Run on physical robot

Mathematica. Used for creating graphs and phase portraits. http://www.wolfram.com

Dynamica. Randall Beer's addition to Mathematica. http://mypage.iu.edu/ rdbeer/

Inkscape. All the illustrations herein are made with Inkscape. http://www.inkscape.org

5 Experiments

5.1 Early experiments

We will present some of our experiments and the problems we ran into. We would like to begin with some of the early experiences we had with the robot. At this point we had already built and painted the environment, and the robot was ready for a test-drive. The process of getting the robot ready to be used is described in the form of a tutorial in Appendix A.

We decided to monitor four out of the eight infrared proximity sensors and drive around in the environment for a while. The sensors gave values ranging from 0 to 3000, 0 meaning that it was in the open, and 3000 indicating that it was touching one of the walls. The sensors are very stable and rarely affected by external factors. However, we ran into some problems that were difficult to locate, due to low battery power. When the battery runs low on power, some of the data is lost during transfer, and the sensor input is generally a bit lower than usual. We quickly decided that we shouldn't conduct any serious experimentation without full battery capacity. Figure 32 shows the input from a single sensor as the robot slowly backs away from a wall.

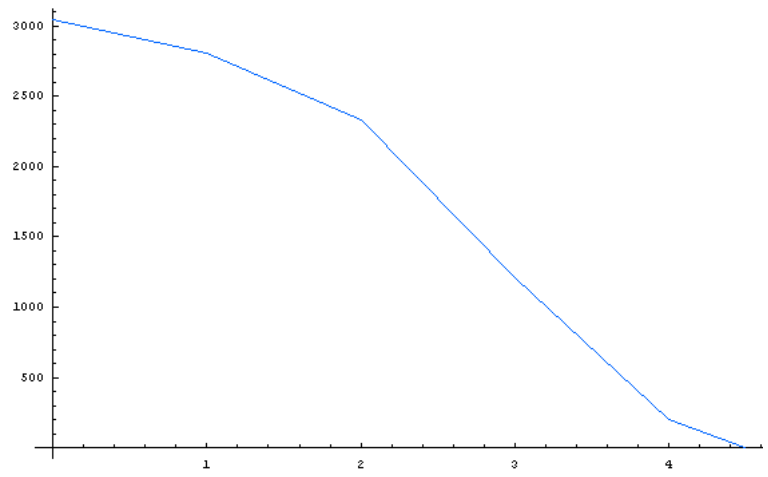

Figure 32: Sensor input at x cm from a wall.

Running into the walls didn't cause any problems as the robot is fairly light, and the floor was slippery enough for the robots' wheels to spin, such that it didn't cause much strain on the motors. The wheel speed can be programmed to run at -1000 (full backwards movement) to 1000 (full speed ahead) where 1000 equals one full rotation on the wheels per second. However, the maximum speeds are quite hard on the motors, so we set the range in our control software to [-500, 500] (500 equals the half of a full rotation on the wheels per second), which is sufficient for our use.

5.1.1 Obstacle-avoidance

The Evolutionary Artificial Neural Network was first put to the test with obstacle-avoidance in mind. The network was still reading input from four sensors and acting on both wheels, using a 4-2-2 topology. The artificial environment was created in the Breve simulator, with both the table and robot as similar to the physical environment as possible. This was achieved by modelling the robot and table in Breve, and linear interpolation of the sensors and wheels in the physical environment. The simulator comes with simulated physics, and this had to be adjusted such that it behaved as much like the real environment as possible. Evolution would typically exploit materials that were too smooth by sliding past them. In our physical environment, the obstacles were quite rough, such that the friction made the robot stop as soon as it hit them. This was solved by increasing the friction on our materials in the simulator. Many of the issues encountered in the physical environment, such as the friction-problem, did not occur in the simulated environment. Typically, only one out of five sets of synaptic weights would work as well in the physical environment as they did in the simulator.

Every individual was evaluated according to

$$\Phi = \sum_{i=1}^{100} (\alpha * W - \beta * S)$$

where i is a measurement at 100 fixed time intervals throughout a trial run. W is the sum of the speed on both wheels at time i and S is the sum of the input on all active sensors at time i, in this case four sensors. Each phenotype is rewarded for forward speed on the motors and punished for backwards movement and sensor input. Evolution was able to exploit this function by running into the wall while maintaining a high speed on the wheels in order to get a fairly high fitness score (and resulting in convergence to an early local optimum). This was solved by introducing the coefficients α and β and keeping $\alpha < \beta$.

After some time, the system evolved an interesting obstacle-avoiding behaviour. However, when driving front-first towards a wall, the evolved weights would cause the robot to either turn left, or right, every time. In other words, if the robot turned left the first time it faced a wall, it would always turn left, making it explore the environment in a circular path. Being unable to distinguish between two situations where the sensor (function) input is the same, is known as the "Perceptual aliasing problem" [28]. The problem scenario is shown in figure 33 on the next page.

5.1.2 Escaping the maze

The purpose of this next test was to see if the robot could solve the task of escaping a simple maze. In order to encourage this behaviour in evolution, the simulator was run for longer than usual, such that the individuals that managed to get out of the maze would benefit more from the time spent in freedom (zero sensor input).

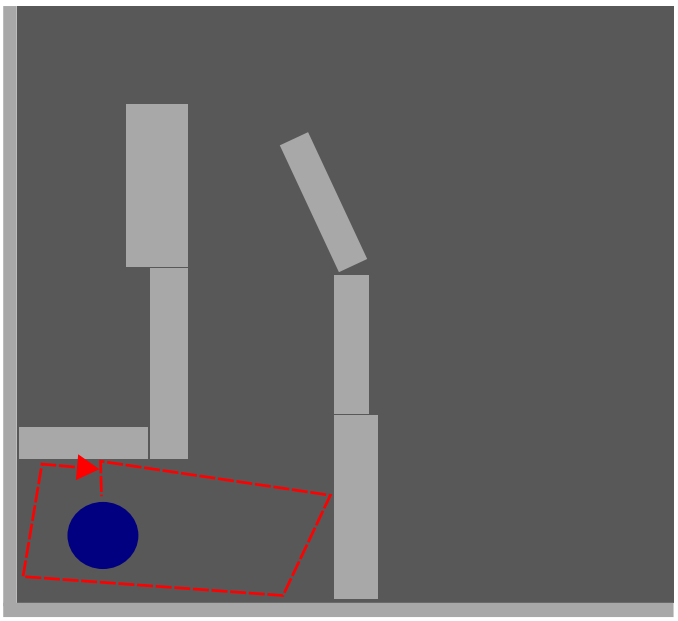

Figure 33: Circular path, using four sensors.

The individuals were evaluated according to

$$\Phi = \sum_{i=1}^{150} (\alpha * W - \beta * S)$$

with i set to 150 fixed time intervals for sampling this time around. Otherwise, the fitness function is the same as the one that was used in the previous test.

Note that all eight sensors were enabled in this test. This had some interesting effects on the robot. Normally when it came close to a wall, it would turn and drive away from it, but still following it from a small distance. With all sensors enabled, it would be punished for staying close to the wall while it turned. This evolved a robot that would back up before turning, and it would also stay further away from the walls in general.

The path taken from one of the best solutions is shown in figure 34 on the following page.

5.2 Perceptual aliasing

The problem of perceptual aliasing as discussed earlier became a problem when we used a classic feed-forward network. The robot was unable to take different decisions when presented with the same input at two different occasions during the same run. In order to solve more complex tasks, our robot should optimally

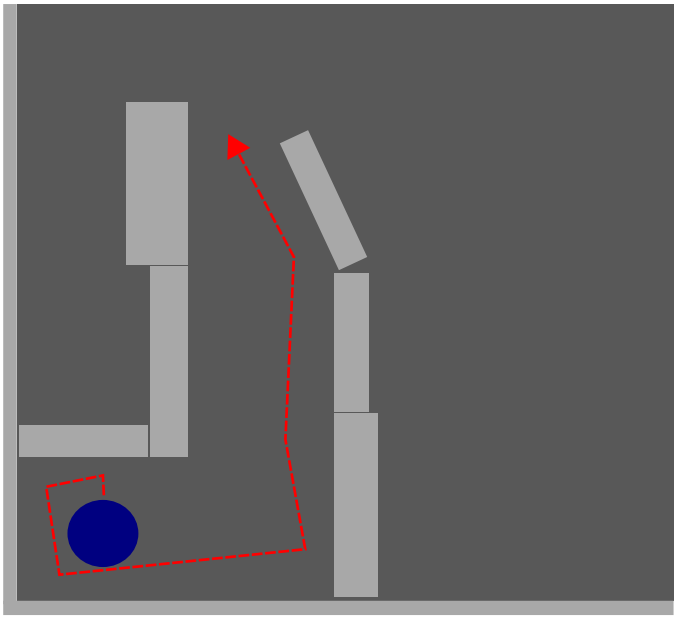

Figure 34: Escaping the maze using eight sensors.

be able to produce different behaviour over time when presented with similar sensory input, depending on the time it receives the input. Our goal was to explore CTRNNs and to see if they could provide the robot with the ability to use perceptual aliasing to our advantage, ie. receive sensory input from some sensor x and behave differently on two separate occasions during a run, when sensor x has the same input.

In order to quickly check that our old artificial neural network and our CTRNN are in fact different when it comes to the ability to solve the problem of perceptual aliasing, we wrote a small test program. In this program we could insert weights found during evolution. We would then be able to see if they had properties that would give different output using the same input at two different times.

This program feeds the networks with 2500 inputs in total. At iteration 100 and 2500 we present the network with one particular input string. The rest of the time we present the network with complete random inputs. The range on the random input is between [-10,10].

For the following tests we are using a 4-4-2 topology without any recurrent neurons. A classic ANN will be used in the first test.

From table 5 on the next page, we can see that net state is the output from each neuron. Not surprisingly the classic ANN does not change its output from iteration 100 to iteration 2500. Since there is no dynamic internal states or weight changes during the iterations. This behaviour of the ANN is expected.

Iteration 100	Running known input((0.5, 0.5, 0.5, 0.5))
Net state	[0.46211715726000974, 0.46211715726000974, 0.46211715726000974, 0.46211715726000974, -0.45055625211872402, -0.020344305625434184, 0.056202296874276025, 0.25425225902456011, 0.059505989381954776, 0.22542321054369299]
Iteration 1000	RANDOM input((4.5852628077288067, 5.735389737880805, -2.8694498683165186, 4.97480286616711)):
Net state	[0.99979189805773494, 0.99997913928276294, -0.9935840339232419, 0.99990451164901095, -0.66419748077911778, -0.39364684157408286, 0.42122002047065987, 0.87513899077800583, 0.21262160374233, 0.7295461580377498]
Iteration 2500	Running known input((0.5, 0.5, 0.5, 0.5)):
Net state	[0.46211715726000974, 0.46211715726000974, 0.46211715726000974, 0.46211715726000974, -0.45055625211872402, -0.020344305625434184, 0.056202296874276025, 0.25425225902456011, 0.059505989381954776, 0.22542321054369299]

Table 5: Classic ANN presented with input over 2500 iterations

When we do the same to a CTRNN using the evolved time constants and evolved weights we will be able to see if the state of the network is enough to produce output which are different. Even when presented with the same input. The following test uses a CTRNN.

From table 6 on the following page we can see that output states illustrate the state of each neuron. We would read motor data from the two last numbers in this list. From iteration 100 to iteration 2500 we can see that there is a change. This happens even when the network is presented with the same input at both iteration 100 and iteration 2500. The time constants alone are able to change the internal state of the network over time, so that it is able to produce different outputs.

With this knowledge we are sure that, at least in theory the CTRNN has the capabilities of solving our perceptual aliasing problems. It should be noted that this is a quite crude test, it is however mathematically provable that CTRNNs have these capabilities [17].

5.3 Recent experiments

Instead of making a major leap from a simple feed-forward network to a full CTRNN, it might be more interesting to gradually extend the old network with new concepts.

With a CTRNN of the same topology and connectivity as the old feed-forward network, a reasonable first step would be to evolve time constants to see how they affect the behaviour. The question then becomes, how many features do we need

Iteration 100	Running known input((1, 1, 1, 1, 0, 0, 0, 0, 0, 0))
Output states	[0.19978596665581355, 0.29870836298144993, 0.16686415629477661, 0.48560578598064924, 0.40379589422399653, 0.81667689906878027, 0.84015543998079789, 0.20287252625492952, 0.99349197156286617, 0.48451966878439079]
Iteration 1000	RANDOM input((0.63760519271426119, 3.5434957725672618, -5.5737544426594976, 2.878331948224325, 0, 0, 0, 0, 0, 0)):
Output states	[0.19332512278743358, 0.34608660588111917, 0.11692019316210628, 0.44102604643444293, 0.21778146983377095, 0.92755922729140083, 0.90800049543878703, 0.16623158732042792, 0.97184723637073223, 0.42018039345697511]
Iteration 2500	Running known input((1, 1, 1, 1, 0, 0, 0, 0, 0, 0)):
Output states	[0.52861861763771201, 0.45158195100837639, 0.23314033249068122, 0.24240861400238797, 0.3300208638115219, 0.85845015314490203, 0.95286425367397454, 0.079893981788752594, 0.99526391646505319, 0.33749464858110939]

Table 6: CTRNN presented with inputs over 2500 iterations

Evolved weights	Recurrence	Time constants	Biases	Self-connections
Yes	None	None	Single	None

Table 7: Configuration features

to add before we get enough dynamics for our tasks, is time constants enough or do we need to go further, and add recurrence? Another interesting question is whether or not added complexity to the CTRNN will result in more complex behaviour. Observing how evolution behaves with more complex CTRNNs is also interesting.

The following tests have the selection mechanism in common. The selection mechanism used was rank selection. These are the first series of testing, starting without recurrence. It is important to add that even if we have no recurrence the CTRNN will have possibility for dynamics which avoids the aliasing problem encountered using a normal feed forward architecture. The reason for this is that the CTRNNs that are being produced are using evolved time constants. Evolution can quickly evolve time constants which can be large enough so that internal dynamics begin to emerge.

The old feed-forward network had evolved weights but a single firing threshold (bias) on all the neurons:

Evolved weights: This binary state tells us whether there is synaptic plasticity during evolution. Note that there is no plasticity during the evaluation of phenotypes, i.e. there is no learning through synaptic plasticity. Without plasticity during evolution, it is possible to look at what what degree of complexity,

in terms of behaviour, it is possible to produce by exploiting neuron dynamics.

Recurrence: This can take on the values: none, limited or full. None meaning the network defaults to feed-forward connections only. Full recurrence means that each neuron connects to all the other neurons, but not to itself unless that is specified in table 7 on the previous page. The various conductivities are displayed graphically, showing the details of the limited recurrent compositions.

Time constants: There can be multiple, single or no time constants. Multiple indicates that each neuron has its own constant whereas single indicates that every neuron shares the same constant. None means that the time constant is ineffective (defaults to 1). In our experiments we allowed for multiple time constants. Evolution was allowed to evolve each individual time constant.

Biases: The biases here are equivalent to the firing thresholds in the old feed-forward network. This entry can take on the values: single, multiple or none, where single means that every neuron shares its bias value with all the other neurons and multiple means that every neuron has its own bias value. None means that the bias is ineffective (defaults to 0). In our experiments we allowed for multiple biases. Evolution was allowed to evolve each individual bias.

Self-connections: This indicates whether the neurons have self-connections, and takes on the values: full, limited or none, where full means that every neuron connects to itself, limited means that some of the neurons have self-connections and none means that there isn't a single neuron with a self-connection.

The idea here is to incrementally extend upon the old network and analyse the results, until it forms a complete CTRNN. Hopefully, this will provide some reasoning about the various parameters in the CTRNN model.

Experiments will be presented in the following matter:

1) Experiment showing that CTRNNs solve the old (from the prestudy) maze quite easily.

2) Experiments in the extended maze.

3) Experiments with slow incrementation in added complexity to the CTRNNs (more recurrence).

Each experiment will roughly follow this schema:

1) Evolutionary parameters

2) Graphical representation of the network, displaying which nodes were recurrent.

3) Figure and description of how the robot moves in the simulator.

4) Figure and description of how the set of weights work on the real robot.

5) Brief per experiment conclusion/explanation.

5.4 Experiment with the first simple maze

When we first started to evolve CTRNNs, we continued with the same maze that we used in our earlier work.

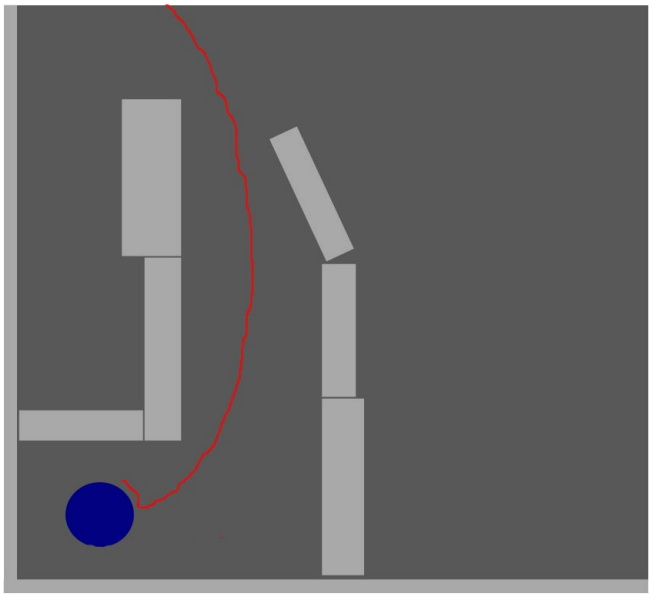

Figure 35: Evolved CTRNN in the old maze

Figure 35 illustrates one of the solutions evolution found for us. The solution angles the robot so it can start on a large circle which moves the robot out of the maze. Evolution quickly found these circling ad-hoc solutions for the maze. The angle made the robot able to escape the maze. But it is so fine tuned to the environment, that even small changes to the environment will make the angle useless and the robot will be unable to escape the maze. These types of non-adaptive ad-hoc solutions are not the types of solutions we find interesting. This led us to extend our maze in an attempt to get results which could be more interesting.

5.5 Experiment 1: Time constants

In this experiment we increase the complexity from a classic feed-forward neural network by using CTRNNs. We use the same 4-4-2 typology but we allow for evolved time-constants.

5.5.1 Evolutionary parameters for experiment 1

The evolutionary parameters for experiment 1 are displayed in table 8 on the next page.

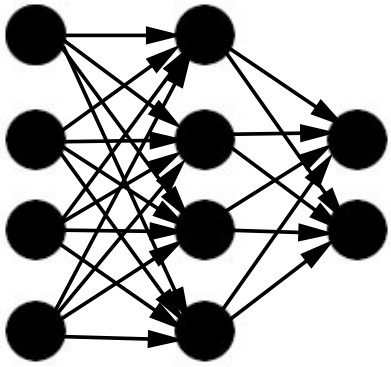

Figure 36: CTRNN connectivity for experiment 1

Population size:	60
Generations:	300
Run-time pr phenotype:	130 seconds
Network configuration:	4, 4, 2 topology, see figure 40 on page 65
Elitism:	off
Mutation parameters:	Gauss-mutation
Gauss parameters:	$\mu(mean) = 0$, $\sigma(variance) = 0.09$
Selection mechanism:	Rank-selection.

Table 8: Evolutionary parameters for experiment 1

Fitness function: Euclidean distance. In addition to this, we added a phenotype termination mechanism. The method gives each phenotype 20 lives. One of these lives is lost when the phenotype has more than 2050 sensory input. This is made so that evolution avoids wall-huggers (which seem to be appearing frequently, unless we stop them). The anti wall-hugger mechanism was added in an attempt to make evolution find solutions more suitable for real-life transfers over to our robots. From experience we noticed that wall-huggers have a high probability of not working on the real robots due to issues with friction.

In figure 37 on the next page we do not plot the minimum fitness for each generation. The reason for this is that the minimum fitness is always 0. In each generation there is at least one individual that triggers the anti-wall hugging mechanism, leaving it with 0 fitness.

As seen on figure 37 on the following page there is a large difference in maximum fitness and average fitness. We are lead to believe that this is due to the fact that not many actually pass the anti-wall hugging mechanism. And this in turn holds the average fitness down. However, evolution seem to climb steadily and continue to find better solutions which leads to a steady increase in fitness. The fitness plot shows an increase in average fitness and an overall increase in maximum fitness.

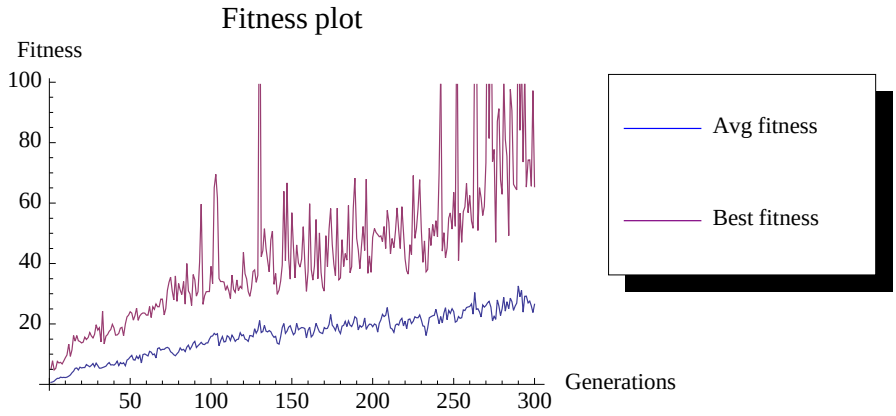

Figure 37: Fitness plot for experiment 1

5.5.2 Observations in the simulator

Figure 38 illustrates how the robot moves in the simulator. The best solution in the last generation had a fitness of 65 when running on the cluster. When it ran on the test machine it had a fitness value of 58. This value is within the expected +/− 20 range. The difference in fitness on the cluster machines and test machines will be discussed and analysed later.

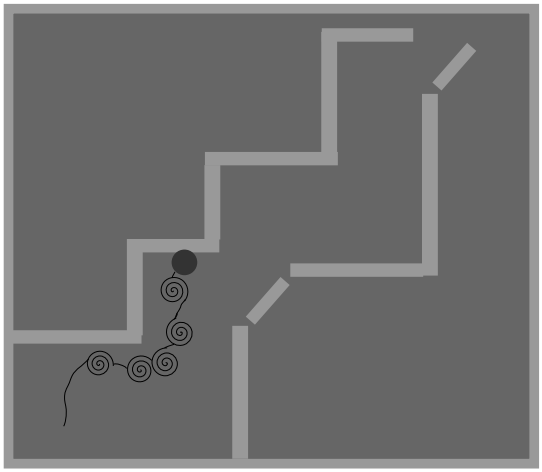

Figure 38: Robot running in the simulator for experiment 1

The robot starts off by running straight forward. It then starts to progress through the early parts of the maze in a circling behaviour. The circling motion is clockwise. During our experiments that use the anti wall-hugging mechanism, spinning behaviour was rewarded with a high fitness value.

When the evolved robot is allowed to continue its run past the time limit used during evolution, it quickly drives straight into the wall and ends there. This occurs shortly(1-2 seconds) after the time-limit is passed. This lead us to believe

that the CTRNN does not generalize, but has instead evolved to maximize its fitness to fit the 130 simulated seconds.

5.5.3 Observations on the real robot

Figure 39 illustrates the behaviour observed when moving the controller onto the real robots.

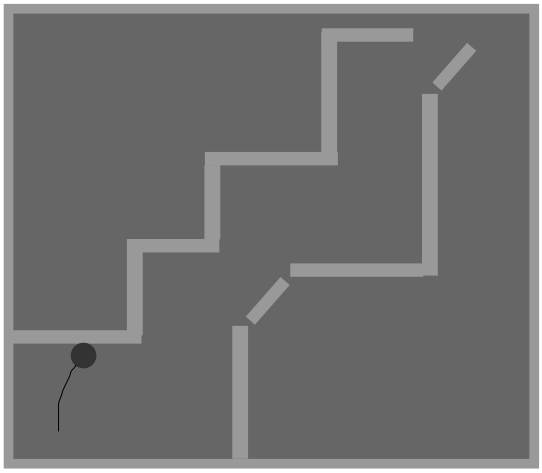

Figure 39: Real robot run for experiment 1

As figure 39 illustrates, the behaviour observed in the simulator is not preserved. The robot instead runs straight into the wall and gets stuck. Both wheels continue their forward speed. The robot seems blind, and does not react to sensory input once it gets close to the wall.

5.5.4 Brief discussion

In a faster simulator it might prove useful to allow evolution to run through more generations. The anti-wall hugging mechanism in essence limits the evolutionary search space of good fitness-values, however it does not limit the size of the genotype search space.

Already the CTRNN is showing promising behaviour in terms of solving the perceptual aliasing problem we experienced using a classic feed-forward ANN. When the robot sees the first wall, the general movement is changed so it can move towards the right. It then sees another wall, and instead of another right turn it is able to turn left. This behaviour is achieved by the robot using spinning behaviour. This spinning behaviour enables the robot to exit the spin at different times so it can head in a direction which is appropriate.

The spinning behaviour observed in the simulated experiment did not transfer well onto the robot, and is heavily affected by the reality problem. Using the anti-wall hugging mechanism could have affected how adaptive the robot is to

Population size:	60
Generations:	300
Run-time pr phenotype:	130sec
Network configuration:	4,4,2 topology, see figure 40
Elitism:	off
Mutation parameters:	Gauss-mutation
Gauss parameters:	$\mu(mean) = 0$, $\sigma(variance) = 0.09$
Selection mechanism:	Rank-selection.

Table 9: Evolutionary parameters for experiment 2

transfer onto the real world. We will have to see what will happen if we disable it.

5.6 Experiment 2: Time constants, no self-connection, no. 2

The previous experiment was made with a rather strict fitness function, which kills phenotypes when they spend too much time too close to the wall, which ends up with either wall sliding or wall-hugging behaviour. However, the result did not prove functioning on the real robot. In order to see what kind of behaviour we would get without the strict phenotype we do the same evolutionary run except that we do not enable the anti wall-hugging mechanism. We continue with the same topology and connectivity as in experiment 1.

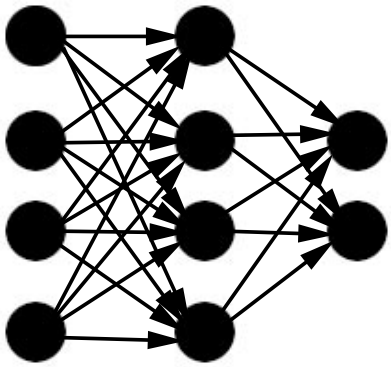

Figure 40: CTRNN connectivity for experiment 2

5.6.1 Evolutionary parameters for experiment 2

The evolutionary parameters used in experiment 2 is displayed in table 9.

Fitness function: Euclidean distance.

Since we removed the anti-wall hugging mechanism it is now useful to plot the minimal fitness per generation, see table 41 on the following page. Evolution

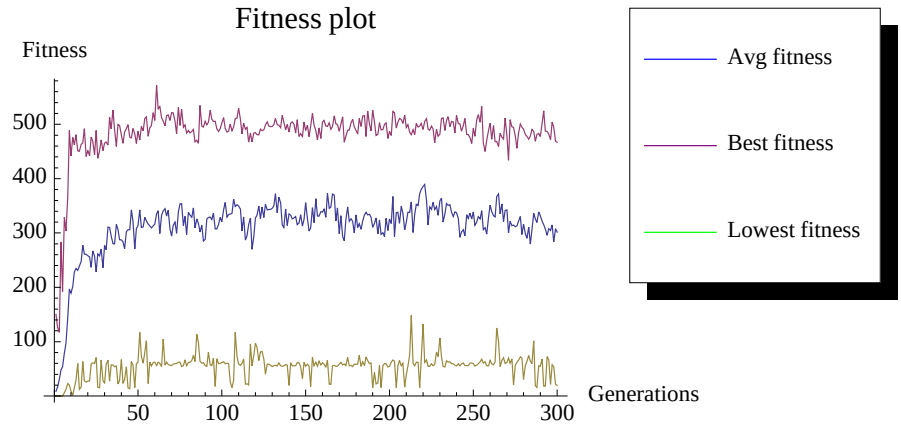

Figure 41: Fitness plot for experiment 2

seems to converge much faster without the anti-wall hugging mechanism, which leads us to believe that we have converged onto a local minimum. Convergence onto a local minimum might be easier as we allow for individuals that slide through the maze as well as those that are able to travel through the maze without hugging the wall often. Even the maximum fitness seems to spike much less. It is also interesting that the overall fitness is so much higher than without the anti wall hugging mechanism. It is likely that evolution finds it easier to discover individuals with high fitness when it is not restricted to solutions that have to avoid walls to the same extent.

5.6.2 Observations in the simulator

Figure 42 on the next page illustrates how the robot moves in the simulator. The gray line indicates the 130 second simulated time run. While the red line indicates how the robot behaves when it is allowed to continue past the time-limit used during evolution.

The gray lined run is good, and it gets very high fitness, the resulting fitness is 467 (compared to experiment 1 which got a fitness of 58). From the illustrations it can seem that this is unusual. However, the run from the end point of experiment 1 to the end point in experiment 2 is slightly longer than what the illustrations may imply, in the simulator. At first glance this is a very good solution found by evolution. It has many desirable behavior characteristics. One of these is that it does not touch the wall that often. However, this solution would not pass our anti-wall hugging mechanism. When running in the simulator this is shown by the way the robot handles the section with the narrow gap mid-run. In order to get through the gap the robot actually hugs the walls closely as it spins around.

Interestingly, when the robot is allowed to continue its run after the time-limit used during evolution, it goes into a dead-lock in the outer parts of the maze. This is illustrated by the red line. Evolution seems to have found a solution

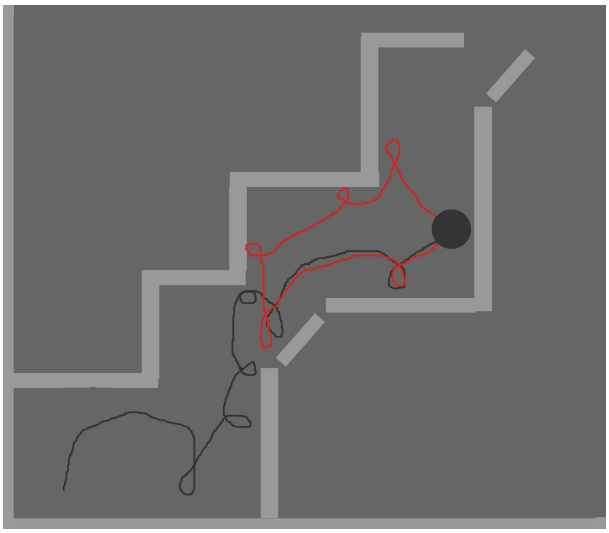

Figure 42: Movement observed in the simulator for experiment 2

that is tailored for the 130 seconds run time. This in turn leads to the robot not being adaptive.

At this point it was hard to see if the robot was actually responding to input or if it was the internal state changes alone that was responsible for the behaviour. In order to test this we placed the robot in the middle of the table, without any walls nearby. We then observed the robot doing only one large circle instead of the turn which makes the robot head in the right direction. This leads us to conclude that this CTRNN is indeed in need of sensory input in order to make decisions.

5.6.3 Observations on the real robot

When the robot controller is placed on the real robot we observe quite different behaviour from that which we observed in the simulator as illustrated on 43 on the following page.

The robot starts off driving straight ahead. It then starts to turn right. This behaviour is also seen in the simulator. However, the robot does touch the wall in the real world and then starts in a circular movement pattern. This, in turn leads to the robot getting stuck on the right side of the first wall.

5.6.4 Brief discussion

In the simulator the robot is yet again able to spin its way out, avoiding the aliasing problems we experienced using classic feed-forward ANNs. The controller is able to do both left and right turns by ending a spinning behaviour at the correct time. This type of behaviour is frequently seen in our CTRNN experiments.

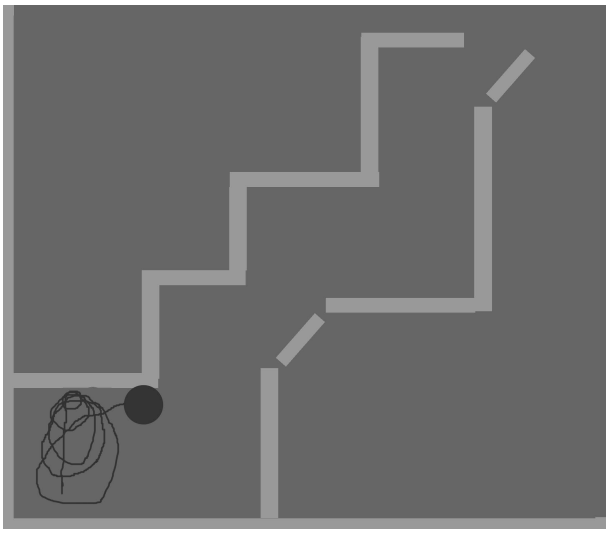

Figure 43: Movement observed on the real robot for experiment 2

As a simulated result this is a fairly good one. Sadly, this is not the case when the controller is placed onto the robot in the real world.

It is hard to understand why the controller fails. At this point we believe there is a series of reasons for this solution not to work. First of all the simulator updates the internal state of the CTRNN much more often and at a slightly more predictable way. The real life robot can suddenly suffer from 0.2 second Bluetooth lag, which can cause noise which in turn leads to instability.

In this experiment we also tested the CTRNN to see if the sensory input was needed to produce behavior. Since the CTRNN was in need of sensory input we are lead to think that the difference from our simulator to the real world is too big for it to produce the same behaviors.

5.7 Experiment 3: Time constants and one self-connection

As an extension from experiment 1 and experiment 2, we are adding recurrence to one single node. This would make us able to see if the added complexity can give us novel behaviour, even though it may only work in the simulator. In this experiment we increase the complexity from a CTRNN without any recurrence to a CTRNN with one self-connecting node. We will continue to use the same 4-4-2 topology. In addition to this we will allow evolution to run through 150 additional generations.

5.7.1 Evolutionary parameters for experiment 3

The evolutionary parameters used in experiment 3 are displayed in table 10 on the next page.

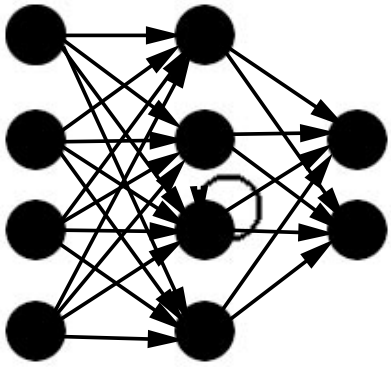

Figure 44: CTRNN connectivity for experiment 3

Population size:	60
Generations:	450
Run-time pr phenotype:	130sec
Network configuration:	4,4,2 topology, see figure 44
Elitism:	off
Mutation parameters:	Gauss-mutation
Gauss parameters:	$\mu(mean) = 0$, $\sigma(variance) = 0.09$
Selection mechanism:	Rank-selection.

Table 10: Evolutionary parameters for experiment 3

Fitness function: Euclidean distance and the anti-wall hugging mechanism. Experiment 2 resulted in some interesting behaviour. However, most of the time we get results that are exploiting friction, and keep sliding on the walls. These results may look promising in the simulator and result in a high fitness value, but it is unlikely that they work on the real robots. This is the reason why we want to continue to experiment using the anti-wall hugging mechanism.

5.7.2 Observations in the simulator

We were unable to produce any good phenotypes using only one self-connection in the middle layer. All attempts converged on an average fitness between 7 and 10 with the highest fitness being around 10. We are also uncertain why our experiments using this configuration have failed. One possibility is that CTRNNs using only one self-connection has a very difficult search space. It is also possible that the neurons on one side of the network saturate, giving a higher chance of spinning behaviour.

5.7.3 Brief discussion

Since we were unable to produce any interesting controllers using only one self-connection, we decided to try to expand the CTRNN with one more self-connection.

Population size:	60
Generations:	300
Run-time pr phenotype:	130sec
Network configuration:	4,4,2 topology, see figure 45
Elitism:	off
Mutation parameters:	Gauss-mutation
Gauss parameters:	$\mu(mean) = 0$, $\sigma(variance) = 0.09$
Selection mechanism:	Rank-selection.

Table 11: Evolutionary parameters for experiment 4

5.8 Experiment 4: Time constants and two recurrent nodes

This experiment will extend the complexity of the CTRNN using two recurrent nodes.

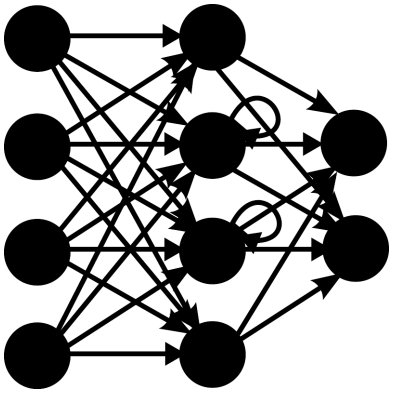

Figure 45: CTRNN connectivity for experiment 4

5.8.1 Evolutionary parameters

The parameters used are shown in table 11.

Fitness function: Euclidean distance. In addition we make use of the phenotype termination algorithm in order to penalize wall-hugger behavior.

As shown in figure 46 on the next page, a steep climb can be seen very early in the evolutionary run, quickly followed by a slow period of minor increase in average fitness. Evolution seems to climb abruptly towards the end, then followed by a convergence in average fitness.

5.8.2 Observations in simulator

The solution seems at first glance to be a rather robust one, as long as the rhythmic behavior is preserved when the robot is about to hit obstacles. A test

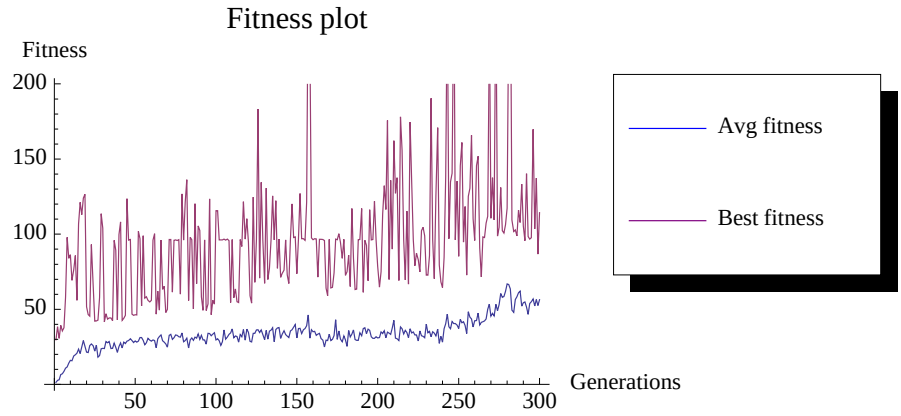

Figure 46: Fitness plot for experiment 4

needs to be done here, in order to see if the CTRNN hits a steady-state, which is often what will happen over long periods of time.

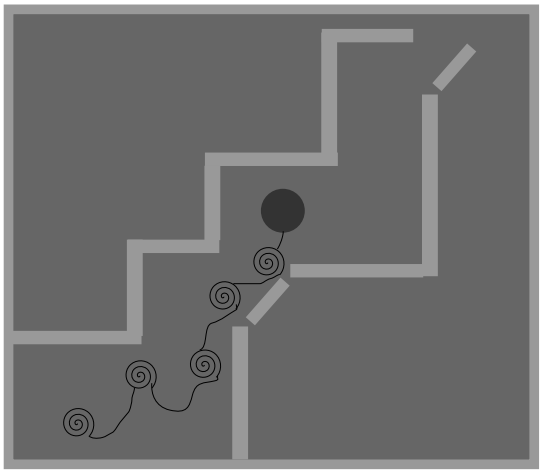

Figure 47: Robot movement in the simulator for experiment 4

Figure 47 shows how a rhythmic behavior between a stationary spin and movement makes the robot travel toward the exit. Rhythmic movement like this is typical for CTRNNs. An interesting question about this type of behavior is to see if the behavior is a result of the combination of sensory input and the internal dynamics. Rather than the CTRNNs internal dynamics alone.

We allowed the simulator to keep running past the time limit in order to observe where the robot would end up. The robot stopped progressing through the maze approximately halfway through. Circling behaviour back and forth was observed when the robot stopped progressing. Evolution seems to have found a solution that progresses through the maze in the given time-limit used during phenotype fitness evaluation, and this solution does not generalize situations well.

5.8.3 Observations on the real robot

As seen on figure 48 the same result was not observed when transferring the evolved CTRNN to the real robots. Instead a small forward drive quickly followed by continuous circling at the same location.

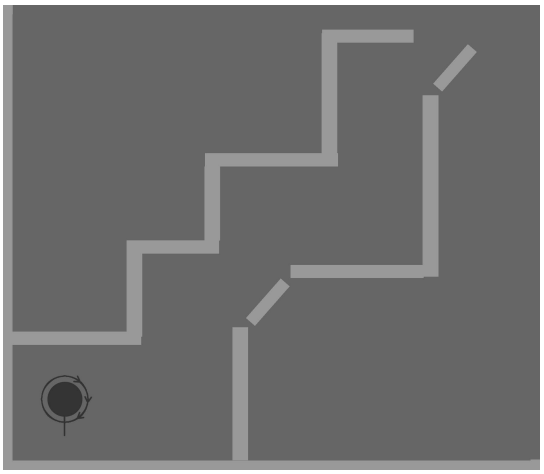

Figure 48: Robot movement on the real robot for experiment 4

5.8.4 Brief discussion

It is clear that our current approaches give decent results in the simulator. However, they do not work on the real robots.

5.9 Future experiments - exploring

Using the simulator without particular modifications or tricks does not seem to work in our experiments. Other research seems to get better results in the real world by adding a lot of noise into the system [27]. As illustrated by the above experiments, noise would have to be added throughout the system, and not just to the sensory data. Varying time limits, noisy sensors and perhaps even noisy time on updates of the internal states could be useful. Behaviour in experiments with added noise would be interesting to see.

Another problem is that the simulator updates the CTRNN a lot more often than what the real world robot is able to. Slowing down the amount of updates to further match the real-world would also be interesting to experiment with.

In addition to this, our goal is to create robots that are able to adapt to the real world. This means that we really want to behavioral diversity. Implementing awards for behavioral diversity into the fitness function is also something we want to experiment with [34].

As a final attempt we did try to evolve CTRNNs with some noise added to the sensors and with reduced update rate on the CTRNN. However, there was not

enough time to produce any good results, considering the amount of time evolution and analysis takes. It could be interesting to see some further experiments with the solutions discussed above.

5.10 Analysis and future work

In our observations we found that the CTRNNs internal state over time is the hardest part to fit onto a real world robot. The internal state that is evolved on the cluster hardware changes in the simulator when run on different hardware. This in turn makes the real life robot behave differently, although similarities can be found. Another interesting but logical consequence is changes in the code which makes updates slower or faster. This affects how often the internal state is being changed and can make the same pair of weights generate different behaviors. This occurs because the behavior is determined by the internal state over time which is dependent of previous states. A delay in update rate will in time result in different behaviour even when the network parameters stay the same.

We unknowingly encountered a similar problem in our early experiments. This led us to remove the threaded code, which minimized the difference from the single-core CPU on the cluster to our dual-core test machines. However, in experiments using the "anti wall-hug" mechanism, we observed that individuals which survive on the cluster, can trigger the "anti wall-hug" algorithm on our test machines. However, the fitness values found on the cluster are very well matched against those on the test machines. Then we found that the difference is in general around $+/-$ 20. Individuals that are found to be the best at the end of each evolutionary run all performed according to their fitness. This led us to believe that the general behavior of the phenotype is preserved when they are moved from the cluster onto the test machines.

When we move the phenotype CTRNN from the test machines onto the robots we do not preserve this behavior. Our CTRNNs seems to fail in most cases. Based on the above reasoning we believe this is caused by how often we are able to update the internal state during evolution and on our test machines compared to how often this is done on the robots.

In an attempt to avoid wall-hugging behaviour, we experimented with a phenotype killer mechanism. The mechanism succeeds in eliminating wall hugging phenotypes. However, due to the nature of this mechanism, evolution seems to create circling behavior to a greater degree. The phenotypes also seemed reluctant to generalize and to progress further when allowed to run on extended time. It seems that evolution tend to favour solutions that are circling when we use this mechanism. This could happen because the circling behaviour makes sure that no sensor are in touch with the wall for many ticks. In addition, we suspect that circling behaviour can make the robot more careful on its direction. In general, our observations using the anti wall-hugging mechanism tends to evolve robots with a circling behaviour.

5.11 Debugging

Due to the complexity of these systems and the way they interact, it can be difficult to find the cause of undesired behavior. The user has to evaluate multiple steps in the evolutionary process as well as the end result - the dynamical neural network. With a series of interconnected components that have been evolved, the flow of data can be difficult to follow, even with the aid of good analytical tools. The data flow in the system is subject to several interpolations which are necessary for the system to work. These interpolations can lead to some loss in precision, though it is questionable whether a high level of accuracy is necessary for analysis of emergent behaviour in dynamical networks. With sufficient attention to detail, it is possible to trace the logic throughout the process to some extent. Dynamical systems theory allows us to look at a general behavioural space, and how neurons interact in a theoretical sense. Analysis of arbitrary-size networks doesn't provide much useful information in terms of solving the problems here, however.

Simple mistakes in the system tend to affect the end result, without leaving any obvious clues along the way. A possible approach to the problem would be to change the encoding of our neural network. Most of the time, the indication that something was broken, appeared as circling behaviour in the final phenotype. This could be caused by a flawed selection mechanism or mutation error, logical errors in the neural network code, low battery power etc. Our dynamical neural networks had two output neurons, specifying the motor speeds on the left and right motors, respectively. Since it was very likely that the two output neurons ended up with different values, any error would be indicated by the robot running in circles, irregardless of the component causing it. A better solution, perhaps, could be to decode the output neurons as general heading and speed indicators. This was pointed out by Randall D. Beer, along with some mistakes in the mutation code and the fitness functions, which solved some of the major problems.

Generally the system is extremely sensitive to the number ranges provided by the genetic algorithm (bias, weights, time constants) and interpolations of these passed to the neural network. Also, the data provided to the robot via the interpolated output from the neural network and correct interpolation of sensory input is crucial for evolving any useful behaviour. Without the proper ranges, the system will not function at all.

Late in our work we were told that evolving time biases for input and output nodes might not be a good idea. This because evolved biases on the output and input nodes could cause problems with the sensory data. We did not have time to implement this into our system as that would require extensive recoding of the evolutionary algorithm. However, we will consider it in our plans for future work.

6 Conclusion

In this report we have looked at some cases of evolutionary robotics in theory and practice. We have looked at some of the limitations of the direct programming approach, and whether we can benefit from continuous-time recurrent neural networks.

In the prestudy, we looked at existing approaches to adaptive behaviour in robotics. Among the most interesting for our context, was the continuous-time recurrent neural network and reinforcement learning. The intricate dynamics found in the continuous-time recurrent neural networks in combination with evolution, make them very appealing for robotics. In other words we are looking for greater diversity in our phenotype space than what the old feed-forward network could provide.

A lot of applications until now have focused on designing controllers for self-organizing systems in order to fulfill a particular function. It can seem that the more classical approaches to achieving adaptivity are stalled by the complexity of the problems that they are trying to solve. The best adaptive systems we can think of, are found in biological systems. We would like to explore the possibilities of evolution further, and get some insight into this complexity. Even though the designer is limited to the specification of a fitness function in terms of steering the search, evolution gives useful results. By selecting individuals that perform well from an overall perspective, and being burdened only with the specification of a simple fitness function, the designer is spared from much of the complexity of the design process. It is important to mention that the selection of a well-defined fitness function is crucial, and not always an easy task. However, this task is often simpler than completely specifying a dynamical environment or solution. In addition to this, evolution provides the designer with creative solutions, as evolution often comes up with unexpected results. Evolution is a motivating approach because you can observe its impressive proof-of-concept everywhere in nature, such as the emergence of advanced sensory organs. Typical examples would involve eyes and ears, and in particular the human brain. We have also observed that biological systems are robust and resilient, and most importantly, adaptive.

This approach to evolving robot controllers has given several good solutions. However, the final controller is designed for a specific task, and making it perform other tasks would require further evolution. For other tasks, we would most likely have to evolve separate neural networks, as it can be difficult to combine multiple functions into one. However, that leads us back to the action selection problem, which in turn would probably require yet another neural network. As the complexity grows, we run the risk of operating with a series of black boxes, and the assumption that evolution will solve all our problems. We have experienced some of the frustration of being unable to see a connection between undesired behaviour and programming errors, and we have marvelled at the clever exploits discovered by Darwinian evolution. The creative solutions that emerge from evolution makes this approach viable when there is no obvious solution using direct programming. The alternatives to the standard engineering approach might be better or more efficient in some sense, and we can certainly learn something from them.

We have tried to demonstrate that there is a number of ways for evolution to be used in robotics and in particular how they affect the neural networks they are allowed to work on. What we have not mentioned is the way evolution can work on hardware instead of existing solely in software.

Artificial evolution came up with solutions that matched the environment quite well, and in unpredictable ways. By unpredictable, we mean in ways that we normally wouldn't think of. It was challenging to circumvent the many exploitations that evolution made within the simulated environment, which wouldn't work in the real environment. We looked at two basic experiments that used an evolutionary artificial neural network in order to produce adaptive behaviour.

We have also made sure that our evolutionary algorithm and neural network is allowed to not only work in simulation but also on the real robots. We feel that this is important, since simulations tend to make assumptions about a number of things, and to make the task more trivial than it really is. Despite the degree of realism in the simulated environment, there are things that you cannot predict.

We solved many technicalities during early development which was related to the E-puck robot. In addition to creating our own evolutionary algorithm and neural network we had to build the simulation environment from scratch using the Breve simulator as a starting point.

In the introduction, one of the things we said that we wanted to look at was whether dynamical neural networks had any benefits compared to our much simpler feed-forward networks. Based on our experiments we can conclude that the use of dynamical neural networks provides us with a much greater diversity in the phenotype space. The dynamics along with the feedback loops (recurrence) provide a form of memory which can enable e.g. rhythmic behaviour, bursting (which is similar to an epileptic seizure where all the neurons fire simultaneously) and some limited prediction. However, the possibility of bursting is not a desired effect, it just makes the structure more biologically plausible. In the simulated environment, we have obtained better and more efficient solutions, though the evolution time has increased significantly along with the behaviour space.

Concerning our first goal, which was to tackle the perceptual aliasing problem, we are somewhat satisfied. In section 5.2 we showed that our CTRNN will respond differently to the same input given at different times. However, with this added diversity, the behavior became difficult to control and led to unexpected solutions. We were looking for a phenotype that would result in efficient solutions, and not anything like the circular wall-following behavior that we experienced with the old feed-forward network. The new solutions are much better in terms of exploration, but they spin a lot, which ideally should not be necessary for solving our tasks. It is important to keep in mind that the robot can not really "see" in the traditional sense; imagine yourself walking blindfolded through a maze, feeling your way with your hands. The only advantage to spinning, is that it provides a more precise description of the environment. There is a gap between each sensor, and spinning can help uncover details, such as corners that might otherwise go unnoticed.

Our second goal was to handle the real world problem. With the simple feed-

forward networks, the evolved phenotypes from the simulator worked on the robots without any difficulty. We would usually run a few evolutionary generations in the real environment after the initial evolution in order to adapt to the lighting conditions and noise in the environment to achieve a nearly identical behaviour. However, as we concluded in the section concerning CTRNNs, with dynamical neural networks this became a very difficult task. Because of instability (bifurcations) in the dynamical system, small changes in the parameter space led to completely unpredictable behaviour. We were unable to solve the real world problem when using our most recent networks and in this sense, using CTRNNs was a step backwards.

In essence, the dynamics provided by CTRNNs felt very rewarding in our simulated environment, but it also made it very difficult to apply our solutions to real world problems. Using CTRNNs might be a step in the right direction if you have sufficient time and analytical skills.

6.1 Future work

With the evolved CTRNNs, we were able to produce phenotypes that solved the maze faster and more efficiently than our feed-forward networks, but only in the simulated environment. We feel that the dynamics of CTRNNs eliminated, to some extent, the perceptual aliasing problem [28]. However, our understanding and evidence of why it works is not complete. Given more time, it would be interesting to create interactive tools to investigate neuron activity during a run. It would also be interesting to be able to take a few steps back in time and examine how previous states would behave differently with alternative parameters.

It would also be interesting to make more experiments with random sensory noise on the simulator in order to see if this would evolve more robust solutions. With sensory noise, one would think that unstable solutions would be filtered out along the evolutionary generations, though we haven't been able to eliminate this problem.

As we mentioned in section 5.11, it was proposed by Professor Randall D. Beer that it could be beneficial to give the output neurons of the network different encodings. Rather than specifying the motor speeds, one neuron could define a general heading and the other could specify motor speeds. It is very likely that this would prevent some of the spinning behaviour, and perhaps result in more efficient solutions. It wouldn't, however, solve the real world problem.

7 Appendix A: Getting started with the E-puck

This guide explains how to get the E-puck robot up and running with Python, or any other language, and Bluetooth. Some of the information here is specific to Mac OS X, but it should be easy to adapt to other operating systems.

(1) Flash the E-puck:

The E-puck must be flashed with software that allows it to read and parse instructions received over bluetooth. For this purpose, we use a program called BTcom, which is available at the official E-puck website. It is located in "Downloads", "Software", "Library", "e-puck library compiled". It comes with instructions that explain how to flash the E-puck. You only have to do this once.

(1.1) Download the e-puck project via svn from "https://gna.org/svn/?group=e-puck"

(1.2) Browse into the directory 'e-puck/program/BTcom'

(1.3) Flash your e-puck with BTcom_default.hex using Tiny PIC bootloader

(2) Connect and pair with the bluetooth device:

Click the bluetooth icon in the top right corner of your screen and click "Open Bluetooth Preferences". Make sure that "on" is checked. Click "Set Up New Device". Make sure that your E-puck is turned on, select "any device" and continue with the wizard. In the list, you should find "e-puck_XXXX". Select it and continue. You will be asked to enter a passkey, which is the same as your pincode. Voila! Bob's your uncle and you are paired with the E-puck.

(3) Connect to the E-puck from Python:

First, we need to locate the bluetooth device. Open up a new terminal window and write:

```
$ ls /dev | grep e-puck
```

Typical output would be something like:

```
cu.e-puck_XXXX-COM1-1
tty.e-puck_XXXX-COM1-1
```

Now that we've found the device, we can connect to it using pyserial. Note that you have to replace XXXX with the pincode of your E-puck.

```
import serial

com = serial.Serial(
    port = "/dev/tty.e-puck_XXXX-COM1-1",
    bytesize = 8,
    baudrate = 115200,
    timeout = 1
)

com.write("\n")
out = com.readline()
```

```
com.write("t,1\n") # Play sound 1
out = com.readline()

com.close()
```

This code has no error-checking, and is the minimum of what you need in order to communicate with the E-puck. For some reason, the E-puck returns "Command unknown" in response to the first instruction you send to it. To remedy this, we can simply send com.write("\n") prior to our set of instructions. Note that com.readline() will empty the buffer on the robot, and should be used after each instruction you send to it. For a basic, error-handling skeleton, we suggest the following:

```
import serial
import sys

def connect():
    "Connect to E-puck through serial-port."
    try:
        com = serial.Serial(
            port = "/dev/tty.epuck_XXXX-COM1-1",
            bytesize = 8,
            baudrate = 115200,
            timeout = 1
        )
    except serial.serialutil.SerialException:
        print "*** E-puck appears to be offline."
        sys.exit(1)
    print "*** Connected to E-puck"
    return com

def disconnect(com):
    "Disconnect from E-puck."
    com.close()
    print "*** Disconnected from E-puck"

def test(com):
    "Your code goes here."
    try:
        com.write("\n")
        out = com.readline()
        com.write("t,1\n")
        out = com.readline()
    except serial.serialutil.SerialException:
        print "*** Port is closed by operating system"
        disconnect(com)
        sys.exit(1)

def main():
    e_puck = connect()
    test(e_puck)
```

```
    disconnect(e_puck)

if __name__ == "__main__":
    main()
```

By now you you are probably wondering what kind of instructions you can send to the E-puck. Here's the complete list:

```
"A"          Accelerometer
"B,#"        Body led 0=off 1=on 2=inverse
"C"          Selector position
"D,#,#"      Set motor speed left,right
"E"          Get motor speed left,right
"F,#"        Front led 0=off 1=on 2=inverse
"G"          IR receiver
"H"          Help
"I"          Get camera parameter
"J,#,#,#,#" Set camera parameter mode,
             width,height,zoom(1,4 or 8)
"K"          Calibrate proximity sensors
"L,#,#"      Led number,0=off 1=on 2=inverse
"N"          Proximity
"O"          Light sensors
"P,#,#"      Set motor position left,right
"Q"          Get motor position left,right
"R"          Reset e-puck
"S"          Stop e-puck and turn off leds
"T,#"        Play sound 1-5 else stop sound
"U"          Get microphone amplitude
"V"          Version of SerCom
```

7.1 Notes

We have used the Python programming language in the above example, but it is important to note that any language that can communicate through a serial COM-port is able to communicate with the E-puck.

7.2 Appendix B: Code

This section briefly describes the code we used in our experiments.

Project homepage:

`http://code.google.com/p/adaptive-robotics/`

Both the source code for the threaded and non-threaded version is available for download via SVN. Code controlling the E-puck is also in our repository.

SVN directories:

```
svn
    branches
        ctrnn
        simulator_cluster -
        simulator_cluster2 - Not-used
        simulator_cluster_no_thread - Non threaded version
        simulator_solo - E-puck controller code, not simulator
    docs - Master thesis documentation
        images
        prestudy
    tags - Not used
    trunk - Not used
        textures
        weights_old
    wiki - Not used
```

`ctrnn.py`

- Neural network library code.

`ea.py`

- Evolutionary Algorithm library code.

`e_puck_sim.py`

- Simulator file written for breve.

`e_puck_vehicle.py`

- E-puck vehicle for the Breve simulator. Based on the Braitenberg vehicle.

`ga_master.py`

- Evolutionary parameters

`draw_ann.py`

- Tool for displaying the neural networks(only works with feed-forward networks).

`controller.py`

- Startup code for the e-puck robots. EA parameters changeable in the code.

8 Appendix C: Notes on how to be able to evolve CTRNNs

This guide was created after we experienced problems related to evolving CTRNNs. This is a short recap of what we struggled with and what we belive is important to keep in mind when evolving CTRNNs.

(1) Correct values into the input nodes.

(2) No recurrence in input or output nodes. This is to avoid "seeing things that aren't there". The network's input and output nodes quickly begin to behave unpredictably when using recurrence.

(3) When writing your own CTRNN library, testing if the code is working correctly is hard. In particular when testing against complex network structures. We found that it was very useful to check the CTRNN code by testing it against other people's working code. In our case we used Proffesor Randall D. Beer's code as a benchmark.

9 Bibliography

[1] Wooldridge M. J. *An Introduction to Multi-Agent Systems*
John Wiley & Sons, 2002, ISBN: 047149691X, chapter 5.

[2] Sipper, Moshe and Sanchez, Eduardo. *A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems*
IEEE Transations on Evolutionary Computation, Vol. 1, No. 1, 1997

[3] Pinker, Steven and Mehler, Jacques. *Connections and Symbols*
Cambridge MA: MIT Press, 1998, ISBN: 0262660644, pp. 1

[4] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*
Prentice Hall, 2nd edition, ISBN: 0-13-080302-2, pp. 712

[5] Sirosh J., Miikkulainen R. *How lateral interaction develops in a self-organizing feature map.*
Proc. IEEE Int. Conf. on Neural Net- works. San Fransisco, CA, (1993).

[6] P. D. Wasserman. *Neural Computing: Theory and Practice. Van Nostrand Reinhold*
New York, (1989)

[7] Nikos A. Vlassis, George Papakonstantinou and Panayiotis Tsanakas. *Robot Map Building by Kohonen's Self-Organizing Neural Networks*
National Technical University of Athens, (1997)

[8] Nilsson N.J. *Principles of Artificial Intelligence*
Springer-Verlag, New York, (1980)

[9] R. Rojas. *Neural Networks*
Springer-Verlag, Berlin, (1996)

[10] Stefano Nolfi and Dario Floreano. *Evolutionary Robotics*
The MIT press, (2000)

[11] Keith Downing, *The Backpropagation Algorithm: Gradient Decent Training of Neural Networks*
http://www.idi.ntnu.no/emner/it3708/lectures/backprop.pdf, (2007)

[12] Francis Heylighen, *The science of self-organization and adaptivity*
Center "Leo Apostel", Free University of Brussels, Belgium.

[13] Robert Callan. *The Essence of Neural Networks*
Prentice Hall Europe, ISBN: 0-13-908732-X, pp. 1, 84-90

[14] Terrence Deacon. *Evolution of Human Behaviour*
http://ls.berkeley.edu/dept/anth/deacon.html, 2007

[15] Sutton R. S, Barto A. G. *Reinforcement Learning: An Introduction*
The MIT Press, 1998, ISBN: 0262193981, section 1.4.

[16] Randall D. Beer. *Toward the Evolution of Dynamical Neural Networks for Minimally Cognitive Behavior*
Santa Fe Institute, (1996)

[17] Randall D. Beer. *On the dynamics of small continuous-time recurrent neural networks*
Case Western Reserve University, (1995)

[18] Tom M. Mitchell. *Machine Learning*
Carnegie Mellon University, International Edition, (1997)

[19] Richard Dawkins. *The Selfish Gene*
Oxford University press, 30th Anniversary edition, ISBN: 0199291152, pp. 21, 2006

[20] Barry McMullin. *John von Neumann and the Evolutionary growith of Complexity: Looking backward, Looking forward...*
Artificial Life Labratory, Massachusetts Insitute of Technology, Artificial Life 6. (2000)

[21] Jesper Blynel. *Evolving Reinforcement learning-like abilities for robots*
Autonomous Systems Lab, Swiss Federal Institute of Technology. (2003)

[22] Mataric M. J. *The Basics of Robot Control*
http://www-robotics.usc.edu/ maja/robot-control.html

[23] Anders Kofod-Petersen. *A Case-Based Approach to realising ambient intelligence among agents*
Doctoral thesis at NTNU, (2007), pp. 51-53.

[24] Maxim Likhachev, Michael Kaess, Zsolt Kira and Ronald C. Arkin. *Spation Temporal Case-Based Reasoning for Efficient Reactive Robot Navigation*
Mobile Robot Labratory, College of Computing, Georgia Institute of Technology, (2005)

[25] Edgar Bermudez Contreras. *A Biologically Inspired Solution for an Evolved Simulated Agent.*
School of Informatics, University of Sussex, (2008)

[26] Robert Hinterding. *Gaussian Mutation and Self-adaption for Numeric Genetic Algorithms*
Department of Computer and Mathematical Sciences, Victoria University(1995)

[27] Jesper Blynel and Dario Floreano. *Exploring the T-Maze: Evolving Learning-Like Robot Behavours using CTRNNs*
Institute of Systems Engineering, Swiss Federal Institute of Technology(EPFL)

[28] Per-Erik Forssen, Bjorn Johansson, and Gosta Granlund. *Learning under Perceptual Aliasing*
Linkoping University, Sweden

[29] Jeff Hawkins. *On Intelligence*
Holt Paperbacks, ISBN: 0805078533, 2005

[30] Randall D. Beer. *CTRNN Class Documentation*
http://mypage.iu.edu/ rdbeer/Software/EvolutionaryAgents/CTRNNDoc.pdf

[31] Andy Blow. *An Investigation into a CTRNN Node*
http://www.artificiallife.co.uk/mikeblow_fcs.pdf

[32] Eduardo Izquierdo. *A bit more on CTRNN transfer functions*
http://edizquierdo.wordpress.com/2007/06/20/a-bit-more-on-ctrnn-transfer-functions/

[33] Gautam Roy, Hyunyoung Lee, Jennifer L. Welch, Yuan Zhao, Vijitashwa Pandey and Deborah Thurston. *A Distributed Pool Architecture for Genetic Algorithms*

[34] Pavel Kordik, Jan Saidl, Miroslav Snorek. *Evolutionary Search for Interesting Behavior of Neural Network Ensembles*

[35] Springer Berlin / Heidelberg. *Embodied Cognitive Science*
ISBN: 978-3-540-77611-6, 2008

[36] Tim Taylor. *A Genetic Regulatory Network-Inspired Real-Time Controller for a Group of Underwater Robots*

[37] Gregory S. Hornby. *Functional Scalability through Generative Representations: the Evolution of Table Designs*