EVOLVING PATTERN-SEEKING ARTIFICIAL LIFE WITH CRÉATÚR

by

Amy de Buitléir

A thesis submitted in partial fulfilment of the

requirements for the M.Sc. in

Software Engineering

Athlone Institute of Technology

2011

Supervisors: Michael Russell and Mark Daly

Department of Electronics, Computer and

Software Engineering

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of MSc is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:

Student ID: A00168093

Date:

Contents

Abstract

This thesis describes a research project to evolve an Artificial Life (ALife) population with sufficient intelligence to discover patterns in data, and to make survival decisions based on those patterns. As part of this research, Créatúr, a reusable software framework for automating experiments with artificial lifeforms, was built. An ALife species called wains was implemented using diploid reproduction, Hebbian learning and Kohonen Self-organising Maps, in combination with novel techniques such as using pattern-rich data as the environment and framing data analysis as an ALife survival problem. The data set used was the MNIST database of handwritten numerals. The first generation of wains mastered the numeral recognition task well enough to thrive. Evolution further adapted the wains to their environment by making them a little more pessimistic (lowering the rate at which they learn from positive experiences, and raising the the rate at which they learn from negative experiences), and also by making their brains more efficient (reducing the number of patterns remembered, and discarding the least useful patterns less frequently).

Acknowledgements

I would like to thank my co-supervisor, Michael Russell, for letting me reach for the stars while keeping me safely tethered to the Earth, and for knowing just how to bring out the best in me. I would also like to thank my co-supervisor, Dr. Mark Daly, for mirroring my enthusiasm, thereby assuring me that my ideas had merit; and for appreciating my comics. Finally, I would like to thank Frank Scaduto for reviewing this thesis in detail and asking such insightful questions, prompting me to write with more clarity and conviction.

List of Figures

List of Tables

Chapter 1

Introduction

> Our understanding of the universe will be severely limited until we have a more definitive view of how much life and consciousness can be explained as emergent phenomena.
>
> John Holland

This thesis describes a research project called Créatúr.[1] The idea for this project came when three chance encounters rekindled my long-term interest in Artificial Intelligence (AI), the creation of computer models of intelligence;[2] Artificial Life (ALife), the creation of computer models of biological lifeforms;[3] emergence,[4] the phenomenon by which simple rules give rise to complex behaviour; and the question of how a mind arises from a brain.

[1] Créatúr, which is pronounced /crʲeːt̪ˠuːrˠ/ [KRAY-toor], is an Irish word for an animal, or an unfortunate person.

[2] The term "artificial intelligence" was first used by John McCarthy in 1955 [1].

[3] The term "artificial life" was first used by Chris Langton in 1986 [2].

[4] The term "emergent" was first used by the psychologist G. H. Lewes [3].

The first encounter was with PolyWorld, an ALife platform which has given rise to biologically realistic behaviours such as predator-prey population cycles. In many ALife systems, the *animats*[5] (artificial animals) do not have anything resembling a brain; their behaviour is genetically hard-wired. In PolyWorld, however, each animat has a brain in the form of a neural network, the structure of which is genetic, and shaped by natural selection. So while ALife has tended to ignore the mind, and AI has tended to ignore the body, PolyWorld models both. I speculated that this way of bringing elements of AI into ALife, of modelling a brain not as a standalone system for making decisions, but in a context where it is responsible for a animat's survival, could produce more intelligent animats.

The second encounter was with the theory of Neural Darwinism, which proposes that some form of evolution by natural selection takes place in the brain. In biology, natural selection provides a step-by-step explanation of not only how life arose from chemicals in the first place, but how complex organs such as the eye can occur[6], and also how the rich and endlessly varying forms of life on this planet came to be. I found the idea that natural selection could similarly explain how a mind emerges from a brain, deeply appealing.

The third encounter was with the Self-organising Map (SOM), a type of neural network which builds a simplified model of the data it receives, and can discover patterns even when the relationships between the input data and the output data are very complex. It seemed to me that SOMs could be a useful component in modelling a brain.

[5]The term "animat" was coined by Stewart W. Wilson in 1991 [4].

[6]Charles Darwin readily admitted the difficulty of explaining how "organs of extreme perfection and complication" [5, pp. 143-146] such as the eye could evolve, though he did outline a possible solution. Richard Dawkins [6, chap. 5] has a very clear explanation of how eyes evolved not once, but many times, in various parts of the animal kingdom.

The aim of my research was to explore how these ideas could be used to build a system that would discover patterns in the environment, and make predictions based on those patterns.

1.1 Document Structure

The structure of this document is outlined below.

Chapter 2: Literature Review establishes the theoretical basis for the Créatúr research project.

Chapter 3: Objectives and Approach describes the objectives and the guiding principles of the Créatúr project, and ends by placing it in the context of a somewhat similar project, PolyWorld.

Chapter 4: Pilot Project: Numeral Recognition with Back-propagation describes a pilot project to gain familiarity with the some of the technologies and tools considered for use in Créatúr, and to assess their suitability. The pilot project implemented a neural network which used back-propagation to recognise handwritten numerals.

Chapter 5: Créatúr: an ALife habitat describes Créatúr, a reusable software framework for automating experiments with artificial lifeforms.

Chapter 6: Dotes: an artificial lifeform describes the first of two experiments performed using the Créatúr habitat. The animats used in this experiment are called dotes.

Chapter 7: Wains: an artificial lifeform describes the second of two experiments performed using the Créatúr habitat. The animats used in this experiment are called wains.

Chapter 8: Conclusions and observations presents some conclusions about the Créatúr research project, and some observations that may be helpful to others beginning an ALife research project.

Chapter 9: Future Directions proposes future directions for research continuing on from, or inspired by, the Créatúr project.

A **Glossary** (which includes the symbols used in equations in this document), list of **Acronyms**, and **Bibliography** are provided.

Appendix A: Methodology discusses the methodology used in the Créatúr research project.

Appendix B: Requirements defines the requirements for the artificial life software at the core of this research project.

Appendix C: Dote Genome describes the encoding scheme for the dote genome.

Appendix D: Wain Genome describes the encoding scheme for the wain genome.

1.2 Nomenclature

In this thesis, the word **Créatúr** can refer to either *a*) the Créatúr habitat, a software framework for running ALife experiments, or *b*) the research project described in this document, for which the Créatúr framework was built. It should be clear from the context which meaning is intended.

Creatures (in italics and without diacritical marks) refers to the ALife game created by Steve Grand.

The word **neuron** is used to refer to either a biological neuron or an artificial neuron.

Chapter 2

Literature Review

> The ALife-AI claim is, "The dumbest smart thing you can do is stay alive." That is, ALife represents a lower bound for AI.
>
> Richard K. Belew

This chapter establishes the theoretical basis for the Créatúr research project. Next, tools and methods from Software Engineering are discussed which, although they did not influence the *nature* of the Créatúr project, did influence the *implementation*. Finally, the main ideas which influence Créatúr are summarised.

This project incorporates concepts from fields as diverse as philosophy, biology, neuroscience, AI and ALife. Figure 2.1 shows how these concepts interrelate; the reader may find it helpful to refer back to this diagram occasionally.

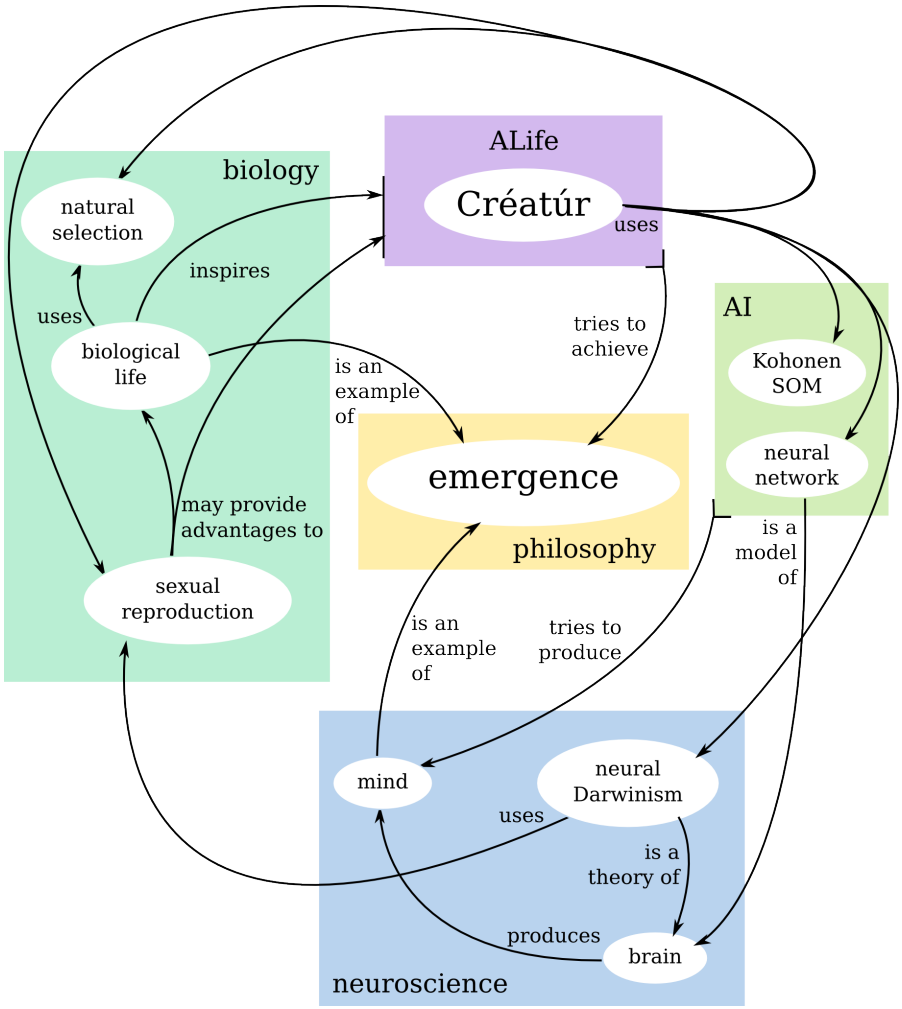


Figure 2.1: Concept map for this thesis. The diagram does not include all of the concepts discussed, only the key concepts and relationships.

2.1 Ideas from philosophy

Cognitive science is an interdisciplinary study concerned with the nature of thought [7]. It emerged as a field of study in the 1950s, and philosophy (together with psychology, linguistics, neuroscience, computer science, and anthropology) has been an ingredient of cognitive science since its inception [8]. Although the involvement of philosophy has lessened somewhat in recent years [9], "philosophy has continued to serve as a source of interesting ideas that have engaged the field" [10]. Two philosophical concepts which the Créatúr project draws from, emergence and the intentional stance, are discussed below.

2.1.1 Emergence

Ant colonies and traffic jams. Flocks of birds and swarms of locusts. Tornadoes and the Internet. These are examples of *emergence*: a phenomenon where the "whole is greater than the sum of its parts" [11, Book VIII, part 6], where individual components behaving according to simple rules give rise to complexity at a higher level. The science writer Steven Johnson called it "the movement from low-level rules to higher-level sophistication" [12]. Perhaps more memorably, the philosopher David Chalmers described it as the phenomenon where "something stupid buys you something smart" [13].

But the very idea of emergence seems to defy logic. In the words of John Holland, who pioneered the field of genetic algorithms, "How can the interactions of agents produce an aggregate entity that is more flexible and adaptive than its component agents?" [14, p. 248] While acknowledging that examples of emergence are abundant, the philosopher Mark Bedau felt there was a contradiction in the

very concept. To highlight the problem, he defined two chief characteristics of emergent phenomena [15].

1. Emergent phenomena are somehow constituted by, and generated from, underlying processes.

2. Emergent phenomena are somehow autonomous from underlying processes.

In an attempt to explain the contradiction, Bedau distinguished between two types of emergence, *weak* and *strong*. Weakly emergent phenomena can be derived from the underlying processes, but only by simulation, whereas strongly emergent phenomena cannot be derived at all. Weak emergence sounds similar to a chaotic system, i.e., a system whose behaviour can be predicted in principle, but which is so sensitive to initial conditions that it is, for all practical purposes, unpredictable. However, while chaos can underlie an emergent system, Bedau felt that all complex systems were weakly emergent, whether chaotic or not. As for strong emergence, Bedau granted the logical possibility and did not rule out its existence, but he found it "uncomfortably like magic".[1] [2]

[1]Bedau also identified a form of emergence that is weaker than weak emergence, which he termed nominal emergence This phenomenon occurs when a property can exist at a macro level but not a micro level. For example, water is fluid and transparent, but the constituent molecules cannot be said to have these properties [16].

[2]Slightly different definitions for weak and strong emergence were given by Chalmers:

> We can say that a high-level phenomenon is strongly emergent with respect to a low-level domain when the high-level phenomenon arises from the low-level domain, but truths concerning that phenomenon are not deducible even in principle from truths in the low-level domain.
>
> We can say that a high-level phenomenon is weakly emergent with respect to a low-level domain when the high-level phenomenon arises from the low-level domain, but truths concerning that phenomenon are unexpected given the principles governing the low-level domain [17].

In this formulation, weak emergence is reminiscent of the philosophy of *reductionism*, which claims that a system can be understood by dividing it into components and studying their prop-

The Créatúr project involves ALife, and to a limited extent, AI. Why discuss emergence at all? One reason is that biology is full of examples of emergence, and any ALife system worthy of the name should demonstrate some emergent behaviour. In trying to achieve this quality, we naturally look to the sources of emergence in nature, and try to incorporate those mechanisms. But perhaps the most compelling reason that a discussion of ALife often includes some mention of emergence is given by Bedau.

> The aggregate global behaviour of the complex systems studied in ALife offers a new way to view emergence… Thus ALife will play an active role in future philosophical debates about emergence and related notions such as explanation, reduction, complexity and hierarchy [18].

As shown in Figure 2.1, the concept of emergence (at least in its weaker form), is a recurring theme in this thesis.[3]

2.1.2 The Intentional Stance

The fundamental way to predict the behaviour of an object (whether alive or not) is to use what the philosopher and cognitive scientist Daniel Dennett calls the *physical stance* [20, p. 15f] [21]: use whatever is known about the laws of physics and constitution of the object. If we drop an object, whether it is a rock, an alarm clock, or a goldfish, we can predict its trajectory based on physics.

However, the physical stance is not as useful for predicting some behaviours of objects. A faster (but riskier) way to make predictions is to use the *design stance*

erties. In contrast, strong emergence is reminiscent of the philosophy of *holism*, which focuses on the study of complex systems as a whole entity.

[3] An excellent guide to the history of emergence theory can be found found in Clayton and Davies [19].

[20, p. 16f] [21]: if an object is designed, it will operate according to that design. For example, if the buttons and dials of an alarm clock are manipulated in the correct way, the alarm will ring some time later. We do not need to take the clock apart and examine its physical properties in detail in order to make this prediction. However, in using the design stance, we assume that the entity *is* designed, and that it is not malfunctioning.

For some types of objects, we can use an even faster method of making predictions. The *intentional stance* [20, p. 17f] [21] treats the object as an agent with beliefs and desires, and assumes that the object will act in accordance with those beliefs and desires. Dennett gives the example of a chess program: the easiest way to predict what it will do is not to analyse the source code, but to think of the program as a "rational" agent that "wants" to win, and "knows" the rules of chess. We can then assume that it will behave rationally, and take the action that it "believes" will lead to victory.

The intentional stance is such a powerful generator of predictions that we use it regularly, and would find it difficult to give it up. We say that a car is "trying" to start on a cold morning, and that a program "knows" the value of x when it gets to a particular line of code. Dawkins used the intentional stance when he wrote *The Selfish Gene* [22]. This stance offers both risk and benefit:

> [The intentional stance] must be used with caution; we must walk a tightrope between vacuous metaphor on the one hand and literal falsehood on the other… Properly understood, it can provide a sound and fruitful perspective… directing our attention to the crucial experiments that need to be conducted. [23, p. 36]

Because it is so concise (and avoids the necessity of sprinkling "scare quotes" on every page), the intentional stance is used in this thesis to describe the behaviour of the animats.

2.2 Ideas from biology

As the focus of this research is to build a flexible system that can recognise patterns and make predictions, the best source for guidance on how to approach this task is Nature itself. Biological life is rich with examples of emergence, and with organisms that regularly solve problems in complex environments and which have the ability to adapt – both in an evolutionary sense and during the lifetime of the individual – to a changing environment. Two ideas from biology which have guided the Créatúr research project, evolution by natural selection, and sexual reproduction, are discussed below.

2.2.1 Evolution by natural selection

Charles Darwin wrote *On the Origin of Species* [24] with two goals in mind: to prove that species evolve, and to explain the mechanism by which it happens – descent with modification (which we now call "natural selection"). Both ideas met with initial resistance, but evolution was accepted first; descent with modification did not begin to gain widespread acceptance until the 1930s. The first key step toward acceptance was the rediscovery, at the turn of the 20th century, of Gregor Mendel's formulation of the laws of inheritance based on his experiments

with pea plants [25, p. 270].[4] Another was when the biologist and statistician R.A. Fisher showed that groups of discrete genes could collectively produce the kind of continuous variation seen in nature, and that natural selection could change the frequency of genes in a population over time [27].

The recipe for evolution is a simple one. It requires the following conditions [28]:

1. *variation*: a continuing abundance of different elements,
2. *heredity or replication*: the capacity to create copies of elements, and
3. *differential fitness*: the number of copies created depends on the interaction between the features of an element with features of the environment.

From these simple conditions, all the complexity and variation of biological life arises; clearly this is an emergent phenomenon. And yet, "the only thing that changes in evolution is the genes" [29]. Dawkins describes genes as "Duplicate Me" programs, and explains this outlook with a delightful use of the intentional stance:

> Elephant genes say: 'Elephant cells, work together to make a new elephant, which must be programmed in its turn to grow and make more elephants, all programmed to duplicate me.' [6, p. 271]

Mutation, and the shuffling of genetic material that occurs in sexual reproduction, makes it possible for a species to adapt to changes in the environment,

[4] Mendel was a contemporary of Darwin, and had read *On the Origin of Species* and agreed with much of it. However, Mendel did not believe that the traits of the parents are blended to produce offspring. Darwin was apparently unaware of Mendel's work [26, p. 23].

provided those changes aren't too rapid. If we view an individual's genome as a solution to the problem of scarcity of resources, and a species as a class of solutions that are partially optimised to a particular ecological niche, then evolution is seen as a process for finding solutions. Although the process of evolution is normally associated with biological organisms, it can occur with any substrate as long as those three conditions are met. It has been argued that other forms of evolution occur in nature. One example is Neural Darwinism, which will be discussed below. Another example, somewhat controversial, is the idea that memes, cultural units of transmission or imitation, undergo a type of evolution [22, p. 192].

Evolution seems to be a promising approach to developing a complex and adaptive system from simple parts. The Créatúr project made use of evolution by making certain aspects of the design of the animat species, particularly those aspects concerned with thinking and decision making, genetically determined. This allowed evolution to improve on the initial design of the animats.

2.2.2 Sexual reproduction

From an evolutionary perspective, sex has a number of disadvantages. It requires a mechanism of "arbitration" when the organism inherits different alleles, or forms of a particular gene, from each parent. Only half of the population can produce eggs. Organisms must invest time and energy in finding mates, they must produce special sex cells, and they can only pass along half of their genes to their offspring. Why did sexual reproduction develop, and why does it persist? Bell called this the "queen of problems in evolutionary biology" [30, p. 19].

Many theories have been proposed to explain how sexual reproduction might

be adaptive; a few examples are mentioned below.[5]

- Repair of chromosome damage may be easier when there is a second chromosome of the same kind [32–34].

- In asexual reproduction, the genome is copied as a block. Therefore, once all the individuals in a population carry at least one deleterious mutation, so will all future generations (except in the unlikely event that a future mutation reverses an existing mutation). Over time, the individuals with the fewest deleterious mutations are lost and the population accumulates more deleterious mutations ("Muller's ratchet"). In populations that reproduce sexually, however, it is possible for offspring to have fewer mutations than their parents, and offspring that end up with more mutations than their parents are likely to be eliminated. In this way, sexual reproduction may help to remove deleterious mutations from the gene pool [35].

- Sex may provide protection against parasites as the host evolves new defences. This is called the Red Queen hypothesis because "it takes all the running you can do to keep in the same place" on the part of host and parasite as they race to keep up with each other's adaptations [36, 37].

These theories suggest that organisms gain primarily from being *diploid* (having two sets of chromosomes in each cell), rather than from having multiple sexes. In fact, not everyone agrees that sexual reproduction is adaptive. It may be that

> Our sexuality does not derive directly from only benefit, in other words, because most, if not all, sexually reproducing animals have

[5] Meirmans and Strand strand provide a good summary [31].

> no exit, no way to opt out of the ancient cycle of meeting, mating, and cell growth to make our bodies [38, p. 112].

In an effort to gain the potential benefits of sexual reproduction, while avoiding the disadvantages, the animats developed for the Créatúr project are diploid, and reproduce sexually (i.e. each parent contributes a set of chromosomes), but they have only one sex. Using diploid reproduction was a key element of the approach used in this project, as will be discussed in Section 3.2.8.

2.3 Ideas from neuroscience

When trying to build a system that can recognise patterns and make predictions, the obvious course of action is to model it after existing systems that have this ability. A good model is found in brains of creatures that can evaluate possible actions and make choices based on that evaluation. These creatures don't merely learn through trial-and-error; they have the ability to filter out some bad choices, and make better-than-chance guesses about what to do next. Dennett calls them *Popperian creatures*, after the philosopher Sir Karl Popper who said that this ability lets "our theories die in our stead" [23, p. 116] [39, p. 244]. This ability is found in mammals, birds, fish, cephalopods, some reptiles, and possibly other creatures as well. Popperian creatures are the subject of neuroscience;

2.3.1 Neural Darwinism

Evolution by natural selection is a simple process that can create highly sophisticated designs. The human brain is undoubtedly a result of evolution, but our genes

are an incomplete blueprint for this vital organ. They specify the components of the brain, but do not provide a detailed wiring diagram. Instead, the brain has to quickly wire itself into a mind during infancy. This process of wiring and rewiring, or neural plasticity, takes place throughout our lives [40].

But how does the brain organise itself? It may be that another type of natural selection occurs within the brain, forming new connections between neurons, and pruning connections that are found to be undesirable. Gerald Edelman [41] proposed the theory of *Neural Darwinism*: Neurons in the cerebral cortex form into clusters called groups. Within each group, there is a pattern of firing where one neuron in the centre is amplified, and the surrounding neurons are suppressed, so there is a kind of competition or differential fitness, satisfying the third condition for evolution (as discussed in Section 2.2.1). There is also variation in these groups, so the first condition is satisfied.

Critics of the theory of Neural Darwinism, including Francis Crick [42], pointed to the absence of replication (the second condition for evolution) in the process. To appreciate why replication is important, it is necessary to understand what happens if an evolutionary process has discovered a good solution, and is now experimenting with variants of that solution. If the original solution is modified to produce a variant, it may turn out to be non-adaptive and get culled. Replication, on the other hand, leaves the original solution intact, so we do not risk losing it if the variations turn out to be less viable. However, Chrisantha Fernando [43] has since proposed mechanisms by which a type of replication might occur in the brain. If evolution does occur in the brain, it would help to explain how a mind emerges from a disorganised tangle of neurons.

A type of evolution was incorporated into the brains of the animats developed

for the Créatúr project. This was a key element of the approach used in this project, as will be discussed in Section 3.2.4.

2.3.2 Pre-wiring the brain

In an effort to learn how thoughts are represented spatially in the brain, a team of researchers led by Mitchell and Just [44, 45] conducted a series of experiments to explore how cognitive states are represented spatially in the brain. They presented a stimulus to volunteers for approximately 3 seconds, and asked them to think about it. The stimulus consisted either of a word such as "bottle", or a simple image such as a drawing of a bottle. All the words and images were of concrete nouns. The subjects were observed using functional Magnetic Resonance Imaging (fMRI). The researchers were able to train a variety of classifiers (machine learning programs) to examine an fMRI scan and identify the stimulus that caused that scan. When they trained the classifier on fMRI scans from one group of people, and used it to examine scans from a different group, it could identify the stimulus with statistically significant accuracy. They then trained a classifier on fMRI scans from people viewing *words*, and used it to decode scans from people viewing *images*, also with accurate results. This suggests that the classifiers were capturing neural activity about the *meaning* of the stimulus, not just the appearance of the object or written word. They also trained a classifier on fMRI scans from bilingual people viewing words in English, and use it to decode scans from the same group of people viewing words in Portuguese, again with accurate results, giving further support to the idea that the spatial representation in the brain was related to the meaning of the stimulus rather than the word used.

The next step was to build a model that could predict neural activity for new words, words that the classifier hadn't trained on. The researchers trained a classifier to recognise a set of 25 verbs, chosen to cover a range of sensorimotor activity. They then examined Google's trillion-word text corpus to discover which verbs co-occur with a pre-selected set of concrete nouns. For example, the verbs "eat" and "taste" often co-occur with the noun "celery", while the verbs "drive" and "wear" do not. They then trained the classifier to predict neural activity in response to a *noun*, as the linear sum of activity in response to *verbs*, weighted according the the probability of that verb co-occurring with the noun. For example:

$$predicted\ activity_{celery} = 0.84 * activity_{eat} + 0.35 * activity_{taste} + \ldots$$

Once the classifier was trained on fMRI scans from one set of nouns, it was given two scans for nouns it had never been trained on. It was able to predict, with statistically significant accuracy, which noun was the stimulus for which scan.

All of this research seems to suggest that *brains have similar wiring*, at least in how they represent concrete nouns and images. But how can this similarity be explained? Perhaps the fact that people have many learning experiences in common, and many of the early learning experiences occur at the same stage of development, leads to similar representations in the brain. Or perhaps we inherit some "pre-wiring". If pre-wiring is beneficial to biological life, it might also be beneficial to ALife. In order to evaluate the benefit to ALife, a type of genetically controlled neural pre-wiring was used in dotes, one of the animat species developed for the Créatúr project. (It was not used in wains, the other species, due to schedule constraints, however it could easily be incorporated.)

2.4 Ideas from AI

AI has been defined as "the scientific understanding of the mechanisms underlying thought and intelligent behaviour and their embodiment in machines" [46]. The most ambitious AI projects attempt to create a program or machine that exhibits the sort of intelligent behaviour found in Dennett's Popperian creatures. Though the interest in "thinking machines" can be traced back to ancient Greece, the mathematician Alan Turing is generally viewed as the founder of modern AI. In a thought experiment, Turing described a simple machine supplied with an infinite stream of tape, capable of reading and writing the symbols '0' and '1' and moving the tape backward and forward. Supplied with a simple "table of behaviour" telling it which action to take, such a machine could perform any mathematical calculation [47, p. 96-107]. Just as Natural Selection shows "the continuity between lifeless matter on the one hand and living things and all their activities and products on the other", Turing "foresaw that there was a traversable path from Absolute Ignorance to Artificial Intelligence, a long series of lifting steps in that Design Space" [48].

A good definition of intelligence has not been established,[6] however, so AI researchers were aiming at a moving target. Programs exist which play chess, compose music, and develop mathematical proofs, yet none of them seem to capture the essence of intelligence. In the words of the cognitive scientist Douglas

[6]Turing defined a test of machine intelligence [49], in which a human judge converses through a text-only interface with a human and a machine. All participants are in separate rooms. If the judge cannot reliably tell the human from the machine, then the machine has passed the test. This "Turing Test", as it is called, is perhaps the best criterion for machine intelligence available. However, it is not viewed as infallible. Humans are not reliable judges; we sometimes attribute intelligence where it does not exist, and overlook intelligence where it does exist. For example, "chatbots", which are extremely simple programs designed to add human-like comments to on-line conversations, regularly fool people.

Hofstadter,

> The ineluctable core of intelligence is always in that next thing which hasn't yet been programmed. This "Theorem" was first proposed to me by Larry Tesler, so I call it Tesler's Theorem: "AI is whatever hasn't been done yet" [50, p. 601].

Nonetheless, AI has produced interesting and practical techniques for solving problems and modelling certain aspects of intelligence. Two ideas from AI which have guided the Créatúr research project, Artificial Neural Networks (ANNs) and Self-organising Maps (SOMs), are discussed below. Using techniques from AI was a key element of the approach used in this project, as will be discussed in Section 3.2.1.

2.4.1 Artificial Neural Networks

An ANN is a computational system, inspired by neuroscience and loosely modelled on biological neurons, that can be used to identify patterns in data, to model relationships between inputs and outputs, and to make predictions based on its input.

Artificial neurons

Conceptually, the basic building block of an ANN is the neuron, shown in Figure 2.2. It is characterised by [51, pp. 13-16, 25-28]

- a set of inputs x_i, usually more than one
- a set of weights w_i associated with each input

- The weighted sum of the inputs $a = \Sigma x_i w_i$
- an activation function $f(a)$ which acts on the weighted sum of the inputs, and determines the output, and
- a single output $y = f(a)$

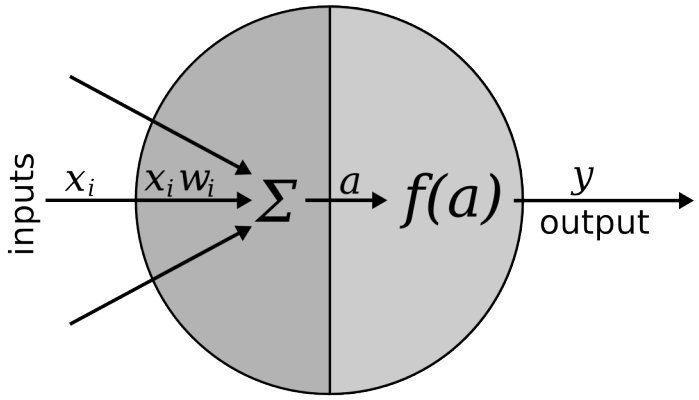


Figure 2.2: An artificial neuron.

Often the input and output signals for artificial neurons are restricted to the binary values 0 and 1. However, other encoding schemes are also used. For example, the signals might be analogue, represented by a continuous stream characterised by a pulse frequency [51, p. 18f].

Hebbian learning

In 1949, the psychologist Donald Hebb postulated a method by which biological brains might form associations.

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic

> change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased [52, p. 62].

This is often summarised as "cells that fire together, wire together". Today the term *Hebbian learning* can refer to any one of a number of mathematical models that implement this type of behaviour. The simplest of these is known as Hebb's Rule, and is given by Equation 2.1,

$$w_i' = w_i + \eta x_i y \tag{2.1}$$

where w_i is the current value of the i^{th} weight and w_i' is the updated value, η is the learning rate, x_i is the i^{th} input, and y is the output. This equation is unstable; if one of the input signals x_i is dominant, then the weight will increase without bound.

The computer scientist Erkki Oja [53] modified Hebb's Rule to solve the stability problem. Oja's Rule is given by Equation 2.2; the symbols have the same meaning as for Hebb's Rule.

$$w_i' = w_i + \eta y(x_i - w_i y) \tag{2.2}$$

Both Hebb's Rule and Oja's Rule are *local learning rules*; they depend only on the neuron's current state and its inputs. Many other learning rules, both local and non-local, have been used in ANNs. One technique for training ANNs, back-propagation, uses a non-local learning rule. Back-propagation will be discussed below and in Chapter 4. First, however, it is necessary to consider how artificial neurons are organised into an ANN.

Feed-forward networks

The most common type of ANN architecture is the *feed-forward network*. In a feed-forward network, the neurons are grouped into layers, as shown in Figure 2.3. Each neuron feeds its output forward to every neuron in the following layer. There is no feedback from a later layer to an earlier one, and no connections within a layer, so there are no loops. The elements of the *input pattern* to be analysed are presented to a *sensor layer*, which has one neuron for every component of the input. The sensor layer performs no processing; it merely distributes its input to the next layer. After the sensor layer there are one or more *hidden layers*; the number of neurons in these layers is arbitrary. The purpose of a hidden layer is to reduce the dimensionality of the data from the previous layer, so that the next layer has a simpler model to learn.[7]. The last layer is the *output layer*; the outputs from these neurons form the elements of the *output pattern.* Hence the number of neurons in the output layer must match the desired length of the output pattern.

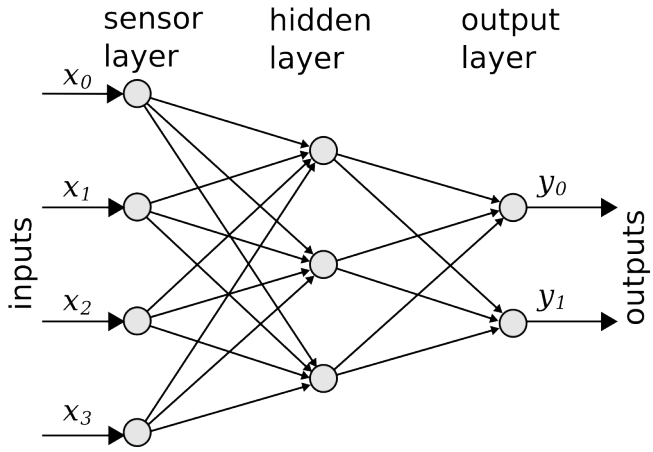


Figure 2.3: A simple neural network.

[7]The hidden layer models its input as a linear combination of the activation function The activation functions used in artificial neurons are usually universal basis functions such as tanh, so given enough neurons, the hidden layer can model the inputs to any degree of accuracy required.

Training the ANN

The *error* is a function of the vector difference between the output pattern and the *target pattern* (desired output). The network can be trained by adjusting the network weights with the goal of reducing the error; there are many techniques for doing this.

Methods for training an ANN can be categorised according to whether they use *supervised training* or *unsupervised training*. In supervised training, the desired response (target pattern) for each input vector in the training set must be known in advance or be calculable. During the training phase, the ANN has access to the target values (usually after it has responded to the input vector), which allows it to calculate the error and make weight corrections intended to reduce the error in future. After training is completed, a separate data set is used to test the ANN. During testing, the ANN is *not* provided with the target values.

One example of a supervised training technique is *back-propagation* [54], which can be used with feed-forward networks. In back-propagation, after an input pattern is propagated forward through the network to produce an output pattern, During the back-propagation phase, each neuron's contribution to the error is calculated, and the network configuration can be modified with the goal of reducing future errors. This technique is described in more detail in Chapter 4.

In unsupervised training, the desired response is not known, and there may or may not be a way to estimate errors. An ANN that uses unsupervised training is most useful for analysing data and finding other ways to represent it (e.g., by compressing it, reducing the dimensionality, or categorising it). One example of a network that uses unsupervised training is the SOM. A SOM is in many ways similar

to an ANN, and is sometimes classed as one, but because it has some unique characteristics, it is discussed separately, in Section 2.4.2. Because Hebbian learning has biological plausibility, and can be used with or without supervised training, it was incorporated into the brains of the animats developed for the Créatúr project.

2.4.2 Kohonen's Self-organising Map

In 1982, Teuvo Kohonen proposed a computational method for analysing high-dimensional data [55, 56]. A Kohonen SOM maps the input vectors onto a regular grid (usually two-dimensional) where each node in the grid has a weight vector representing a model of the input data. (The components in a SOM are usually referred to as nodes rather than neurons.) Kohonen's technique ensures that any topological relationships within the input data are also represented in the grid. The training of the grid is *unsupervised*, no target pattern is required. Some features that distinguish a SOM from more typical ANNs include: the activation function is the identity function, only one output node is active at a time, there is competition between nodes to represent the input pattern, and the network itself becomes a map which preserves the topology of the input data.

The algorithm for implementing a SOM is straightforward. For each input vector (input pattern) presented, the following steps are performed.

1. A *winning node* is selected. The node chosen is the one that is most similar (in some sense) to the input vector. Typically it is the node with the smallest Euclidean distance[8] between its weight vector and the input vector that is

[8]In Cartesian coordinates, if $p = (p_1, p_2, ..., p_n)$ and $q = (q_1, q2, ..., q_n)$, the Euclidean distance is given by $\sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$

selected as the winning node.

2. The weight vector of the winning node is adjusted to make it slightly more similar to the input vector.

3. The weight vectors of all nodes within a given radius of the winning node are also adjusted, by an amount which is smaller the further the node is from the winning node.

As more and more inputs are received, nodes that are physically close end up responding to similar patterns in the input data; this preserves its topology. The update performed in steps 2 and 3 takes the general form shown in Equation 2.3,

$$\mathbf{w_i'} = \mathbf{w_i} + \eta f(i,k)(\mathbf{x} - \mathbf{w_i}) \tag{2.3}$$

where $\mathbf{x}$ is the input vector, k is the index of the winning node, i is the index of the node being updated, $\mathbf{w_i}$ is the current value of the weight vector, $\mathbf{w_i'}$ is the updated value, η is the learning rate, and $f(i,k)$ is the neighbourhood function, which has value 1 when $i = k$ and falls off with the distance between the nodes i and k. The neighbourhood function is often an exponential function.

Because SOMs do not require supervised training, and are well-suited to finding patterns in data, a simplified SOM was incorporated into the brains of wains, one of the animat species developed for the Créatúr project.

2.5 Ideas from ALife

ALife is a field which attempts to create life-like behaviour using software, hardware, biochemistry or other media. The computer scientist Christopher Langton, who is considered one of the founders of the field, described it as broadening the scope of biology, which in practise is restricted to the study of carbon-based life (owing to the lack of any other kind being available for study). By studying "life-as-it-could-be" in addition to "life-as-we-know-it", Langton suggested, we could isolate the essential properties of life. [57]

Although Créatúr uses ideas from many areas of study, it is implemented as an ALife project. Inspiration for the project was drawn from two prior ALife projects: PolyWorld and *Creatures*. These projects, along with the drive for complexity in ALife, and options for reproduction, are discussed below. First, however, we will look at a program that established a "gold standard" for ALife. That program is Tierra.

2.5.1 Tierra

As an undergraduate, Tom Ray [58] studied ants in the Costa Rican rain forest. He became fascinated with the complex relationships between army ants and the birds, butterflies and other insects that follow the colony on raids. He continued on to complete a master's and doctorate at Harvard, where he became frustrated with the inability of ecology to explain the types of interdependent relationships he had witnessed in the army ants [59]. To explore these relationships, Ray decided to create synthetic life in a computer. His goal in developing Tierra was to "to synthesise rather than simulate life"; The inhabitants of Tierra are self-replicating

programs which evolve by natural selection. In most ALife systems of the time, each individual had a genome with a set of pre-defined genes and allelic forms, and survived (or not) according to a fitness function designed by the developer. Ray saw this as a limitation; he wanted a system that permitted "open-ended evolution". The inhabitants of Tierra run in a virtual machine which has been ported to many operating systems, including Linux (x86 and 64 bit Alpha), many Unix platforms, and Windows 2000.

Tierra developed a diverse ecology. The instruction set included jump commands, which directed the processor to another memory location, thereby supporting loops. One mutation caused programs to incorrectly report their size to the operating system; allowing them to execute part of another animat's code. These "parasites" could only replicate in the presence of the longer "host". Subsequently, another mutation granted some programs a degree of "immunity" to these parasites; their code permits the parasites to replicate a few times, but eventually eliminates them. Yet another mutation allowed the parasites to bypass this immunity. Eventually this "arms race" led to the development of hyper-parasites and hyper-hyper-parasites. Perhaps more surprisingly, "social hyper-parasites" developed, which share code to their mutual benefit.

The macro-evolutionary patterns found in longer runs included periods of slow evolutionary change, punctuated by periods of rapid change. Ray describes this as a parallel to the theory of *punctuated equilibrium*, as proposed by paleontologists Niles Eldredge and Stephen Jay Gould [60]. Another naturalistic feature in Tierra is the "Lotka-Volterra" cycle, a pattern found in predator and prey populations [61]. With complex behaviours such as these, Tierra, showed that ALife systems could realistically model biological life.

It seems unlikely that Tierra would have achieved this much diversity if a fitness function had been used. For this reason, no fitness function other than survival itself was used in the Créatúr project. This was a key element of the approach used in this project, as will be discussed in Section 3.2.5.

2.5.2 PolyWorld

PolyWorld [62–66] is a cross-platform (Linux, Mac OS X) program written by Larry Yaeger to explore issues in ALife. Simulated organisms reproduce, and survive by finding and eating food in the environment, or by fighting, killing and eating each other. Polyworldians are *haploid*; they only have one set of genetic material. An organism's behaviour is controlled by its "brain", which is an ANN using Hebbian learning. The ANN is neither empty or random at birth, it inherits some initial wiring from its parents. Like Tierra, PolyWorld has no fitness function. Some interesting behaviours have been observed after prolonged evolution, including predator-prey population cycles that mimic those in nature, and mimicry.

Researchers who worked on PolyWorld quickly discovered that "evolution loves to cheat" [64]. For example, one group of animats (dubbed the "indolent cannibals") found a bizarre solution to the problem of obtaining food. They would remain in tight clusters, rarely moving from their birthplace. They would mate, eat the offspring [64], fight with each other, and eat the casualties [62]. This behaviour was dramatically reduced by imposing an energy cost for having offspring, and by eliminating the food source that remained after an animat died. For this reason, everything that the animats in the Créatúr project do, even just being alive, has an associated energy cost. This was a key element of the approach used in this

project, as will be discussed in Section 3.2.6.

In PolyWorld, the colour of an animat is an RGB triplet. The red component indicates how aggressive the animat currently is, and the blue indicates how badly it wants to mate. The green component is genetically determined; this was implemented to support kin selection. Some tribalism has been observed in PolyWorld, but "usually you have to trick it" [64]. The animats used in Créatúr also have a genetically controlled appearance, to allow for the possibility of kin selection in future. This was a key element of the approach used in this project, as will be discussed in Section 3.2.9.

One criticism [67] that has been made of PolyWorld is that there is little evidence that the animats are becoming smarter as a result of evolution. The Hebbian learning that takes place during an animat's lifetime appears to be overwhelmingly responsible for the results. One cause of this may be that the PolyWorld genomes had limited control over the brain structure. To address this concern, evolution was allowed to control multiple aspects of the brains of animats used in Créatúr in the hope that improvements to the brain would occur over generations. In addition, improvements in decision-making was measured both within one lifetime, and over generations.

2.5.3 *Creatures*

Creatures [68] is an ALife computer game created by Steve Grand. It was one of the first commercial games to use machine learning. In the game, a user hatches animated creatures called Norns, and teaches them how to talk, feed themselves, and protect themselves against predators. Users are able to create new "species"

using selective breeding and genetic "tinkering". Norns also have a simplified biochemistry, haploid genetics, and an ANN "brain". Grand chose to "fake" instinct by allowing the organism to learn from its environment *in utero*. The game was popular due to the diversity of the resulting animats, and was seen as providing insight into how real world organisms may function and evolve.

Inspired by Grand's discussion of the importance of finding some way to "bootstrap" knowledge into the young norns the author provided something similar, but more biologically realistic for the animats in Créatúr: the young learn about their environment while still under parental care. This was a key element of the approach used in the Créatúr project, as will be discussed in Section 3.2.7.

2.5.4 Complexity

The most ambitious ALife projects attempt to create the type of complexity that emerges in biology.

> Artificial life research can in many instances be characterised as a search for the surprising… Biological life is full of surprises and therefore ALife should be as well [69].

One criticism frequently levied at ALife projects is that the environment is not rich enough to evolve complexity.

> Ecologists have long recognised that the complexity of an organism's behaviour is related to the environment it must "solve". [70]

The importance of of a complex environment inspired the author to use a rich data source as the environment (as will be discussed in Section 3.2.2), and to make

data analysis be the survival problem (discussed in Section 3.2.3). These were key elements of the approach used in the Créatúr project. However, a more complex environment alone may not be sufficient; the animats themselves, and their genomes, may need to be more complex.

> Some people believe that natural selection given an infinite space of genetic possibilities will inevitably produce more and more complex adaptations. But soft artificial life models like Tierra, Avida, and Echo show conclusively that those mechanisms are in general insufficient to produce a trend of increasing complexity. The proof is simple: The models embody those mechanisms but they don't exhibit the requisite behaviour. Mechanisms like natural selection in an infinite space of genetic possibilities might be necessary for explaining the trend, but they are not sufficient [71].

Partly to address this concern, the animat genomes used in the Créatúr are complex, thereby allowing evolution to have a greater influence over the animats' design.

2.5.5 Reproduction

Whether or not sexual reproduction confers an adaptive advantage to biological lifeforms, it may be beneficial for ALife. Calabretta et al [72] compared haploid and diploid genotypes in a genetic algorithm that evolved weights for ANNs controlling a robot. The solutions which caused the robot to explore more of the environment were allowed to reproduce. They found that the diploid populations tended to have lower average fitness, but higher peak fitness than the haploid populations.

This gave the diploid populations an advantage in a varying environment. Similar results had been found earlier by Smith and Goldberg when working with genetic algorithms for search and optimisation [73].

> [In] the changing environment diploids exhibit another feature that characterises them with respect to haploids, i.e., their capacity to keep a genetic ”memory” of the past that can be useful when the population must re-adapt to an environment to which it has already adapted in the past. Haploids have all their genes expressed and therefore their entire genetic endowment becomes adapted to the current environment. When the environment changes, the negative effects on fitness are felt more strongly by haploids than by diploids whose genetic ”memory”, recorded in their non- expressed genes, allows them to be less negatively affected by environmental changes [72].

The majority of animats species that have been created are haploid. This simplifies the implementation, but perhaps more importantly, it also allows all animats in a population to be potential mating partners. The heavy use of computational resources by animats forces researchers to work with small populations, and having two sexes would halve the number of mating opportunities [64].

2.6 Software Engineering

The Créatúr project required the development of new software. The previous sections reviewed the literature that influenced *what* would be built; this section reviews the literature that influenced *how* it would be built and tested.

2.6.1 Paradigm: Functional Programming

Functional programming is an approach to software development that focuses on the evaluation of functions in the mathematical sense. A function is *pure*; it is free of side-effects and will yield the same result each time it are invoked with the same value. As a result, an expression in functional code is *referentially transparent*; it can be replaced with its value without changing the result. This is consistent with the way expressions behave in mathematics. In fact, functional programming is based on *lambda calculus*, which is a formal system developed by Alonzo Church [74] for defining functions, function applications, and recursion.

Most programming languages widely used in commercial software development (such as C/C++, Java and Visual Basic) are *imperative*; the programmer specifies an algorithm for solving the problem. In contrast, functional programming is *declarative*; the programmer writes a series of definitions leading to the desired result, but does not specify how the computation is performed. The compiler is free to choose any implementation that returns the correct result. Of course, a program completely without side effects would not be very useful (it could not perform any I/O, for example). Functional programming languages provide mechanisms to isolate side-effects, state data, and mutable data from the functional parts of the program.

The primary motivation for the increasing interest in functional programming languages is that programs written in them are easier to parallelise. The lack of side effects makes it simpler to reason about concurrency, and allows the compiler to generate code that is optimised for concurrent processing [75]. The support for concurrency that functional languages provide is of particular interest in ANNs, as

their usefulness is often limited by the number of neurons that they can simulate.

Another advantage is that it is generally easier to understand and predict the behaviour of a functional program. Functional programming can improve modularity; modular programs are generally easier to understand [76]. Referential transparency makes it possible to conduct *equational reasoning* on the code. For example, if $y = f(x)$ and $z = h(y)$, then the second definition can safely be rewritten as $z = h(f(x))$ without affecting the result. A program written in a functional language resembles mathematical notation. This has obvious benefits when solving mathematical problems such as neural networks. As will be shown in this paper, there is a clear relationship between the equations and the final code. This allows the programmer to focus on the equations, leaving the compiler to handle the implementation.

Referential transparency also allows the compiler to defer the computation of a value until it is needed, or even to skip the computation if it turns out not to be required. A program that avoids unnecessary computations will usually be more efficient.

An additional advantage of functional programming is the greater availability of tools for property-based testing. In property-based testing, the programmer writes assertions about logical properties that a function should fulfil; the tool then generates the necessary input data and runs the tests, looking for cases where the assertions fail. In practise, the programmer can test the software more thoroughly, and with less effort, than would be possible using only conventional unit test methods. Property-based testing tools are especially easy to use with neural networks implementations; the inputs, targets, and patterns are generally numeric, and the tools can easily generate pseudo-random data in a suitable range [77, 78]. Because

of these advantages, a functional programming language was used for Créatúr.

Of the most well-known functional languages, Haskell, F, List, Erlang, Scala, and OCaml, only Haskell is a pure functional language. QuickCheck, a property-based testing tool, is available for Haskell. Also, benchmark tests [79] indicated that Haskell would perform well. For these reasons (and after confirming Haskell's suitability using a pilot project, described in Chapter 4), Haskell was chosen as the programming language for the Créatúr project. Brief introductions to Haskell and QuickCheck are provided below.

2.6.2 Programming language: Haskell

Haskell [80] is a purely functional programming language named after the logician Haskell Curry. Haskell uses strong static typing. Some basic features of the Haskell programming language are demonstrated below using simple examples.

The factorial of a positive integer n is the product of all integers from 1 to n. A Haskell definition of the factorial function is shown below.

```
fact 0 = 1
fact n = n * fact (n-1)
```

The syntax for function invocation is *function-name param1 param2 . . .* Parentheses are not normally required.

```
fact 7
```

Usually it is not *necessary* to specify a type signature for a function; in most cases the compiler can determine an appropriate type signature. However, providing a type signature can make the programmer's intention clearer.

```
fact :: Int -> Int
```

```
fact 0 = 1
fact n = n * fact (n-1)
```

The symbol `::` is read ”has type” and introduces a type specification. The notation `Int -> Int` indicates that the function `fact` takes one `Int` (integer) parameter, and returns an `Int`.

Consider the type signature for the following function, which takes two `Int` parameters and returns a `Bool` (Boolean).

```
f :: Int -> Int -> Bool
```

The reader may be surprised by the presence of two `->` operators, but this notation hints at something very important and fundamental to functional programming: the concept of *currying*, or partial function application. The `->` operator is right-associative. Parentheses can be added as shown below without changing the meaning; this will help to illustrate how currying works.

```
f :: Int -> (Int -> Bool)
```

Written this way, another way to view `f` emerges. Instead of viewing it as a function that takes two parameters, it can be thought of as a function takes one `Int`, and returns a second function that takes an `Int` and returns a `Bool`. One way to take advantage of this is to define a new function that *partially applies* `f`.

```
g :: Int -> Bool
g = f 3
```

Thus g is a function which, when given a parameter *x*, returns a function which invokes `f` with 3 as the first parameter, and *x* as the second parameter. For example, suppose `f` is defined as follows:

```
f :: Int -> (Int -> Bool)
f x y = x > y
```

Then `g 4 = f 3 4 = 3 > 4 = false`.

2.6.3 Property-based testing with QuickCheck

Choosing to use a functional programming language allowed the use of more advanced testing techniques than are generally available for imperative developers. Unit testing is the one of the key techniques for verifying imperative code. In this technique, the tester writes code to test individual cases. For some applications, determining the desired result for each test case can be time-consuming, which necessarily limits the number of cases that can be tested.

Property-based testing tools such as QuickCheck [77] take a different approach. The tester defines properties that should hold for all cases, or at least for all cases satisfying certain criteria. In most cases, QuickCheck can automatically generate suitable pseudo-random test data and verify that the properties are satisfied, saving the tester's time. In keeping with the test-driven development [81] methodology, these properties could be defined and the tests automated before any code is written for the unit under test. In fact, it should be possible to define a significant number of testable properties during requirements analysis.

QuickCheck can also be invaluable in isolating faults, and finding the simplest possible test case that fails. This is partially due to the way QuickCheck works: it begins with "simple" cases (e.g., setting numeric values to zero or using zero-length strings and arrays), and progresses to more complex cases. When a fault is found, it is typically a minimal failing case. If the default functions provided

by QuickCheck for generating pseudo-random test data are not suitable, the tester can write custom functions.

2.7 Summary

The Créatúr project, and this thesis, incorporate ideas from a variety of fields. Those ideas are summarised below, with references to the section in which the topic was discussed.

There exist systems which operate according to simple rules, but which exhibit complex behaviour that would be impractical or impossible to predict by merely examining the rules. This behaviour is termed *emergent* (2.1.1). The variety of biological life is an emergent phenomenon of the DNA molecule. Similarly, the mind is widely thought to be an emergent phenomenon of the brain. The concept of emergence (at least in its weaker form), is a recurring theme in this thesis.

The intentional stance treats an object as an agent with beliefs and desires, and assumes that the object will act in accordance with those beliefs and desires (2.1.2). Used with caution, the intentional stance can make it easier to discuss and predict the behaviour of objects. It can offer insight, and suggest experiments that should be performed. The intentional stance is used to describe the behaviour of animats in the Créatúr project.

Evolution by natural selection may occur in any substrate, provided that it satisfies the three conditions (2.2.1). It can be used to develop a complex and adaptive system from simple parts, which is one of the goals of both AI and ALife. It may even be beneficial to include more than one type of evolution. A type of evolution by natural selection may even occur in the brain (2.3.1). This may help to explain

how a mind arises from a brain. It may be useful to implement a form of Neural Darwinism in AI and ALife projects. A type of evolution was incorporated into the brains of the animats developed for the Créatúr project.

Sexual reproduction may be adaptive for both biological lifeforms and animats (2.2.2). Most ALife projects use haploid organisms, but diploid organisms may have an evolutionary advantage (2.5.5). The animats developed for the Créatúr project are diploid, and reproduce sexually (i.e. each parent contributes a set of chromosomes), but they have only one sex.

Human brains have similar wiring, at least in how they represent concrete nouns and images (2.3.2). A possible explanation for this is that brains inherit some "pre-wiring"; it may be useful to implement some genetically-determined "pre-wiring" in the brains of animats. A type of genetically controlled neural pre-wiring was used in dotes, one of the animats developed for the Créatúr project.

AI has produced interesting and practical techniques for modelling certain aspects of intelligence, including ANNs (2.4.1) and SOMs (2.4.2). Hebbian learning (2.4.1) is a technique used in ANNs where "cells that fire together, wire together". Many types of equations can be used to update the neural weights, including Hebb's Rule and Oja's Rule. Hebbian learning was incorporated into the brains of the animats developed for the Créatúr project. The SOM is particularly useful for recognising patterns in data. A simplified SOM is incorporated into the brains of wains, one of the animat species developed for the Créatúr project.

ALife programs can develop diverse ecologies in which biological features such as parasitism, predator prey cycles, and punctuated equilibrium naturally emerge (2.5.1, 2.5.2). However, just as in biology, everything an animat does should have a cost (2.5.2). It may be beneficial to have a mechanism that protects animats until

they have learned how to survive. Ways to accomplish this include implementing some sort of instinct (2.5.3) or parental care. In order to produce complex behaviour, it may be necessary to have a complex environment and a complex genome (2.5.4). The Créatúr project uses a pattern-rich data source as the environment, and uses data analysis as the survival problem. The animat genomes used in the Créatúr are complex.

It is generally easier to understand and predict the behaviour of a program if it is written in a functional language (2.6.1). Functional programming languages provide better support for property-based testing, in which the programmer writes assertions about logical properties that a function should fulfil, and the tool then generates and runs the tests. The Créatúr project uses Haskell, a functional programming language (2.6.2), and QuickCheck, a property-based testing tool.

Chapter 3

Objectives and Approach

This chapter describes the objectives and the guiding principles of the Créatúr project.

3.1 Objectives

The objectives of the Créatúr research project were:

1. To evolve an ALife population with sufficient intelligence to discover patterns in data, and to make survival decisions based on those patterns.

2. To create a population that adapts to its environment through both evolution and lifetime learning.

3. To evolve an ALife population with some general-purpose problem solving skills, that could be used as "seeds" for projects requiring specialised skills. It should be feasible to create a specialised population by starting with an existing population of animats with basic intelligence and introducing new

survival challenges gradually until the animats have developed the new skills required. furthermore, it should be faster to do this than it would be to evolve a specialised population “from scratch”.

3.2 Approach

The key elements of the approach used in the Créatúr research project are described below. This approach is based on the research as described in the literature review in Chapter 2. The key elements of the approach are mapped to the objectives in Appendix B.

3.2.1 Combine AI and ALife

As discussed in section 2.4, the most ambitious AI projects attempt to produce the sort of intelligent behaviour found in Dennett’s Popperian creatures. However, it has so far proved impossible to define the characteristics that would convince us that a machine or a program is intelligent. Perhaps we would not recognise intelligence unless it has a familiar form: a creature using intelligence to survive in a complex environment. Perhaps it isn’t even meaningful to speak of intelligence in any other context.

Most ALife projects use fairly simple brain designs; this is sufficient to model many aspects of biology. But if the animats had more sophisticated brains, using techniques from AI such as neural networks, perhaps they would exhibit more complex behaviour.

3.2.2 Use data as the environment

As discussed in 2.5.4, a complex environment may be necessary to ALife project However, simulating a complex environment usually requires a large amount of programming and is processor-intensive. To address this, the environment used for the Créatúr project is data – which can be as complex as desired. The inhabitants would survive – or not – based on their ability to discover patterns in the data and to react appropriately.

In a sense, any ALife population can be said to live in a universe of data; the animats are computer programs, and all programs do is manipulate data. But there is a fundamental difference in Créatúr: The environment is not a simulation; it is real… real data. There is no layer of abstraction between the inhabitants of the Créatúr program and their environment. This has several advantages:

- It is not necessary to decide in advance how sophisticated the animats should be in order to solve the problem of interest. Assuming the problem is solvable, that sufficient time is allowed, and that the animats can evolve more complexity, then the animats will become just as complex as they need to be in order to "solve" their environment.

- It is easier to analyse the behaviour of the system because it can be described naturally, using the intentional stance. For example, "the animat saw this string of data, (mis)identified it as a source of edible food, and attempted to eat it. It lost too much energy as a result, and died."

- The Créatúr habitat can be used in two ways: as a tool for studying ALife, or as a tool for analysing complex data and discovering patterns.

- When the user's goal is to study ALife (rather than to perform data analysis), then any data set can be used as the environment provided it contains patterns for the animats to discover. If a more complex environment is desired, simply use more complex data. However, there must be patterns in the data; random data is complex, but not useful.

Based on the literature review undertaken (see Chapter 2), using data directly as the the environment is a novel approach to ALife.

3.2.3 Frame data analysis as a survival problem

Most data analysis problems can be re-framed as survival problems for animats. Consider the most basic needs of any animal: eating, mating, and self-preservation. In order to meet these needs, an animal must discover and recognise patterns, and make predictions. Table 3.1 shows some examples of this analysis. Of course, these predictions are not necessarily experienced as conscious thoughts in the animal's mind. However, since animals tend to act in ways that favour their survival, it is reasonable to view an action that an animal takes as a prediction that the outcome of that action will benefit that animal. This is an example of the intentional stance.

If the actions that an animat takes can be viewed (using the intentional stance) as a prediction that the animat will benefit by that action, then we can get the animats to make predictions of interest to us by arranging the environment so that good predictions are beneficial to the animat's survival. Table 3.2 lists some examples. In order to re-frame these applications as survival challenges for animats, the data itself becomes a universe to interact with, and the pattern of interest becomes a food source. By "eating" an object (string of data), an animat makes a

Table 3.1: Basic survival needs viewed as problems of pattern recognition and prediction.

need	pattern	sample prediction
food	self	If I try to eat this bit of flesh, I will feel pain.
food	prey	This entity is edible, and looks weak. I can catch it without expending too much of my energy.
reproduction	others of my kind	This entity looks healthy and is a suitable mating partner.
reproduction	offspring	This entity is my child; I must protect it.
self-preservation	parent	This entity will feed me and protect me.
self-preservation	predator	This entity is a threat; I should flee.

prediction that the object is edible (i.e., a pattern of interest), and that it will be rewarded with energy. If desired, patterns could be grouped into types, and a basic need associated with each. For example, patterns of one type might reduce hunger when identified, in another, boredom. Imposing a hierarchy of needs (e.g., hunger, then mating, then boredom) allows the discovery of patterns of one type to be prioritised over those of another type.

Based on the literature review undertaken (see Chapter 2), framing data analysis as a survival problem is a novel approach to ALife.[1]

[1]Genetic programming and evolutionary programming could, in a sense, be said to frame data analysis as a survival problem, in that each individual is a solution to a problem of interest, and the individual lives or dies according to how successful it is as a solution. However, in this thesis the term "survival problem" is used in a more biologically realistic sense: the animats must recognise patterns in order to find food and mate. Furthermore, the animats used in Créatúr are not *solutions*, they are problem *solvers*.

Table 3.2: Software applications viewed as problems of pattern recognition and prediction. Note the similarity to Table 3.1.

application	pattern	sample
handwriting recognition	The first symbol is 'h'. The next symbol is 'O', 'o' or '0'. The next symbol is either 't' or '+'.	This string is the word "hot".
log data analysis	This data entry is anomalous.	Probably an intrusion attempt; report it.
Quality of Service	The mobile unit was at position p_1 at time t_1, and is now at position p_2.	The user is moving out of range, re-route the call.

3.2.4 Use multiple kinds of evolution

The ultimate goal of ALife and AI is to build systems that match the complex and purposeful behaviour found in nature. However,

> This presents a dilemma: we do not understand such behaviours well enough to program them into a machine. So we must either increase our understanding until we can, or create a system which outperforms the specifications we give it. The first possibility includes the traditional top-down methodology, which appears as inappropriate here as it has so far proved to be for (symbolic) artificial intelligence... The second option is to create systems which somehow outperform the specifications given them and which are open to producing increasingly complex advantageous behaviours. Evolution in nature has no (explicit) evaluation function. [This] is why novel structures and behaviours emerge [67].

Projects such as Tierra and PolyWorld used evolution effectively, creating models that were life-like in many aspects. Their success, combined with the theory of Neural Darwinism, suggest that using multiple kinds of evolution might support more life-like behaviour. The following kinds of evolution are used in Créatúr:

Traditional evolution Rather than have a fixed brain design, use genetics to specify the parameters and allow evolution to improve the brain design. Also allow the animats to genetically inherit some "pre-wiring" of the brain. This should allow later generations to be more intelligent than earlier generations.

Neural Darwinism Use natural selection *within* the brain, as it operates, to form new connections and patterns, and to prune connections and patterns that prove to be least useful. This should allow an individual to continue learning throughout its lifetime, and to react to changes in the environment. (The changes to the brain made by Neural Darwinism would not be passed on to any offspring.)

Based on the literature review undertaken (see Chapter 2), using multiple kinds of evolution is a novel approach to ALife.

3.2.5 No fitness function except survival

In Genetic Algorithms, fitness functions are used in situations where it is possible to objectively assess the quality of a solution. But in ALife and AI, the goal is to build animats that are similar to biological lifeforms. In biology the only way to assess the fitness of a particular combination of genes is to compare the survival

of the sub-population carrying them to the population at large; there is no other criterion. Nature does not have an explicit fitness function, and as discussed in Sections 2.5.1 and 2.5.2, Tierra and PolyWorld did not use fitness functions. Neither does Créatúr.

3.2.6 No free lunch

Everything an animat does should have a cost. This design principle is based on a lesson learned from PolyWorld's "indolent cannibals" (described in Section 2.5.2). In Créatúr, mating, having offspring, and rearing them requires energy. They also lose energy through a "metabolism tax" based on brain complexity, as in PolyWorld.

3.2.7 Protect the young while they learn

A brand-new, empty brain is useless. An animat must have some basic skills in order to fend for itself. One biological solution to this is *instinct*, an inherent behaviour that does not need to be learned. However, since Créatúr uses evolution to develop the brain architecture, it would be difficult to simulate instinct by creating some initial connections; we don't know which concept will be associated with a particular neuron. If instincts are needed, evolution will have to implement them. This may take the form of "pre-wiring", as described in Section 3.2.4.

A second biological solution is *parenting*. Depending on the species, parents may protect their offspring while they learn, model adaptive behaviours that the offspring will emulate, or even actively teach them adaptive behaviours. In Créatúr, parents fill the first two roles. Offspring "observe" their parents and learn

from their behaviour by sharing in the outcome. For example, if the parent eats something edible, the energy gained will be shared between parent and child, allowing the child to make a connection between food and the reduction of hunger. Children cannot act independently; they can only benefit or suffer through the actions of the nurturing parent. There are no predators in Créatúr to defend against; nevertheless parents indirectly protect their offspring by virtue of the fact that they are less likely to make fatal mistakes than the child would if left to survive alone. Based on the literature review undertaken (see Chapter 2), protecting the young while they learn is a novel approach to ALife.

3.2.8 Use diploid animats

As discussed in 2.5.5, sexual reproduction may be beneficial in an ALife population, however it is seldom used because it halves the number of mating opportunities, and ALife populations are usually small due to processor limitations. The animats used in this project are diploid, although they only have one sex. This provides a way to exploit some of the advantages of sexual reproduction, while still maximising the number of mating opportunities.

3.2.9 Provide a means for animats to estimate degrees of kinship

In the biological world, appearance can be used to distinguish between animals of one's own species and animals of other species (or sub-species). PolyWorldians have a genetically-determined appearance for this reason, as discussed in 2.5.2. Although it was not expected that the lifeforms in Créatúr would demonstrate kin

selection or speciation in the short term, this might occur eventually. To support this possibility, the animats have a genetically-determined appearance, and they can sense the appearance of potential mates.

3.3 Summary

The key elements of the approach used in the Créatúr project are:

- Combine AI and ALife.
- Use data as the environment. ★★★
- Frame data analysis as a survival problem. ★★★
- Use multiple kinds of evolution. ★★★
- No fitness function except survival.
- No free lunch.
- Protect the young while they learn. ★★★
- Use diploid animats.
- Provide a means for animats to estimate degrees of kinship.

Elements marked with ★★★ are novel approaches to ALife, based on the literature review undertaken.

Chapter 4

Pilot Project: Numeral Recognition

Early in the Créatúr project, the author identified the following needs:

- to gain familiarity with the some of the technologies (neural networks and the challenges involved in training them, functional programming, property-based testing) considered for use in Créatúr
- to gain familiarity with the some of the tools (Haskell, QuickCheck, the MNIST database) considered for use in Créatúr
- to assess the suitability of these technologies and tools

To meet those needs, a smaller application was developed as a pilot project. The pilot project was to implement a neural network which would recognise handwritten numerals. This chapter describes that pilot project.

4.1 The MNIST database

This pilot project used the MNIST database of handwritten digits, which consists of a training set of 60,000 examples and a test set of 10,000 examples [82, 83]. The digits have been size-normalised and centred in a fixed-size image. This database was used because it requires minimal pre-processing, has been widely used for testing neural network training methods, and results for common methods are readily available.

The MNIST database was created using black-and-white images of handwritten numerals collected from U.S. Census Bureau employees and high-school students. The images were re-sized to fit into a 20x20 pixel square while preserving their aspect ratio, and anti-aliased, resulting in an image with 256 grey levels. Each image was then placed on a 28x28 pixel white background so that the centre of mass of the pixels was positioned in the centre of the 28x28 square. A sample image is shown in Figure 4.1. The MNIST data was used as-is; without additional image processing.

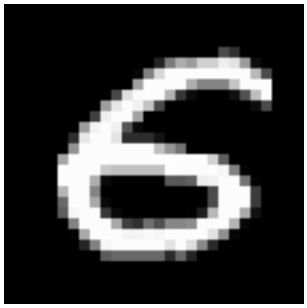


Figure 4.1: A sample numeral from the MNIST database.

4.2 The network

The network used in this pilot project is a feed-forward network, as described in 2.4.1. There is only one hidden layer. The 784 pixels from the image are presented to the input layer, which contains 784 neurons. The output layer has 10 neurons, one for each of the digits. The size of the hidden layer can be adjusted to achieve a balance between accuracy and performance.

4.3 Back-propagation

Back-propagation is a common method of training artificial neural networks. It was chosen for the pilot project because it is easy to implement and results for it are available [84], not because it was anticipated that Créatúr would use back-propagation.

After an input pattern is propagated forward through the network to produce an output pattern, the output pattern is compared to the target (desired) pattern, and the error is then propagated backward. During the back-propagation phase, each neuron's contribution to the error is calculated, and the network configuration can be modified with the goal of reducing future errors. Back-propagation is a supervised training method, so the correct answers for the training set must be known in advance, or be calculable. A simple "no-frills" back-propagation algorithm was used; it is described below. This simple algorithm is sufficient for demonstrating a functional approach to neural networks.

The *error* is a function of the vector difference between the output pattern and the *target pattern* (desired output). The network can be trained by adjusting the

network weights with the goal of reducing the error. Back-propagation is one technique for choosing the new weights [54]. This is a *supervised training* process; the network is presented with both the input pattern as well as the target pattern. The error from the output layer is propagated backward through the hidden layers in order to determine each layer's contribution to the error. This process is illustrated in Figure 4.2. The weights in each layer are then adjusted to reduce the error for that input pattern.

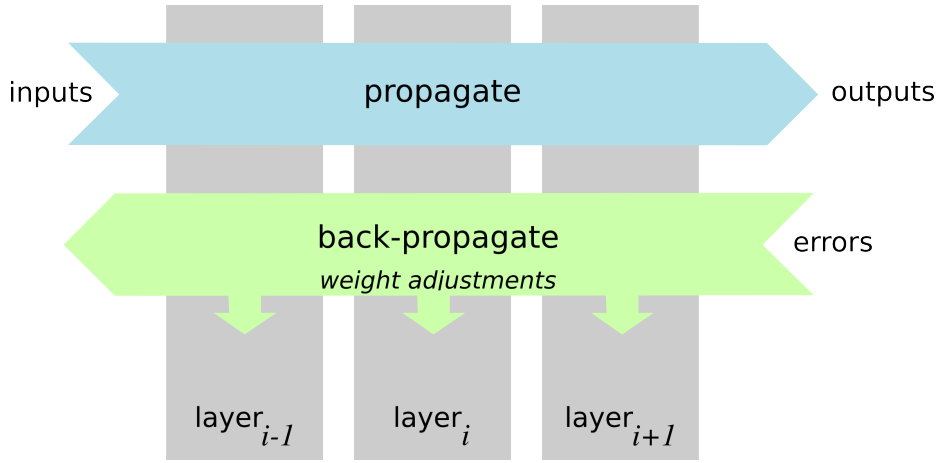


Figure 4.2: Back-propagation

4.4 Building a neural network

4.4.1 Building a neuron

In this implementation, matrices were used to represent the weights for the neurons in each layer. The matrix calculations are performed using Alberto Ruiz's `hmatrix` [85, 86], a purely functional Haskell interface to basic matrix computations and other numerical algorithms in GSL [87], BLAS [88, 89] and LAPACK

[90,91]. With a matrix-based approach, there is no need for a structure to represent a single neuron. Instead, the implementation of the neuron is distributed among the following entities

- the inputs from the previous layer
- the output to the next layer
- a column in the weight matrix
- an activation function (in this implementation, the same function is used for all neurons in all layers except the sensor layer)

The weight matrix was implemented using the `Matrix` type provided by `hmatrix`. The inputs, outputs and patterns are all column vectors; these use the `Matrix` type, via the synonym `ColumnVector`. In Haskell, the `type` keyword defines an alternative name for an existing type; it does not define a new type.[1]

```
type ColumnVector a = Matrix a
```

The final element needed to represent the neuron is the activation function. The implementation of this shows one of the advantages of a functional approach. Like most most functional programming languages, Haskell supports *first-class functions*; a function can be used in the same way as any other type of value. It can be passed as an argument to another function, stored in a data structure, or returned as result of function evaluation. The user can supply any activation function as an argument at the time the network is created.

[1] A complete code listing is available from the author, along with a sample character recognition application.

It is convenient to create a structure to hold both the activation function and its first derivative. (The back-propagation algorithm requires that the activation function be differentiable, and the derivative is required to apply the back-propagation method.) This helps to reduce the chance that the user will change the activation function and forget to change the derivative. This type is defined using Haskell's *record syntax*, and include a string to describe which activation function is being used.

```
data ActivationSpec
    = ActivationSpec
        {
            asF :: Double -> Double
          , asF' :: Double -> Double
          , desc :: String
        }
```

The first field, `asF`, is the activation function, which takes a `Double` (double precision, real floating-point value) as input and returns a `Double`. The second field, `asF'`, is the first derivative. It also takes a `Double` and returns a `Double`. The last field, `desc`, is a `String` value containing a description of the function.

Accessing the fields of a value of type `ActivationSpec` is straightforward. For example, if the name of the value is `s`, then its activation function is `asF s`, its first derivative is `asF' s`, and its description is `desc s`.

As an example of how to create a value of the type `ActivationSpec`, here is one for the identity function $f(x) = x$, whose first derivative is $f'(x) = 1$.

```
identityAS = ActivationSpec
    {
```

```
        asF = id
      , asF' = const 1
      , desc = "identity"
    }
```

The function `id` is Haskell's predefined identity function. The definition of `asF'` may seem puzzling. The first derivative of the identity function is 1, but it is not possible to write `asF' = 1`. Why not? Recall that the type signature of `asF'` is `Double -> Double`, so the expression assigned to it must take a `Double` and returns a `Double`. However, `1` is just a single number. It could be of type `Double`, but not `Double -> Double`. Any easy way to solve this issue, is to use the predefined `const` function, which takes two parameters and returns the first, ignoring the second. Partially applying it (supplying `1` as the first parameter), yields a function that takes a single parameter, and always returns the value `1`. So the expression `const 1` can satisfy the type signature `Double -> Double`.

The hyperbolic tangent is a commonly-used activation function; the appropriate `ActivationSpec` is defined below.

```
tanhAS :: ActivationSpec
tanhAS = ActivationSpec
    {
        asF = tanh
      , asF' = tanh'
      , desc = "tanh"
    }


tanh' x = 1 - (tanh x)^2
```

This takes advantage of Haskell's support for *first-class functions* to store functions in a record structure, and to pass functions as parameters to another function (in this case, the `ActivationSpec` constructor).

4.5 Building a neuron layer

To define a layer in the neural network, a record structure containing the weights and the activation specification is used. The weights are stored in an $n \times m$ matrix, where n is the number of inputs and m is the number of neurons. The number of outputs from the layer is equal to the number of neurons, m.

```
data Layer
    = Layer
        {
          lW :: Matrix Double
        , lAS :: ActivationSpec
    }
```

The weight matrix, `lW`, has type `Matrix Double`. This is a matrix whose element values are double-precision floats. This type is defined in the `hmatrix` package. The activation specification, `lAS` uses the type `ActivationSpec`, defined earlier. Again this uses the support for *first-class functions*; a value of type `Layer` is created by passing a record containing function values into another function, the `Layer` constructor.

4.5.1 Assembling the network

The network consists of a list of layers, and a parameter to control the rate at which the network learns new patterns.

```
data BackpropNet
  = BackpropNet
    {
      layers :: [Layer]
    , learningRate :: Double
    }
```

The notation `[Layer]` indicates a list whose elements are of type `Layer`. A value of type `BackpropNet` can be created using an expression such as the following.

```
layer = Layer { lW=w, lAS=s }
```

Of course, the number of outputs from one layer must match the number of inputs to the next layer. This is ensured by not exporting the `BackpropNet` constructor outside the module, and instead requiring the user to call a special function to construct the network. It is necessary to verify that the dimensions of a consecutive pair of network layers are compatible; The following function will report an error if a mismatch is detected.

```
checkDimensions :: Matrix Double -> Matrix Double -> Matrix Double
checkDimensions w1 w2 =
  if rows w1 == cols w2
       then w2
       else error "Inconsistent dimensions in weight matrix"
```

Assuming that no errors are found, `checkDimensions` simply returns the second layer in a pair.

The constructor function should invoke `checkDimensions` on each pair of layers. In an imperative language, a *for loop* would typically be used. In functional languages, a recursive function could be used to achieve the same effect. However, there is a more straightforward solution using an operation called a *scan*. There are several variations on this operation, and it can proceed either from left to right, or from right to left. This uses the predefined operation `scanl1`, read "scan-ell-one" (not "scan-eleven"). The letter l indicates that the scan starts from the left, and the numeral 1 indicates the variant that takes no starting value.

```
scanl1 f [x1, x2, x3, ...] == [x1, f x1 x2, f (f x1 x2) x3, ...]
```

Applying `scanl1 checkDimensions` to a list of weight matrices gives the following result (again assuming no errors are found).

```
scanl1 checkDimensions [w1, w2, w3, ...]
  == [w1, checkDimensions w1 w2,
          checkDimensions (checkDimensions w1 w2) w3, ...]
  == [w1, w2, w3, ...]
```

Therefore, if the dimensions of the weight matrices are consistent, this operation simply returns the list of matrices. The next task is to create a layer for each weight matrix supplied by the user. In an imperative language, the program might operate on each element in the weight matrix list using a *for loop*. In Haskell, the `map` function achieves the same goal. The expression `map buildLayer checkedWeights` will return a new list, where each element is the result of applying the function `buildLayer` to the corresponding element in the list of weight matrices. The def-

inition of `buildLayer` is simple, it merely invokes the constructor for the type `Layer`, defined earlier.

```
buildLayer w = Layer { lW=w, lAS=s }
```

Using the operations discussed above, the constructor function, `buildBackpropNet`, can now be defined.

```
buildBackpropNet
  :: Double
  -> [Matrix Double]
  -> ActivationSpec
  -> BackpropNet
buildBackpropNet lr ws s = BackpropNet { layers=ls, learningRate=lr }
  where checkedWeights = scanl1 checkDimensions ws
        ls = map buildLayer checkedWeights
        buildLayer w = Layer { lW=w, lAS=s }
```

The primary advantage of using functions such as `map` and `scanl1` is not that they save a few lines of code over an equivalent *for loop*, but that these functions more clearly indicate the programmer's intent.

4.6 Running the Network

4.6.1 A closer look at the network structure

The network consists of multiple layers of neurons, numbered from 0 to L, as illustrated in Figure 4.3. Each layer is fully connected to the next layer. Layer 0 is the sensor layer. It performs no processing; each neuron receives one component

of the input vector **x** and distributes it, unchanged, to the neurons in the next layer. Layer L is the output layer. The layers $l = 1..(L-1)$ are hidden layers. z_{lk} is the output from neuron k in layer l.

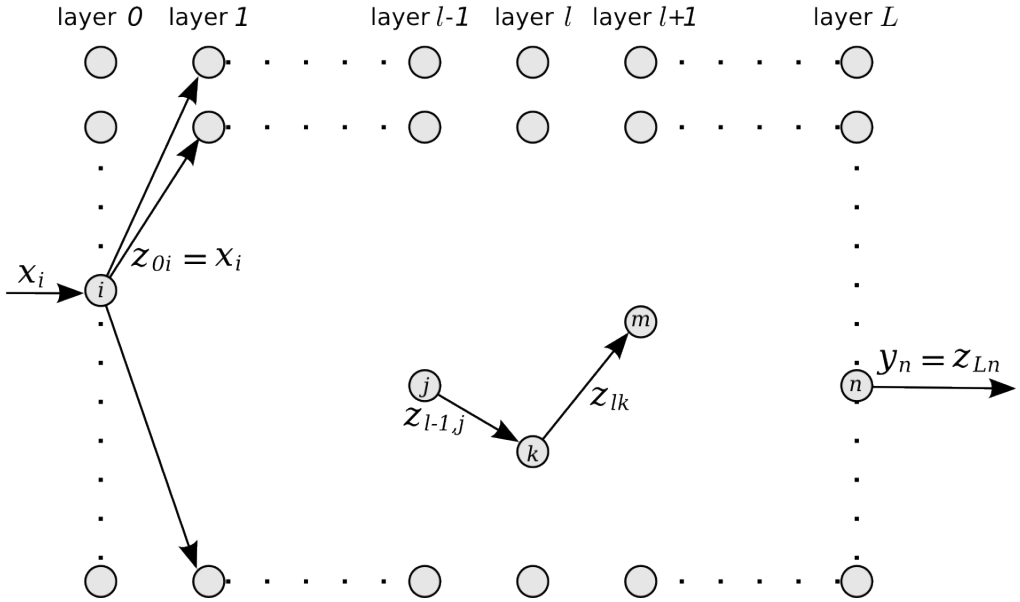


Figure 4.3: Propagation through the network.

The following notation is used.

- x_i is the ith component of the input pattern.
- z_{li} is the output of the ith neuron in layer l.
- y_i is the ith component of the output pattern.

4.6.2 Propagating through one layer

The activation of neuron k in layer l is

$$a_{0k} = x_k \tag{4.1}$$

$$a_{lk} = \sum_{j=1}^{N_{l-1}} w_{lkj} z_{l-1,j} \qquad l > 0 \tag{4.2}$$

where

- N_{l-1} is the number of neurons in layer $l-1$.
- w_{lkj} is the weight applied by the neuron k in layer l to the input received from neuron j in layer $l-1$. (Recall that the sensor layer, layer 0, simply passes along its inputs without change.)

The activation for layer l can be expressed using a matrix equation.

$$\mathbf{a_l} = \begin{cases} \mathbf{x} & l = 0 \\ \\ \mathbf{W}_l \mathbf{x} & l > 0 \end{cases} \tag{4.3}$$

The output from the neuron is

$$z_{lk} = f(a_{lk}) \tag{4.4}$$

where $f(a)$ is the activation function. For convenience, the function `mapMatrix` is defined; it applies a function to each element of a matrix (or column vector). This is analogous to Haskell's `map` function. The layer's output is calculated using the Haskell expression `mapMatrix f a`, where `f` is the activation function.

If the only goal is to propagate the input through the network, only the output from the final layer, $\mathbf{z}_L$, is required. However, the intermediate calculations must be kept because they will be required during the back-propagation pass. All of the necessary information is kept in the following record structure. Note that anything

between the symbol -- and the end of a line is a comment, and is ignored by the compiler.

```
data PropagatedLayer
    = PropagatedLayer
        {
          -- | The input to this layer
          pIn :: ColumnVector Double
          -- | The output from this layer
        , pOut :: ColumnVector Double
          -- | First derivative of the activation function for this layer
        , pF'a :: ColumnVector Double
          -- | The weights for this layer
        , pW :: Matrix Double
          -- | The activation specification for this layer
        , pAS :: ActivationSpec
        }
    | PropagatedSensorLayer
        {
          -- | The output from this layer
          pOut :: ColumnVector Double
        }
```

This structure has two variants. For the sensor layer (`PropagatedSensorLayer`), the only information needed is the output, which is identical to the input. For all other layers (`PropagatedLayer`), the full set of values is required. Now it is possible to define a function to propagate through a single layer.

```
propagate
  :: PropagatedLayer
  -> Layer
  -> PropagatedLayer
propagate layerJ layerK = PropagatedLayer
        {
          pIn = x
        , pOut = y
        , pF'a = f'a
        , pW = w
        , pAS = lAS layerK
        }
  where x = pOut layerJ
        w = lW layerK
        a = w <> x
        f = asF ( lAS layerK )
        y = mapMatrix f a
        f' = asF' ( lAS layerK )
        f'a = mapMatrix f' a
```

The operator `<>` performs matrix multiplication; it is defined in the `hmatrix` package.

4.6.3 Propagating through the network

To propagate through the entire network, a sensor layer is created to provide the inputs, and use another scan operation, this time with `propagate`. The `scanl` func-

tion is similar to the `scanl1` function, except that it takes a starting value.

```
scanl f z [x1, x2, ...] == [z, f z x1, f (f z x1) x2), ...]
```

In this case, the starting value is the sensor layer.

```
propagateNet
  :: ColumnVector Double
  -> BackpropNet
  -> [PropagatedLayer]
propagateNet input net = tail calcs
  where calcs = scanl propagate layer0 (layers net)
        layer0 = PropagatedSensorLayer{ pOut=validatedInputs }
        validatedInputs = validateInput net input
```

The function `validateInput` verifies that the input vector has the correct length, and that the elements are within the range [0,1]. Its definition is straightforward.

4.7 Training the network

This section uses the matrix equations for basic back-propagation as formulated by Hristev [92, Chapter 2]. The back-propagation algorithm requires each layer to be operated on in turn, using the results of the operation on one layer as input to the operation on the next layer. The input vector $\mathbf{x}$ is propagated *forward* through the network, resulting in the output vector $\mathbf{z}_L$, which is then compared to the target vector $\mathbf{t}$ (the desired output). The resulting error, $\mathbf{z}_L - \mathbf{t}$ is then propagated *backward* to determine the corrections to the weight matrices:

$$W_{new} = W_{old} - \mu \nabla E \tag{4.5}$$

where μ is the learning rate, and E is the error function. For E, the sum-of-squares error function, defined below, can be used.

$$E(W) \equiv \frac{1}{2} \sum_{q=1}^{N_L} [z_{Lq}(x) - t_q(x)]^2 \tag{4.6}$$

where z_{Lq} is the output from neuron q in the output layer (layer L). The error gradient for the last layer is given by:

$$\nabla_{z_L} E = \mathbf{z}_L(x) - \mathbf{t} \tag{4.7}$$

The error gradient for a hidden layer can be calculated recursively according to the equations below.[2]

$$(\nabla E)_l = [\nabla_{z_l} E \odot f'(a_l)] \cdot z_{l-1}^T \quad \textit{for layers } l = \overline{1, L} \tag{4.8}$$

$$\nabla_{z_l} E = W_{l+1}^t \cdot [\nabla_{z_{l+1}} E \odot f'(a_{l+1})] \quad \textit{calculated recursively from L-1 to 1} \tag{4.9}$$

The symbol $\odot$ is the *Hadamard*, or element-wise product.

4.7.1 Back-propagating through a single layer

The result of back-propagation through a single layer is stored in the structure below.[3]

[2] See [92, Chapter 2] for the derivation.

[3] The expression $\nabla_{z_l} E$ is not easily represented in ASCII text, so the name "dazzle" is used in the code.

```
data BackpropagatedLayer
    = BackpropagatedLayer
        {
          -- | Del-sub-z-sub-l of E
          bpDazzle :: ColumnVector Double
          -- | The error due to this layer
        , bpErrGrad :: Matrix Double
          -- | Value of 1st derivative of the activation function
        , bpF'a :: ColumnVector Double
          -- | The input to this layer
        , bpIn :: ColumnVector Double
          -- | The output from this layer
        , bpOut :: ColumnVector Double
          -- | The weights for this layer
        , bpW :: Matrix Double
          -- | The activation specification for this layer
        , bpAS :: ActivationSpec
        }
```

The next step is to define the `backpropagate` function. For hidden layers, equation (4.9) is used.

```
backpropagate
  :: PropagatedLayer
  -> BackpropagatedLayer
  -> BackpropagatedLayer
backpropagate layerJ layerK = BackpropagatedLayer
        {
```

```
          bpDazzle = dazzleJ
        , bpErrGrad = errorGrad dazzleJ f'aJ inputJ
        , bpF'a = pF'a layerJ
        , bpIn = pIn layerJ
        , bpOut = pOut layerJ
        , bpW = pW layerJ
        , bpAS = pAS layerJ
        }
    where dazzleJ = wkT <> (dazzleK * f'aK)
          dazzleK = bpDazzle layerK
          wkT = trans ( bpW layerK )
          errK = bpErrGrad layerK
          f'aK = bpF'a layerK
          f'aJ = pF'a layerJ
          inputJ = pIn layerJ
errorGrad :: ColumnVector Double -> ColumnVector Double ->
                ColumnVector Double -> Matrix Double
errorGrad dazzle f'a input = (dazzle * f'a) <> trans input
```

The function `trans` calculates the transpose of a matrix. The operator `*` used in the definition of `dazzleJ` appears between two column vectors, `dazzleK` and `f'aK`, so it calculates the Hadamard (element-wise) product rather than a scalar product. The final layer uses equation (4.7).

```
backpropagateFinalLayer
    :: PropagatedLayer
    -> ColumnVector Double
```

```
    -> BackpropagatedLayer
backpropagateFinalLayer l t = BackpropagatedLayer
        {
          bpDazzle = dazzle
        , bpErrGrad = errorGrad dazzle f'a input
        , bpF'a = (pF'a l)
        , bpIn = pIn l
        , bpOut = pOut l
        , bpW = pW l
        , bpAS = pAS l
        }
        where dazzle =  pOut l - t
              f'a = pF'a l
              input = pIn l
```

4.7.2 Back-propagating through the network

The `scanl` function, which operates on an array from left to right, was discussed above. For the back-propagation pass, `scanr`, which operates from right to left, is used. Figure 4.4 illustrates how `scanl` and `scanr` will act on the neural network. The boxes labelled `pc` and `bpc` represent the result of each propagation operation and back-propagation operation, respectively. Viewed in this way, it is clear that `scanl` and `scanr` provide a layer of abstraction that is ideally suited to back-propagation.

The definition of the `backpropagateNet` function is very similar to that of `propagateNet`.

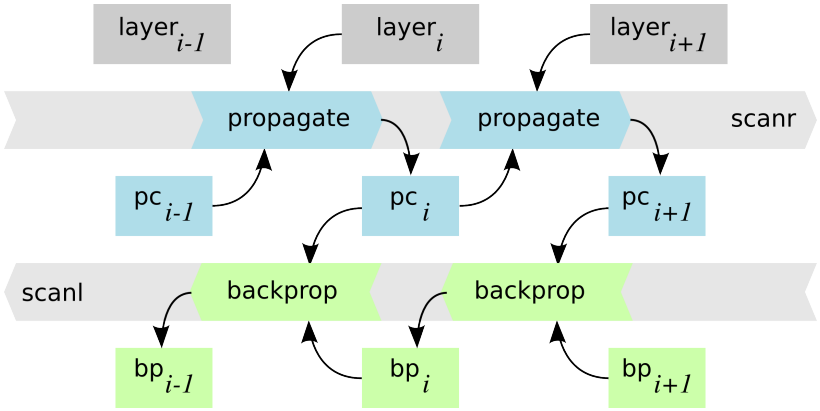


Figure 4.4: A schematic diagram of the implementation.

```
backpropagateNet
  :: ColumnVector Double
  -> [PropagatedLayer]
  -> [BackpropagatedLayer]
backpropagateNet target layers =
    scanr backpropagate layerL hiddenLayers
  where hiddenLayers = init layers
        layerL = backpropagateFinalLayer (last layers) target
```

4.7.3 Updating the weights

After the back-propagation calculations have been performed, the weights can be updated using equation (4.5).

```
update :: Double -> BackpropagatedLayer -> Layer
update rate layer = Layer
        {
          lW = wNew
```

```
        , lAS = bpAS layer
        }
    where wOld = bpW layer
          delW = rate .* bpErrGrad layer
          wNew = wOld - delW
```

The operator `.*` performs element-wise multiplication of a matrix by a scalar.

4.8 Testing

An in-depth look at QuickCheck is beyond the scope of this thesis. Instead, one example is used to illustrate the value of property-based testing. What properties should a neural network satisfy, no matter what input data is provided? One property it should satisfy is that if the network is trained once with a given input pattern and target pattern, and immediately run on the same input pattern, the error should be reduced. Another way of saying this is that training should reduce the error in the output layer, unless the error is negligible to begin with. The property is defined as follows.

```
-- | Property: training reduces error in the final (output) layer
trainingReducesFinalLayerError ::
  (ColumnVector Double, Layer, ColumnVector Double) -> Property
trainingReducesFinalLayerError (x, l, t) =
    classifyRange "len x " n 0 25 $
    classifyRange "len x " n 26 50 $
    classifyRange "len x " n 51 75 $
    classifyRange "len x " n 76 100
```

```
    (errorAfter < errorBefore || errorAfter < 0.01)
        where n = inputWidth l
              pl0 = PropagatedSensorLayer{ pOut=x }
              pl = propagate pl0 l
              bpl = backpropagateFinalLayer pl t
              errorBefore = P.magnitude (t - pOut pl)
              lNew = update 0.0000000001 bpl
              plNew = propagate pl0 lNew
              errorAfter =  P.magnitude (t - pOut plNew)

-- | Testable property:
-- | Training reduces error in the final (output) layer
prop_trainingReducesFinalLayerError :: Property
prop_trainingReducesFinalLayerError =
    forAll arbLayerTestData trainingReducesFinalLayerError
```

The function `trainingReducesFinalLayerError` takes a tuple consisting of an input pattern `x`, an output layer `l`, and a target pattern `t`, and returns a Boolean to indicate whether or not the property holds. (A *tuple* is a way to package several values as a single value. Unlike lists, the elements in a tuple do not have to be of the same type.) This particular property only checks that training works for an output layer; the complete implementation tests other properties, including the effect of training on hidden layers. The resulting property returns true if the error before training is less than the error after, or if the error is already negligible (less than 0.01 for this test). The `classifyRange` statements are useful when running the tests interactively; they display a brief report indicating the distribution

of the test coverage. The function `trainingReducesFinalLayerError` specifies that a custom generator for pseudo-random test data, `arbLayerTestData`, is to be used. The generator `arbLayerTestData` ensures that the ”simple” test cases that QuickCheck starts with consist of short patterns and a network with a small total number of neurons. This helps to ensure that if there are errors, the first failing test case found will be easier to analyze.

The test is run in `GHCi`, a Haskell interpreter.

```
ghci> quickCheck prop_trainingReducesFinalLayerError
+++ OK, passed 100 tests:
62% len x 0..25
24% len x 26..50
12% len x 51..75
 2% len x 76..100
```

By default, QuickCheck runs 100 test cases. Of these, 62% of the patterns tested were of length 25 or less. More test cases can be requested; the test of 10,000 cases below ran in 20 seconds.[4] It would not have been practical to write unit tests for this many cases, so the benefit of property-based testing as a supplement to unit testing is clear.

```
ghci> quickCheckWith Args{replay=Nothing, maxSuccess=10000,
maxDiscard=100, maxSize=100} prop_trainingReducesFinalLayerError
+++ OK, passed 10000 tests:
58% len x 0..25
25% len x 26..50
```

[4]On a 3.00GHz quad core processor running Linux

```
12% len x 51..75
 3% len x 76..100
```

4.9 Summary

Haskell provides operations such as `map`, `scanl`, `scanr`, and their variants, that are particularly well-suited for implementing neural networks in general, and back-propagation in particular. These operations are not unique to Haskell; they are part of a category of functions commonly provided by functional programming languages to factor out common patterns of recursion and perform the types of operations that would typically be performed by loops in imperative languages. Other operations in this category include *folds*, which operate on lists of values using a combining function to produce a single value, and *unfolds*, which take a starting value and a generating function, and produce a list.

Functional programming has some clear advantages for implementing mathematical solutions. There is a straightforward relationship between the mathematical equations and the corresponding function definitions. Note that in the back-propagation example, it was only necessary to create data structures and write definitions. At no point were instructions provided on how to sequence the operations. The final results were defined in terms of intermediate results, which were defined in terms of other intermediate results, eventually leading to definitions in terms of the inputs. The compiler is responsible for either finding an appropriate sequence in which to apply the definitions, or reporting an error if the definitions are incomplete.

Property-based testing has obvious benefits. With minimal effort, the applica-

tion can be tested very thoroughly. But the greatest advantage of property-based testing may be its ability to isolate bugs and produce a minimal failing test case. It is much easier to investigate a problem when the matrices involved in calculations are small.

Functional programming requires a different mind-set than imperative programming, which can lead to a conceptual mismatch. Textbooks on neural network programming usually provide derivations and definitions, but with the ultimate goal of providing an algorithm for each technique discussed. The functional programmer needs only the definitions, but would be wise to read the algorithm carefully in case it contains additional information not mentioned earlier.

Functional programming may not be suited to every programmer, or to every problem. However, some of the concepts demonstrated here can be applied in imperative languages. Some imperative languages have borrowed features such as first-class functions, maps, scans and folds from functional languages. And some primarily functional languages provide mechanisms for doing object-oriented programming. Crossover between these two paradigms is beneficial because it provides programmers with more ways to approach problems.

As a result of this pilot project, it was decided to use Haskell, QuickCheck, and the MNIST database in the Créatúr project.

Chapter 5

Créatúr: an ALife habitat

This chapter describes Créatúr, a reusable software framework developed for this project, which automates experiments with artificial lifeforms and encapsulates functionality common to most ALife species. The framework is described from the perspective of a potential user; for a more formal requirements definition, see Appendix B. For advice on how to set up an experiment using Créatúr, see Section 8.1.2.

Créatúr consists of *a*) a daemon (background task) which is responsible for scheduling and running all events in the environment, and *b*) a library of functions for implementing common behaviours such as eating, mating, and metabolism. The user provides the implementation of the animats and other objects in the virtual environment. This architecture is illustrated in Figure 5.1.

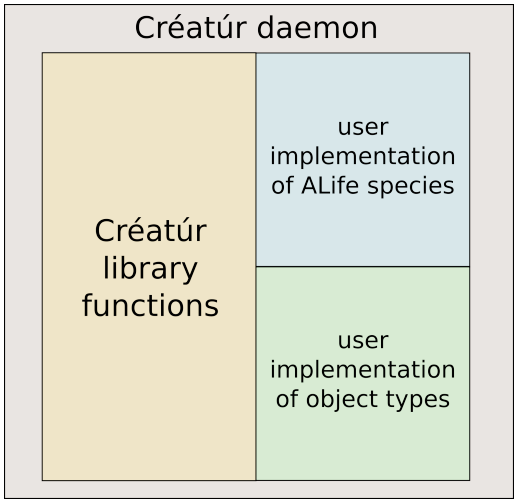


Figure 5.1: Créatúr architecture

5.1 Features of Créatúr

Créatúr automates tasks which are independent of the particular animats implementation. It provides the features listed below.

- Generating an initial population with random genes, based on a user-specified genome.
- Maintaining a minimum population by generating and adding new random animats. This feature is primarily used to diagnose problems. If a population is dying off, it may be easier to investigate the reason while the experiment is still running.
- Assigning a unique ID to all animats, whether born or generated. This is for convenience in tracing the activities of a particular animat
- Removing dead animats from the population and archiving them for further study. An animat dies when its energy level reaches zero; there is no fitness

function.

- Representing data as objects in the environment. This makes it easy to use the data as the ALife environment. For example, each record in a database table could be an object.

- Scheduling random encounters of animats with objects in the environment, and between pairs of animats.

- When an encounter occurs, presenting the relevant sensory inputs to the animats involved, along with that animat's current status. The appropriate course of action in a given situation may depend on whether the animat is currently hungry, passionate, or bored, so the animat needs this information.

- Regularly deducting energy from animats to simulate metabolic requirements.

- "Reading" the animats' decisions in response to encounters, and enacting the consequences. For example, if an animat chooses to eat something, the result may be an increase in the animat's energy level. If an animat chooses to mate with another animat, the result is the birth of a new animat.

- Implementing reproduction between animats, and transferring some energy from both parents to the offspring.

- Keeping offspring with one parent until the offspring reaches maturity. This gives the child time to learn about the environment, and to learn survival techniques from the parent.

- Apportioning energy gains and losses between a parent and the child it is caring for. This helps to ensure that the child survives until maturity.
- Recording events, decisions, outcomes, and statistics to a rotating log file.
- Running as a system daemon which the user can start, stop, and restart as needed.
- Regularly saving the state of all animats, and persisting across invocations of the daemon. This allows the user to examine the population while Créatúr is running. Each animat is saved in a separate file. If a file containing one animat's data is corrupted (e.g., in a power failure or disk crash), the file can be deleted and the loss of the animat can be interpreted as an accidental death. However, if all animats are written to a single file which becomes corrupted, it might be tedious or difficult to recover the population.
- Keeping the system running, even when "errors" occur. A "bug" might have consequences that evolution could exploit in interesting ways. For this type of application, the user should make the decision to halt a trial or continue, not the software.

5.2 Créatúr set-up

In order to run Créatúr, the user must provide implementations for *a*) one or more species of animats, and *b*) one or more types of objects for the animats to interact with. Créatúr is compiled with the user implementations to produce an executable program. The requirements for user implementations are listed below.

5.2.1 Animats

Each species implementation must make the features and operations listed below available to Créatúr.

- methods for determining information about the appearance of all animats in the environment.
- a method to feed sensory information to an animat about the environment (exteroception), and about its current state and physical needs (interoception)
- a method to read the action the animat has decided to take in response.
- a method for creating random strings of genetic material, suitable for constructing an initial population.
- a method which, given two alleles, applies the appropriate dominance relationship (see Section 5.3.3). This allows the user to implement dominant and recessive alleles, incomplete dominance, etc.
- methods to encode a gene sequence to a byte string, and decode a byte string into a gene sequence. The encoding scheme must be designed so that any random byte string can be decoded into a valid gene sequence. Requiring the genome design to be robust in this way helps to ensure that any useful partial solutions are not lost; they remain available to future generations. Also, evolution may proceed more quickly if all matings result in an animat. This reduces the possibility that evolution might get temporarily stuck in a

"dead end". There is no guarantee that the animat resulting from a particular gene sequence will be viable, however.

- a method for constructing an animat from its genome, used for reproduction.
- methods for determining and updating the animat's current energy level, passion level, boredom level, age, etc.
- methods for saving and restoring the animat's state.
- a method for determining the animat's metabolism, i.e., the energy it uses just to stay alive. This allows the user to influence the evolution of the animat species. For example, if the metabolism cost is partially based on brain size, this can help to ensure that the species doesn't evolve excessively large brains (which would consume excessive processor time).
- a method for determining whether or not an animat is mature.
- a method for determining an animat's level of devotion to potential offspring.

As long as the features listed above are provided, the animat can live in the Créatúr habitat; the internal implementation of the animat does not matter. For example, the animat might have a simple brain that makes decisions according to hard-coded rules, or it might have a sophisticated brain that learns from experience.

5.2.2 Objects

The user can create one or more types of objects for the animats to interact with. At the minimum, a "food" object is required (unless there are multiple animat species forming a complete food chain). Each object implementation must make the features and operations listed below available to Créatúr.

- a sequence of numeric values representing the object's description appearance (see Section 5.3.1). The description field is used for logging. Delta energy is positive for food, negative for poisonous objects.
- the energy gain (or loss) provided to an animat that eats the object.
- if other drives are implemented (e.g. boredom), the effect on those drives if the animat chooses a particular action. Delta passion is usually positive to discourage animats from attempting to mate with objects, instead encouraging animats to learn identify and mate with others of their species. Delta boredom is positive for toys.

5.3 A closer look at Créatúr

5.3.1 Object encounters

Créatúr schedules encounters between animats and objects in the environment. When it is time for an object encounter, Créatúr selects a random animat from the population, and a random object from the set of available objects. It presents the appearance of the object to the animat's sensors (exteroception), along with information about the animat's current state and physical needs (interoception),

and reads the animat's output signals to determine the course of action chosen. It then implements the pre-defined consequences of that action, and returns the animat to the population. Some typical examples of consequences are listed below.

- If the animat chooses to eat the object, and the object is edible, give energy to the animat.
- If the animat chooses to eat the object, and the object is poisonous, take energy from the animat.
- If the animat chooses to play with an object, and that object is "interesting", reduce the animat's boredom level.
- If the animat chooses to mate with an object, increase the animat's passion level. (This is useful for encouraging animats to learn identify others of their species, and to mate with them rather than with objects.)

5.3.2 Animat encounters

Créatúr also schedules encounters between pairs of animats. When it is time for an animat encounter, Créatúr selects two random animats from the population. It then presents the appearance of each animat to the other animat's sensors (exteroception), along with information about its own current state and physical needs (interoception), and reads the animats' output signals to determine the course of action chosen by each. It then implements the pre-defined consequences of that action, and returns both animats (and any resulting offspring) to the population. Some typical examples of consequences are listed below.

- If either animat wants to mate, it will lose some energy. This "flirting tax" simulates the cost of courting. (The time invested in courting could have been spent hunting for food instead.)
- If both animats want to mate, and the potential dam is not currently rearing a child, reproduction occurs. (See Section 5.3.3.)
- If potential sire wants to mate, but the potential dam is currently rearing a child, increase passion level of the potential sire. (Assuming that an animat's appearance indicates whether or not it is rearing a child, this might be useful for encouraging animats to choose mates that are more likely to be receptive.)
- If the animats choose to play with each other, reduce their boredom levels. (This might be useful for promoting more complex social interactions.)

5.3.3 Reproduction

Créatúr was designed to work with *diploid* animats. Each animat has two complete sequences of building instructions, either one of which would be sufficient to create an animat. By loose analogy with biology, an instruction is called a *gene*, and the various settings for a particular instruction are called *alleles*. When two animats mate, they each donate one string of genetic material to the offspring. Again by analogy with biology, we call this single string a *gamete*. To produce a gamete, Créatúr makes copies of the two strings of genes from the parent, and performs one or more of the operations listed below, in decreasing order of probability.

- *crossover*: breaking the strings at corresponding locations, and swapping the tails

- *cutting and splicing*: breaking the sequences at non-corresponding locations, and swapping the tails, thereby ending up with two sequences of different length

- *mutation*: randomly altering a bit in in one of the sequences

Afterward, one of the two resulting sequences is randomly selected as the gamete, and the other is discarded. The offspring receives one gamete from each parent, and thus ends up with two sequences of instructions. Thus, the offspring contains a mixture of genetic information from both parents.

Since the two sequences of instructions in a diploid animat's genome are generally not identical, before constructing the offspring the sequences are merged into a single sequence of instructions which we call the *blueprint*. As in biology, when the genetic instructions at corresponding locations differ, one instruction may take precedence over the other (i.e., has dominance), or the result may be a blending of the two instructions (i.e., incomplete dominance). These dominance relationships are enforced by the animat implementation. The implementation provides a method which, given two alleles, applies the appropriate dominance relationship and returns the resulting instruction for the blueprint. Créatúr invokes this method for each pair of genes, and compiles the blueprint. The animat implementation also provides a method for constructing an animat from the blueprint. Créatúr invokes this method to complete the process of reproduction.

5.3.4 Parenting

When mating occurs, one parent is arbitrarily chosen to be the sire, and one the dam. Both parents donate a fraction of their current energy to the offspring; after

that, the sire's role in parenting ends. The offspring remains with the dam until maturity, and shares in its experiences, thereby learning some basic survival skills. During this time, the dam shares a fraction of all energy gains or losses with the offspring.

5.3.5 Créatúr Time

Créatúr maintains a program counter, called Créatúr Time. This counter is used to schedule events. Créatúr is designed to run on a workstation that may be used for other purposes at the same time. The advantage of using a counter rather than system clock time is that it ensures that the availability of food and mating opportunities is not affected by the amount of processing time allocated to the Créatúr daemon, or by stopping and restarting the daemon. It also allows meaningful comparison of experiments performed on computer systems with different hardware and processing capacity.

5.4 Implementation and testing

In order to give the reader an idea of how Haskell and QuickCheck were used in the implementation of Créatúr, a short excerpt from the code will be presented and discussed.[1] The `Crossover` module provides crossover operations, and is used to create gametes for reproduction. This module resides in the `Genetics` package. The first step is declaring the module and the functions that are *exported* (visible to other packages). The method `randomCutAndSplice` is a general form of crossover. The method `randomCrossover` is a convenience method that can be used when

[1] A complete code listing is available from the author.

both strings are cut at the same position. These methods are both exported, along with `testModule`, which automatically tests the module.

```
module Genetics.Crossover
  (
    randomCutAndSplice,
    randomCrossover,
    testModule
  ) where
```

This module requires two library packages. QuickCheck (`Test.QuickCheck`) was introduced in Section 2.6.3. `Control.Monad.Random` provides a source of random values.

```
import Control.Monad.Random
import Test.QuickCheck
```

Next, an internal function is defined that takes two strings and the positions at which they should be cut, cuts them, swaps the tails, and splices them to form two new lists.

```
cutAndSplice :: Int -> Int -> ([a], [a]) -> ([a], [a])
cutAndSplice n m (as, bs) = (cs, ds)
    where cs = as1 ++ bs2
          ds = bs1 ++ as2
          (as1, as2) = splitAt n as
          (bs1, bs2) = splitAt m bs
```

As discussed in Section 2.6.2, the symbol `::` is read ”has type” and introduces a type specification. The type specification for a function is a list of the types of each

input parameter, followed by the type of the output parameter, all separated by the symbol `->`. So the function `cutAndSplice` takes an `Int`, another `Int`, something of type `([a], [a])`, and returns something of type `([a], [a])`.

But what is this cryptic-looking `([a], [a])`? It may be easiest to read this from the inside out, beginning with the symbol `a`. All types in Haskell begin with a capital letter, so `a` cannot be a type. Instead, `a` is a *type variable*, a placeholder indicating that we can use any type we wish for `a`. The notation `[a]` indicates a list[2] of `a`s, whatever `a` is. So `[a]` could be a list of `Int`s, `Char`s, or anything we wish. (However, the same type must be used everywhere `a` appears in this function's specification.)

The notation (*type1*,*type2*) indicates a *tuple*, which is a way to package several values as a single value. Unlike lists, the elements in a tuple do not generally have to be of the same type. However, in this case the tuple has two elements, both of type `[a]`, and as explained above, the same type must be supplied for all instances of `a`. So for the third parameter we could supply a value of type `([Int], [Int])`, or `([Char], [Char])`, for example, but not `([Int], [Char])`. Finally, the type of the value returned by `cutAndSplice` will be the same type as that third input parameter.

Créatúr uses `cutAndSplice` on byte strings (lists of bytes). However, there is nothing in its implementation that depends in any way on the type of the list elements. Because the specification uses a type variable rather than an explicit type, the function `cutAndSplice` is *polymorphic*[3] (can be used with any type).

[2]In other languages, the terms *array* or *sequence* might be used in place of *list*.

[3]In the author's experience, most programming languages require extra effort on the part of the programmer to create a polymorphic method. In Haskell, it is usually trivial to write a polymorphic function.

Everything after the first line is the function's implementation. The operator `++` represents list concatenation, and the Haskell library function `splitAt` splits a list at the specified position, returning the two halves. The implementation can be read as a series of mathematical definitions. Given the input parameters `n`, `m`, and `(as, bs)`,[4] the output is `(cs, ds)`, where `cs` is formed by splicing the first part of `as` with the second part of `bs`, and `ds` is formed by splicing the first part of `bs` with the second part of `as`. In other words, the result is the tuple

```
(a[0..n-1] ++ b[m..], b[0..m-1] ++ a[n..])
```

Remember, however, that functions in Haskell consist only of expressions and definitions; there is no guarantee that the computations will be performed in any particular order – except of course that if one expression depends on another, the second expression will be evaluated first. In the function `cutAndSplice`, the definitions for `as1`, `as2`, `bs1`, and `bs2` will be evaluated before `cs` and `ds`, but apart from that, the order of evaluation is not fixed.

Table 5.1 provides examples of the use of `cutAndSplice`.

Table 5.1: Examples using cutAndSplice

Expression	Result
`cutAndSplice 2 5 ("abcdef", "ABCDEF")`	`("abF","ABCDEcdef")`
`cutAndSplice 3 1 ("abcd", "ABCDEFG")`	`("abcBCDEFG","Ad")`
`cutAndSplice 4 4 ("abcdef", "ABCDEF")`	`("abcdEF","ABCDef")`

If n <= 0 or m <= 0, the corresponding list will be completely transferred to the other, effectively exchanging the position of the lists within the tuple. Table 5.2 provides examples.

[4] In Haskell, lists are often given two-letter variable names where the second letter is *s*, such as `xs`, pronounced "exes".

Table 5.2: Examples using cutAndSplice with a zero or negative index

Expression	Result
`cutAndSplice 0 4 ("abcdef", "ABCDEF")`	`("EF","ABCDabcdef")`
`cutAndSplice (-2) 4 ("abcd", "ABCDEFGH")`	`("EFGH","ABCDabcd")`
`cutAndSplice 5 0 ("abcdef", "ABCDEF")`	`("abcdeABCDEF","f")`

If n or m are greater than or equal to length of the corresponding list, that list will not be transferred. Table 5.3 provides examples.

Table 5.3: Examples using cutAndSplice with an index greater than the length of the input list

Expression	Result
`cutAndSplice 10 0 ("abcdef", "ABCDEF")`	`("abcdefABCDEF","")`
`cutAndSplice 0 0 ("", "ABCDEF")`	`("ABCDEF","")`

Now the function `randomCutAndSplice` can be defined, which invokes `cutAndSplice` with random values. However, recall that functions in Haskell are *referentially transparent*; any expression can be replaced by its value without changing the behaviour of the program. In other words, given the same inputs, a function should always return the same value. This is equivalent to saying that functions should not have side effects. If `randomCutAndSplice` behaves randomly, it will violate referential transparency.

Of course, a program completely without side effects would not be very useful (it could not perform any I/O, for example). Haskell provides mechanisms to isolate side-effects, state data, and mutable data from the purely functional parts of the program. This is accomplished through the use of *monads*,[5] which are pro-

[5]The name comes from the term monad in category theory.

gramming structures that represent computations. The main method in a Haskell program runs in the `IO` monad, which is the only context that allows I/O. In fact, the `IO` monad is a sort of "anything goes" zone where the usual constraints are relaxed.

A full explanation of monads is beyond the scope of this thesis; instead, the function `randomCutAndSplice` will be used to illustrate the use of monads.

```
randomCutAndSplice :: (RandomGen g) => ([a], [a]) -> Rand g ([a], [a])
randomCutAndSplice (as, bs) = do
    n <- getRandomR (0,length as - 1)
    m <- getRandomR (0,length bs - 1)
    return (cutAndSplice n m (as, bs))
```

The first line is the type signature for the function. As before, the lowercase letters `a` and `g` are type variables, but in this case `g` cannot be just any type; it has been constrained. The expression `(RandomGen g) =>` can be read as "for all types `g`, where `g` is an instance of `RandomGen`. And the expression `Rand g ([a], [a])` means that instead of returning a value of type `([a], [a])`, this function will return a computation that can calculate the result. That may not seem to be much of an improvement, but in fact these computations can be chained together to form more complex computations. Eventually this particular computation will be evaluated in the main method of Créatúr, in the `IO` monad.

Everything after the first line is the function's implementation. The first thing the reader will notice is that the syntax used in this function looks very different than in previous functions. In fact, it looks like imperative code! This "do notation" (note the word `do` in the function definition) allows a series of computations to be chained together. The reader might guess what this function does, i.e.,

1. Assign a random value between `0` and the length of the first input list, minus one, to `n`.

2. Assign a random value between `0` and the length of the second input list, minus one, to `m`.

3. Invoke the function `cutAndSplice` with the values `n`, `m`, and the two input lists, and return the value.

That description is essentially correct. However, the function `return` does *not* cause the function to exit. In Haskell, `return` simply takes a normal value and converts it into a computation that will return that value. A function using this "do notation" will return a computation that executes the statements inside it in sequence and returns the last expression in the sequence. In this example, the last expression does contain the word `return`, but that is not always the case.

The next function defined in this module, `crossover`, is quite simple. It is simply a convenience method that calls `cutAndSplice` using the input parameter for both indices.

```
crossover :: Int -> ([a], [a]) -> ([a], [a])
crossover n = cutAndSplice n n
```

The function `randomCrossover` is very similar to `randomCutAndSplice`, except that it uses the same random value for both indices.

```
randomCrossover :: (RandomGen g) => ([a], [a]) -> Rand g ([a], [a])
randomCrossover (as, bs) = do
    n <- getRandomR (0,length as - 1)
    return (crossover n (as, bs))
```

The use of QuickCheck was illustrated in Section 4.8, so it will suffice to mention the testable properties that were defined for this module. The first one states that the sum of the lengths of the two strings is not altered by the cut-and-splice operation.

```
prop_cutAndSplice_preserves_sum_of_lengths ::
  Int -> Int -> (String, String) -> Property
prop_cutAndSplice_preserves_sum_of_lengths n m (as, bs) =
  property $ length as' + length bs' == length as + length bs
    where (as', bs') = cutAndSplice n m (as, bs)
```

The other property states that the sum of the lengths of the two strings is not altered by the crossover operation.

```
prop_crossover_preserves_sum_of_lengths :: Int -> (String, String) -> Property
prop_crossover_preserves_sum_of_lengths n (as, bs) =
  property $ length as' + length bs' == length as + length bs
    where (as', bs') = crossover n (as, bs)
```

Finally, the `testModule` method invokes QuickCheck to run all of the tests defined for this module.

```
testModule :: IO ()
testModule = do
  putStrLn $ ">>>>> Testing " ++
    "Genetics.Crossover.prop_cutAndSplice_preserves_sum_of_lengths"
  quickCheckWith (stdArgs { maxSuccess=1000, maxDiscard=100, chatty=True})
    prop_cutAndSplice_preserves_sum_of_lengths


  putStrLn $ ">>>>> Testing " ++
```

```
    "Genetics.Crossover.prop_crossover_preserves_sum_of_lengths"
  quickCheckWith (stdArgs { maxSuccess=1000, maxDiscard=100, chatty=True})
    prop_crossover_preserves_sum_of_lengths
```

5.5 Summary

Créatúr is a reusable software framework for automating experiments with artificial lifeforms. It automates tasks which are independent of the animat species. The user provides the implementation of the animats and other objects in the virtual environment. Créatúr supports common behaviours such as eating, mating, and playing; users can implement additional behaviours as needed. Créatúr places very few constraints on the animat implementation; making this framework usable for a wide variety of ALife projects.

Créatúr animats live in a world of data, and survive by discovering patterns. The appearance of each animat and object is a string of data, so finding food or mating partners requires recognising patterns in that data. There is no fitness function except survival. Animats and objects are selected at random and allowed to interact. Créatúr animats are diploid, and are immature at birth. They remain with a parent and learn by observation until they are mature. The age of maturity is genetically determined.

As will be seen in Chapters 6 and 7, the Créatúr framework has already been used with two very different ALife species. Although this framework was designed to support some novel approaches to ALife (using data as the environment, framing data analysis as a survival problem, using multiple kinds of evolution, and protecting the young while they learn), it could support a variety of ALife projects.

By encapsulating the functionality common to most ALife habitats and species, the Créatúr framework can save development time.

Chapter 6

Dotes: an artificial lifeform

This chapter describes the first of two attempts to achieve the research objectives outlined in Chapter 3. The animats used in this experiment are called *dotes*[1]. There is one type of object for the dotes to interact with: berries, which are a potential food source defined by an RGB colour and the amount of energy that they provide when eaten. Some berries provide small negative amounts of energy, making them mildly poisonous. (However, eating poisonous berries will only be fatal if it reduces the dote's energy to zero or below.) The implementation of dotes and berries satisfies the requirements in Section B.2 with IDs beginning with "USR-".

The structure of this chapter is outlined below.

Section 6.1: The dotes describes the appearance of dotes, how they eat and lose energy, mate, rear children, think, learn, and forget. It also describes the extent to which each of these traits is genetically determined.

Section 6.2: Dote Genetics describes how dotes are constructed from their genes.

[1] *Dote* (rhymes with coat) is a Hiberno-English term of endearment usually applied to children or small animals.

Section 6.3: Implementation and testing presents excerpts from the code in order to show how Haskell and QuickCheck were used in the implementation. The excerpts presented implement the learning rules used by neurons.

Section 6.4: Experimental set-up describes the procedure used when setting up and running experiments with dotes.

Section 6.5: Results and Interpretation analyses the results obtained using dotes in the Créatúr framework.

Section 6.6: Summary summarises the key points from this chapter.

6.1 The dotes

6.1.1 Appearance

The appearance of a dote is an RGB colour (a triple containing values for red, green and blue), which is determined by the *colour gene*. When a dote encounters another, the stimuli presented to its brain include the colouring of the second dote. This information could help dotes judge how genetically similar they are, which could eventually allow them to judge how closely related they are to potential mating partners, supporting kin selection or speciation.

6.1.2 Eating and metabolism

Dotes have an energy level, e, between 0 and 1. When a dote encounters a berry it has the option to eat it or reject it. Eating results in an energy gain or loss; rejecting

a berry has no effect on energy. Once a dote's energy reaches 1 it is full; continuing to eat is not beneficial, but it is not harmful.

At regular intervals, dotes lose some energy through a *metabolism tax*, denoted $e_{metabolism}$ and given by Equation 6.1. This metabolism tax is determined by the complexity and processing requirements of the brain; this should prevent dotes from evolving excessively large, inefficient brains. If a dote's energy reaches 0, it dies and is removed from the population.

$$e_{metabolism} = e_{neuron} n_{neurons} + e_{connection} n_{connections} + e_{thinking} t_{thinking} \quad (6.1)$$

where

$e_{metabolism}$ is the metabolism tax

e_{neuron} is the energy cost per neuron

$n_{neurons}$ is the number of neurons in the dote's brain

$e_{connection}$ is the energy cost per neural connection

$n_{connections}$ is total number of neural connections in the dote's brain

$e_{thinking}$ is the energy cost per update of the brain state

$t_{thinking}$ is the number of brain updates the dote makes per decision

6.1.3 Mating

Dotes have a passion level, p, which begins at 0 and gradually rises to a maximum of 1 if the dote does not mate. When a dote encounters another dote, it has the option to try to mate with it, or to ignore it. If it chooses to mate, it loses a small amount of energy for the time investment of "flirting"; this might eventually en-

courage the animats to develop a strategy for identifying receptive mates. If both partners choose to mate, the passion level of both dotes is set to zero and a child is produced. The actual process of reproduction was described in Section 5.3.3.

6.1.4 Child rearing

Mating always results in a child. At birth, each parent donates a fraction of its current energy to the child. In addition, the dam donates a fraction of all its energy gains or losses from food to the child until the child is mature. In both cases, the fraction is specified by the *devotion gene*.

After a child is born, it remains with the dam until the child is mature. During this time, it shares in the dam's energy gains or losses from food, receives the same external stimuli as the dam, and learns from those experiences. The age of maturity is specified by the *maturation time gene*.

6.1.5 Thinking

A dote brain is an unstructured, heterogeneous, neural network. The number of neurons is determined by the *start neuron genes* and *end neuron genes*. At birth, the output level of every neuron in the brain is set to a random value.

All connections between neurons are one-way, but a pair of connections can be used to facilitate bi-directional communication between neurons. Cycles within the network are permitted. The number of neurons, and the learning rule and forgetting rule of each neuron are determined genetically, and do not change during a dote's lifetime. In addition, the genome specifies a "starter set" of connections between neurons, although these connections can be broken and new ones formed

as a dote learns from its environment.

A connection has a *weight* parameter; when a neuron updates, it sets its output to the weighted sum of its inputs (i.e., the activation is the identity function). The weight of a connection is initially set to 0.1, and is regularly adjusted according to the target neuron's learning rule. Table 6.1 identifies the neuron allocation.

Table 6.1: Neurons in the dote brain

neuron	purpose
0	output for eat/don't eat decision
1	output for mate/don't mate decision
2	output, reserved
3	output, reserved
4	output, reserved
5-12	input, object code
13-15	input, object colour
16+	hidden neurons

Every time a dote encounters something in the environment, an input vector is presented to the neural network which then runs for a number of cycles to select a response. This input vector includes the appearance of the object, and the dote's current energy and passion levels. Each element of the input vector is a value between 0 and 1. During each cycle, every neuron in the brain updates its weights based on the inputs from the neurons to which it is connected, according to a learning rule. The learning rule, forgetting rule, and operational parameters of each neuron are determined genetically. The number of cycles is specified by the *thinking time gene*.

Inspired by the theory of Neural Darwinism, neural connections are generated and pruned by an evolutionary process. After each decision made by a dote, two

neurons are chosen at random, and a connection is built from one to the other if it doesn't already exist. A connection has a *health* parameter, which represents the estimated usefulness of that connection. The health of a connection is initially set to 1, and is regularly adjusted according to the target neuron's forgetting rule. If health reaches zero, connection is pruned.

The *connection source gene* is used to pre-wire the brain of a newborn dote Connections built in this way are subject to the same forgetting rules as connections created later.

6.1.6 Learning

A *learning rule* is the means by which an artificial neuron adjusts the weights between itself and other neurons (or direct inputs). The *learning rule gene* determines the learning rule used by each individual neuron. Two forms of Hebbian Learning (as discussed in Section 2.4.1) are supported: Oja's Rule and Hebb's Rule. Because Hebb's rule is unstable, it was not used in constructing starter populations. However, it is an available allele in the dote genome, and mutation could cause it to be selected for. It is conceivable that evolution will find a use for it; therefore the option was made available.

An additional learning rule, called "No Learning", instructs the neuron not to make any weight adjustments (although it can still form new connections). This rule was developed for testing; it was not used in starter populations. However, mutation could cause it to be selected for.

6.1.7 Forgetting

After a dote "thinks" (i.e., processes sensory inputs and makes a decision), two neurons in its brain are chosen at random, and a connection is created between them. A *forgetting rule* is the mechanism by which connections that turn out to be useful are preserved, while those that turn out not to be useful are pruned. A connection is useful if the outputs of the two neurons involved are strongly correlated (either negatively or positively).

The usefulness of a connection is represented by a parameter called its *health*. Initially, all connections start with a weight of 0.1 and a health of 1. The health of a connection is updated every time it is read.

The *forgetting rule gene* determines the forgetting rule used by each individual neuron. Under the *basic forgetting rule*, a connection gains health by having a weight that is close to either 0 (indicating a strong negative correlation) or 1 (indicating a strong positive correlation). A connection loses health by being close to 0.5, because that indicates that there is little correlation between the neurons. The formula used is given by Equation 6.2.

$$h' = h + \rho\Big[4 * \Big(w - \frac{1}{2}\Big)^2 - h\Big] \tag{6.2}$$

where h is the current value of the connection health, h' is the updated value, ρ is the forgetting rate, and w is the weight associated with this connection by the target neuron.

An alternative forgetting rule, called "No Forgetting", instructs the neuron never to prune any of its connections (although it can still form new connections). This rule was developed for testing; it was not used in starter populations. How-

ever, mutation could cause it to be selected for.

6.2 Dote Genetics

Because the focus of this research was the brain, dotes have a very simple body and metabolism, and most of the dote genetic traits relate to the brain. The dote genome consists of instructions encoded as a series of bytes. The encoding scheme is explained in Appendix C. Any byte that cannot be interpreted as part of one of the other gene sequences will be treated as a no-op instruction, which has no effect. This ensures that all gene sequences are valid, and can be used to construct a dote.

6.2.1 Genetic dominance

As discussed in Section 5.3.3, dominance relationships must be defined to handle the situation when the alleles at corresponding locations in the two gene sequences differ. For most homologous gene combinations, a type of genetic blending was implemented by taking the average of the two values. The minimum of the two values for the connection source gene is used in the hope that it will result in smaller, more efficient brains. The merging of two learning rule genes depends on the rules they specify; Oja's Rule takes precedence over all rules, and "No Learning" takes precedence over Hebb's Rule. More detailed information about the dominance relationships is provided in Section C.2.

6.2.2 Dote assembly

The technique for constructing a dote from its genome is inspired more by a factory assembly line than by biology. The instructions in the blueprint are followed one by one, as described in Table 6.2. After the dote is assembled, the neurons in its brain are set to random values.

6.3 Implementation and testing

In order to give the reader an idea of how Haskell and QuickCheck were used in the dote implementation, some excerpts from the code will be presented and discussed.[2] These excerpts implement the learning rules used by neurons. The first step is to define the `LearningRate` type, which is just a synonym for `Double`. Learning rates should normally lie in the closed interval $[0, 1]$. However, this range is not enforced; evolution may find it useful to exceed these ranges in some cases.

```
type LearningRate = Double
```

As discussed in Section 6.1.6, three learning rules are currently implemented: Hebb's rule, Oja's rule, and a "No Learning" rule. (Only Oja's Rule was used in the starter population.) So there are three constructors for `LearningRule`, one for each of the rules. The constructors for Hebb's Rule and Oja's Rule both take one parameter of type `LearningRate`. The "No Learning" rule constructor does not take any parameters. The notation `deriving (Eq, Show, Read)` allows the type to inherit standard functions for equality, display, and parsing.

```
data LearningRule
```

[2] A complete code listing is available from the author.

Table 6.2: Genes interpreted as instructions for assembling a dote

Instruction	Action
devotion gene	Set the dote's devotion to any future offspring to the specified value (possibly overriding a previous setting).
maturation time gene	Set the time the dote spends with its dam to the specified value (possibly overriding a previous setting).
start neuron gene	If a neuron is currently being assembled, but has not been added to the brain, discard it. Begin a new neuron and set the learning rule to the same value as used for the previous neuron. (If this is the first neuron, use Oja's Rule with a rate of 0.1.) Set the forgetting rule to the same value as used for the previous neuron. (If this is the first neuron, use the basic forgetting rule with a rate of 0.01.)
learning rule gene	If a neuron is currently being assembled, set the learning rule for the current neuron to the specified value (possibly overriding a previous setting). Otherwise, ignore the instruction.
forgetting rule gene	If a neuron is currently being assembled, set the forgetting rule for the current neuron to the specified value (possibly overriding a previous setting). Otherwise, ignore the instruction.
connection source gene	If a neuron is currently being assembled, create a connection from the neuron at the specified index to the current neuron. Otherwise, ignore the instruction. (If the index is greater than than $n - 1$, where n is the final number of neurons in the brain, the connection is invalid and will eventually be pruned.) Only the source of the connection is specified by the gene; the neuron currently being defined is always the target.
end neuron gene	If currently building a neuron, add it to the list of finished neurons. Otherwise, ignore the instruction.
colour gene	Set the dote's colouring to the specified value (possibly overriding a previous setting).
thinking time gene	Set the time the dote spends thinking about a decision to the specified value (possibly overriding a previous setting).
no-op gene	Take no action.

```
  = Hebb LearningRate
    | Oja LearningRate
    | NoLearning deriving (Eq, Show, Read)
```

The function `updateWeight` has a different implementation for each of the learning rules. It takes a learning rule, an output signal, an input signal, and a connection weight, and returns an updated weight. When `updateWeight` is called, one of the three implementations will be chosen by performing pattern matching on the learning rule. The notation `(Hebb r)` will match a value that was created using the Hebb's Rule constructor. If it matches, the symbol `r` will be assigned the learning rate, and the expression on the right side of the equals sign will be evaluated and returned. The reader may find it helpful to compare the code with Equations 2.1 and 2.2. Note that this code uses `r` in place of η for the learning rate.

```
updateWeight
  -- | The learning rule
  :: LearningRule
  -- | The current value of the output signal
  -> Signal
  -- | The current value of the input signal
  -> Signal
  -- | The current weight
  -> Weight
  -- | The updated weight
  -> Weight
updateWeight (Hebb r) y x w = w + r * x * y
updateWeight (Oja r) y x w = w + r*y*(x - y*w)
```

```
updateWeight NoLearning _ _ w = w
```

The next set of examples illustrates one of the tests that was run on the code above. Although QuickCheck will generate random test data, there are situations where greater control over the test inputs is required. Fortunately, it is easy to define custom generators for test data. For example, as mentioned above, learning rates should normally lie in the closed interval [0, 1]. This range is not enforced, but many of the properties that we'd like to test will only work if the learning rate is in this range. The code below first defines the interval, and then creates a generator called `arbLearningRate` that will select random values in that interval using the `choose` function.

The notation `:: (LearningRate, LearningRate)` tells Haskell that we want the value `(0, 1)` to be interpreted as a tuple containing two values of type `LearningRate` as opposed to, say, two `Int`s or two `Double`s.

```
learningRateInterval = (0, 1) :: (LearningRate, LearningRate)
arbLearningRate = choose learningRateInterval
```

Similarly, a generator called `arbWeight` selects suitable test values for the connection weights.

```
type Weight = Double
weightInterval = (0,1) :: (Weight, Weight)
arbWeight = choose weightInterval
```

As mentioned in Section 2.4.1, Hebb's Rule is unstable, therefore we want to avoid using it for most tests. The following code creates a generator that only returns instances of Oja's Rule, with arbitrary learning rates produced by the `arbLearningRate` generator defined above.

```
instance Arbitrary LearningRule where
  arbitrary = do
    rate <- arbLearningRate
    return $ Oja rate
```

With those definitions, we can now verify that the implementation of `updateWeight` satisfies one of the expected properties. If the input and output values are correlated, Oja's Rule should increase the connection weight – unless it is already set to 1. That property is tested by the following code. The last line can be read as "for an arbitrary weight `w`, either the expression `updateWeight l 1 1 w` is greater than `w`, or `w` is equal to one".

```
prop_potentiation_happens11 :: LearningRule -> Property
prop_potentiation_happens11 l =
  forAll arbWeight $ \w -> updateWeight l 1 1 w > w || w == 1
```

6.4 Experimental set-up

This section describes the procedure used when setting up and running experiments with dotes.

6.4.1 Generating a starter population

For each trial, between 200 and 1000 gene sequences in the order specified in Figure 6.1 were generated. The brain size for the initial population was set to 30 neurons, based on an assumption that this would be sufficient for evolution to construct a reasonably accurate, if inefficient, neural network. The number of initial connec-

tions was set to 0 or 1, chosen at random. All other gene parameters were chosen at random from their respective domains. Each gene sequence was duplicated and a dote was constructed from the resulting pair of (identical) sequences. These dotes formed the starter population for a trial run of Créatúr.

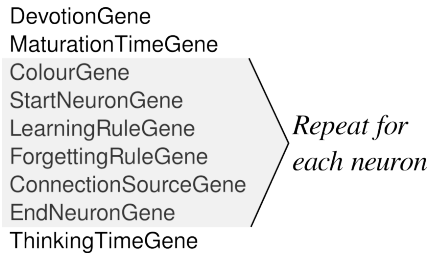


Figure 6.1: Gene sequence of starter population

As the program ran, the population size and the number of births were monitored. If it seemed unlikely that the population would become self-sustaining (i.e., that the population size would remain above the pre-set minimum), logs were analysed to determine the main factors leading to dote deaths, and a starter population that might be more viable was generated for the next trial.

6.4.2 Configuring the Ecosystem

The dote configuration file allows the user to specify the items listed below.

- the directory containing the population
- the username the daemon will run as
- the minimum population level
- the number of neurons any artificially-generated (i.e., not born) dotes will have

- the interval between metabolism tax levies, in number of Créatúr clock ticks
- the energy requirement per neuron
- the energy requirement per neural connection
- the energy requirement per brain update
- the energy gained by eating edible berries
- the energy lost by eating poisonous berries

Choosing good configuration values is not an easy task. If nutrition is too plentiful, the dotes won't be forced to learn to identify edible berries; they can compensate for the poisonous berries they eat by eating more berries overall. But if nutrition is too scarce, the dotes may die out before they can evolve the skill to identify edible berries. Once the dotes can reliably avoid poisonous berries, if the amount of nutrition from edible berries offered meets their metabolic needs, then the population will be stable. This section describes the configuration strategy that was used.

Assume that the goal is to have a stable target population of n dotes. The metabolism tax is set to be applied every n ticks of the Créatúr clock. Ideally, the dotes would eat all of the edible berries that are offered to them, and avoid all of the poisonous berries. In that situation, assuming that the population size matches the metabolism cycle, each dote will be offered, on average, one berry per metabolism cycle. However, only 50% of berries are edible. In order to balance the energy from the edible berries against the metabolism costs,

$$\bar{e}_{metabolism} \approx \frac{1}{2} e^{+}_{berry} \tag{6.3}$$

where $\boxtimes_{metabolism}$ is the average metabolism tax paid by a dote (see Equation 6.1), and e^{+}_{berry} is the energy provided by a edible berry. (In this equation, it is assumed that the dote only eat edible berries.)

The exact values are not critical; what matters is that the equation balances. However, dotes will receive a berry every other metabolic cycle *on average*; they must be able to survive several cycles without food. A dote's maximum energy is 1, so setting $\boxtimes_{metabolism} = 0.1$ is a good compromise, which would require $e^{+}_{berry} = 0.2$. This should be suitable once the dotes have adapted to their environment. In the beginning, however, they will not distinguish between the different types of berries. It is necessary to give them extra energy so that the ones that are slightly better at the task can live long enough to reproduce. A value of $e^{+}_{berry} = 0.4$ ensures that some members of the starter population survive long enough to have offspring and rear them to maturity.

Returning to the metabolism tax, we have from 6.1,

$$0.1 = e_{neuron} n_{neurons} + e_{connection} n_{connections} + e_{thinking} t_{thinking} \tag{6.4}$$

The first goal was to get a viable population that had adapted to the berry task. Based on the allocation in Table 6.1, it was estimated that evolution would be able to construct a reasonably accurate, if inefficient, neural network using 30 neurons, with approximately 50 connections, and 25 updates. Somewhat arbitrarily, the values listed below were chosen for most of the trials.

$e_{neuron} = 0.000033$

$e_{connection} = 0.0000033$

$e_{thinking}$ = 0.00034

Given the metabolic "budget" of 0.1, this would allow for a generous 100 neurons, 500 connections, and 255 update cycles. All of the configuration parameters were chosen with the intent to make it easy for the first generation to survive and produce offspring. Once that was achieved, the Créatúr daemon was halted, the configuration changed to make the environment slightly more challenging, and then the daemon was restarted using the same population. As each trial ran, the population was monitored for signs that the dotes were learning to distinguish between poisonous and non-poisonous berries.

6.5 Results and Interpretation

This section summarises the results obtained for dotes running in Créatúr, and then analyses the data from the last trial.

6.5.1 Summary of results from early trials

In an early trial, the maturation time gene was set to a random value in the range 0 to 25,535. The logs showed that offspring remained with their parents so long that they were an excessive drain on the parent's resources, resulting in the death of both parent and child. In subsequent trials, the range of the age of maturity in the starter population was capped at 200 to reduce this problem. This only directly affected the starter population; it would still have been possible for evolution to eventually drive the value above 200.

A trial with a single dote demonstrated that it could learn a training signal

with supervised training, as shown in Figure 6.2. A simple signal representing a ripe berry was presented to the neural net, and the network was allowed to update. The process was repeated 10 times. The colour indicates the output level of each neuron, at each time step. At time zero, the neurons have a wide range of output levels because they were initialised to random values. The neuron representing the eating decision has a low output initially (indicating "no") but eventually produces a high output (indicating "yes").

Dotes ate edible berries and poisonous berries in similar proportions; this indicates that they had not learned to distinguish between the two types. Figure 6.3 shows the results from the final run; earlier runs produced similar results. A series of changes was made to the configuration settings and the genetic make-up of starter population. Dotes continued to eat edible and poisonous berries in similar proportions; indicating that they had not learned the difference.

Making the environment harsher (by making the poisonous berries more poisonous) did not trigger berry discrimination. Instead, the population fell below the pre-set minimum (at which point new random dotes are generated and added to the population) and showed no sign of recovery. Figure 6.4 shows the results from the final run; earlier runs produced similar results.

Because the neural nets that form the dotes' brains are unstructured, many had connections that terminated at input neurons. As a result, the inputs would often be modified as the dote "thought". This behaviour can be seen in Figure 6.2. Originally it was hoped hoped that evolution would either select against this type of configuration, or find a way to make use of it. Perhaps it would have happened eventually, but since no signs of learning were observed, the brain was modified so it would rewrite the input signal at every update. A population with this new

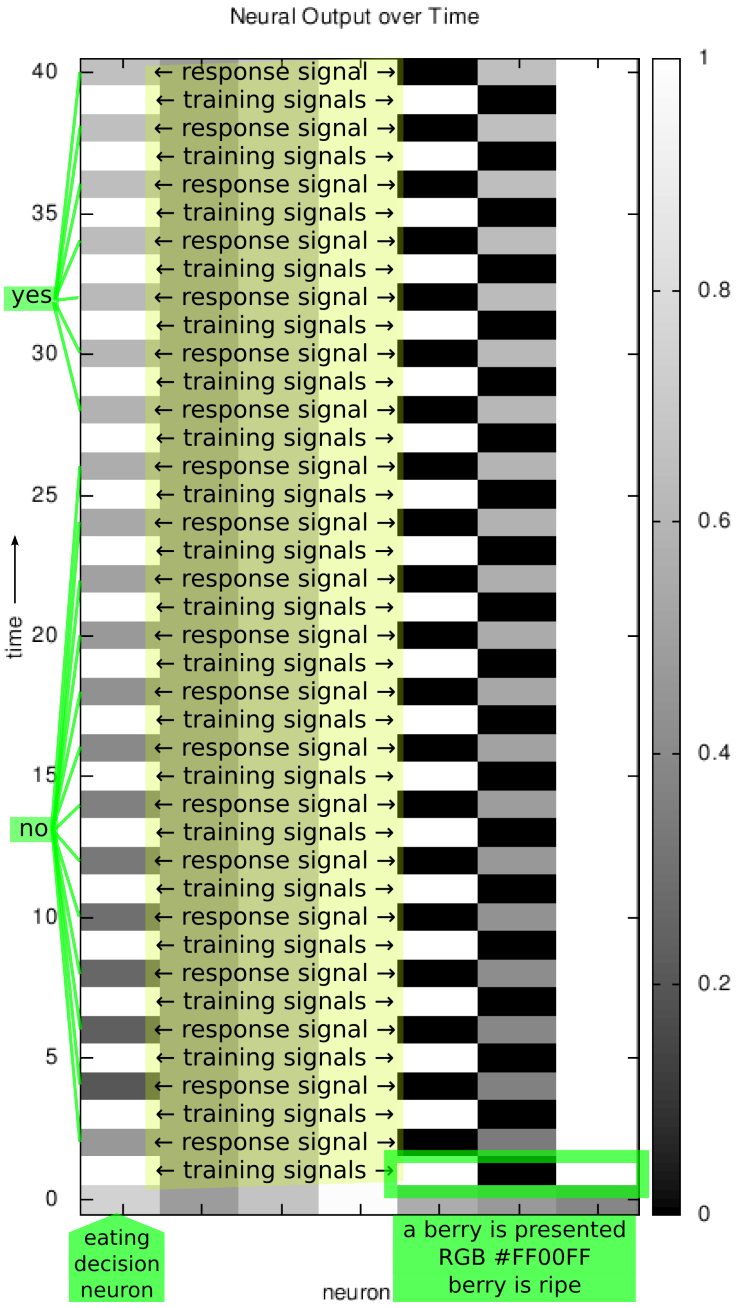


Figure 6.2: Supervised training of a dote's brain

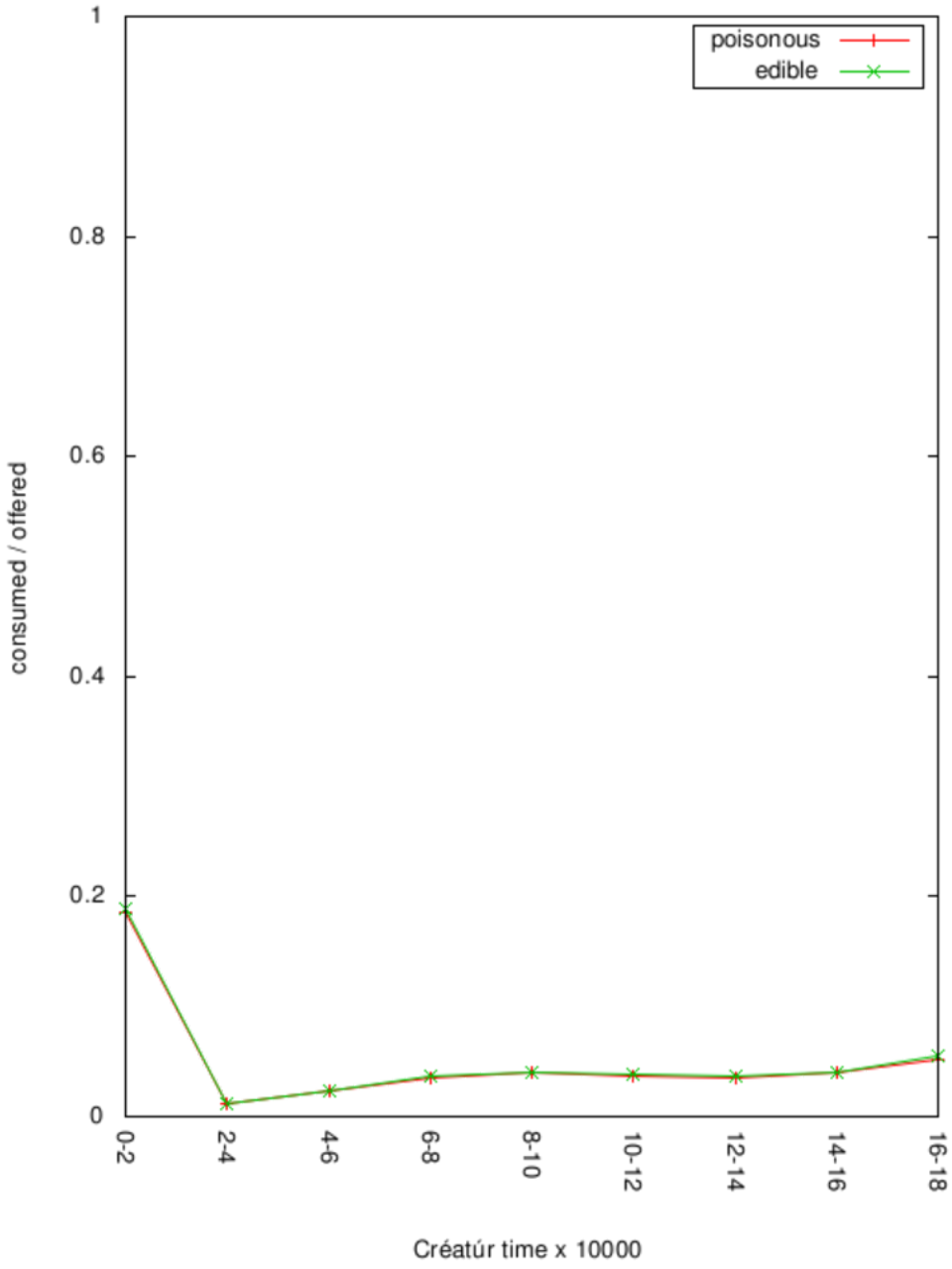


Figure 6.3: Dote eating patterns over time

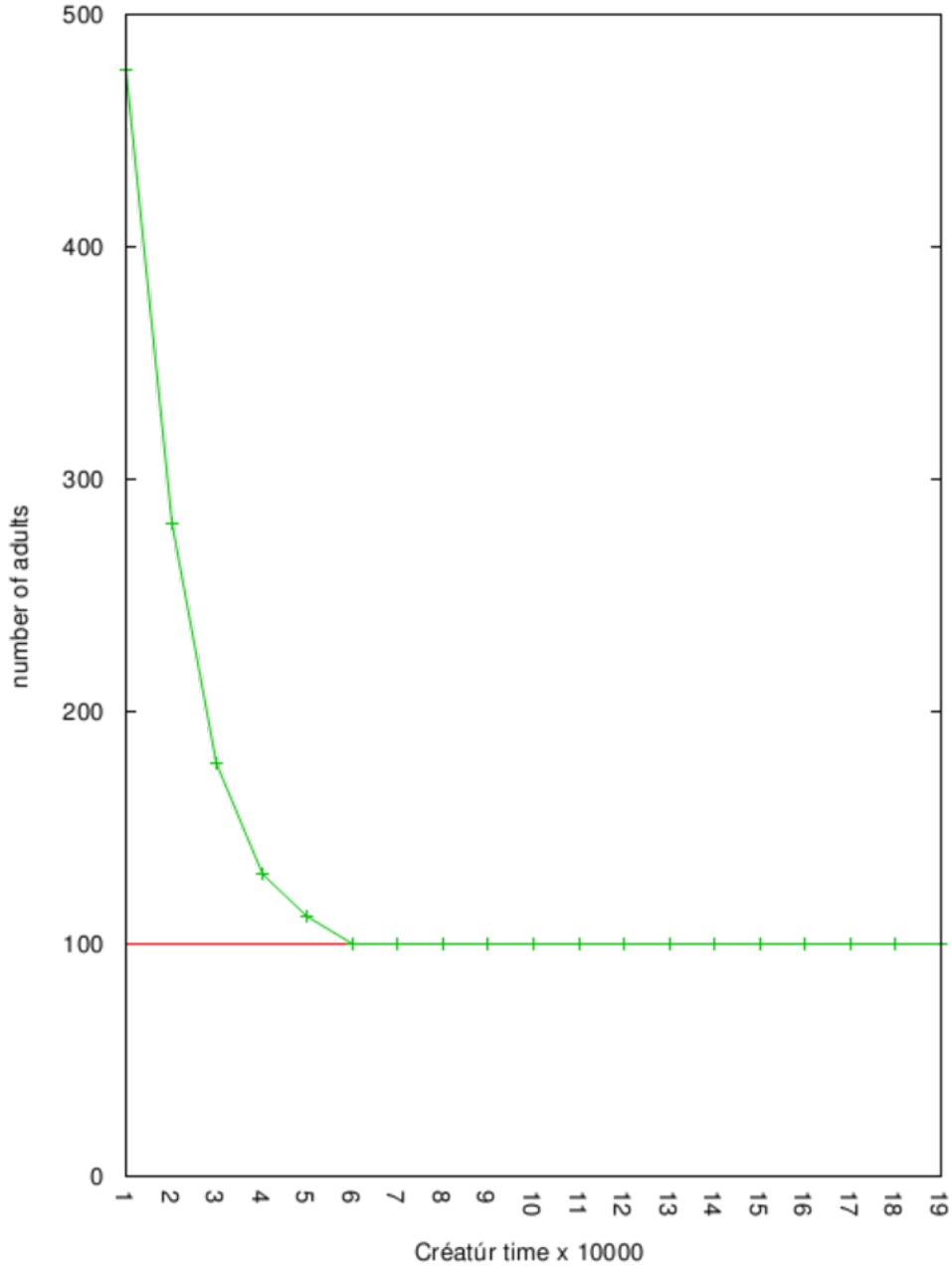


Figure 6.4: Dote population as a function of time. The red line indicates the minimum population level. If the population falls below this level, new dotes are randomly generated and added to the population.

feature was evolved, but it did not demonstrate any ability to distinguish between edible and poisonous berries.

Initially, the neural nets that formed the dotes' brains received no training. It was hoped that evolution would wire some sort of "reflection" circuit into the brain, so that the dotes would learn from their mistakes. Whether or not this would have occurred eventually, the analysis showed no sign that any of the dotes were learning. Therefore, a limited type of training was implemented. After a dote eats something, it should know afterwards if that was a good decision or bad decision (based on whether it gained or lost energy). The dote implementation was modified so that after eating, the neural net would be trained for a few cycles with that information. No training occurred if the dote chose not to eat (because it would be biologically unrealistic for the dote to know the outcome of an action not taken). A population with this new feature was evolved, but the dotes continued to eat edible and poisonous berries in similar proportions, indicating that they did not learn to distinguish between them.

6.5.2 Results from final trial

This was the last trial performed with dotes. The version of the software used for this trial incorporated all of the changes described in Section 6.5.1. The set-up for this trial is shown in Table 6.3.

Figure 6.3 shows the eating patterns of the dotes in the final trial. The results are typical of those obtained in trials in which an attempt was made to force the dotes to learn by imposing an energy penalty for eating a poisonous berry. Half of the berries offered to the dotes were edible, half poisonous. The dotes showed

Table 6.3: Set-up for final trial with dotes

item	value
initial population	500
minimum population	100
numNeurons	30
metabolic cycle	1000 ticks
e_{neuron}	3.3e-4
$e_{connection}$	3.3e-6
$e_{thinking}$	3.4e-4
e^{+}_{berry}	0.4
e^{-}_{berry}	-0.1

no preference in their eating patterns, indicating that they did not learn to distinguish between edible and poisonous berries. Furthermore, even though the penalty was low compared to the potential energy reward from an edible berry, the dotes tended to avoid eating altogether. During the time shown, only one child survived to maturity.

Figure 6.4 shows that the population did not become stable. Instead, it plummeted rapidly, and showed no signs of recovery.

The elders

At the time the experiment was stopped, five dotes from the starter population of 500 were still alive: 86 pi, 97 pi, 308 pi, 391 pi, 455 pi. (The surname "pi" indicates the name of the population.) For convenience, these dotes are called the "elders". How did they remain alive so long? Table 6.4 compares some statistics about the elders to the population averages, providing clues to the answer.

Table 6.4: Elder dotes

	86 pi	97 pi	308 pi	391 pi	455 pi	avg	pop. avg.
Age	197	197	197	197	197	197	44.62
Energy	0.3714	0.9	0.1084	0.7	0.1747	0.4509	0.5356
Time since last mated	3399	3411	3403	3315	3423	3390.2	841.11
Thinking time	0	0	4	0	3	1.4	75.46
Neurons	30	30	30	30	30	30	30
Connections	896	898	899	896	896	897	451.51
Metabolism	0.003947	0.003950	0.005317	0.003947	0.004967	0.004425	0.02817

On average the dotes in the population had mated approximately 841.11 clock ticks ago, whereas the elders last mated approximately 3390.2, a four-fold increase. This might seem to suggest that the elders mate far less often, and therefore spend less time rearing children and sharing food with them. However, the "time since last mating" counter is zero when a dote is born or randomly generated. The age of the elders was 197 clock ticks, over four times the average age of a dote (44.62). Most dotes in the population were too young to have much of a mating history, so there is no evidence that the elders actually mate less often, or that this contributes to their longevity.

The row labelled "thinking time" indicates the number of brain updates that the dotes perform when making a decision; it provides a better clue to the longevity of the elders. The average dote performs 75.46 brain updates for each decision, but elders 86, 97, and 391 perform no brain updates at all, and elders 308 and 455 only perform 4 and 3 updates, respectively. The elders had functioning brains; they simply weren't using them. Substituting the values from Table 6.3 into Equation

6.1, we have

$$e_{metabolism} = (0.00033)n_{neurons} + (0.0000033)n_{connections} + (0.00034)t_{thinking} \tag{6.5}$$

Since the metabolic cost is largely based on the number of brain update cycles, the elders' metabolic costs are extremely low. As can be seen in the last row of Table 6.4), the metabolic cost for elders is 0.004425 on average, (approximately one-sixth of the population average of 0.02817). Since each dote begins life with an energy of 1, we would expect the elders to live a fairly long life *even without eating*, as shown in Equation 6.6. Indeed, an analysis of the logs showed that elders 86, 308, and 455 never eat.

$$lifespan = \frac{1}{0.004425} = 225\,ticks \tag{6.6}$$

By contrast, elders 97 and 455 eat every berry offered. As shown in Table 6.3, edible berries provided 0.4 units of nutrition, while poisonous berries provided -0.1 units. Edible and poisonous berries were available in equal amounts, resulting in an average energy gain of 0.15 units per berry. This explains why elders 97 and 455 have much higher energy levels than elders 86, 308, and 455.

6.6 Summary

The key points about the dote implementation are summarised in this section, with references to the section in which the topic was discussed.

The appearance of a dote is an RGB colour, which is genetically determined

(6.1.1). The objects in the dote universe include berries and other dotes. Berries are edible or poisonous according to their RGB colour (6.1.2). When a dote encounters an object, the appearance of that object is presented to the dote's senses, along with information about the dote's current state (6.1.5). It can then choose to eat (6.1.2) or attempt to mate with (6.1.3) the object. After a child is born, it remains with the dam until it is mature (6.1.4).

A dote brain is an unstructured, heterogeneous, neural network (6.1.5). The number of neurons, their learning rules, and their forgetting rules are all genetically determined. The hope was that evolution would design a suitable brain from the components provided. Dotes did learn to mate successfully and eat regularly, but have been unable to raise children to maturity (6.5.2). However, in the experiments performed thus far, dotes have not learned to distinguish between edible and poisonous berries.

The first objective was to produce a stable population of dotes with the ability to identify patterns in data. The partly random nature of evolution makes it impossible to know in advance how long it will take to find an acceptable solution, or if it will ever succeed under the conditions provided. Extremely slow progress can be indistinguishable from lack of progress. However, it did not seem likely that dotes would achieve this objective in the time allotted for this research.

After reviewing the results, the author concluded that it would be more practical to create a functional brain, and let evolution improve on it. Therefore, the second attempt to achieve the research objectives outlined in Chapter 3 used a new ALife species was designed, with a new brain. This second attempt is described in Chapter 7.

Chapter 7

Wains: an artificial lifeform

This chapter describes the second of two attempts to achieve the research objectives outlined in Chapter 3. This attempt featured a new ALife species, with a new brain. There are two types of objects in the universe, listed below.

- Wains, an artificial life form.
- Numerals, a potential food source or toy which consists of a 28x28 grey-scale image of a handwritten numeral. Each numeral has different characteristics and uses.

The implementation of wains and numerals satisfies the requirements with IDs beginning with "USR-", in Section B.2. For the convenience of a reader who may be more interested in this implementation, no knowledge from the previous chapter is assumed. As a result, there is some repetition of information. Those who have read the previous chapter will find Table 7.1 useful; it summarises the key differences between *dotes*, the animats used in the previous experiment, and

wains, the animats used in this experiment.[1]

Table 7.1: Key differences between dotes and wains

dotes	wains
Dotes can eat, mate with, or ignore things in their environment.	Wains can eat, mate with, play with, or ignore things in their environment.
The food source is berries.	The food source is handwritten numerals.
When a dote encounters an object, it receives sensory information about the class of the object (berry or dote) in addition to the object's appearance.	The only sensory information that the wain receives is the object's appearance.
The appearance of a dote or a berry is an RGB triple.	The appearance of a wain or a numeral is a 28x28 grey-scale image.
Dotes have two internal senses: energy level and time since last mated.	Wains have three internal senses: hunger level, boredom level, and passion level.
The brain has an ad hoc structure, designed over generations by evolution. Also, during one individual dote's lifetime, connections are created and pruned by an evolutionary process.	The brain design is fixed, but some parameters are evolve over generations. Also, during one individual wain's lifetime, patterns are created and pruned by an evolutionary process.

The structure of this chapter is outlined below.

Chapter 7.1: The wain describes the appearance of wains, how they eat and lose energy, mate, rear children, play, learn, and make decisions. It also describes the extent to which each of these traits is genetically determined.

[1] *Wain* (rhymes with mean or rain) is a word for "child", commonly used in Donegal and Northern Ireland.

Chapter 7.2: Wain Genetics describes how wains are constructed from their genes.

Chapter 7.3: Implementation and testing presents excerpts from the code in order to show how Haskell and QuickCheck were used in the implementation. The excerpts presented implement part of the brain.

Chapter 7.4: Experimental set-up describes the procedure used when setting up and running experiments with wains.

Chapter 7.5: Results and Interpretation analyses the results obtained using wains in the Créatúr framework.

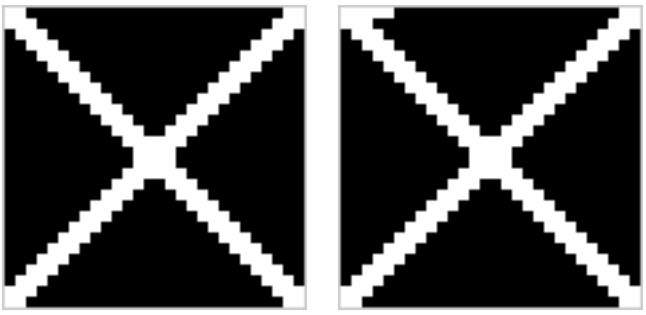
Chapter 7.6: Summary summarises the key points from this chapter.

7.1 The wain

7.1.1 Appearance

The appearance of a wain is an 28x28 grey-scale image; the value of each pixel is specified by an *appearance gene*. The appearance of the wains in the starter population is shown in Figure 7.1a; this shape was designed to be easy for the wains to distinguish from numerals. Wains learn to identify others of their species by their appearance. (To clarify, they only learn to distinguish between objects that are suitable for mating with, and objects that are not.) Because a wain's appearance is genetically determined, wains could eventually learn to judge how closely related they are to potential mating partners. If the wain is currently raising a child, a small temporary modification is made to the wain's image, as shown in

Figure 7.1b. This could eventually help wains identify mates that are likely to be receptive.



(a) Default (b) While rearing a child

Figure 7.1: Appearance of wains in the initial population. Note the additional white pixels in the upper left of (b).

Over time, mutation will cause the appearance of some wains to differ from that of the initial population. The appearance of wains will provide a rough guide of the genetic variance in the population. If eventually they were to diverge into separate species, wains would be able to distinguish between their own kind and the "other" in the same way they distinguish between numerals. For example, the new species might look like an X with an added curlicue, or with an incomplete "arm".

7.1.2 Eating and metabolism

Wains have an energy level, e, between 0 and 1. In some contexts the hunger level, h, where $h = 1 - e$, is more convenient to use. Some numerals are edible; eating them provides energy to a wain. Others are mildly poisonous; eating them decreases a wain's energy. (However, eating poisonous food will only be fatal if it reduces the wain's energy to zero or below.) If a wain rejects a numeral, it will

not receive the energy gain (or loss). Since a numeral can be both a food item and a toy, as described in Section 7.1.5, the wain's choice of action should depend on both how hungry it is and how bored it is.

Once a wain's energy reaches 1 it is full; continuing to eat is not beneficial, but it is not harmful. At regular intervals, wains lose some energy through a *metabolism tax*, denoted $e_{metabolism}$ and given by Equation 7.1. This metabolism tax is determined by the complexity and processing requirements of the brain; this prevents wains from evolving excessively large, inefficient brains. If a wain's energy reaches 0, it dies and is removed from the population.

$$e_{metabolism} = e_{iq}(n_{ex} + 10\, n_{pat} + n_{int}) \tag{7.1}$$

where

$e_{metabolism}$ is the metabolism tax

e_{iq} is a multiplier relating the brain complexity to its metabolic costs

n_{ex} is the number of external inputs to the wain's brain

n_{pat} is the maximum number of patterns the wain's brain can differentiate

n_{int} is the number of internal inputs to the wain's brain

7.1.3 Mating

Wains have a passion level, p, between 0 and 1. When a wain encounters another wain, if it chooses to mate, it loses a small amount of energy for the time investment of "flirting". If both partners choose to mate, and the wain that is randomly selected to be the dam is not currently rearing a child, the passion level of both wains is set

to zero and a child is produced. The actual process of reproduction was described in Section 5.3.3.

7.1.4 Child rearing

Mating always results in a child. When two adults mate, each donates a fraction of its current energy to the resulting child. In addition, the dam donates a fraction of all the food it eats to the child until the child is mature. In both cases, the fraction is specified by the *devotion gene*.

After a child is born, it remains with the dam until it is mature. The age of maturity is specified by the *maturation time gene*. During this time, it shares in the dam's food. It also builds a set of patterns based on its experiences, as will be described in 7.1.7. However, it does not make decisions, or learn from mistakes, until it is mature.

7.1.5 Play

Wains have a boredom level, b, between 0 and 1. A wain's boredom level is increased slightly, by a user-configurable amount, at the same time the metabolism tax is applied. Some numerals are fun; playing with them reduces the wain's boredom. Others are boring; they either increase the wain's boredom or have no effect. Wains are configured to find each other slightly boring so that mating will be a more attractive option than playing. Ignoring a numeral or another wain has no effect on boredom. Since a numeral can be both an edible food and a toy, the wain's choice of action should depend on both how hungry it is and how bored it is.

Once a wain's boredom reaches 0, continuing to play is not beneficial, but it is not harmful. A bored wain will not experience any ill-effects; the option to play with objects was merely introduced to give the wains a richer life, which might drive evolution to produce better brains.

7.1.6 Brain Structure

The brain structure for wains was designed in advance; evolution was allowed to fine-tune the parameters and discover the factors that should influence a wain's decisions. The brain consists of two parts, a classifier and a decider, as illustrated in Figure 7.2. It receives the external inputs to the wain's senses, and categorises the input vector as belonging to one of the patterns that the wain knows. The decider chooses the course of action based on the pattern ID, along with the wain's internal status. These components are described below in more detail.

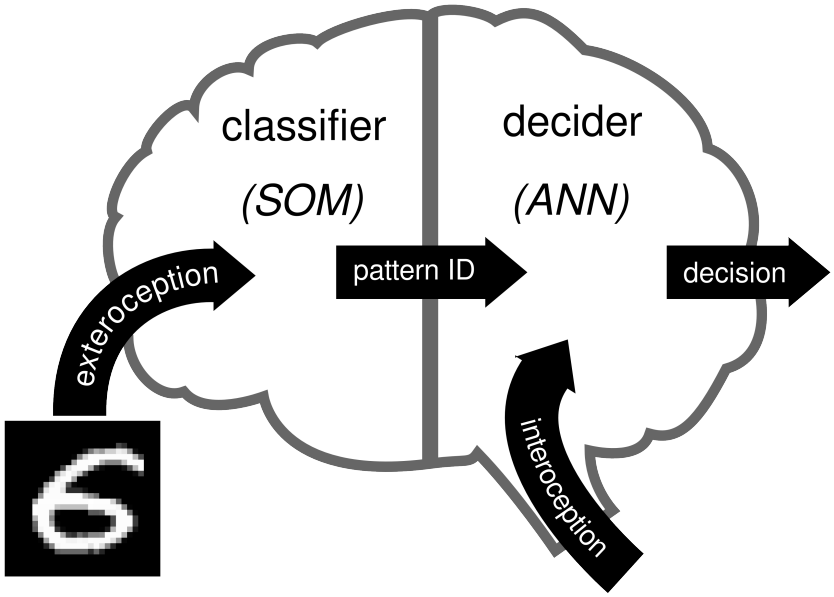


Figure 7.2: A schematic diagram of a wain brain.

7.1.7 Learning patterns

A wain may receive tens of thousands of unique input vectors during its lifetime; it must be able to discover and recognise patterns in order to develop general guidelines for making decisions. This is the job of the classifier, which is a modified Kohonen SOM (Self-Organising Map). Two modifications were made to the SOM. One was to periodically erase the least useful node so that it can learn a new pattern, thereby implementing a type of Neural Darwinism; this will be discussed shortly. The other modification sacrifices the topology-preserving feature of the SOM to allow faster processing. In a traditional SOM (described in Section 2.4.2), once a winning node has been selected, its weights, and those of its neighbours, are updated. In the modified SOM used in wains, only the winning node is updated. This allows for faster processing; however the resulting map does not preserve the topology of the input data. In this case, the goal is for wains to discover patterns in handwritten numerals, not to decide whether a handwritten two is more similar to a nine or a three. The modified SOM was designed to be as simple as possible, while still performing the task of identifying patterns in data.

The brain receives two types of inputs: internal and external. The number of external inputs is specified by the *exteroception capacity gene*. When deciding how to react to its environment, a wain must also take into account its own status. This internal data includes factors such as the wain's current hunger, passion, and boredom levels. The number of internal inputs is specified by the *interoception capacity gene*.

Although the classifier is working with images of handwritten numerals and other wains, there is no requirement that objects be classified into exactly 11 pat-

terns (10 for the digits from 0 to 9 and 1 for wains). The number of patterns that a wain can recognise is specified by the *pattern capacity gene*. The goal is for the animats to discover and remember patterns that are useful to their survival, not to implement a numeral recognition system. For example, a wain might identify two separate patterns for the numeral 2; one written with a loop (see Figure 7.3a), and one without (see Figure 7.3b).

(a) With loop (b) Plain

Figure 7.3: Samples of different styles of the numeral 2, from the MNIST database.

Evolution is free to create wains with as many patterns as desired. However, the metabolism tax is partly based on the number of patterns; this should prevent the evolution of excessively large, inefficient brains. Each wain encounters numerals in a different order, so one wain might identify all handwritten 3s as pattern #7, while another wain identifies them as pattern #2.

When a wain is born, it has no experience of the world, and therefore it must be able to learn patterns very quickly. The patterns that a wain recognises in its environment are fluid categories which shift or become broader or narrower according to the inputs that the wain receives during its lifetime. The rate at which the SOM modifies each pattern node is specified by the *pattern learning rate gene*.

Later in life, the wain has a workable set of patterns that have helped it to

survive. It must continue to learn, but it would not be practical to make dramatic changes to its view of the world based on one experience. For this reason, the pattern learning rate should decay over time. This rate of decay is specified by the *pattern learning rate decay gene*.

The number of patterns that a wain can remember is limited. If a pattern turns out not to be useful, it may be better to forget it and start fresh. The number of matches for each pattern is tracked. At intervals of specified by the *Edelman cycle gene*,[2] the least useful pattern (the one that has the fewest matches) is discarded, and the node is randomised so that it can learn a new pattern. This creates competition between patterns, thus implementing a type of Neural Darwinism. The number of matches for all patterns is then cleared, so that the new pattern has a fair chance to compete for survival.

7.1.8 Making decisions

The core of the decider is a basic Hebbian neural network. Each row in the weight matrix represents an action (eat, mate, play, ignore) and each column represents one of the pattern IDs produced by the classifier. The input vector consists of the following information, which is called the *context*.

- the wain's current hunger level, a number between 0 and 1
- the wain's current boredom level, a number between 0 and 1
- the wain's current passion level, a number between 0 and 1

[2]Named for Gerald Edelman, who proposed the theory of Neural Darwinism.

- a list of values, where each element is zero except the element whose index corresponds to the pattern ID

The values of the four output neurons represents a proportional vote for each of the potential actions (eat, mate, play, ignore). The potential actions and their respective votes form a weighted list; the final decision is chosen by weighted random selection. This element of randomness ensures that the wains will occasionally take risks. An action that had a bad outcome under one set of circumstances may have a good outcome in a different situation.

There is one case where the brain makes a decision without consulting the decider. If the wain's energy is below 0.1, the brain will choose to eat any object that the wain encounters.

7.1.9 Learning to make better decisions

If a wain chooses an action other than "ignore", it should know afterwards if that was a good decision or bad decision. To support this, the wain's happiness is measured as a function of its energy, passion and boredom, as shown in Equation 7.2. The parameters 7 and 3 were chosen after some experimentation, but they clearly reflect the order of priorities: eating, mating, then playing.

$$happiness = e * 7 + (1 - p) * 3 + (1 - b) \tag{7.2}$$

The wain's happiness is calculated before a decision is made, and again after the action is taken. If happiness increased, the decider's neural network is trained with the context vector, using a positive learning rate. Otherwise, a negative learning rate is used for training. However, a wain should not permanently alter its be-

haviour based on one experience; it may have misidentified the pattern, or the result might be different under other conditions. The rate at which the decider reinforces decisions with positive outcomes is specified by the *positive learning rate gene*. The rate at which it reinforces decisions with negative outcomes is specified by the *negative learning rate gene*.

Wains need to be able to respond to changes in their environment. To allow this, all of the weights in the neural network decay over time. The rate at which the weights decay is specified by the *decider forgetting rate gene*.

Because the wain can never know whether or not ignoring an object was a good decision, the values in the "ignore" row in the weight matrix are always zero. This row is not currently required; it is reserved for future use.

7.2 Wain Genetics

Since the focus of this research was the brain, wains have a very simple body and metabolism, and most of the genetic traits relate to the brain. The differences between the brain design of dotes and wains requires significant differences in the genome. The wain genome consists of instructions encoded as a series of bytes. The encoding scheme is explained in Appendix D. Any byte that cannot be interpreted as part of one of the other gene sequences will be treated as a no-op instruction, which has no effect. This ensures that all gene sequences are valid, and can be used to construct a wain.

7.2.1 Genetic dominance

As discussed in Section 5.3.3, dominance relationships must be defined to handle the situation when the alleles at corresponding locations in the two gene sequences differ. For most homologous gene combinations, a type of genetic blending was implemented by taking the average of the two values. The minimum of the two values for the maturation time gene was used in the hope that the wains would reproduce faster and evolve more quickly. For the exteroception capacity gene, interoception capacity gene, and pattern capacity gene, the minimum of the two values was used in the hope that it would result in smaller, more efficient brains. More detailed information about the dominance relationships is provided in Section D.2.

7.2.2 Wain assembly

The way in which a wain is constructed from its genome is inspired more by a factory assembly line than by biology. The instructions in the blueprint are followed one by one, as described in Table 7.2.

7.3 Implementation and testing

In order to give the reader an idea of how Haskell and QuickCheck were used in the dote implementation, some excerpts from the code will be presented and discussed.[3] These excerpts implement the modified SOM described in Section 7.1.7.

The first step is to define a SOM node. Each node has a vector `weights` representing a model of the input data. It also has a unique identifier, `index`. In a

[3] A complete code listing is available from the author.

Table 7.2: Genes interpreted as instructions for assembling a wain

Instruction	Action
devotion gene	Set the wain's devotion to any future offspring to the specified value (may override a previous setting).
maturation time gene	Set the time the wain spends with its dam to the specified value (may override a previous setting).
exteroception capacity gene	Set the number of external sensory inputs to the specified value (may override a previous setting).
interoception capacity gene	Set the number of internal sensory inputs to the specified value (may override a previous setting).
pattern capacity gene	Set maximum number of patterns the classifier can learn to the specified value (may override a previous setting).
pattern learning rate gene	Set the pattern learning rate to the specified value (may override a previous setting).
pattern learning rate decay gene	Set rate of decay of the pattern learning rate to the specified value (may override a previous setting).
Edelman cycle gene	Set cycle time for pruning patterns to the specified value (may override a previous setting).
positive decider learning rate gene	Set the learning rate for decisions with good outcomes to the specified value (may override a previous setting).
negative decider learning rate gene	Set the "unlearning" rate for decisions with bad outcomes to the specified value (may override a previous setting).
decider forgetting rate gene	Set rate of decay for weights in the decider's neural net to the specified value (may override a previous setting).
appearance gene	Add a pixel to wain's appearance with the specified value.
no-op gene	Take no action.

typical SOM, each node would be identified by its co-ordinates within the grid; this would be used to preserve the topology of the input data. In the modified SOM used in wains, topology is not preserved, so all that is required is that the identifier be unique. The node has an additional field that would not be required in a typical SOM: `score`. This field tracks the number of times that this node's pattern has been matched, which is a measure of how useful the node is. Periodically, the least useful node is erased. This allows the node to learn a new pattern, one that may be more useful.

```
data Node = Node
  {
    weights :: [Double],
    index :: Int,
    score :: Int
  } deriving (Show, Read, Eq)
```

The function `buildNode` creates a new SOM node. When a node is created, the weight vector is typically set to small random values. The expression `ws <- getRandomRs (0.0, 5.0)` creates an infinite list of random values between 0 and 5, bound to `ws`. How can the computer hold an infinite list? To be more precise, `ws` is a *thunk*, a promise to evaluate as much of the list as we require. In the next line, the expression `(take n ws)` takes the first `n` elements of `ws`, where `n` is the desired length of the pattern that this node will learn. Thus, only `n` values from the infinite list are calculated. The use of thunks allows Haskell to work with infinite lists very efficiently.

```
buildNode :: (RandomGen g) => Int -> Int -> Rand g Node
buildNode n i = do
```

```
  ws <- getRandomRs (0.0, 5.0)
  return $ Node (take n ws) i 0
```

The function `trainNode` is called once the winning node, or best matching unit (BMU), is known. If the node is the BMU, it calls the `adjustWeights` function to train it.

```
trainNode :: Double -> [Double] -> Int -> Node -> Node
trainNode learningRate xs bmu n =
  if index n == bmu
     then incScore $ adjustWeights learningRate xs n
     else n
```

The `adjustWeights` function adjust the node's weight vector to make it slightly more similar to the input pattern. First, it zero-pads the input vector (in case the input vector is shorter than the weight vector), and binds the result to `xsSafe`. Effectively `xsSafe` is an infinite list, with the input data followed by a series of zeroes. However, only a finite number of those values will ever be evaluated. Next, it computes the vector difference between the node's weight vector and `xsSafe`, binding the result to `diffs`. (It is at this point that the required number of elements in `xsSafe` are evaluated, after which the thunk is no longer needed.) The vector difference is multiplied by the `learningRate` to produce a list of `deltas`, which are used to adjust the node's weight vector.

```
adjustWeights :: Double -> [Double] -> Node -> Node
adjustWeights learningRate xs n = n { weights=ws' }
  where ws = weights n
        xsSafe = xs ++ repeat 0
```

```
        diffs = zipWith (-) xsSafe ws
        deltas = map (learningRate *) diffs
        ws' = zipWith (+) ws deltas
```

The function `incScore` is called when a node matches a pattern.

```
incScore :: Node -> Node
incScore n = n { score=score n + 1 }
```

The function `forgetByScore` is called periodically. If the input node is the one with the lowest score, indicating that the pattern learned by the node hasn't been very useful, the node is erased so that it can learn a new pattern. All nodes have their scores reset to zero at this time, so that the new node has a fair chance to compete in the next round.

```
forgetByScore :: (RandomGen g) => Int -> Node -> Rand g Node
forgetByScore s n =
  if score n <= s
     then buildNode k i -- replace this pattern with an empty one
     else return $ n {score = 0 } -- put all patterns on an equal footing
       where k = (length . weights) n
             i = index n
```

QuickCheck will generate random test data for simple values, but for more complex data structures, a custom generator is required. `sizedArbNode` is a generator that produces nodes in an arbitrary state. The parameter `n` is an indication of how complex of a value is desired. QuickCheck begins with simple values, and then moves on to more complex values, until a test fails or the desired number of tests has been run. If a test fails, QuickCheck then tries to recreate the failure with

the simplest possible value (`n` = 0); this behaviour is extremely valuable in isolating faults.

But what are the qualities that make a value simple or complex? That depends on the nature of the data, but QuickCheck has some very useful defaults. By default, the simplest number is zero. Lists of numbers are made simpler by replacing their elements with zeroes, or by truncating the list.

By using the parameter `n` somewhere in the generator, the programmer can define simplicity to mean whatever is most useful for testing the application.[4] In `sizedArbNode`, a vector of `n` random values is used for the node weights. If a test fails, QuickCheck will try to find a node with the shortest possible weight vector, containing as many zeroes as possible.

```
sizedArbNode :: Int -> Gen Node
sizedArbNode n = do
  ws <- vector n
  i <- arbitrary
  s <- arbitrary
  return $ Node ws i s
```

`Arbitrary` is a typeclass. A Haskell typeclass is similar to a Java interface; any types belonging to a typeclass must implement the functions defined by the typeclass. Alternatively, it can use default function implementations, if the typeclass provides them. To make it easier to use the type `Node` with QuickCheck, it is declared to be an instance of `Arbitrary`, using the `sizedArbNode` generator defined above.

[4]The programmer could also create multiple generators, each using a different concept of simplicity, as appropriate for diagnosing different kinds of faults.

```
instance Arbitrary Node where
  arbitrary = sized sizedArbNode
```

Here is an example of a property that was tested using QuickCheck: training a node should never change the length of its weight vector.

```
propAdjustWeightsDoesntChangeLength
  :: Double -> [Double] -> Node -> Property
propAdjustWeightsDoesntChangeLength lr xs n =
  property $ (length . weights) n' == (length . weights) n
    where n' = adjustWeights lr xs n
```

7.4 Experimental set-up

This section describes the procedure for setting up and running experiments with wains.

7.4.1 Generating a starter population

For each trial, a small number (from 200 to 1000) of gene sequences was generated using the order and parameters specified in Table 7.3. (The same parameters were used for all trials.) Each gene sequence was duplicated and a wain was constructed from the resulting pair of (identical) sequences. These wains formed the starter population for a trial run of Créatúr.

7.4.2 Configuring the Ecosystem

The wain configuration file allows the user to specify the items listed below.

Table 7.3: Gene sequence for the starter population

Instruction	Parameter
devotion gene	A random integer between 0 and 255.
maturation time gene	A random integer between 100 and 500. This range was chosen based on the experience with wains.
exteroception capacity gene	784, allowing input of 28x28 grey-scale images
interoception capacity gene	3 (for energy, passion, boredom).
pattern capacity gene	A random integer between 11 and 25. This range was chosen after experimenting with a stand-alone implementation of the modified SOM.
pattern learning rate gene	A random integer between 90 and 110, which encodes for a learning rate between 0.90 and 1.10. This range was chosen after experimenting with a stand-alone implementation of the modified SOM.
pattern learning rate decay gene	A random integer between 230 and 255, which encodes for a decay rate between 0.90 and 1.00. This range was chosen after experimenting with a stand-alone implementation of the modified SOM.
Edelman cycle gene	A random integer between 50 and 100000. This range was chosen arbitrarily.
positive learning rate gene	A random integer between 10 and 30, which encodes a learning rate between 1 and 3. This range was chosen after experimenting with a standalone Hebbian neural net.
negative learning rate gene	A random integer between 10 and 30, which encodes a learning rate between 1 and 3. This range was chosen after experimenting with a standalone Hebbian neural net.
decider forgetting rate gene	A random integer between 230 and 255, which encodes a forgetting rate between 0.90 and 1.00. This range was chosen after experimenting with a standalone Hebbian neural net.
appearance gene	A sequence of 784 genes, encoding the image shown in Figure 7.1

- the directory containing the population
- the username the daemon will run as
- the directory containing the MNIST images
- the minimum population level
- the interval between metabolism tax levies, in number of Créatúr clock ticks
- e_{iq}, the multiplier relating the brain complexity to its metabolic costs
- the energy provided by each numeral, if eaten
- the boredom relief provided by each numeral, if played with
- $p_{crossover}$, the probability that crossover resulting in equal lengths will occur when gametes are created
- $p_{cut-and-splice}$, the probability that crossover resulting in non-equal lengths will occur when gametes are created
- $p_{mutation}$, the probability that mutation will occur when gametes are created

Based on the experience with dotes, values were already known that were likely to work well to ensure that the wains had sufficient food available. At the start of early wain trials, food provided abundant energy. The logs were monitored to verify that the wains were learning to distinguish between the different kinds of objects in their environment, and make appropriate decisions. A little at a time, the amount of energy provided by food was reduced, to attempt to drive the population to make even smarter decisions. After each adjustment, the population size was monitored to ensure that the population was still stable.

7.5 Results and Interpretation

This section presents the results obtained using wains. The data presented is from the final trial. The population used in this trial was named *Ardara* The set-up for this trial is shown in Tables 7.4 and 7.5. The values chosen for $p_{crossover}$, $p_{cut-and-splice}$, and $p_{mutation}$ are not biologically realistic, but were chosen so that evolution might be observed during the time allotted for this research.

Table 7.4: Set-up for final wain trial

item	value
initial population	100
minimum population	50
metabolic cycle	1000 ticks
e_{iq}	0.000033
$p_{crossover}$	0.1
$p_{cut-and-splice}$	0.01
$p_{mutation}$	0.001

7.5.1 Population stability

As shown in Figure 7.4, the population was self-sustaining. After the starter population was created, no wains were added except through birth. At Créatúr clock times 744138, 752352, 791676, and 806168, the poisonous numerals (7, 8 and 9) were made successively more poisonous, to see how the wains would cope. (Table 7.5 shows the initial and final values for these numerals.) As shown in Figure 7.5, the first three changes seemed to have no effect, but after the last change, a dip was observed in the population size, followed by a recovery. The poison levels remained

Table 7.5: Numeral characteristics for final wain trial

object	energy	boredom relief	comment
0	1.0	0	edible, boring
1	0.8	-0.1	edible, fun
2	0.6	-0.2	edible, fun
3	0.3	-0.3	edible, fun
4	0.2	-0.6	edible, fun
5	0.1	-0.8	edible, fun
6	0	-1.0	edible, fun
7	-0.05 (initial) -0.1 (final)	-0.2	poisonous, fun
8	-0.08 (initial) -0.2 (final)	0	poisonous, boring
9	-0.11(initial) -0.3 (final)	0	poisonous, boring
wain	-0.05	0.1	poisonous, boring

at the new (higher) settings, but the wains had adapted. Section 7.5.5 will provide some insight into how they adapted.

7.5.2 Eating patterns

Figure 7.6 shows how the first generation of wains successfully learned to distinguish between edible and non-edible foods. The time period shown is from the first generation, so this graph reflects learning during a single lifetime. During this period, each wain encountered 52 examples of each handwritten numeral, on average. Some of the wains in the initial population died quickly. If these wains were less adept than average at distinguishing numerals, after their death the accuracy of the population as a whole would rise, even if the remaining individuals made no further improvement. To account for this effect, data excluding the wains that died during the period shown is also plotted; these are the lines labelled "filtered".

Figure 7.7 shows the eating pattern over a longer period of time. Only wains aged between 1100 and 1200 Créatúr clock ticks are included. The age range was restricted so that the data would not be affected by changing demographics in the population (e.g., a sudden influx of young wains reaching maturity at the same time.) The time span shown includes 12 generations of wains. As can be seen from the graph, overall, the wains tend to eat edible numerals and avoid the poisonous ones. In particular, they avoid trying to eat each other. If a wain does try to consume another wain, the first wain loses some energy; this mimics the cost of the fight that would likely ensue in the biological world.

Eating a poisonous numeral can be classified as an mistake on the part of a wain, but *not* eating an edible numeral is not necessarily a mistake. If a wain is

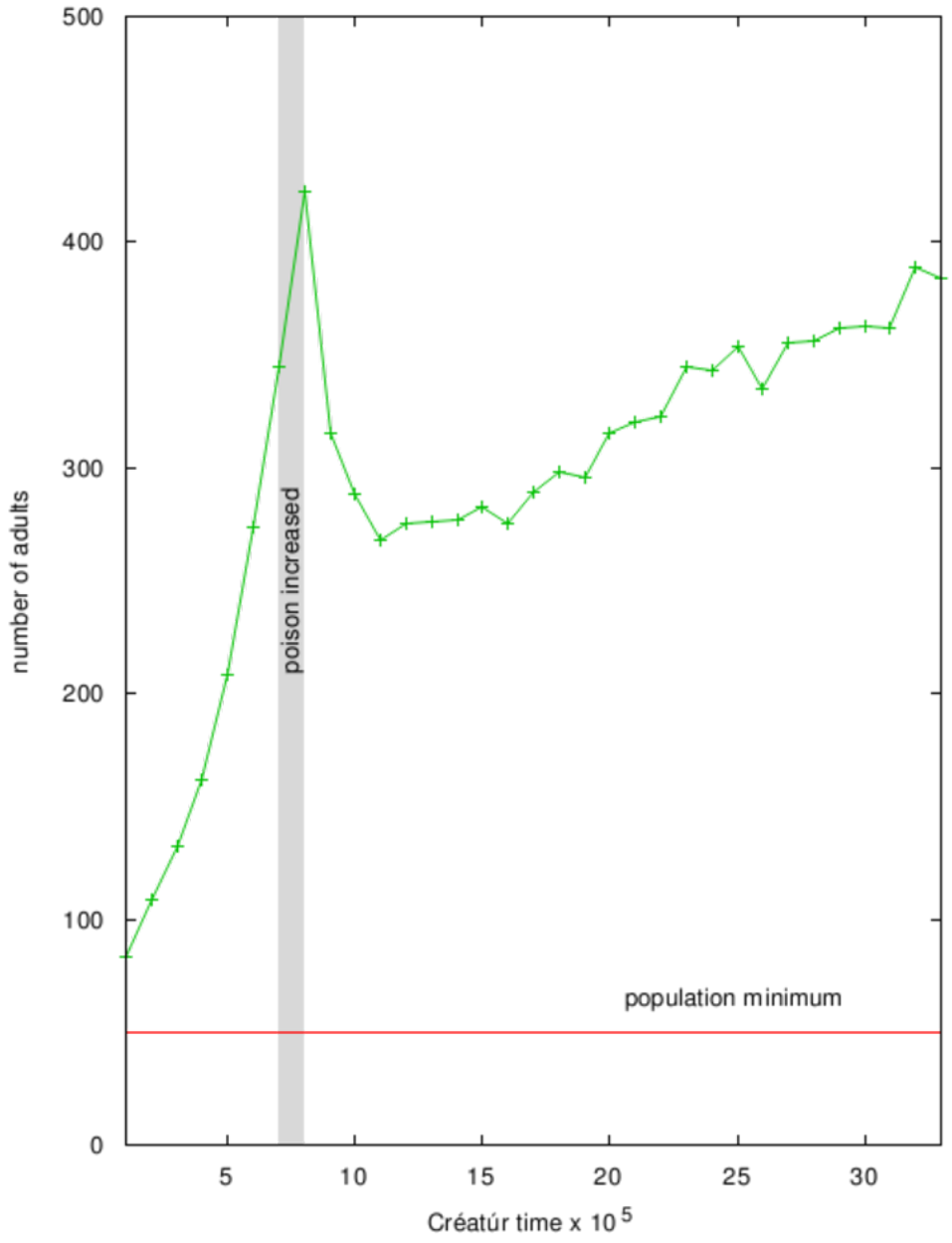


Figure 7.4: Wain population growth. The time span shown includes 12 generations of wains. The grey vertical band indicates a series of adjustments to the toxicity of poisonous numerals.

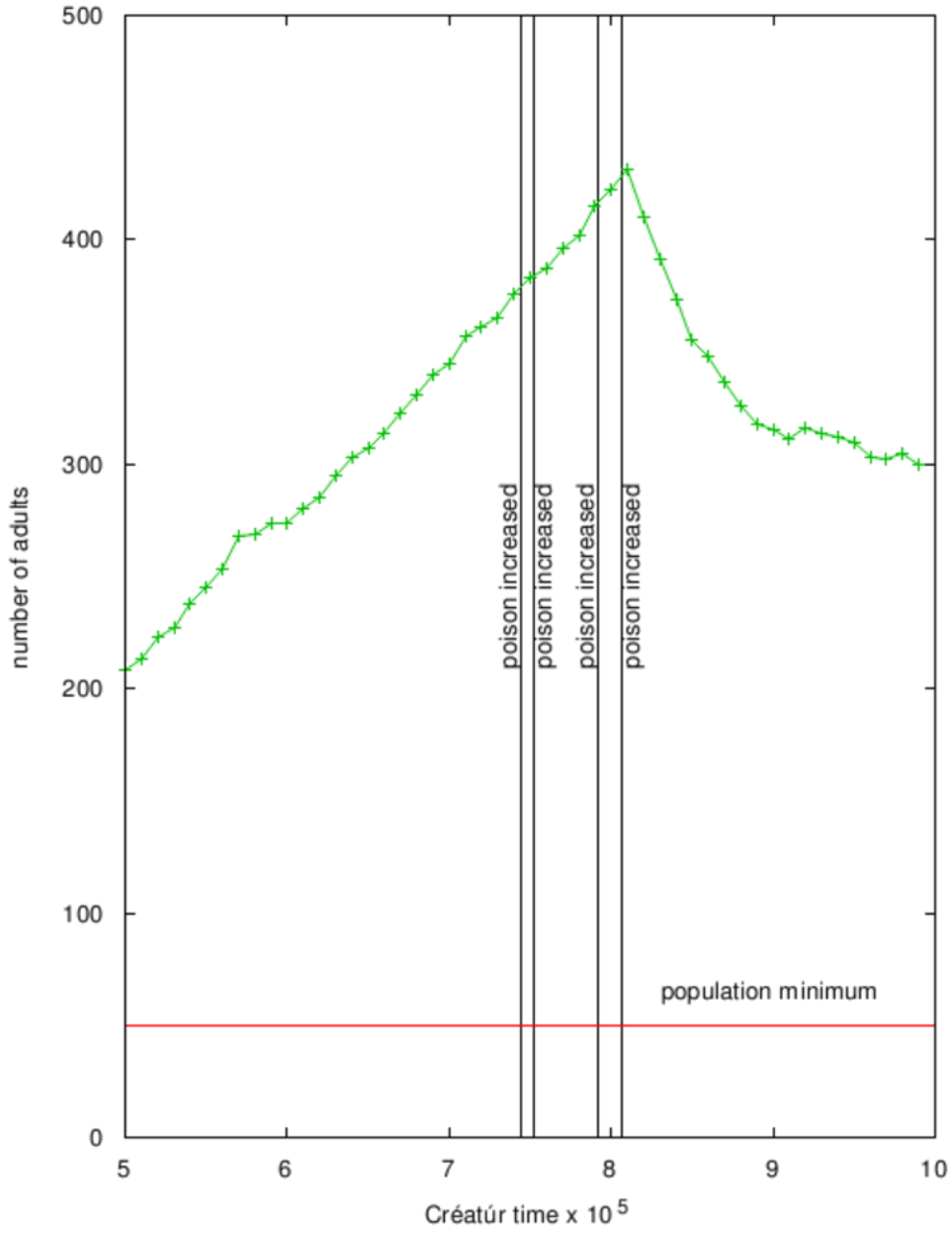


Figure 7.5: Wain population changes in response to a harsher environment. The four vertical lines indicate adjustments to the toxicity of poisonous numerals.

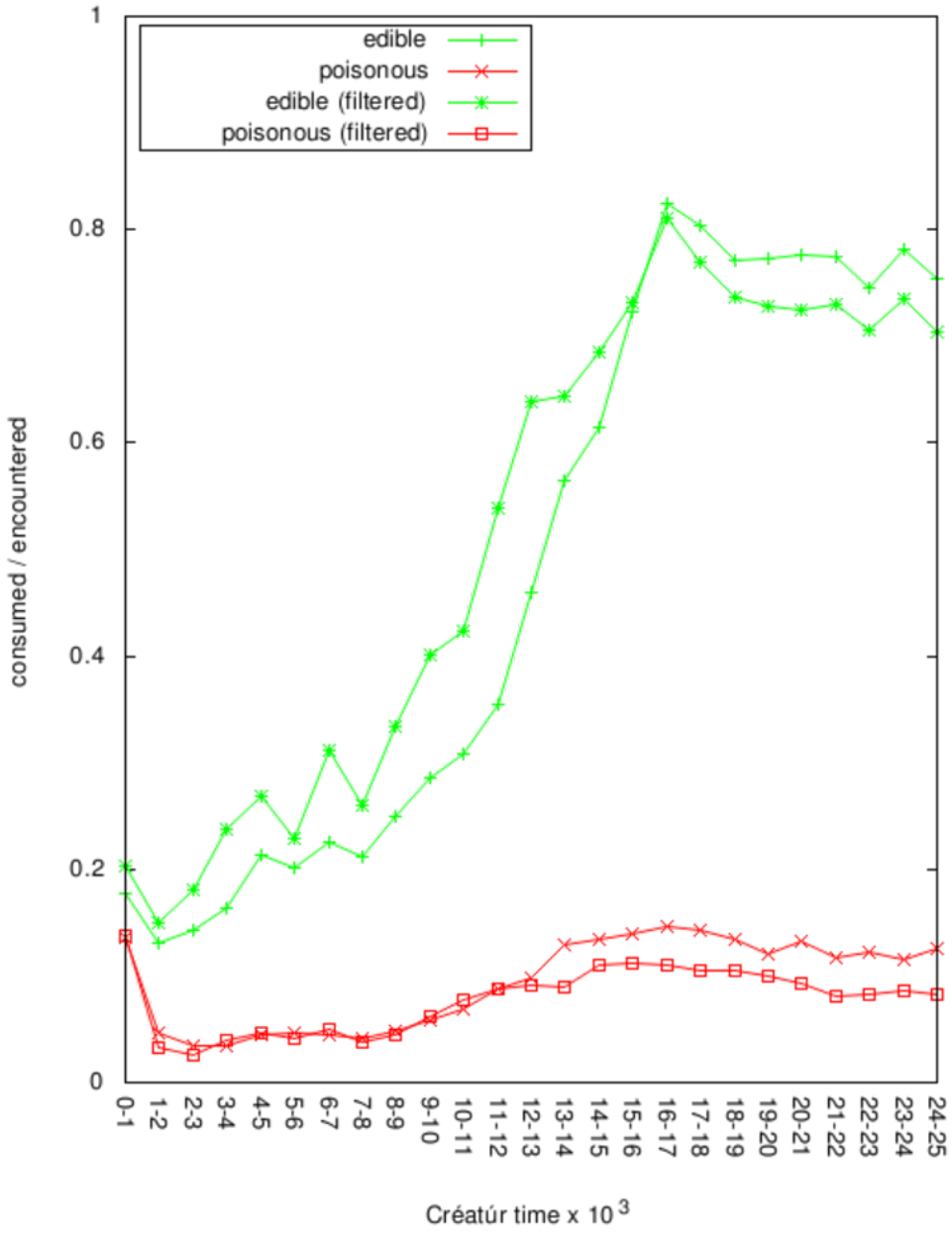


Figure 7.6: First-generation wain eating patterns. Graph shows the fraction of encounters where the wains decided to eat (or try to eat) the object. Green indicates edible objects; red indicates poisonous objects.

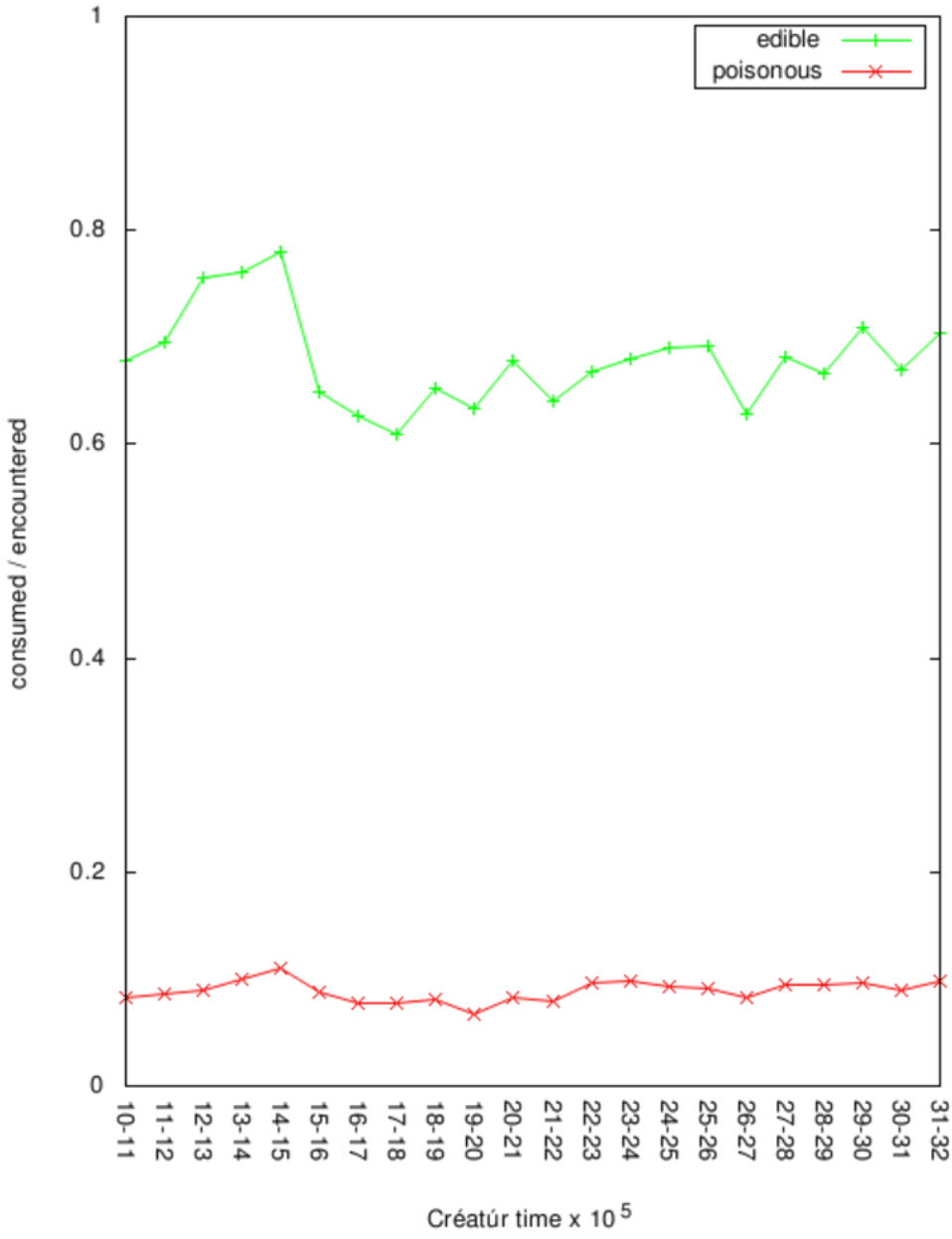


Figure 7.7: Wain eating patterns. Graph shows the fraction of encounters where the wains decided to eat (or try to eat) the object. Green indicates edible objects; red indicates poisonous objects.

not hungry, eating will not make it happier, so the numeral may have more value as a toy (by reducing the wain's boredom level, increasing its happiness). Or, if the numeral is boring (i.e., playing with it will not reduce boredom), then ignoring it is just as sensible as eating it.

However, wains do make mistakes, as can be seen from the number of poisonous numerals eaten. There are four reasons why a wain would eat a poisonous numeral. In no particular order, they are:

1. Inexperience: The wain has not yet learned which numerals are poisonous.

2. Misidentification: One of the wain's patterns matches two or more numerals, at least one of which is edible.

3. Taking risks: As discussed in Section 7.1.8, the element of randomness ensures that the wains will occasionally choose a decision that is unlikely to have a good outcome.

4. Starvation: As discussed in Section 7.1.8, if the wain's energy is below 0.1, the brain will choose to eat any object that the wain encounters.

Figure 7.8 shows that misidentification plays a significant role. Only wains aged between 1100 and 1200 Créatúr clock ticks are included. By the time a wain reaches 1100, it has encountered thousands of numerals,[5] so inexperience is not a factor. As shown this graph, handwritten 8s, which are poisonous, are eaten far more often than other poisonous numerals. One reason for this may be confusion

[5]At each tick of the Créatúr clock, a numeral is offered to one of the wains in the population. Every 1000 ticks, the metabolism process executes, and all creatures age by one unit. As a rough estimate, we can assume that there are 250 wains in the population, so out of the 1000 numerals offered in a cycle, a particular wain would receive approximately 4.

between 8s and 3s, which are edible. Similarly, handwritten 4s, which are edible, are eaten far less often than other edible numerals. Confusion between 4s and 9s, which are poisonous, may account for this.

There is no direct way to find out which numeral a wain thinks a particular image is. The question is meaningless, because the wain has no concept of numerals. All it has are patterns, and those patterns are in no particular order. If a wain chooses to eat an image containing a handwritten numeral, it is because that image is a reasonable match for one of the wain's patterns, and in the wain's experience, images that match that pattern that are usually edible.

However, there is an indirect way to gain some insight into the mistakes that wains make. Figure 7.9 shows the SOM of a young adult wain. Each node in the SOM has a vector of 784 weights; these have been converted into a 28x28 matrix of greyscale values, allowing us to see the patterns that this wain has identified. To clarify, the wains do not actually receive sensory inputs in two dimensions. The appearance of objects in their environment is presented to the SOM as a one-dimensional vector, so they are unaware that pixel 1 and pixel 29 are adjacent because they are in the same column in subsequent rows. In any case, the figure does show that some of the patterns identified by this wain are ambiguous, and that confusion of 3s with 8s, and 4s with 9s, is quite plausible.

7.5.3 Mating patterns

As can be seen in Figure 7.10, wains quickly learned to prefer other wains as potential mating partners, rather than numerals. Only wains aged between 1100 and 1200 Créatúr clock ticks are included in the graph. So far, no cause for the

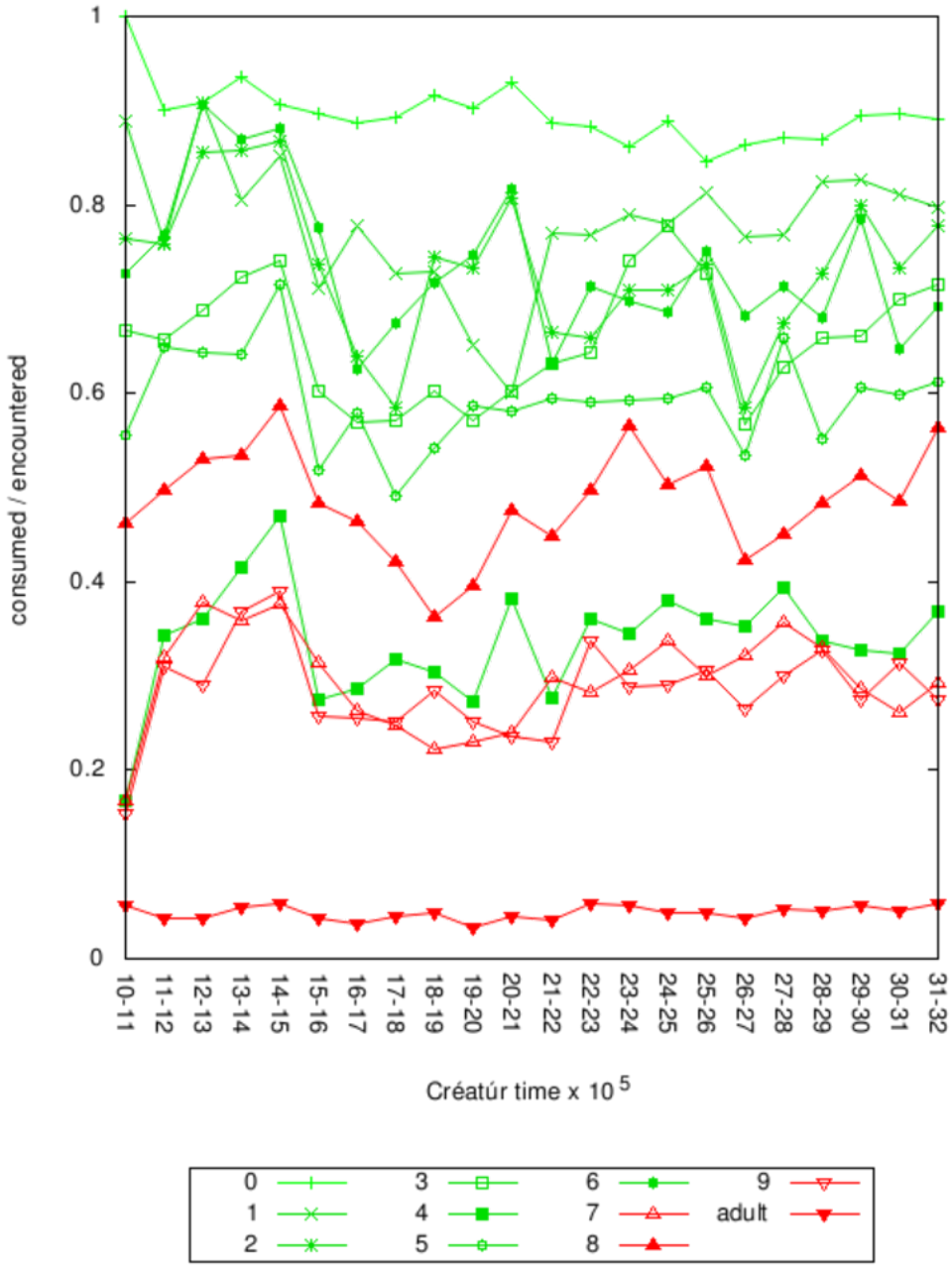


Figure 7.8: Detailed wain eating patterns. Graph shows the fraction of encounters where the wains decided to eat (or try to eat) the object. Green indicates edible objects; red indicates poisonous objects.

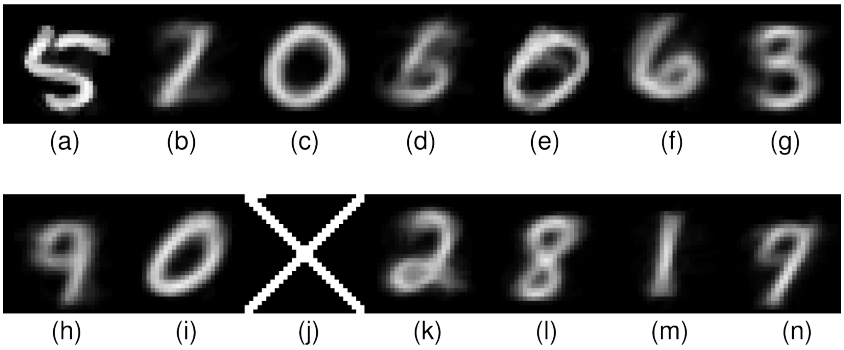


Figure 7.9: A typical SOM. This the SOM of a young adult wain. Node (j) is a clear match for X, the appearance of a wain. Most of the numerals are well-defined. As is typical, that there are two nodes that might match a 2: node (k) would match a 2 with a loop, while node (b) would match a plain 2 (or a heavily slanted 1). There is no clear representation of the numeral 4; nodes (h) and (n) are probably the best match for a 4, but they are even better matches for a 9. Node (d) might match a 5 or a 6.

temporary increase in flirtations with objects at Créatúr time 20-2100000 has been determined. Recall from Section 7.1.3 that when a wain encounters another wain, if it chooses to mate, it loses a small amount of energy for the time investment of "flirting". If a flirtation is unsuccessful, the unlucky suitor ends up with less energy and no reduction in passion, and therefore has a lower happiness level. It appears that wains have learned to flirt when reproduction is likely, as demonstrated by the fact that they generally flirt in less than one-third of encounters, yet the population is thriving. There are three reasons why a wain's flirtation would be unsuccessful; these are listed below.

- The other wain has chosen not to mate.
- The other wain has been selected to be the dam, and it is currently rearing a child.
- This wain has been selected to be the dam, and it is currently rearing a child.

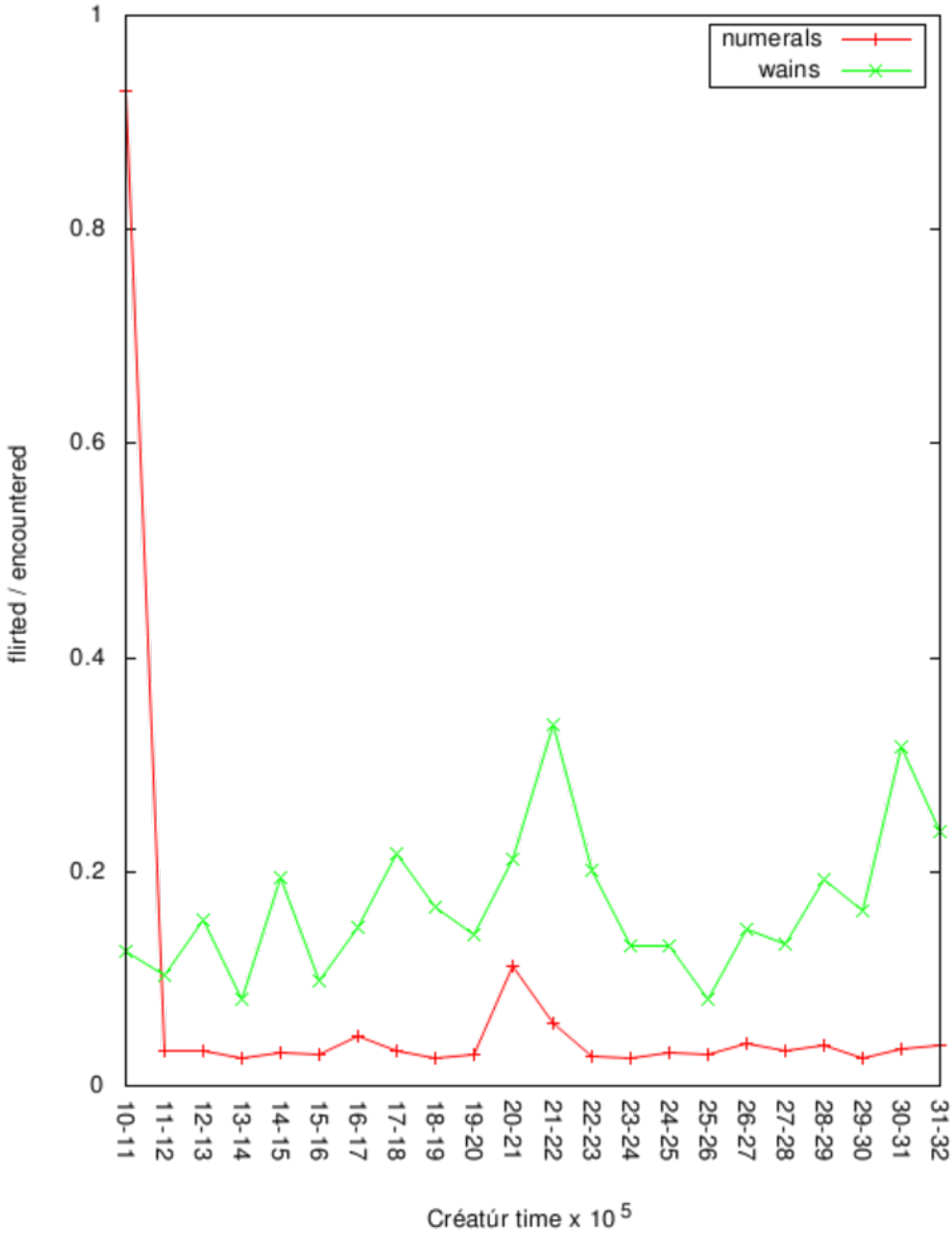


Figure 7.10: Wain flirting patterns. Graph shows the fraction of encounters where the wain decided to mate (or try to mate) with the object.

A wain currently has no way to know another wain's passion level; if it did, it might be able to anticipate the other wain's decision, which would help it avoid situation (1). A wain's appearance indicates if it is raising a child. Wains do see each other's appearance, so they could use that information to avoid situation (2). In order to do that, they would need to have two separate SOM patterns, one for a wain with a child, and one without. However, in the 500+ brain scans performed to date on wains, the only instances where there are multiple X-like patterns in the SOM would differentiate between mutant wains and normal wains, as will be discussed in Section 7.5.6.

It was intended that wains would have a sensory input indicating if it was currently raising a child, but this was inadvertently omitted from the wain implementation. However, a wain does know its current passion level. Since the passion level is reset to zero after mating, it could be used as a very rough measure of the likelihood that this wain is currently raising a child. It is a rough measure because a low passion level only indicates that the wain has mated recently; it does not indicate whether the wain was the sire or dam in the last mating. Flirting only when its passion level is high would help a wain avoid situation (3).

But is this strategy being used? Figure 7.11 shows the values for one element of the decision weight matrix: the element that relates the wain's passion level to the likelihood that the wain will flirt, given an opportunity. As explained in Section 7.1.8, the weight matrix is multiplied with the vector of sensory inputs to create a weighted list of possible actions. The action is then chosen by random weighted selection. However, all actions have a minimum weight of 1, to ensure that wains take risks occasionally. Therefore, if a weight is negative, the action may still be selected, but only rarely.

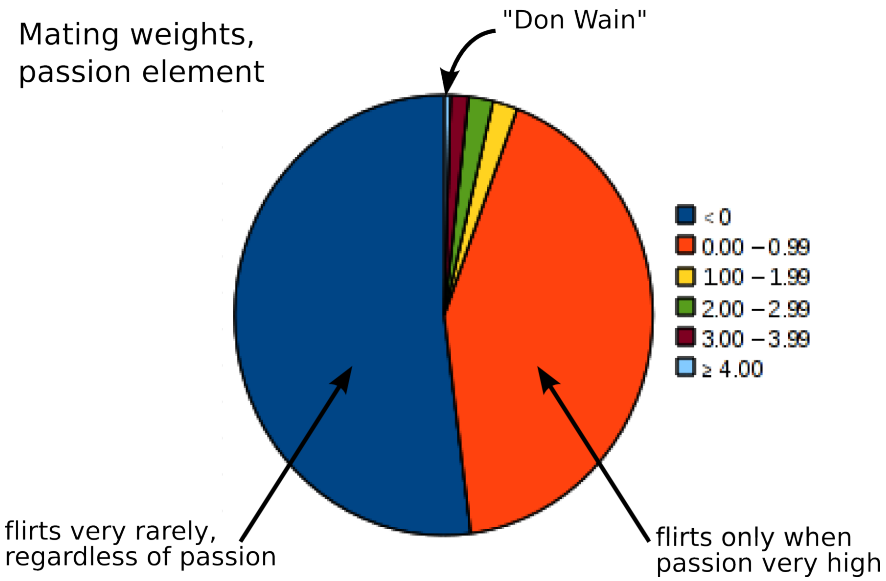


Figure 7.11: Wain mating weights. Graph shows the values for one element of the decision weight matrix; the element that correlates the wain's passion level with the likelihood that it will flirt, given an opportunity. The statistics are taken from the wains that were alive at time 3308000.

Over half of wains leave the decision whether to flirt or not entirely up to chance, by having a negative weight at this location in the weight matrix. Somewhat less than half of the population do seem to follow the strategy outlined above; i.e., they are unlikely to flirt unless their passion levels are high. There are a few wains in the population that have a very high(>4) weight, which means that even a low passion level makes them very likely to flirt. They have been given the nickname "Don Wains" as a nod to the legendary lover, Don Juan.

7.5.4 Play patterns

A wain never suffers any ill-consequences from playing with a numeral, and only a minor increase in boredom for playing with another wain. At best, the object will lower its boredom levels. At worst, the wain loses out on a potentially better

opportunity, such as eating the object (if it is edible), or mating with it (if it is another wain). Figure 7.12 shows the play patterns for wains. Only wains aged between 1100 and 1200 Créatúr clock ticks are included in the graph. Note that the rates for the numerals in this graph are roughly in the inverse order of Figure 7.8, which suggests that the wains prefer to play with the numerals that they are most reluctant to eat.

7.5.5 Wain evolution

Learning rate genes

Recall from Section 7.1.9 that the *positive learning rate gene* controls the rate at which the decider reinforces decisions with positive outcomes, while the *negative learning rate gene* controls the rate at which it reinforces decisions with negative outcomes. Figure 7.13 shows how the value of these two genes changed over time in the Ardara population. Over time, a slow but steady reduction in the value of the positive learning rate gene can be seen. The effect of this would be to make wains less likely to assume an action is wise because it had a good outcome on one or two occasions; instead, they wait for more evidence.

Another change that can be seen in Figure 7.13 may explain how wains adapted to the increases in toxicity of the poisonous numerals, described in Section 7.5.1. Immediately after this change was made, the value of the negative learning rate gene increased significantly, and eventually levelled off. The effect of this would be to make the wains more likely to avoid actions that had bad outcomes. Taking the changes to both of these genes into account, it seems that the wains have evolved a slightly more pessimistic view of their environment than was present in the initial

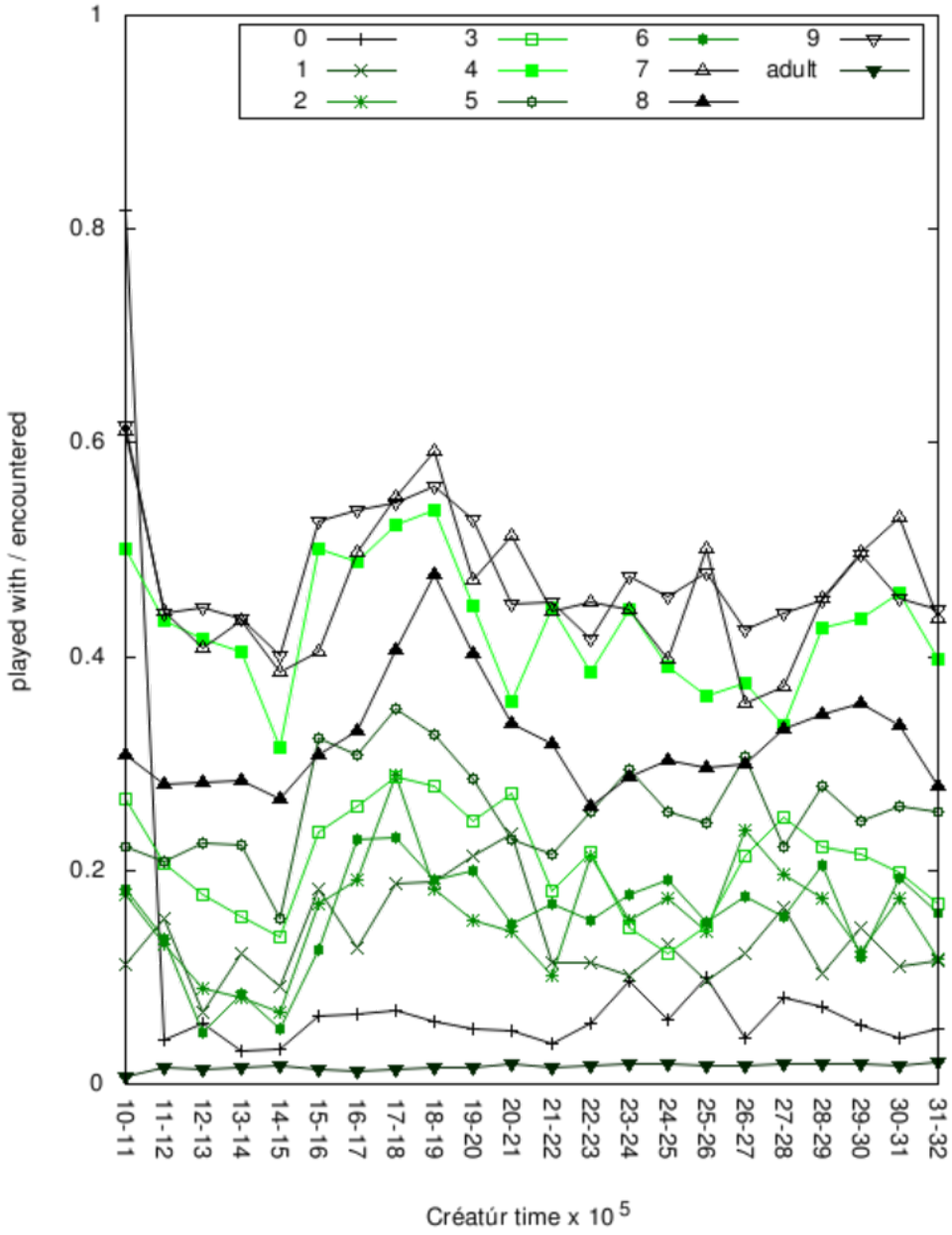


Figure 7.12: Wain play patterns. The lighter green lines indicate fun objects (those which reduce boredom); dark green to black indicates boring objects.

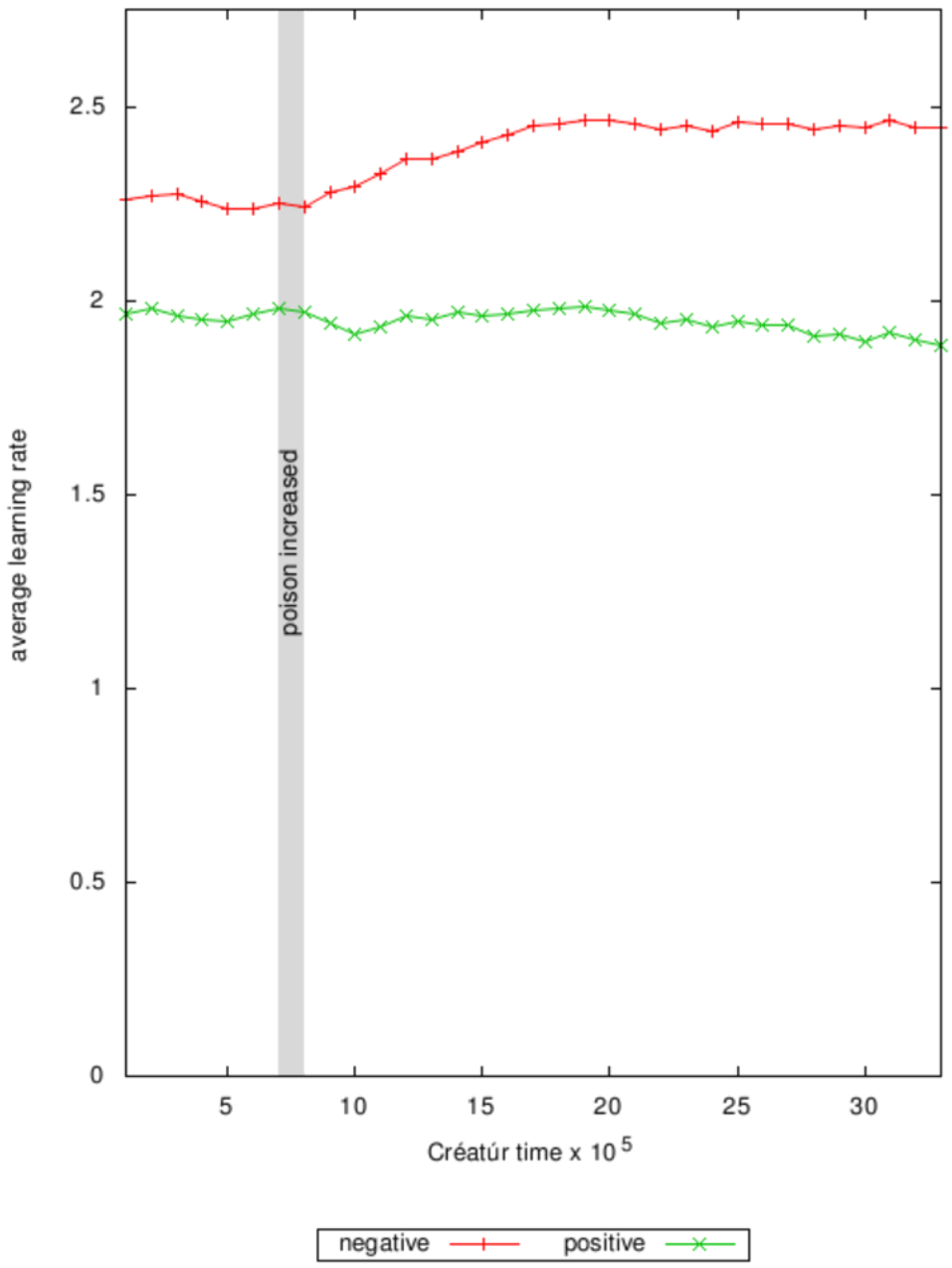


Figure 7.13: Evolution of decider component in the brains of wains. The grey vertical band indicates a series of adjustments to the toxicity of poisonous numerals. The graph shows a slow but steady trend reducing the positive learning rate. The negative learning rate increased after the adjustments, but seems to have levelled off. The time span shown includes 12 generations of wains.

population.

Pattern capacity gene

As discussed in Section 7.1.2, the metabolism tax paid by wains is partly dependent on the pattern capacity of the SOM, which is determined by the pattern capacity gene. If a wain can reduce the number of patterns that it learns, without sacrificing its ability to identify enough edible food to survive, then it has an evolutionary advantage. As shown in Figure 7.14, wains have approximately two fewer patterns now than the initial population did. The population continues to thrive, however, as evidenced by Figure 7.4.

Edelman cycle gene

Recall from Section 7.1.7 that at intervals specified by the *Edelman cycle gene*, the least useful pattern (the one that has the fewest matches) is discarded, and the node is randomised so that it can learn a new pattern. In the initial population, the cycle was set to a random integer between 50 and 100000. Figure 7.15 shows how the cycle has evolved over time. The cycle increased, then levelled off, but it may be increasing again.

7.5.6 Mutations

Given the high mutation rate used in this trial ($p_{mutation}$ = 0.001), it was not surprising that mutations appeared in the second generation of the Ardara population. One area where it is particularly easy to observe the effect of mutations is in the appearance of the wains. Figure 7.16 shows some representative examples.

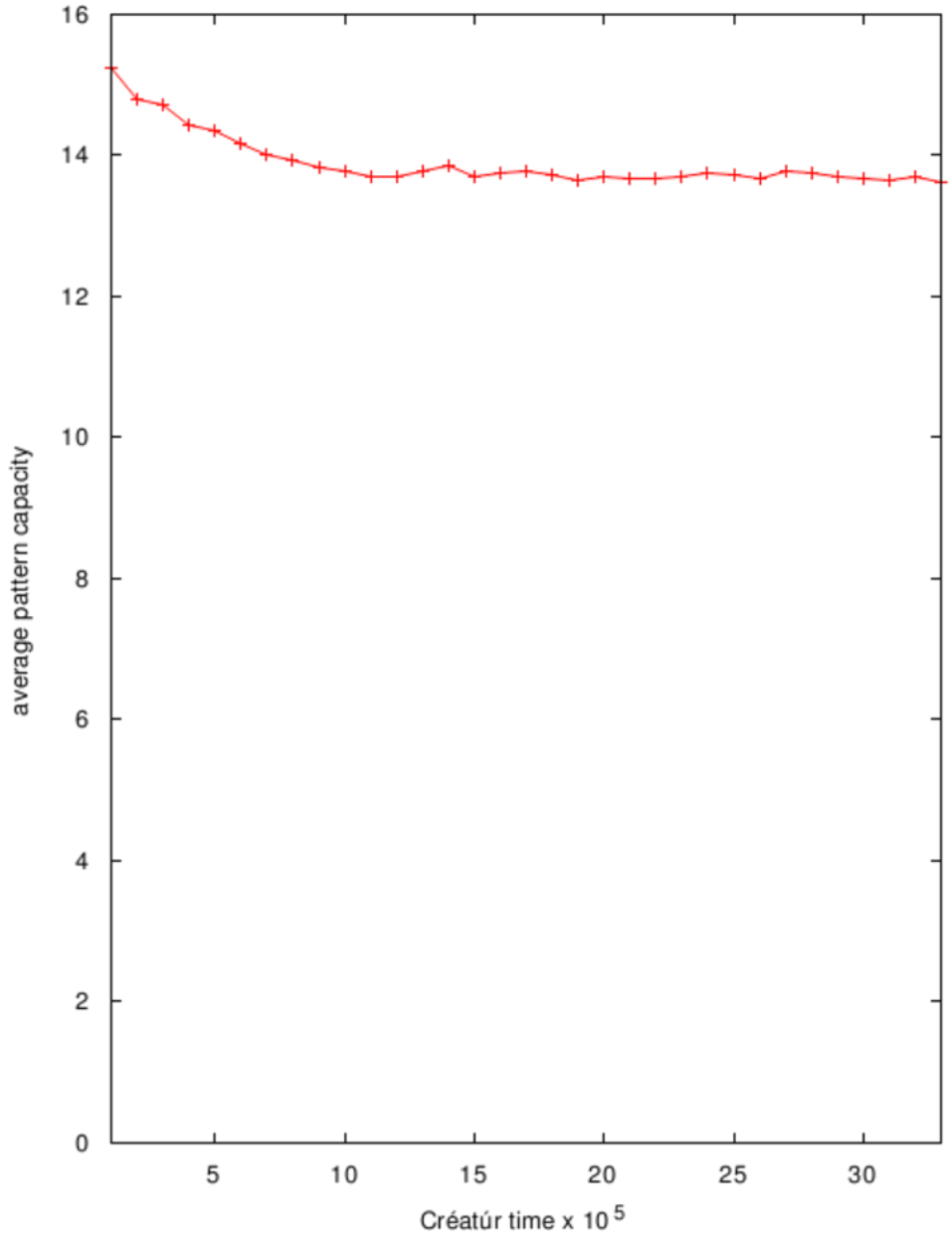


Figure 7.14: Evolution of pattern capacity for wains. The graph shows a trend toward reducing the number of patterns stored, making the SOM more efficient. The time span shown includes 12 generations of wains.

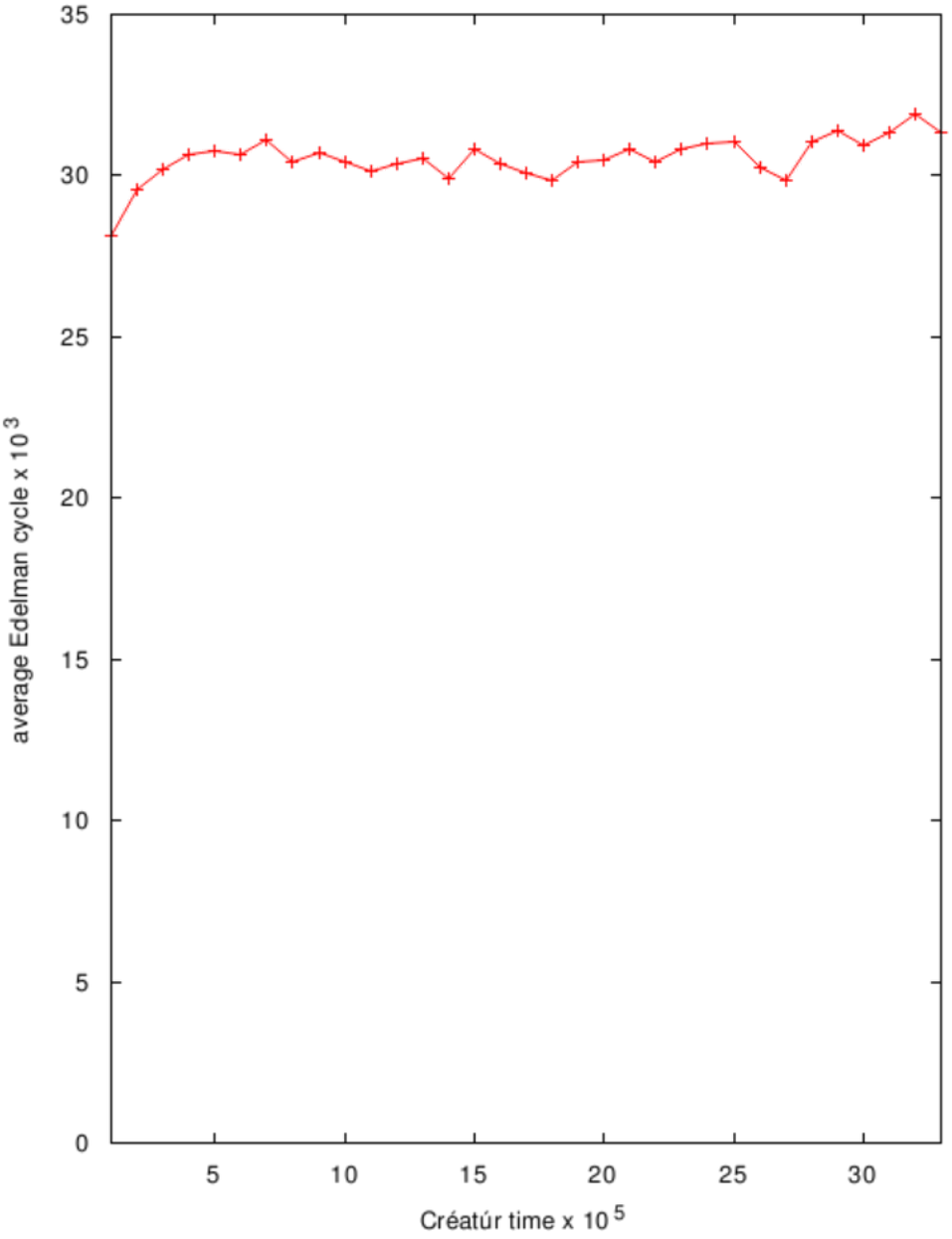
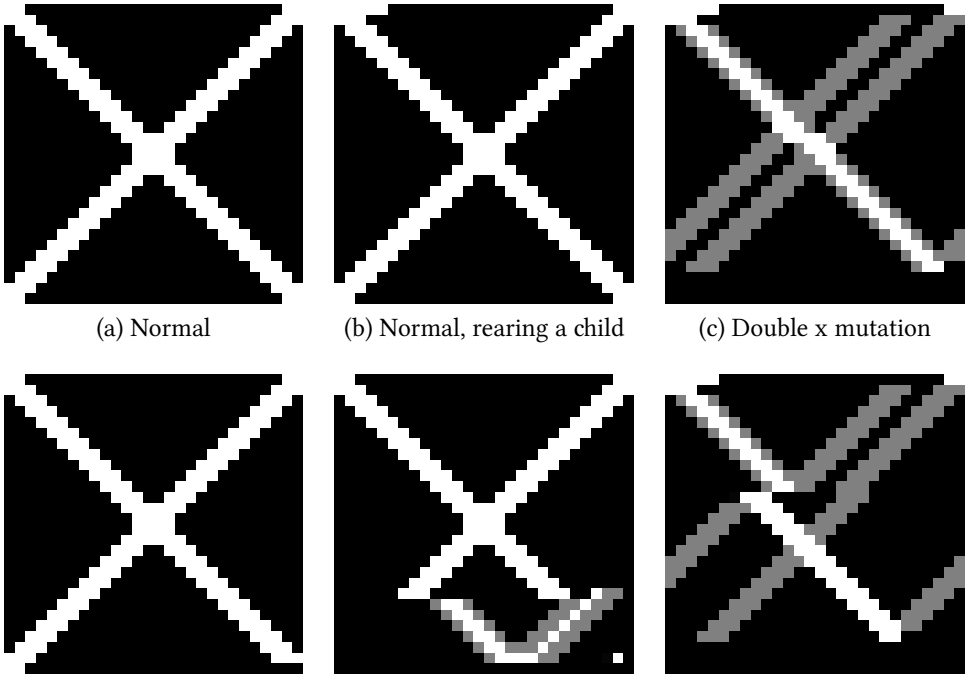


Figure 7.15: Evolution of Edelman cycle for wains. The graph shows an trend toward lengthening the Edelman cycle. The time span shown includes 12 generations of wains.



(a) Normal (b) Normal, rearing a child (c) Double x mutation

(d) Short southeast mutation (e) Reflected south mutation (f) Picasso mutation

Figure 7.16: Wain appearance mutations

Figure 7.17 shows the family history of one particular mutation. The “missing south” mutation caused the sequence of appearance genes to be truncated, so that the lower half of the X is missing. If the offspring inherit appearance genes for the lower half from only one parent, those genes will be expressed. Therefore, the “missing south” mutation is a recessive trait, although it involves a sequence of genes rather than a single gene.

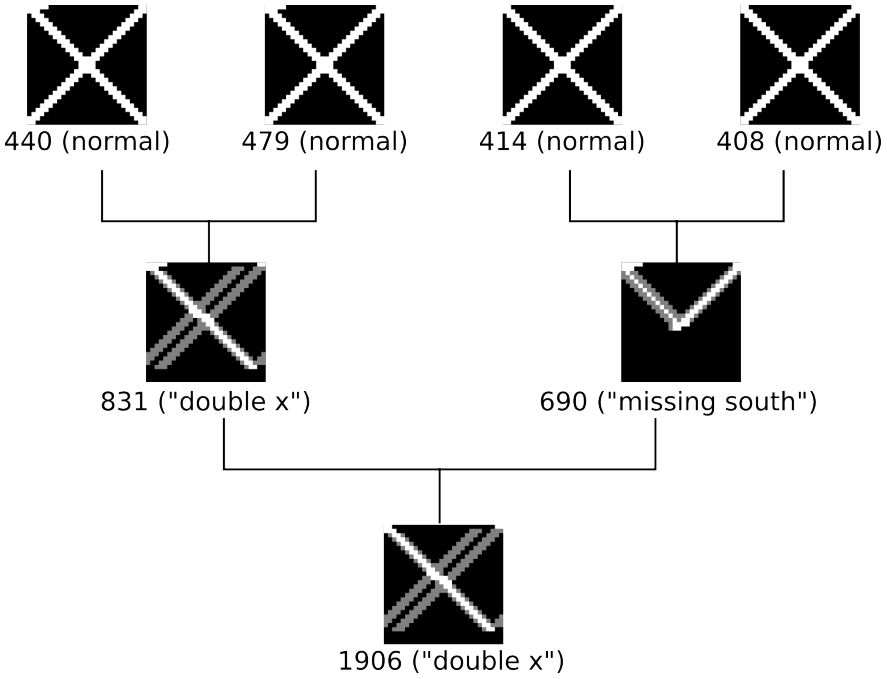


Figure 7.17: Inheritance of a mutation. The numbers are the wain IDs.

Once mutants appeared, it was to be expected that wains would develop ways to recognise them. Figure 7.18 shows a SOM with two separate patterns for wains, one for normal wains, and one for mutants. As this wain is itself a mutant, it now has a way to identify wains that are likely to be related to it. Given time, this might lead to kin selection.

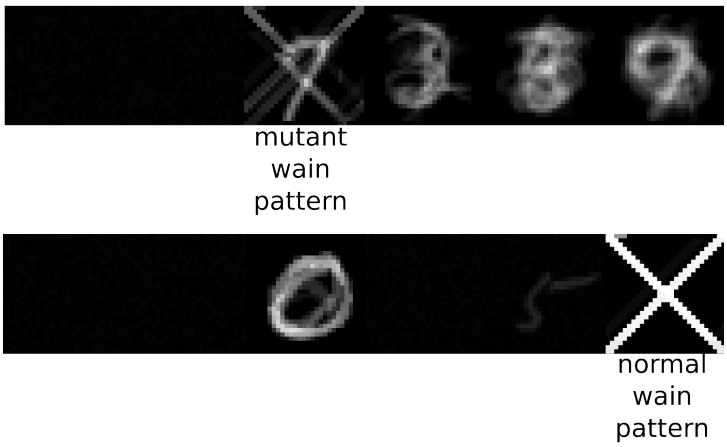


Figure 7.18: A mutant-detecting SOM. This wain's SOM shows two separate patterns for wains. This wain is itself a mutant, of the "reflected south" variety.

7.5.7 Risk-taking and evolution

After reviewing the results, it seems that wains too often take actions that are predicted to have bad outcomes. This happens because every action receives a minimum weighting of one, which was implemented so that the wains would occasionally take risks. An action that had a bad outcome under one set of circumstances may have a good outcome in a different situation. One potential area for improving the decider would be to make the amount of risk-taking genetic. It seems likely that evolution would make the wains a little more risk-adverse.

7.6 Summary

The key points about the wain implementation are summarised in this section, with references to the section in which the topic was discussed. The appearance of a wain is an 28x28 grey-scale image, which is genetically determined (7.1.1). Wains in the starter population had the image of an X as their appearance; since

then, several mutations have arisen producing wains with different appearances (7.5.6).

The objects in the wain universe include images of handwritten numerals, and other wains. Some numerals are edible; eating them provides energy to a wain. Others are mildly poisonous; eating them decreases a wain's energy(7.1.2). When a wain encounters an object, the appearance of that object is presented to the wain's senses, along with information about the wain's current state (7.1.8). It can then choose to eat (7.1.2), attempt to mate with (7.1.3), or play (7.1.5) with the object. After a child is born, it remains with the dam until it is mature (7.1.4).

Wains have brains which contain a classifier and a decider (7.1.6). The classifier is a SOM; it identifies patterns in sensory inputs that a wain receives during its lifetime. The decider is a Hebbian network; it makes decisions based on the patterns identified by the classifier and learns from the results. The brain implements a form of Neural Darwinism; patterns compete with each other, and the least useful pattern is pruned periodically.

The wains discovered patterns in the data (7.5.2, 7.5.3, 7.5.4), and they thrived (7.5.1). Not only did individual wains learn to make better decisions during their lifetime (7.5.2), but changes were made to the brain over several generations that improved the decision-making ability of the wains (7.5.5). When poisonous numerals became more toxic, wains adapted (7.5.1), primarily by modifying their learning rates through evolution (7.5.5). Evolution also made the brains of wains more efficient, by reducing the number of patterns that the SOM stored, without affecting the wain's ability to identify sufficient food to survive (7.5.5).

Chapter 8

Conclusions and observations

This chapter presents some general observations from the Créatúr research project that might be applicable to other ALife projects as well. Finally, it presents some conclusions about this project.

8.1 Observations

During the literature review, the author found information about ALife *implementations*, but very little advice on setting up an experiment, monitoring the population, and identifying problems. This section contains the sort of information that the author would have found useful. Some of the advice is specific to aspects of the approach used in this project, but most of it is general enough to apply to any ALife project.

8.1.1 Analysis tools

The availability of good analysis tools is crucial to understanding why a population is succeeding or failing. Guessing why a population isn't behaving as expected and tweaking parameters to try to solve the problem is an exercise in frustration. Some of the types of tools that were found most useful are listed below.

- tools to analyse large quantities of log data in order to discern general patterns of behaviour in the population
- tools to monitor the brain activity of an animat and represent it visually
- tools visually represent the structure (neural connections) in a dote's brain
- tools to analyse a dote's genes, allowing the researcher to verify that dominance and blending were working as expected

8.1.2 Strategy for using Créatúr

Chapters 6 and 7 described how Créatúr was used in experiments with two different types of animats. The procedure below is the one found to be most successful during those experiments. It may also be a useful guide to others beginning an ALife research project.

Design the survival problem

When using animats to solve a non-biological problem, consider re-framing it as a survival problem in order to take advantage of evolution. The most straightforward way to do this is to use the problem as a source of energy for the animats. For example, if the goal is for the animats to recognise patterns in data (e.g., identifying

handwritten numerals), then successfully identifying those patterns could result in an increase of energy for the animats, while mis-identifying a pattern could result in an energy loss. The strings of data to be examined become objects in the animats' environment.

Design the animat

In designing the animats, one important consideration is the genome. Choosing which characteristics to make inheritable involves a trade-off. Life on this planet is a testament to evolution's ability to design solutions to meet difficult and changing requirements, so the temptation for the programmer faced with a difficult question is to let evolution find the answer. However, there are no guarantees that an answer exists, or that evolution will find it, at least not in the timescale of a research project.

Start with an "easy" environment

Begin with an environment in which it is trivial for the animats to survive. For example, the metabolism rate might be very low initially, allowing animats to survive for long periods without food. After the animats are mating and eating, gradually increase the metabolism rate to encourage faster evolution. This is usually quicker than starting with a challenging environment and waiting for evolution to produce animats that can survive in it.

Ensure that the animats mate

Diagnose and correct any problems that prevent the animats from mating. The animats must be able to recognise a mating opportunity, and take advantage of it. This should be the first goal of any ALife experiment, because once it is achieved,

evolution may be able to assist with the remaining goals. If mating is not occurring, it may indicate a problem with the design of the animat, or a bug in its implementation. Once mating occurs, at least occasionally, the frequency of mating is likely to increase without further intervention because evolution will select for animats that mate regularly. Eventually the population should be self-sustaining

Ensure that the animats eat

Once the animats are mating occasionally, diagnose and correct any problems that prevent them from eating. The animats must be able to recognise food in their environment, and take advantage of some opportunities to eat. If the initial environment is "easy", animats will only need to eat occasionally. Later on, the harshness of the environment can gradually be increased, and evolution will select for animats that eat more regularly.

Ensure that offspring are self-sufficient

Once the animats are mating and eating, ensure that at least some offspring become self-sufficient. If offspring are not self-sufficient at birth, they should be nurtured until they are self-sufficient. However, they should not cause an excessive drain on the parent. Once a small percentage of the offspring become self-sufficient, evolution should improve the survival rate, perhaps by selecting for offspring that are smarter at birth, or by selecting for better parents.

Gradually make the environment more challenging

Recall that evolution requires three conditions: (1) variation, (2) heredity, and (3) differential fitness. Once the animats are mating, eating, and raising viable off-

spring, the first two conditions are satisfied. However, since the initial environment is an "easy" one, there may be little difference in survival rates between animats with different traits. Now it is time to "turn up the heat"; i.e., to alter the environment to ensure that the best-adapted animats are more likely to survive. In a sense, the steps above were preparation; this is where the real experiment begins and the animats start to "solve" their environment.

8.2 Conclusions

As discussed in Section 3.1, the first objective of the Créatúr project was to evolve an ALife population with sufficient intelligence to discover patterns in data, and to make survival decisions based on those patterns. The first ALife implementation, dotes, did not achieve this objective. However, the second implementation, wains, was successful. The wains have learned to distinguish between objects in their environment by identifying patterns in data representing the objects' appearance. Based on those patterns, they have learned to make good decisions about which objects to eat, which ones to try to mate with, and which ones to play with.

The second objective was to create a population that adapts to its environment through both evolution and lifetime learning. The wain implementation achieved this objective as well. Even without benefit of evolution, the first generation mastered the numeral recognition task well enough to thrive. Evolution further adapted the wains to their environment by making them a little more pessimistic (lowering the rate at which they learn from positive experiences, and raising the the rate at which they learn from negative experiences), and also by making their brains more efficient (reducing the number of patterns and increasing the Edelman

cycle) without adversely affecting their accuracy.

The third objective was to evolve an ALife population with some general-purpose problem solving skills, that could be used as "seeds" for projects requiring specialised skills. This objective has not yet been achieved; more research is needed, as discussed in Chapter 9.

In their current form, wains are not as accurate at identifying handwritten numerals as a trained neural network would be; however, that was not the purpose of this research. The numeral recognition task was a "toy problem"; it helped to prove the design and demonstrate the capabilities of the wains. The wains were not told what kinds of patterns to look for, which suggests that, given time to evolve, they would be able to find patterns in large, complex data sets, where the researcher may not know what patterns exist, or where to look for them. If a project required an ALife population to work on problem of similar size and complexity, some time might be saved by starting with the wains, as evolution has already tweaked their design. However, to have more general application, the wains will need much larger brains.

This project was successful in meeting the first two objectives, and the author expects that with more research, it would meet the third objective. After only twelve generations, the wain population has exhibited complex, interesting behaviour. In addition, the project produced Créatúr, a reusable software framework for ALife projects. The Créatúr framework has already been used with two very different ALife species, and should be applicable to a variety of ALife projects. By encapsulating the functionality common to most ALife habitats and species, the Créatúr framework can save development time in future projects.

Chapter 9

Future Directions

This chapter proposes future directions for research continuing on from, or inspired by, the Créatúr project. Any of the directions could be pursued independently of the others.

9.1 Richer interaction

While maintaining the philosophy that "the data is the environment", Créatúr could be made more biologically realistic by allowing the animats to move freely within the environment. This would give animats a measure of control over which objects and other animats they interact with. The data could be used to add an element of geography; some paths would be easier to travel than others, different parts of the environment would have a different mix of resources for the animats to exploit. This could support isolation of populations, and eventual speciation. Animats might be allowed to rearrange the environment, constructing shelters and barriers for production, and possibly promoting co-operation. In situations where

Créatúr is being used to analyse data rather than to study ALife itself, the ability to rearrange data would be subject to "natural" laws that would ensure the integrity of the relationships within the data.

It could be beneficial to allow animats to have additional types of interactions with each other, such as fighting, sharing, and trading. To support this, an animat should be able to sense aspects of another animat's current state, such as their aggression, passion, and strength. Providing a means for animats to observe the behaviour of others might result in a type of cultural transmission.

9.2 More realistic ecology

Allowing dotes and wains (and other future animat species) to co-exist would make the ecology more biologically realistic. Predator species and prey species might be introduced. Animats might be given a more complex biochemistry, requiring multiple types of nutrients available from diverse sources of food. Currently the food in Créatúr is inert; introducing plants as a food source, subject to their own evolutionary paths would allow the researcher to study a variety of relationships in the ecosystem.

Some changes that might promote faster evolution include working with larger populations, imposing a maximum lifespan, opening up the genome to give evolution more control over animat design, and allowing the genetic code itself to evolve. Rates for mutation and crossover, which are currently fixed, could be allowed to evolve as well.

9.3 Better brains

Although the dotes did not exhibit any evidence of learning during this project, their brain architecture is far more flexible than that of the wains. It is possible that, given enough time, evolution would have designed a working dote brain. However, it might be more practical to design a simple yet functional brain using the dote architecture, and then let evolution refine that design. New alleles encoding additional learning rules could be added.

During childhood, the only learning that wains perform lies in training the SOM to recognise patterns in the data. They do not begin to learn what actions are best in response to those patterns until they are mature. Wains might behave more intelligently if they had a a period between childhood and adulthood, an adolescence during which they observed the choices made by their parents and learned from the outcomes. In this way, by the time they are independent, they are pre-equipped to make better choices.

9.4 Benchmarking

The validity of Créatúr as an ALife environment could be established by benchmarking it against nature. Studies of predator and prey population cycles, classical conditioning and operant conditioning (particularly shaping and chaining of behaviours) could be studied, and compared to biological models.

Créatúr could also be used to model a possible evolutionary trend toward complexity by applying standard complexity measurements to Créatúr, and observing the results over time, particularly with regard to neural complexity. Similar stud-

ies were performed for PolyWorld [63, 65], it would be interesting to compare the results.

9.5 Improved support for solving real-world problems

Another avenue for research would be to evolve an ALife population with some general-purpose problem solving skills, that could be used as a "seed" for projects requiring specialised skills. It should be feasible to create a population adapted for a particular task by starting with an existing population of animats with basic intelligence and introducing new survival challenges gradually until the animats have developed the new skills required. Furthermore, it should be faster to do this than it would be to evolve a specialised population "from scratch". It might be advisable to have several seed populations, each of which has proved to be well-adapted to solving a particular class of problem.

Glossary

activation A function used by an artificial neuron in an ANN, which acts on the weighted sum of the inputs to that neuron and determines its output. 36, 71–74, 78, 79, 117

allele One of the possible forms that a given gene may take. 28, 43, 101, 118, 120, 151, 247

animat An artificial animal. 16, 26, 28, 30, 31, 33, 40, 41, 43–48, 54–61, 63–66, 93–102, 111, 113, 116, 139, 140, 147, 185–188, 190, 191, 193, 194, 196–198, 201, 220–222, 224–233, 237–240, 243–246

appearance In Créatúr, this is a catch-all term for properties which are available to the senses of an animat.. 225

Ardara The name of the wain population used in the final trial of Créatúr. 160

b Boredom level of an animat. 144

back-propagation A method for altering the weights in a neural network to reduce error. Used in conjunction with supervised learning. 39, 69, 70, 72, 79, 82, 83, 86, 87, 91

blueprint In the Créatúr project, a set of instructions for creating an animat. 102, 121, 151

chaotic system A system whose behaviour can be predicted in principle, but which is so sensitive to initial conditions that it is for all practical purposes, unpredictable. 23

classifier In mathematics and neural networks, a machine learning program. In wains, the component of the brain that builds a set of patterns representing the types of objects that it encounters. 32, 33

context The sensory inputs presented to a wain, including information about the wain's internal state as well as external sensory inputs. 148, 149

crossover Breaking a pair of gene sequences, and swapping their tails. Sometimes the term crossover is reserved for the special case where the sequences are broken at corresponding locations, while the term *cutting and splicing* is used for the more general case where the cuts may be at non-corresponding locations, thereby ending up with two sequences of different length. 101, 103, 159, 191, 229

cutting and splicing See *crossover*. 102

daemon A computer program that runs in the background and does require user interaction. 93

dam In the Créatúr framework, the animat chosen to rear the child. See *sire*. 101–103, 116, 122, 138, 143, 144, 152, 170, 172, 183, 230

declarative programming A programming paradigm that defines the result of a computation rather than the actions to be performed. 49

deserialise In the Créatúr framework, reading an animat from a file. See *serialise*. 232

design stance Predicting the behaviour of an object by assuming that the object will operate according to its design. 24

diploid In biology, a diploid organism has two sets of chromosomes in each cell. By extension, a diploid ALife organism contains two sets of building instructions. 29, 47, 48, 55, 101

dominance An effect where a child inherits two different versions of a gene, and one gene is expressed while the other has no effect. 102, 120, 151

dote An artificial lifeform in the Créatúr habitat. 17, 18, 33, 55, 113–122, 125–130, 134–140, 150, 151, 159, 188, 191, 192, 222, 236, 237

e Energy level of an animat. 114, 142

e^{+}_{berry} The energy provided by a edible berry. 128, 135

e^{-}_{berry} The energy cost from eating a poisonous berry. 135

$e_{connection}$ The energy cost per neural connection. 115, 128, 135

e_{iq} A multiplier relating the brain complexity to its metabolic costs. 143, 159, 160

emergence A phenomenon where individual components behaving according to simple rules give rise to complexity at a higher level. 15, 22, 54

$e_{metabolism}$ The amount of energy an animat loses at every tick of the Créatúr clock. 115, 143

$\bar{e}_{metabolism}$ The average amount of energy that animats lose at every tick of the Créatúr clock. 128

e_{neuron} The energy cost per neuron. 115, 128, 135

$e_{thinking}$ The energy cost per brain update. 115, 129, 135

evolutionary development A software development methodology in which an initial implementation is built, user feedback is obtained, and new versions are produced as needed. 223

export In programming, making data or methods in one component visible to other components. 75, 103, 104

exteroception The perception of stimuli that originate outside the body; the process by which an organism perceives the external world. 97, 99, 100, 227

feed-forward network In a feed-forward network, the neurons are grouped into layers. The flow of data is from the sensor layer to the output layer, without any loops. 38, 39, 69

forgetting rule The mechanism by which connections that turn out to be useful are preserved, while those that turn out not to be useful are pruned. 116–118

functional programming A programming paradigm that treats expressions as mathematical functions, and avoids side effects of computation. 49

gamete In biology, a sex cell. In Créatúr, a sequence of genes donated by one parent. 101–103, 229

gene A unit of heredity. 101, 120, 151

genome The set of genes for an organism. 102, 116, 229, 230, 232, 233

h The hunger level of an animat. 142

haploid An organism or cell having only one sequence of genes. 44, 46–48, 55

Haskell A purely functional programming language named after the logician Haskell Curry. 51, 56, 70–74, 79, 90, 91, 103, 105, 106, 121, 151

Hebb's Rule A method of updating weights for an artificial neuron. 37, 55, 118, 120

Hebbian learning Any of a number of algorithms for updating neurons in which "cells that fire together, wire together". 37, 40, 55

hidden layer An internal layer of artificial neurons in an ANN. 38

holism The view that a system should be studied as a whole entity. 24

homologous Refers to genes for the same characteristics, at the same location in the gene sequence. 120, 151

imperative programming A programming paradigm that focuses on computation as a series of actions performed in a specified order, which change the state of the program. 49

incomplete dominance An effect where a child inherits two different versions of a gene, and one gene is expressed more strongly than the other. 102

input pattern The input presented to an ANN. 38

intentional stance Predicting the behaviour of an object by treating it as an agent with beliefs and desires, and assuming that the object will act in accordance with those beliefs and desires. 25, 54, 59, 60

interoception The perception of stimuli that originate inside the body; the process by which an organism perceives its own condition. 97, 99, 100, 227

lambda calculus A formal system developed by Alonzo Church [74] for defining functions, function applications, and recursion. 49

learning rule The means by which an artificial neuron adjusts the weights between itself and other neurons (or direct inputs). 116, 117

local learning rule A learning rule which depends only on the neuron's current state and its inputs. 37

meme A unit of cultural transmission. 28

metabolism tax An amount of energy deducted from organisms in the Créatúr universe, related to brain complexity. 115, 128, 143

monad In functional programming, a structure that represents computations. 107, 108

Muller's ratchet A theory of how sexual reproduction may help to remove deleterious mutations from the gene pool. 29

mutation In ALife, randomly altering a bit in a gene sequence. 102, 159, 191, 229

$n_{connections}$ The number of neural connections in the brain. 115

Neural Darwinism A theory proposed by Gerald Edelman which states that connections in the brain undergo a type of natural selection. 16, 31, 146, 148, 183, 220

neural network See artificial neural network. 50

neural plasticity The process whereby the brain "wires" and "re-wires" itself. 31

neuron A nerve cell, or a node in an ANN. 19, 31, 35–38, 40, 50, 64, 121

n_{ex} The number of exteroceptive inputs to the brain. 143

n_{int} The number of interoceptive inputs to the brain. 143

n_{pat} The maximum number of patterns that the brain can recognise. 143

$n_{neurons}$ The number of neurons in the brain. 115

node In mathematics, a vertex in a graph. In neural networks, another term for an artificial neuron. 40, 41, 151

nominal emergence A phenomenon occurs when a property can exist at a macro level but not a micro level. 23

non-local learning rule A learning rule which has other dependencies besides the neuron's current state and its inputs. 37

Oja's Rule A method of updating weights for an artificial neuron. 37, 55, 118, 120, 122

output layer The layer of artificial neurons in an ANN which produces the output signals. 38

PolyWorld An alife program written by Larry Yaeger. 16, 17, 42, 44, 45, 63, 64

p Passion level of an animat. 115, 143

$p_{cut-and-splice}$ The probability that crossover resulting in non-equal lengths will occur when gametes are created. 159, 160

$p_{mutation}$ The probability that mutation will occur when gametes are created. 159, 160, 177

$p_{crossover}$ The probability that crossover resulting in equal lengths will occur when gametes are created. 159, 160

persistence In computer science, the ability for data to be preserved between executions of a program. 96

physical stance Using whatever is known about the laws of physics and constitution of an object to predict its behaviour. 24

polymorphic In computer science, the ability for code to work with more than one data type. 105

Popperian creature A creature that can evaluate possible actions and make choices based on that evaluation. 30, 34, 58

punctuated equilibrium A theory that states that evolution, rather than being gradual, tends to happen in short bursts, interspersed with long periods of little or no change to the species. 43, 55

QuickCheck A property-based software testing tool. 53, 54, 56, 88, 90, 103, 104, 110, 121, 151

Red Queen hypothesis A theory that sexual reproduction may provide protection against parasites as the host evolves new defenses. 29

reductionism The view that a system can be understood by dividing it into components and studying their properties. 23

referential transparency A property where any expression can be replaced by its value without changing the behaviour of the program. 49, 50, 107

RGB A colour model in which the red, green, and blue components are specified. 45, 113, 114

sensor layer The layer of artificial neurons in an ANN which receives the input signals. 38

serialise In the Créatúr framework, converting an animat into a format that can be stored in a file. See *deserialise*. 232

sire In the Créatúr framework, the animat not chosen to rear the child. See *dam*. 101–103, 172

strong emergence A phenomenon that cannot be derived from the underlying processes. 23

supervised training A training method in which the desired response (target values) for each input vector in the training set is known and provided to the neural network during training. 39, 69, 70

target pattern The desired output of an ANN for a particular input pattern. 39, 40

test-driven development A software development methodology credited to Kent Beck [81] where the developer writes a (failing) test case for a new function or feature, and then writes the code to pass the test. 53

thunk A computation that is delayed until the result is needed. 153

$t_{thinking}$ The number of brain updates performed when making a decision. 115

tuple An ordered list of elements. In Haskell, elements in a list must all be of the same type, but elements in a tuple need not be. 89, 105

type variable A placeholder for a type. 105

unsupervised training A training method in which the desired responses (target values) for the input vectors in the training set are not known. 39, 40

wain An artificial lifeform in the Créatúr habitat. 18, 33, 41, 55, 139–153, 157–163, 167, 168, 170, 172–174, 177, 181–183, 188, 189, 191, 192, 222, 242, 243, 246

weak emergence A phenomenon can be derived from the underlying processes, but only by simulation. 23

winning node In a SOM, the node whose weight vector is most similar to the input pattern. 40, 41

Acronyms

AI Artificial Intelligence. 15, 16, 20, 24, 34, 35, 54, 55, 58, 62, 63, 219, 220

ALife Artificial Life. 15, 16, 18–20, 24, 33, 42–47, 54, 55, 57–63, 65, 66, 93, 95, 111, 112, 138, 139, 184–186, 188, 189, 191–193, 219–221, 223–225, 227

ANN Artificial Neural Network. 35, 37–40, 44, 46, 47, 49, 55

fMRI functional Magnetic Resonance Imaging. 32, 33

SOM Self-organising Map. 16, 35, 39–41, 55, 146, 151, 153, 183, 192, 219

Bibliography

[1] McCarthy J, Minsky ML, Rochester N, Shannon CE. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence; 1955. The date for the origin of the term ”artificial intelligence” is often given as 1956. However, although the conference was in 1956, the proposal which contains the term was written in 1955. http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html. Available from: `http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html`.

[2] Langton CG. Studying artificial life with cellular automata. Phys D. 1986 October;2:120–149. Available from: `http://portal.acm.org/citation.cfm?id=25201.25210`.

[3] Lewes GH. Problems of Life and Mind: The principles of certitude. From the known to the unknown. Matter and force. Force and cause. The absolute in the correlations of feeling and motion. Appendix: Imaginary geometry and the truth of axioms. Lagrange and Hegel: the speculative method. Action at a distance. Problems of Life and Mind. Tr

”ubner & co.; 1875. Available from: http://books.google.ie/books?id=ouPK_DUE-KYC.

[4] Wilson SW. The Animat Path to AI. In: Proceedings of the first international conference on simulation of adaptative behavior : From animals to animats; 1991. p. 15–20.

[5] Darwin C. The origin of species. Everyman’s library. Dent; 1936. Available from: http://books.google.ie/books?id=gZeHvxthDkQC.

[6] Dawkins R. Climbing mount improbable. Penguin science. Penguin; 1997.

[7] Willaford W. Dictionary of Philosophy of Mind - cognitive science; 2004 [cited 2011-07-02 01:26:30]. Available from: http://philosophy.uwaterloo.ca/MindDict/cognitivescience.html.

[8] Miller GA. The cognitive revolution: a historical perspective. Trends in Cognitive Sciences. 2003;7(3):141–144. Available from: http://www.sciencedirect.com/science/article/pii/S1364661303000299.

[9] Barsalou LW. Introduction to 30th Anniversary Perspectives on Cognitive Science: Past, Present, and Future. Topics in Cognitive Science. 2010 Jul;2(3):322–327. Available from: http://doi.wiley.com/10.1111/j.1756-8765.2010.01104.x.

[10] Gentner D. Psychology in Cognitive Science: 1978-2038. Topics in Cognitive Science. 2010 Jul;2(3):328–344. Available from: http://doi.wiley.com/10.1111/j.1756-8765.2010.01103.x.

[11] Aristotle. Metaphysics. The Internet Classics Archive; 350BCE. Translated by W. D. Ross. Available from: http://classics.mit.edu/Aristotle/metaphysics.html.

[12] Johnson S. Emergence : the connected lives of ants, brains, cities and software. London: Allen Lane; 2001.

[13] Chalmers DJ. Thoughts on Emergence; 1990. Post to comp.ai.philosophy newsgroup. Available from: http://consc.net/notes/emergence.html.

[14] Holland JH. Emergence : from chaos to order. Oxford: Oxford University Press; 2000.

[15] Bedau MA. Weak Emergence. In: Tomberlin J, editor. Philosophical Perspectives: Mind, Causation, and World. vol. 11. Malden, MA: Blackwell Publishers Inc.; 1997. p. 375–399. Available from: http://dx.doi.org/10.1111/0029-4624.31.s11.17.

[16] Bedau MA. Downward Causation and the Autonomy of Weak Emergence. Principia. 2002;6(1):5–50.

[17] Chalmers DJ. Strong and Weak Emergence. In: The Re-Emergence of Emergence. Oxford University Press; 2006. .

[18] Bedau M. The Scientific and Philosophical Scope of Artificial Life. Leonardo. 2002;35(4):395–400.

[19] Clayton P, Davies P. The re-emergence of emergence: the emergentist hypothesis from science to religion. Oxford University Press; 2006. Avail-

able from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.114.1724\&rep=rep1\&type=pdf.

[20] Dennett DC. The intentional stance. Bradford Books. MIT Press; 1989. Available from: http://books.google.ie/books?id=Qbvkja-J9iQC.

[21] Dennett DC. Intentional Systems Theory. In: The Oxford handbook of philosophy of mind. Oxford handbooks. Claredon Press; 2009. Available from: http://books.google.ie/books?id=FT-USif1E7YC.

[22] Dawkins R. The selfish gene. Oxford University Press; 2006. Available from: http://www.worldcat.org/isbn/9780199291144.

[23] Dennett D. Kinds of Minds: The Origins of Consciousness. London: Phoenix; 1997.

[24] Darwin C. On the Origin of Species or the Preservation of Favoured Races in the Struggle for Life. Project Gutenberg; 1859. Available from: http://www.gutenberg.org/cache/epub/1228/pg1228.txt.

[25] Nelson RW. Darwin, Then and Now: The Most Amazing Story in the History of Science. iUniverse.com; 2009. Available from: http://books.google.ie/books?id=je2Ms5kQCNcC.

[26] Hasan H. Mendel and the laws of genetics. Primary sources of revolutionary scientific discoveries and theories. Rosen Pub. Group; 2005. Available from: http://books.google.ie/books?id=lXcRCag4rR8C.

[27] Fisher RA, Bennett JH. The genetical theory of natural selection: a complete variorum edition. Oxford University Press; 1999. Available from: http://books.google.com/books?id=sT4lIDk5no4C.

[28] Dennett DC. Consciousness Explained. Penguin; 1993.

[29] Fuller LWS. Transcript of the debate between Professor Steve Fuller and Professor Lewis Wolpert at Royal Holloway College; 2007 [cited 2011-07-03 01:11:46]. Available from: http://www.bcseweb.org.uk/index.php/Main/RoyalHollowayCollegeDebate.

[30] Bell G. The masterpiece of nature: the evolution and genetics of sexuality. Croom Helm applied biology series. Croom Helm; 1982. Available from: http://books.google.com/books?id=q5g9AAAAIAAJ.

[31] Meirmans S, Strand R. Why Are There So Many Theories for Sex, and What Do We Do with Them? Journal of Heredity. 2010;101(Supplement 1):S3–S12. Available from: http://jhered.oxfordjournals.org/cgi/doi/10.1093/jhered/esq021.

[32] Gensler HL, Bernstein H. DNA damage as the primary cause of aging. The Quarterly Review of Biology. 1981 Sep;56(3):279–303. PMID: 7031747. Available from: http://www.ncbi.nlm.nih.gov/pubmed/7031747.

[33] Bernstein H, Byerly HC, Hopf FA, Michod RE. Genetic damage, mutation, and the evolution of sex. Science (New York, NY). 1985 Sep;229(4719):1277–1281. PMID: 3898363. Available from: http://www.ncbi.nlm.nih.gov/pubmed/3898363.

[34] Michod RE, Bernstein H, Nedelcu AM. Adaptive value of sex in microbial pathogens. Infection, Genetics and Evolution. 2008 May;8(3):267–285. Available from: http://linkinghub.elsevier.com/retrieve/pii/S156713480800004X.

[35] Muller HJ. The relation of recombination to mutational advance. Mutation Research/Fundamental and Molecular Mechanisms of Mutagenesis. 1964;1(1):2–9. Available from: http://www.sciencedirect.com/science/article/pii/0027510764900478.

[36] Hamilton WD, Axelrod R, Tanese R. Sexual Reproduction as an Adaptation to Resist Parasites (A Review). Proceedings of The National Academy of Sciences. 1990;87:3566–3573.

[37] Neiman M, Koskella B. Sex and the Red Queen. In: Schön I, Martens K, Dijk P, editors. Lost Sex. Dordrecht: Springer Netherlands; 2009. p. 133–159. Available from: http://www.springerlink.com/index/10.1007/978-90-481-2770-2_7.

[38] Margulis L, Sagan D. Dazzle gradually: reflections on the nature of nature. Sciencewriters Series. Chelsea Green Publishers; 2007. Available from: http://books.google.ie/books?id=XccH9fgT8AgC.

[39] Popper KR. Objective knowledge: an evolutionary approach. Clarendon Press; 1972. Available from: http://books.google.ie/books?id=v3sIAQAAIAAJ.

[40] Gritti A. Adult neural stem cells: plasticity and developmental potential. J Physiol Paris. 2002 Jan;96(1-2):81–90.

[41] Edelman GM. Neural Darwinism : the theory of neuronal group selection. Basic Books; 1987.

[42] Crick F. Neural Edelmanism. Trends in Neurosciences. 1989;12(7):240–248. Available from: http://www.sciencedirect.com/science/article/B6T0V-485CVT9-J/2/2086e24dfd1847320fa33368933147c2.

[43] Fernando C, Karishma KK, Szathmáry E. Copying and Evolution of Neuronal Topology. PLoS ONE. 2008 Nov;3(11):e3775+. Available from: http://dx.doi.org/10.1371/journal.pone.0003775.

[44] Mitchell TM, Hutchinson R, Niculescu RS, Pereira F, Wang X, Just M, et al. Learning to Decode Cognitive States from Brain Images. Mach Learn. 2004;57(1-2):145–175. Available from: http://www.cs.cmu.edu/{~}tom/mlj04-final-published.pdf.

[45] Mitchell T. YouTube - Brains, Meaning and Corpus Statistics; 2009. Available from: http://www.youtube.com/watch?v=QbTf2nE3Lbw.

[46] AAAI Website. AITopics / AIOverview; 2011. Available from: http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/AIOverview.

[47] Hodges A. Alan Turing: the enigma. London: Vintage; 1992.

[48] Dennett D. Colloquium Papers: Darwin's "strange inversion of reasoning". Proceedings of the National Academy of Sciences. 2009;106(Supplement_1):10061–10065. Available from: http://www.pnas.org/cgi/doi/10.1073/pnas.0904433106.

[49] Turing AM. Computing Machinery and Intelligence. Mind. 1950;LIX:433–460.

[50] Hofstadter DR. Gödel, Escher, Bach: an eternal golden braid. New York: Vintage Books; 1980.

[51] Gurney K. An introduction to neural networks. 1st ed. Address: Routledge; 1997.

[52] Hebb DO. The Organization of Behavior: A Neuropsychological Theory. New edition ed. New York: Wiley; 1949.

[53] Oja E. Simplified neuron model as a principal component analyzer. Journal of Mathematical Biology. 1982 Nov;15(3):267–273. Available from: http://dx.doi.org/10.1007/BF00275687.

[54] Rumelhart DE, Hinton GE, Williams RJ. Learning internal representations by error propagation. 1986;p. 318–362. Available from: http://portal.acm.org/citation.cfm?id=104293.

[55] Kohonen T. Self-organized formation of topologically correct feature maps. Biological Cybernetics. 1982 Jan;43(1):59–69. Available from: http://dx.doi.org/10.1007/BF00337288.

[56] Kohonen T. Self-organizing maps. 3rd ed. Springer series in information sciences, 30. Springer; 2001.

[57] Langton CG. Artificial Life. In: Langton CG, editor. Artificial life: the proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, held September, 1987, in Los Alamos, New Mexico. Addison-Wesley, Redwood City, CA; 1989. p. 1–48.

[58] Ray TS. An approach to the Synthesis of Life. In: C Langton, C Taylor, J D Farmer, S Rasmussen, editor. Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity. vol. XI. Redwood City, CA: Addison-Wesley; 1991. p. 371–408.

[59] Ward M. Virtual organisms. London: Pan; 2000.

[60] Eldredge N, Gould SJ. Punctuated Equilibria: An Alternative to Phyletic Gradualism. In: (ed) Schopf TJM, editor. Models in Paleobiology. San Francisco.: Freeman, Cooper and Co., San Francisco; 1972. p. 82–115.

[61] Shao J, Ray TS. Maintenance of Species Diversity by Predation in the Tierra System. In: Harold Fellermann, Mark D¨rr, Martin Hanczyc, Lone Ladegaard Laursen, Sarah Maurer, Daniel Merkle, Pierre-Alain Monnard, Kasper Støy, Steen Rasmussen, editor. Artificial Life XII: Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems. Cambridge, Massachusetts London, England: The MIT Press; 2010. Available from: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2\&tid=12433.

[62] Yaeger L. Computational genetics, physiology, metabolism, neural systems, learning, vision, and behavior or PolyWorld: Life in a new context. In: Artificial Life III, Vol. XVII of SFI Studies in the Sciences of Complexity, Santa Fe Institute; 1993. p. 263–298. Available from: http://www.beanblossom.in.us/larryy/polyworld.html.

[63] Yaeger L, Sporns. Evolution of neural structure and complexity in a computational ecology. In: Proceedings of the Tenth International Conference

on Simulation and Synthesis of Living Systems (ALifeX. MIT Press; 2006. p. 330–336.

[64] Griffith V. YouTube - Polyworld: Using Evolution to Design Artificial Intelligence; 2007. Available from: http://www.youtube.com/watch?v=_m97_kL4ox0 [cited 2009-11-30 18:26:42].

[65] Yaeger L, Griffith V, Sporns O. Passive and driven trends in the evolution of complexity. In: Bullock S, Noble J, Watson R, Bedau MA, editors. Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems. MIT Press, Cambridge, MA; 2008. p. 725–732. Available from: http://alifexi.alife.org/papers/ALIFExi_pp725-732.pdf.

[66] Yaeger LS. How evolution guides complexity. HFSP Journal. 2009;3(5):328. Available from: http://www.beanblossom.in.us/larryy/Yaeger2009_HowEvolutionGuidesComplexity_HFSP.pdf.

[67] Channon AD, Damper RI. Towards the evolutionary emergence of increasingly complex advantageous behaviours. International Journal of Systems Science. 2000;31(31):843–860.

[68] Grand S. Creation : life and how to make it. London: Phoenix; 2001.

[69] Thorén H, Gerlee P. Weak Emergence and Complexity. In: Fellermann H, Dörr M, Hanczyc MM, Laursen LL, Maurer S, Merkle D, et al., editors. Artificial Life XII. Cambridge, MA, USA: The MIT Press; 2010. Available from: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2\&tid=12433.

[70] Belew RK. Artificial Life: A Constructive Lower Bound for Artificial Intelligence. IEEE Intelligent Systems. 1991;6:8–15.

[71] Bedau M. The arrow of complexity hypothesis (abstract). In: Bullock S, Noble J, Watson R, Bedau MA, editors. Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems. MIT Press, Cambridge, MA; 2008. p. 750. Available from: http://alifexi.alife.org/papers/ALIFExi-pp741-823-abstracts.pdf#page=750.

[72] Calabretta R, Galbiati R, Nolfi S, Parisi D. Two is better than one: A diploid genotype for neural networks; 1996.

[73] Smith RE, Goldberg DE. Diploidy and Dominance in Artificial Genetic Search. Complex System. 1992;6:251–285.

[74] Church A. An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics. 1936 Apr;58(2):345–363. Available from: http://dx.doi.org/10.2307/2371045.

[75] Sutter H. A Fundamental Turn Toward Concurrency in Software. Dr Dobb's Journal. 2005;30(3):16–23.

[76] Hughes J. Why Functional Programming Matters. The Computer Journal. 1989 Feb;32(2):98–107. Available from: http://dx.doi.org/10.1093/comjnl/32.2.98.

[77] Claessen K, Hughes J. QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. New York, NY, USA:

ACM; 2000. p. 268–279. Available from: http://dx.doi.org/10.1145/351240.351266.

[78] Arts T, Hughes J, Johansson J, Wiger U. Testing telecoms software with quviq QuickCheck. In: ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang. New York, NY, USA: ACM; 2006. p. 2–10. Available from: http://dx.doi.org/10.1145/1159789.1159792.

[79] Which programming languages are fastest?;. Available from: http://shootout.alioth.debian.org/u64q/which-programming-languages-are-fastest.php?gcc=on\&gpp=on\&scala=on\&ghc=on\&sbcl=on\&ocaml=on\&fsharp=on\&hipe=on\&erlang=on\&calc=chart.

[80] Peyton-Jones S. Haskell 98 language and libraries : the revised report. Cambridge U.K. New York: Cambridge University Press; 2003. Available from: http://www.worldcat.org/isbn/9780521826143.

[81] Beck K. Test-driven development : by example. Boston: Addison-Wesley; 2003.

[82] LeCun Y, Cortes C. MNIST handwritten digit database; 2010 [cited 2010-02-22 00:36:34]. Available from: http://yann.lecun.com/exdb/mnist/.

[83] LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-Based Learning Applied to Document Recognition. In: Proceedings of the IEEE; 1998. p. 2278–2324. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665.

[84] LeCun Y, Bottou L, Orr G, Müller K. Efficient BackProp. In: Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science. Springer Berlin / Heidelberg; 1998. p. 546. Available from: http://dx.doi.org/10.1007/3-540-49430-8_2.

[85] Ruiz A. hmatrix. codehaskellorg. 2010 [cited 2010-06-20 22:42:44];Available from: http://code.haskell.org/hmatrix/.

[86] Ruiz A. A simple scientific library for Haskell; 2009. Available from: http://code.haskell.org/hmatrix/hmatrix.pdf.

[87] Galassi M. GNU Scientific Library : reference manual for GSL version 1.12. Network Theory; 2009.

[88] Foundation NS, of Energy D. BLAS. Netlib Repository at UTK and ORNL. 2010 [cited 2010-06-20 22:47:52];Available from: http://www.netlib.org/blas/.

[89] Dongarra J. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard. International Journal of High Performance Computing Applications. 2002 Feb;16(1):1. Available from: http://dx.doi.org/10.1177/10943420020160010101.

[90] Foundation NS, of Energy D. LAPACK – Linear Algebra PACKage. Netlib Repository at UTK and ORNL. 2010 [cited 2010-06-20 22:45:15];Available from: http://www.netlib.org/lapack/.

[91] Anderson E. LAPACK users' guide. 3rd ed. Philadelphia: Society for Industrial and Applied Mathematics; 1999.

[92] Hristev RM. The ANN Book. 1st ed. [publisher unknown]; 1998. Available from: ftp://ftp.informatik.uni-freiburg.de/papers/neuro/ANN.ps.gz.

[93] Sommerville I. Software engineering. International computer science series. Addison-Wesley; 2007. Available from: http://books.google.ie/books?id=B7idKfL0H64C.

[94] Bullock S, Noble J, Watson R, Bedau MA, editors. Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems. MIT Press, Cambridge, MA; 2008.

Epigraphs

Chapter 1: Our understanding of the universe will be severely limited until we have a more definitive view of how much life and consciousness can be explained as emergent phenomena. John Holland [14, p. 248].

Chapter 2: The ALife-AI claim is, "The dumbest smart thing you can do is stay alive." That is, ALife represents a lower bound for AI. Richard K. Belew [70].

Appendix A

Methodology

This chapter discusses the methodology used in the Créatúr research project.

A.1 Initial research objectives

Originally, this project was conceived as an AI project, without an ALife component. The research objectives defined at the beginning of the project were to:

1. Build a high-level computer model of SOMs with neuronal copying, incorporating the approach outlined by Fernando et al.Chrisantha Fernando [43], and using SOM maps as described by Kohonen.
2. Using a suitable geospatial data set, train the model to discover relationships in the input data, and to make predictions based on that data. Investigate the following using the computer model:
    - Determine whether or not the model makes useful and reliable predictions.
    - Identify the computational requirements (including time, processing power and data storage).
    - Define criteria which enable the user to tell when the model is getting "close" to a solution.
    - Identify the conditions under which the model follows "dead ends", discarding higher-quality solutions in favour of ones that seem promising but which don't pan out.
3. Investigate whether all three mechanisms (A, B and C) proposed by Fernando et al. are required for obtaining useful results.

4. Train both a traditional artificial neural network and a program using a traditional genetic algorithm with the same geospatial data, and use it to make similar predictions. Compare the accuracy of the predictions with those from the model with neuronal copying. Also compare the processing requirements, length of time required, practicality, etc.

A.2 Exploratory literature search

The first step was a exploratory search of the literature to understand the current state of the art with respect to AI and Neural Darwinism. From this starting point, promising lines of enquiry were followed into biology, philosophy, neuroscience, and ALife, while maintaining the focus on how evolutionary processes could be used to synthesise intelligence and create a system that would explore data and discover patterns.

A.3 Research question

As a result of the literature review, the research question emerged: Would an AI project that incorporates some form of Neural Darwinism yield a more intelligent artificial brain?

A.4 Final research objectives

Also as a result of the literature review, the objectives of the project evolved. The author became convinced that intelligence in the abstract, separate from the problems of survival, might be too difficult to define, and was therefore not a useful concept. As a result, the project was re-conceived, as an ALife project with an AI component. The final objectives of the Créatúr research project were:

1. To evolve an ALife population with sufficient intelligence to discover patterns in data, and to make survival decisions based on those patterns.

2. To create a population that adapts to its environment through both evolution and lifetime learning.

3. To evolve an ALife population with some general-purpose problem solving skills, that could be used as "seeds" for projects requiring specialised skills. It should be feasible to create a specialised population by starting with an existing population of animats with basic intelligence and introducing new survival challenges gradually until the animats have developed the new skills

required. furthermore, it should be faster to do this than it would be to evolve a specialised population "from scratch".

A.5 Approach

Based on the initial review of the literature, a possible way to answer the research question emerged: evolve an ALife population with sufficient intelligence to discover patterns in data, and to make survival decisions based on those patterns. This approach, and the rationale behind it, is discussed in more detail in Chapter 3.

A.6 Focused literature search

With an approach identified for answering the research question, the next step was a more focused examination of the literature to find techniques and approaches that might be incorporated in the design, implementation, and testing. The ideas that seemed most promising were presented in Chapter 2 and summarised in Section 2.7.

A.7 Requirements

The ideas gathered from the literature were then translated into a set of requirements. The decision was made to split the development into two parts, *a*) a reusable framework which would be responsible for scheduling and running events such as eating, mating, and metabolism, and *b*) the animat implementation. The requirements are documented in Appendix B.

A.8 Pilot project

In order to gain familiarity with, and assess the suitability of, some of the technologies and tools considered for use in Créatúr, a smaller application was developed as a pilot project. The pilot project is described in Chapter 4. At the conclusion of the pilot project, the tools and technologies for Créatúr were finalised.

A.9 Framework

Next, the framework was implemented and tested. The framework is described in Chapter 5.

A.10 Evolving a species

The first animat species, the dote, was designed, implemented and tested. Next, the species was installed into the framework, and an initial population was generated and allowed to evolve. Over the course of several trials, the process for establishing a stable population was refined and made more efficient. Basic tools were developed as needed to extract and summarise information from the logs and data files. The design and implementation of the dote species is described in Chapter 6.

While analysing the behaviour, learning, and evolution of the dotes, several areas for improvement were identified, primarily in the design of the brain. When it appeared that the dotes would not achieve the research objectives during the time allotted for the project, the decision was made to develop a new species to incorporate these ideas.

Based on the analysis of the results obtained with the dotes, a new animat species, the wain, was designed, implemented and tested. The species was installed into the framework, and an initial population was generated and allowed to evolve. More sophisticated analysis tools and improved logging made it easier to monitor the progress of the species. Adaptation and learning were observed both during the lifetime of an individual, and over generations. The design and implementation of the wain species is described in Chapter 7.

Appendix B

Requirements

This appendix defines the requirements for the artificial life software at the core of this research project. This information is included in order to document the development process. The process used to develop software for the Créatúr project is what the computer scientist Ian Sommerville terms *evolutionary development*:[1] [93, p. 68f] an initial implementation was developed, evaluated, and then refined as needed.

B.1 User needs

The objectives defined in Section 3.1 are summarised below, reworded slightly for convenience in tracing requirements. Here and throughout this chapter, uppercase strings (e.g., OBJ-1, ALIFE-EAT) are used as unique IDs, which are referenced in the traceability matrices.

OBJ-1 To evolve an ALife population…

OBJ-1 …with sufficient intelligence to discover patterns in data, and to make survival decisions based on those patterns.

OBJ-1 …that adapts to its environment through both evolution and lifetime learning.

OBJ-1 …with some general-purpose problem solving skills, that could be used as "seeds" for projects requiring specialised skills.

As can be seen from these objectives, Créatúr is *structured* as an ALife project. Some of the user needs for Créatúr are common to most ALife projects; these are listed below, and assigned unique IDs for convenience in tracing requirements.

[1] Not to be confused with evolutionary programming or evolutionary computation.

ALIFE-EAT In order for an animat to survive, it must eat.

ALIFE-MATE In order for the population to thrive, animats must mate.

ALIFE-GENE In order for the population to evolve, animat reproduction must involve gene mixing, mutation, or both.

ALIFE-OPP In order for the ecosystem to be viable, it must provide opportunities for animats to eat and mate.

ALIFE-INIT The system should provide a way to generate an initial population.

ALIFE-DAEMON ALife experiments may run for days or months, so the system should run as a background daemon that can be started, stopped, and re-started.

ALIFE-DATA In order to be usable for research, the system must provide access to information about the animats while the experiment runs.

To this list, we can add the key elements of the approach defined in Section 3.2.

APR-AI-ALIFE Combine AI and ALife. (See Section 3.2.1.)

APR-DATA-ENV Use the data as the environment. (See Section 3.2.2.)

APR-DATA-SUR Use data analysis as a survival problem. (See Section 3.2.3.)

APR-MULT-EVO Use multiple kinds of evolution. (See Section 3.2.4.)

APR-NO-FIT No fitness function except survival. (See Section 3.2.5.)

APR-NO-FLUNCH No free lunch. (See Section 3.2.6.)

APR-NURTURE Protect the young while they learn. (See Section 3.2.7.)

APR-DIPLOID Use diploid animats. (See Section 3.2.8.)

APR-KINSHIP Provide a means for animats to estimate degrees of kinship. (See Section 3.2.9.)

The user needs for Créatúr include both lists. Table B.1 traces the user needs to the objectives.

Table B.1: User needs traceability matrix

	OBJ-0	OBJ-1	OBJ-2	OBJ-3
ALIFE-EAT	x			
ALIFE-MATE	x			
ALIFE-GENE	x			
ALIFE-OPP	x			
ALIFE-INIT	x			
ALIFE-DAEMON	x			
ALIFE-DATA	x			
APR-AI-ALIFE		x		
APR-DATA-ENV		x		x
APR-DATA-SUR		x		x
APR-MULT-EVO			x	
APR-NO-FIT	x			
APR-NO-FLUNCH	x			
APR-NURTURE		x	x	x
APR-DIPLOID	x			
APR-KINSHIP	x			

B.2 Requirements

The requirements for the Créatúr software are discussed below. Créatúr consists of a framework for automating ALife experiments, plus a user-provided implementation for the animats and other objects in the virtual environment. Requirements allocated to the framework itself have IDs that begin with the prefix FRAME-; Table B.2 maps these requirements to the user needs that they are derived from. Requirements that the user implementation must satisfy have IDs that begin with the prefix USR-; Table B.3 maps these requirements to the framework and user needs that they are derived from.

B.2.1 Appearance

In accordance with the principle of *using data as the environment* [APR-DATA-ENV], objects in Créatúr are created from byte strings, which can be thought of as representing that object's properties. Some or all of these properties will be available to an animat's senses, as will be discussed in Section B.2.2. The properties that are available to the senses are collectively referred to as the *appearance* of the object. This term is used for convenience; there is nothing inherently visual about

these properties. In fact, Créatúr could be used to model animats with multiple senses, including those found in nature (such as sight, hearing, smell, taste, touch, echolocation, or the ability to detect electric fields), or even "invented" senses (such as an ability to detect molecular structure).

Just as the objects in the environment have an appearance, so do the animats themselves. In accordance with the principle of *providing a means for animats to estimate degrees of kinship* [APR-KINSHIP], the appearance of the animats is genetically determined. While an object's appearance is fixed, an animat's appearance may depend partly on its current state (e.g., whether or not it is currently rearing a child).

Thus, the requirements listed below were defined.

FRAME-APPEAR The appearance of an object or an animat shall be represented by a byte string.

USR-OBJAPP When an object is created, the implementation shall specify the object's appearance.

USR-ANIAPP The implementation shall provide a method to return an animat's appearance.

USR-APPGEN The appearance of an animat shall be primarily genetically determined.

B.2.2 Senses

In accordance with the principle of *combining AI and ALife* [APR-AI-ALIFE], each animat will have a brain to make decisions that affect the animat's survival. The brain will need information about the environment, and the animat's current state.

As discussed in Section B.2.1, the appearance of an object or an animat is represented as a byte string, so a survival problem such as identifying food requires some form of data analysis. This is in accordance with the principle of *using data analysis as a survival problem* [APR-DATA-SUR].

Thus, the requirements listed below were defined. The phrase "animat's current state" is deliberately left undefined; the information that should be included depends on the implementation and the nature of the experiment.

FRAME-OBJSEN In an encounter between an animat and an object, Créatúr shall present the object's appearance to the animat's senses.

FRAME-ANISEN In an encounter between two animats, Créatúr shall present each animat's appearance to the other animat's senses.

FRAME-NCHEAT The only information provided to an animat shall be provided via the senses.

USR-EXTERO The implementation shall provide methods to input information from the environment (exteroception).

USR-INTERO When the implementation feeds sensory data to an animat, it shall add information about the animat's current state (interoception).

B.2.3 Encounters

An ALife habitat must provide opportunities [ALIFE-OPP] for the animats to eat and mate. These events are scheduled by the software. In order to simulate the irregular availability of these opportunities in the biological world, the objects and animats for each encounter are selected at random.

A program counter is used to schedule these encounters. The advantage of using a counter rather than system clock time is that it ensures that the frequency of food and mating opportunities is not affected by the amount of processing time allocated to the Créatúr daemon, or by stopping and restarting the daemon. It also allows meaningful comparison of experiments performed on computer systems with different hardware and processing capacity.

Thus, the requirements listed below were defined.

FRAME-ENC Créatúr shall schedule all encounters between animats, and of animats with objects.

FRAME-RANOBJ Créatúr shall provide a mechanism to select a random object from the environment.

FRAME-RANANI Créatúr shall provide a mechanism to select a random animat from the population.

FRAME-COUNT Events in Créatúr shall be scheduled using a program counter which only advances when the program is running.

B.2.4 Decisions

In accordance with the principle of *combining AI and ALife* [APR-AI-ALIFE], the animats will have the ability to make decisions that affect their survival and reproduction. Thus, the requirements listed below were defined.

FRAME-DECIDE In any encounter, Créatúr shall read the animat(s) sensory outputs to determine the decision made by the animat(s).

FRAME-CONSEQ Créatúr shall implement the consequences of any decision made by an animat to eat, mate, play, or ignore.

USR-DECIDE The implementation shall provide methods to read an animat's decision.

USR-DELTA The implementation shall provide the following information about objects in the environment: delta energy if eaten, delta passion if flirted with, delta boredom if played with.

USR-CONSEQ The implementation shall provide methods to change an animat's energy, passion, or boredom levels.

B.2.5 Learning

In accordance with the principle of *combining AI and ALife* [APR-AI-ALIFE], the animats should have the ability to learn from their experiences. In accordance with the principle of *using multiple kinds of evolution* [APR-MULT-EVO], some form of Neural Darwinism should occur in the animat's brain. Thus, the requirements listed below were defined.

FRAME-THINK After an encounter, Créatúr shall give the animats involved the opportunity to reflect on the outcome and learn from it.

USR-NEURALD The implementation of the animat's brain shall incorporate a form of Neural Darwinism.

B.2.6 Eating

Animats must eat in order to survive [ALIFE-EAT]. Furthermore, in accordance with the principle of *framing data analysis as a survival problem* [APR-DATA-SUR], the only way for animats to acquire energy is by eating an object. Some objects may provide negative amounts of energy, making them poisonous. (However, eating poisonous food will only be fatal if it reduces the animat's energy to zero or below.) Thus, the requirements listed below were defined.

FRAME-EATNRG If an animat encounters an object and decides to eat it, Créatúr shall adjust the animat's energy by the amount of energy provided by the object.

B.2.7 Mating

In order to maintain the population, animats must mate [ALIFE-MATE]. In accordance with the principle that there should be *no free lunch* [APR-NO-FLUNCH], animats must give up some energy in order to try to attract a potential mate. This parallels the situation in biology where a male may have to fight off rivals, build a nest, perform a mating display, etc. This "flirting tax" may encourage the animats to choose mates that are more likely to be receptive. Thus, the requirements listed below were defined.

FRAME-FLIRT Flirting shall have an energy cost.

B.2.8 Reproduction

Evolution requires that animat reproduction involve gene mixing, mutation, or both [ALIFE-GENE]. In accordance with the principle of *using diploid animats* [APR-DIPLOID], the requirements listed below were defined.

FRAME-GAMETE If both animats decide to flirt, Créatúr shall generate a gamete from each parent's genes, applying crossover and random mutation. It shall then combine the gametes to produce the genome for the child, and then construct a child based on its genome.

USR-GENOME The implementation shall provide a method that returns an animat's genome.

USR-BUILD The implementation shall provide a method that builds an animat from a pair of gene sequences.

USR-EXPR The implementation shall provide a method which, given a pair of genes, determines how that gene shall be expressed in the animat.

USR-ENCODE The implementation shall provide methods to encode a gene sequence to a byte string, and decode a byte string into a gene sequence.

USR-GENRTR If a gene sequence is encoded to a byte string, and subsequently decoded, the result shall be identical to the starting sequence.

USR-DECALL All bytes strings shall be decodable into a gene sequence.

B.2.9 Birth

An animat must have some energy at birth (otherwise it would die immediately). In accordance with the principle that there should be *no free lunch* (see Section 3.2.6), the starting energy for the child is donated by the parents. This parallels biology where in some species, one or both parents must invest time and energy before their offspring are born (e.g., keeping eggs warm, providing extra nutrition for the mother). Thus, the requirements listed below were defined.

FRAME-CHINRG When an animat is born, a fraction of each parent's energy, specified by the parent's genome, shall be transferred to the child.

USR-DEVOTE The implementation shall provide a method to return the animat's parental devotion.

B.2.10 Child-rearing

In accordance with the principle of *protecting the young while they learn* [APR-NURTURE], the requirements listed below were defined.

FRAME-DAM When an animat is born, one parent shall arbitrarily be selected to be the dam. The child shall remain with the dam until the child matures or dies.

FRAME-CHIEXT In all encounters, a child shall receive the same external sense data as its parent.

FRAME-CHIINT In all encounters, a child shall receive sensory information about its current state.

FRAME-CHISHA A child shall share in the energy gains and losses of its dam, according to a ratio specified in the dam's genome.

FRAME-CHISEP When a child reaches maturity, as specified by its genome, it shall be separated from the dam.

USR-MATURE The implementation shall provide a method to indicate whether or not an animat is mature.

Note that requirement [USR-DEVOTE] defined in the previous section is also relevant here.

B.2.11 Metabolism

In accordance with the principle that there should be *no free lunch* (see Section 3.2.6), animats lose energy throughout their lives. This simulates a biological metabolism. Thus, the requirements listed below were defined. Note that the metabolism rate may not be constant; it might be based on factors that change, such as strength, or the number of neural connections.

FRAME-METAB At the clock intervals specified in the configuration file, Créatúr shall deduct energy from all animats, and increment each animat's age.

USR-METAB The implementation shall provide a method to calculate the energy cost of an animat's metabolism.

USR-AGE The implementation shall provide a method to increment an animat's age.

B.2.12 Population

Créatúr should perform some basic maintenance functions. One of these is to generate an initial population [ALIFE-INIT]. Consider that if a population falls too low, it is unlikely to recover. The user may need to investigate why this has occurred, which may be easier to do while the system is still running. For this reason, Créatúr should provide a way to replenish the population automatically. (This replenishment is intended only as a diagnostic aid. In this research project, any population that requires replenishment is considered to be unsuccessful; this is noted where the results are presented.)

Finally, in accordance with the principle that there should be *no fitness function except survival* [APR-NO-FIT], an animat only dies if its energy falls to zero. Thus, the requirements listed below were defined.

FRAME-INITPOP The software shall provide a mechanism to generate an initial population, with arbitrary genes, of a given size.

FRAME-MINPOP If the population falls below a pre-set minimum, a new animat with random genes shall be generated and added to the population.

FRAME-DEATH When an animat's energy falls to zero, it shall be removed from the population and archived.

USR-ARBGEN The implementation shall provide a method to produce an arbitrary gene sequence suitable for an animat in the starter population.

B.2.13 Automation and maintenance

Créatúr runs as a daemon [ALIFE-DAEMON]. It must be reliable and robust; the user should make the decision to halt a trial or continue, not the software. Thus, the requirements listed below were defined.

FRAME-DAEMON Créatúr shall run as a system daemon, with a mechanism to start, stop, or restart.

FRAME-NOHALT The software shall not throw exceptions or otherwise halt in response to an error.

FRAME-INDFILE Animats shall be saved in individual files.

FRAME-FILRTR The format of the save file shall be such that the animat read from it is identical to the original.

FRAME-SAVE After an animat has an encounter with an object or another animat, its state shall be saved.

USR-SER The implementation shall provide methods to serialise and deserialise animats.

USR-SERRTR If an animat is serialised and subsequently deserialised, the resulting animat shall be identical to the the original.

B.2.14 Analysis tools

Créatúr must provide access to information about the animats while the experiment runs [ALIFE-DATA]. Thus, the requirements listed below were defined.

FRAME-ID Créatúr shall assign a unique ID to all animats, whether born or generated.

FRAME-LOG Créatúr shall provide a logging mechanism.

FRAME-ROTATE Créatúr shall automatically rotate log files when they reach a specified size.

FRAME-STATS Créatúr shall periodically log statistical information about the population, including: population size, average age, average energy, average passion level, average length of genome, average parental devotion, average maturation time.

USR-OBJAPP When an object is created, the implementation shall specify a descriptive name for the object.

USR-STATS The implementation shall provide information about an animat, including: age, energy, passion level, length of genome, parental devotion, maturation time.

Table B.2: Framework requirements traceability matrix

FRAME-...	ALIFE-EAT	ALIFE-MATE	ALIFE-GENE	ALIFE-OPP	ALIFE-INIT	ALIFE-DAEMON	ALIFE-DATA	APR-AI-ALIFE	APR-DATA-ENV	APR-DATA-SUR	APR-MULT-EVO	APR-NO-FIT	APR-NO-FLUNCH	APR-NURTURE	APR-DIPLOID	APR-KINSHIP
APPEAR									x							
OBJSEN								x		x						
ANISEN								x		x						
NCHEAT								x		x						
ENC				x												
RANOBJ				x												
RANANI				x												
COUNT				x		x										
DECIDE								x								
CONSEQ								x								
THINK								x								
EATNRG	x															
FLIRT		x											x			
GAMETE			x												x	
CHINRG													x			
DAM														x		
CHIEXT														x		
CHIINT														x		
CHISHA														x		
CHISEP														x		
METAB														x		
INITPOP					x											
MINPOP																
DEATH												x				
DAEMON						x										
NOHALT						x										
INDFILE						x										
FILRTR						x										
SAVE						x										
ID							x									
LOG							x									
ROTATE							x									
STATS							x									

Table B.3: User implementation requirements traceability matrix. Columns without x's have been eliminated to save space.

	ALIFE-GENE	APR-AI-ALIFE	APR-DATA-ENV	APR-DATA-SUR	APR-MULT-EVO	APR-NURTURE	APR-KINSHIP	FRAME-APPEAR	FRAME-APPGEN	FRAME-OBJSEN	FRAME-DECIDE	FRAME-CONSEQ	FRAME-EATNRG	FRAME-GAMETE	FRAME-CHINRG	FRAME-DAM	FRAME-METAB	FRAME-INITPOP	FRAME-NOHALT	FRAME-FILRTR	FRAME-SAVE	FRAME-LOG	FRAME-STATS
USR-OBJAPP			x					x															
USR-ANIAPP							x	x															
USR-APPGEN							x	x	x														
USR-EXTERO		x		x						x													
USR-INTERO		x								x													
USR-DECIDE											x												
USR-DELTA												x	x										
USR-CONSEQ												x											
USR-NEURALD					x																		
USR-GENOME														x									
USR-BUILD	x													x									
USR-EXPR	x													x									
USR-ENCODE	x													x									
USR-GENRTR	x													x									
USR-DECALL																		x					
USR-DEVOTE															x								
USR-MATURE						x										x							
USR-METAB																	x						
USR-AGE																	x						
USR-ARBGEN																		x					
USR-SER																					x		
USR-SERRTR																				x			
USR-OBJAPP																						x	
USR-STATS																							x

Appendix C

Dote Genome

C.1 Gene encoding

The dote genome consists of instructions encoded as a series of bytes. The first byte indicates the type of instruction, or *gene*, as shown in Table C.1.

Table C.1: Dote genes

byte 0	Gene
0	devotion gene
1	maturation time gene
2	start neuron gene
3	learning rule gene
4	forgetting rule gene
5	connection source gene
6	end neuron gene
7	colour gene
8	thinking time gene
9 to 0xFF	no-op gene

The byte indicating the gene type may be followed by one or more bytes of data. When the string is converted to a list of genes, one or more bytes containing zeroes are added to the end if needed to fill in the missing data for the last gene in

the list. Thus, all byte strings are valid gene sequences; a dote can be constructed from any byte sequence. The format of each gene is explained below.

C.1.1 Devotion gene

Table C.2 shows the encoding of the *devotion gene*. The parameter *devotion* is a number between 0 and 255 which controls how much energy parents share with offspring. When the animat is created, this value is divided by 255 to yield a fraction between 0 and 1.

Table C.2: Devotion gene

byte 0	byte 1
0	*devotion*

C.1.2 Maturation time gene

Table C.3 shows the encoding of the *maturation time gene*. The parameter *time* is a 16-bit integer specifying the age at which the animat becomes mature and is separated from the dam.

Table C.3: Maturation time gene

byte 0	byte 1	byte 2
1	*time*	

C.1.3 Colour gene

Table C.4 shows the encoding of the *colour gene*. The parameters *red*, *green*, and *blue* are numbers between 0 and 255.

Table C.4: Colour gene

byte 0	byte 1	byte 2	byte 3
7	*red*	*green*	*blue*

C.1.4 Start neuron gene

Table C.5 shows the encoding of the *start neuron gene*.

Table C.5: Start neuron gene

byte 0
2

C.1.5 End neuron gene

Table C.6 shows the encoding of the *end neuron gene*.

Table C.6: End neuron gene

byte 0
6

C.1.6 Learning rule gene

Tables C.7, C.8, and C.9 show the encoding of the *learning rule gene*. The second byte indicates the learning rule. Although the values 3 to 0xFF are reserved for future learning rules, currently these values will select the "No Learning" rule. The parameter η specifies the learning rate. It is a number between 0 and 255. When the animat is created, this value is divided by 255 to yield a fraction between 0 and 1.

Table C.7: Learning rule gene: Oja's rule

byte 0	byte 1	byte 2
3	1	η

Table C.8: Learning rule gene: Hebb's rule

byte 0	byte 1	byte 2
3	0	η

Table C.9: Learning rule gene: "No Learning"

byte 0	byte 1
3	2

C.1.7 Forgetting rule gene

Tables C.10 and C.11 show the encoding of the *forgetting rule gene*. The second byte indicates the forgetting rule. The values 2 to 0xFF are reserved for future learning rules. Currently, however, these values will select the "No Forgetting" rule. The parameter ρ specifies the forgetting rate. It is a number between 0 and 255. When the animat is created, this value is divided by 255 to yield a fraction between 0 and 1.

Table C.10: Forgetting rule gene: basic forgetting rule

byte 0	byte 1	byte 2
4	0	ρ

Table C.11: Forgetting rule gene: "No Forgetting"

byte 0	byte 1
4	1

C.1.8 Connection source gene

Table C.12 shows the encoding of the *connection source gene*. The parameter *source* is a 16-bit integer specifying the index number of the source neuron.

Table C.12: Connection source gene

byte 0	byte 1	byte 2
5	*source*	

C.1.9 Thinking time gene

Table C.13 shows the encoding of the *thinking time gene*. The parameter *time* is an 8-bit integer specifying the number of times the brain state is recalculated.

Table C.13: Thinking time gene

byte 0	byte 1
8	*time*

C.1.10 No-op gene

Table C.14 shows the encoding of the *No-op gene*. A byte that cannot be interpreted as part one of the genes described above will be treated as a no-op instruction.

Table C.14: No-op gene

byte 0
9 to 0xFF

C.2 Genetic dominance

Tables C.15 and C.16 illustrate how pairs of genes are expressed as a single instruction for building an animat.

Table C.15: Relation between alleles and genotype

allele 1	allele 2	blueprint instruction
NoopGene	NoopGene	NoopGene
DevotionGene x	DevotionGene y	DevotionGene $\frac{x+y}{2}$
MaturationTimeGene x	MaturationTimeGene y	MaturationTimeGene $\frac{x+y}{2}$
ThinkingTimeGene x	ThinkingTimeGene y	ThinkingTimeGene $\frac{x+y}{2}$
ColourGene r_1 g_1 b_1	ColourGene r_2 g_2 b_2	ColourGene $\frac{r_1+r_2}{2}$ $\frac{g_1+g_2}{2}$ $\frac{g_1+g_2}{2}$
StartNeuronGene	StartNeuronGene	StartNeuronGene
LearningRuleGene lr_1	LearningRuleGene lr_2	See table C.16
EndNeuronGene	EndNeuronGene	EndNeuronGene
ConnectionSourceGene x	ConnectionSourceGene x	ConnectionSourceGene $min(x, y)$
any other combination		NoopGene

Table C.16: Expression of learning rule genes

		Parent 2		
		Hebb y	Oja y	No learning
Parent 1	Hebb x	Hebb $\frac{x+y}{2}$	Oja y	No learning
	Oja x	Oja x	Oja $\frac{x+y}{2}$	Oja x
	No learning	No learning	Oja y	No learning

Appendix D

Wain Genome

D.1 Gene encoding

The wain genome consists of instructions encoded as a series of bytes. The first byte indicates the type of instruction, or *gene*, as shown in Table D.1.

Table D.1: Wain genes

byte 0	Gene
0	devotion gene
1	maturation time gene
2	exteroception capacity gene
3	interoception capacity gene
4	pattern capacity gene
5	pattern learning rate gene
6	pattern learning rate decay gene
7	positive learning rate gene
8	negative learning rate gene
9	decider forgetting rate gene
10	Edelman cycle gene
11	appearance gene
12 to 0xFF	no-op gene

The byte indicating the gene type may be followed by one or more bytes of data. When the string is converted to a list of genes, one or more bytes containing zeroes are added to the end if needed to fill in the missing data for the last gene in the list. Thus, all byte strings are valid gene sequences; a wain can be constructed from any byte sequence. The format of each gene is explained below.

D.1.1 Devotion gene

Table D.2 shows the encoding of the *devotion gene*. The parameter *devotion* is a number between 0 and 255. When the animat is created, this value is divided by 255 to yield a fraction between 0 and 1.

Table D.2: Devotion gene

byte 0	byte 1
0	*devotion*

D.1.2 Maturation time gene

Table D.3 shows the encoding of the *maturation time gene*. The parameter *time* is a 16-bit integer specifying the age at which the animat becomes mature and is separated from the dam.

Table D.3: Maturation time gene

byte 0	byte 1	byte 2
1	*time*	

D.1.3 Exteroception capacity gene

Table D.4 shows the encoding of the *exteroception capacity gene*. The parameter *exteroceptionCapacity* is a 16-bit integer specifying the number of external inputs; which is also the length of the input vector to the SOM.

Table D.4: Exteroception capacity gene

byte 0	byte 1	byte 2
2	*exteroceptionCapacity*	

D.1.4 Interoception capacity gene

Table D.5 shows the encoding of the *interoception capacity gene*. The parameter *interoceptionCapacity* is an 8-bit integer specifying the number of internal inputs. The length of the input vector to the Decider is the sum of the *interoceptionCapacity* value and the *patternCapacity* value specified by the pattern capacity gene.

Table D.5: Interoception capacity gene

byte 0	byte 1
3	*interoceptionCapacity*

D.1.5 Pattern capacity gene

Table D.6 shows the encoding of the *pattern capacity gene*. The parameter *patternCapacity* is an 8-bit integer specifying the number of internal inputs. The length of the input vector to the Decider is the sum of the *patternCapacity* value and the *interoceptionCapacity* value specified by the interoception capacity gene.

Table D.6: Pattern capacity gene

byte 0	byte 1
4	*patternCapacity*

D.1.6 Pattern learning rate gene

Table D.7 shows the encoding of the *pattern learning rate gene*. The parameter *rate* is a number between 0 and 255. When the animat is created, this value is divided by 100 to yield a number between 0 and 2.55.

Table D.7: Pattern learning rate gene

byte 0	byte 1
5	*rate*

D.1.7 Pattern learning rate decay gene

Table D.8 shows the encoding of the *pattern learning rate decay gene*. The parameter *rate* is a number between 0 and 255. When the animat is created, this value is divided by 255 to yield a fraction between 0 and 1.

Table D.8: Pattern learning rate decay gene

byte 0	byte 1
6	*rate*

D.1.8 Edelman cycle gene

Table D.9 shows the encoding of the *Edelman cycle gene*. The parameter *ticks* is a number between 0 and 255.

Table D.9: Pattern learning rate gene

byte 0	byte 1
11	*ticks*

D.1.9 Decider learning rate genes

Table D.10 shows the encoding of the *positive learning rate gene*. Table D.11 shows the encoding of the *negative learning rate gene*. The parameter *rate* is a number between 0 and 255. When the animat is created, this value is divided by 10 to yield a number between 0 and 25.5.

Table D.10: Positive decider learning rate gene

byte 0	byte 1
7	*rate*

Table D.11: Negative decider learning rate gene

byte 0	byte 1
8	*rate*

D.1.10 Decider forgetting rate gene

Table D.12 shows the encoding of the *decider forgetting rate gene*. The parameter *rate* is a number between 0 and 255. When the animat is created, this value is divided by 10 to yield a number between 0 and 25.5.

Table D.12: Decider forgetting rate gene

byte 0	byte 1
9	*rate*

D.1.11 Appearance Gene

Table D.13 shows the encoding of the *appearance gene*. The parameter *pixel* is a number between 0 and 255 indicating a grey-scale value in the wain's appearance.

Table D.13: Appearance Gene

byte 0	byte 1
10	*pixel*

D.1.12 No-op gene

Table D.14 shows the encoding of the *no-op gene*. A byte that cannot be interpreted as part one of the genes described above will be treated as a No-op instruction.

Table D.14: No-op gene

byte 0
9 to 0xFF

D.2 Genetic dominance

Table D.15 indicates how pairs of alleles result in a single instruction being added to the blueprint.

Table D.15: Relation between alleles and genotype

allele 1	allele 2	blueprint instruction
NoopGene	NoopGene	NoopGene
DevotionGene x	DevotionGene y	DevotionGene $\frac{x+y}{2}$
MaturationTimeGene x	MaturationTimeGene y	MaturationTimeGene $min(x, y)$
ExteroceptionCapacity-Gene x	ExteroceptionCapacity-Gene y	ExteroceptionCapacity-Gene $min(x, y)$
InteroceptionCapacity-Gene x	InteroceptionCapacity-Gene y	InteroceptionCapacity-Gene $min(x, y)$
PatternCapacityGene x	PatternCapacityGene y	PatternCapacityGene $min(x, y)$
PatternLearningRate-Gene x	PatternLearningRate-Gene y	PatternLearningRate-Gene $\frac{x+y}{2}$
PatternLearningRate-DecayGene x	PatternLearningRate-DecayGene y	PatternLearningRate-DecayGene $\frac{x+y}{2}$
PositiveDecider-LearningRateGene x	PositiveDecider-LearningRateGene y	PositiveDecider-LearningRateGene $\frac{x+y}{2}$
NegativeDecider-LearningRateGene x	NegativeDecider-LearningRateGene y	NegativeDecider-LearningRateGene $\frac{x+y}{2}$
DeciderForgettingRate-Gene x	DeciderForgettingRate-Gene y	DeciderForgettingRate-Gene $\frac{x+y}{2}$
AppearanceGene x	AppearanceGene y	AppearanceGene $\frac{x+y}{2}$
any other combination		NoopGene