Data Mining Using Artificial Life with Popperian-level Intelligence

by

Amy de Buitléir

A thesis submitted in partial fulfilment of the

requirements for the Ph.D. in

Software Engineering

Athlone Institute of Technology

2017

Supervisors: Mark Daly and Michael Russell

Faculty of Engineering and Informatics

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of PhD 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 **6**

Acknowledgements **7**

List of Figures **8**

List of Tables **10**

Preface **11**

1 Introduction **13**

1.1 Document structure 15
1.2 Major contributions 18
1.3 Conventions used in this thesis 20

2 Literature review **21**

2.1 Artificial Intelligence 21
2.2 Artificial Life 26
2.3 Evolution 27
2.4 Dennett's Tower of Generate-and-Test 28
2.5 Reproduction 32
2.6 Gene expression 33
2.7 Data mining 34
2.8 Agent-based approaches to data mining 37
2.9 The SOM algorithm 38
2.10 Wains 40
2.11 Automatic speech recognition 44
2.12 Functional programming concepts 46
2.12.1 Domain-specific languages 47
2.12.2 Monads 47
2.12.3 Datatype-generic programming 48

2.13 Data sets . . . 49
2.13.1 MNIST . . . 49
2.13.2 TI46 . . . 50
2.13.3 ISP traffic . . . 50
2.13.4 New York City weather data . . . 51
2.14 Summary . . . 52

3 Approach **57**

4 Improving the Créatúr framework **60**
4.1 Artificial Life genetics and recombination . . . 62
4.2 Gene encoding . . . 65
4.3 Reproduction . . . 68
4.4 Gene recombination . . . 71
4.5 Gene expression . . . 73
4.6 Constructing an agent from its *genome* . . . 76
4.7 Limitations . . . 82
4.8 Summary . . . 83

5 Improving the wain **85**
5.1 Architecture . . . 85
5.2 Condition . . . 88
5.3 Metabolism . . . 91
5.4 Appearance . . . 91
5.5 Brain . . . 92
5.6 Genetics and reproduction . . . 93
5.7 Child-rearing . . . 95
5.8 Summary . . . 95

6 The Self-generating model **97**
6.1 Self-Generating Model . . . 102
6.2 Experimental set-up . . . 103
6.2.1 Experiment 1: Early accuracy . . . 104
6.2.2 Experiment 2: Full training run . . . 106
6.3 Results and interpretation . . . 107
6.3.1 Experiment 1: Early accuracy . . . 107
6.3.2 Experiment 2: Full training run . . . 107
6.4 Summary . . . 112

7 **Improving the brain** **114**
7.1 Seeking inspiration 114
7.2 Making decisions 115
7.3 The SGM and genetics 117
7.4 A new brain architecture 118
7.5 Summary 124

8 **Classification with wains** **125**
8.1 Experimental set-up 127
8.1.1 Classifying images 128
8.1.2 Classifying audio samples 129
8.1.3 Training and testing 133
8.2 Results and interpretation 134
8.3 Summary 136

9 **Forecasting with wains** **138**
9.1 Experimental set-up 138
9.1.1 Predicting ISP traffic 139
9.1.2 Predicting weather 143
9.2 Results and interpretation 147
9.3 Summary 152

10 **Conclusions** **155**
10.1 Synopsis 155
10.2 Conclusions 159
10.3 Future directions 160
10.3.1 Making decisions and managing systems 161
10.3.2 Improved configurations 162
10.3.3 Smarter agents 163

Glossary **164**

Acronyms **178**

Bibliography **179**

Appendices **205**

A **An introduction to Haskell** **206**

B **A heuristic approach to configuring experiments with wains** **209**

Abstract

This thesis proposes a strategy for developing Artificial Life (ALife) agents with Artificial Intelligence (AI) that are capable of performing a variety of data mining tasks, and demonstrates it using agents called wains.

There is a continuing need for new data mining techniques, especially to cope with data patterns that change over time, or to process data streams in real-time. While data mining has adopted heavily from AI (especially as regards machine learning), the use of ALife has been restricted to agents with only rudimentary intelligence (e.g., swarm intelligence, ant colony optimisation). These Skinnerian creatures (in Dennett's terminology) only learn through operant conditioning, a trial-and-error method, which means they are likely to try a number of "stupid" moves before discovering the "smart" moves.

Wains live in an environment of pure data; for them, discovering patterns in data is a survival problem. They were also Skinnerian creatures. In this research project they were given the ability to decide which actions are worth evaluating and to predict the outcome of an action, making them Popperian creatures (in Dennett's terminology). The new wains were shown to be capable of extracting knowledge from complex data sets in a variety of domains, including images, audio samples, ISP traffic, and weather. Based on the literature review, this is the first time an ALife species with Popperian-level AI has been applied to *data mining*. This is a new direction, but a promising one, as shown by the experimental results presented in this thesis.

Acknowledgements

Many people have contributed in various ways to my research over the years, and to this thesis. First, I would like to thank my supervisors, Dr. Mark Daly and Mike Russell at AIT, for their guidance and enthusiastic support. I enjoyed some "geek pilgrimages" to the annual Dublin Maker Faire with Mark, and many fruitful lunchtime discussions over Thai food. Mike helped me identify my dreams and make them come true. His keen editorial eye taught me to be a better writer. I would also like to thank my external supervisor, Prof. Daniel Heffernan at Maynooth University, for his suggestions and warm encouragement.

I have been fortunate to interact regularly with the talented researchers at Ericsson in Athlone. Dr. Sven van der Meer gave me loads of excellent advice about research and publication. The many lunchtime discussions we had about wains always filled me with energy, enthusiasm, and confidence. Dr. John Keeney could always be relied on for practical, down-to-earth advice; he helped me keep my sense of humour. I enjoyed several philosophical discussions with Dr. MingXue Wang about the nature of artificial life. I am also grateful to John O'Regan, also at Ericsson, who arranged the funding for my PhD, and provided time and resources to support my research.

I would also like to thank Ronan Flynn at AIT for suggesting that I apply wains to the task of automated speech recognition, provided a data set to work with, and directed me to resources for learning about ASR.

List of Figures

1.1 Key topic dependencies 16

2.1 Topic map 22
2.2 Dennet's Tower of Generate-and-Test 29
2.3 Sample images from the MNIST database 50

4.1 Crossover 71
4.2 Cut-and-splice 71

5.1 Architecture for a typical experiment. 86

6.1 Decision-making using a classifier that preserves topology. . . . 98
6.2 Decision-making using a classifier that does not preserve topology. 99
6.3 Early accuracy comparison 108
6.4 Small SOM after all training images have been presented. 109
6.5 Small SGM after all training images have been presented. 109
6.6 Model stability in SOM and SGM. 110
6.7 Model usage in SOM and SGM. 111
6.8 Processing time for SOM and SGM. 111
6.9 SOM and SGM accuracy 112

7.1 The decision-making process. 119
7.2 A simple *p-score* function. 120
7.3 A more sophisticated p-score function. 120

8.1 Architecture for classifying images and audio samples. 128

9.1 Architecture for predicting stream data. 139
9.2 SMA forecasting of ISP traffic with different window widths. . . 147
9.3 WMA forecasting ISP traffic with different window widths. . . . 148
9.4 Error in ISP traffic *prediction*. 149
9.5 SMA forecasting of next day's high temperature with different window widths. 150

9.6 WMA forecasting of next day's high temperature with different window widths. 150
9.7 Error in next day's high temperature prediction. 151

List of Tables

2.1 Missing records in New York City weather data 51

4.1 The Recombination DSEL. 74
4.2 The Reader DSEL. 77
4.3 The DiploidReader DSEL. 80

6.1 Configuration of SOM and SGM in Experiment 1. 105
6.2 Analysis of MAD between MNIST images. 106
6.3 Configuration of SOM and SGM in Experiment 2. 107

7.1 Example of a signature 118
7.2 Unnormalised probabilities for signature in Table 7.1. 121
7.3 Normalised probabilities from Table 7.2. 121
7.4 Hypotheses based on probabilities in Table 7.3. 122
7.5 Sample payoff matrix. 123

8.1 Configuration for working with MNIST images 130
8.2 Examples of stretching, illustrated using a character string . . . 131
8.3 Configuration of brain for working with audio samples. 132
8.4 Comparison of image classification results. 134
8.5 Comparison of audio classification results. 135

9.1 Configuration for predicting ISP traffic. 141
9.2 Gene pool of initial population for predicting ISP traffic. 142
9.3 Configuration for predicting next day's high temperature. . . . 145
9.4 Gene pool of initial population for predicting next day's high temperature. 146

Preface

Some passages in this thesis have been quoted verbatim from the papers listed below, which were written during my PhD research.

Paper I. Amy de Buitléir, Mark Daly, Michael Russell, and Daniel Heffernan. A functional approach to sex: Reproduction in the Créatúr framework. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming: 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014.* Revised Selected Papers, volume 8843 of *Lecture Notes in Computer Science*, pages 68-83. Springer International Publishing, 2015. ISBN 978-3-319-14674-4.

Paper II. Amy de Buitléir, Mark Daly, and Michael Russell. The Self-Generating Model: an Adaptation of the Self-organizing Map for Intelligent Agents and Data Mining. *Proceedings of the Second International Symposium on Artificial Life and Intelligent Agents, Birmingham, 14-15 June 2016.* (In Press) Springer.

Paper III. Amy de Buitléir, Ronan Flynn, Michael Russell and Mark Daly. An architecture for pattern recognition and decision-making. In Elio Tuci, Alexandros Giagkos, Myra Wilson and John Hallam, editors,

From Animals to Animats 14: 14th International Conference on Simulation of Adaptive Behavior, SAB 2016, Aberystwyth, UK, August 23-26, 2016, Proceedings, pp. 22–33, Springer International Publishing, Cham.

Chapter 1

Introduction

This thesis describes a research project to discover if Artificial Life (ALife) agents with Artificial Intelligence (AI) are capable of performing data mining. The project builds upon an earlier research project [1, 2] I undertook during 2009-2011 in pursuit of an MSc. The goal of that project was "to evolve an ALife population with sufficient intelligence to discover patterns in data and make survival decisions based on those patterns, and to create a population that adapts to its environment through both evolution and lifetime learning" [1, p. 9]. The products of that research included a computational ecosystem, or framework, for artificial life experiments, called Créatúr[1], and a species of artificial agents, called wains[2], which had evolved the ability to discover patterns in data and make decisions based on those patterns.

The research described in this thesis builds upon the foundation of my MSc research by applying lessons learned and exploring new directions. One of the

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

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

findings of the earlier research was that wains too frequently made unwise decisions, given their knowledge of the environment. As a result, my first focus was finding a means to improve their decision-making ability. Dennett's *Tower of Generate-and-Test* [3, 4, p. 83-98, 5, p .258-262] provided a useful framework for thinking about the cognitive power of agents, whether biological or artificial. Using Dennett's terminology, most ALife agents are either *Darwinian creatures* (which have a fixed design, and can only adapt to their environment through evolution) or *Skinnerian creatures* (which can learn through a form of operant conditioning). In contrast, a *Popperian creature* has a sort of "mental laboratory" where they can "try out" an action and predict the results, giving them more cognitive power than Skinnerian or Darwinian creatures. This pointed the way for improving the decision-making ability of wains, i.e., redesign their brains to raise them to the Popperian level. Thus we have the first research question of this thesis:

> **Research Question 1**: Will giving wains a mechanism to predict the outcomes of possible actions, and to choose the action with the best predicted outcome, make them better decision-makers?

Earlier research showed that wains could discover patterns in data. This led to my second focus, discovering if these improved Popperian wains could perform data mining. There is a continuing need for new data mining techniques, especially to cope with data patterns that change over time, or to process data streams in real-time. After considering the various types of tasks included under the data mining umbrella, I decided that Popperian wains would be most likely to succeed at classification and forecasting. This led to the following research

questions:

> **Research Question 2:** Can Popperian wains learn to classify data with accuracy and speed comparable to traditional classification methods?

> **Research Question 3:** Can Popperian wains learn to forecast future values in a data stream, with accuracy and speed comparable to traditional forecasting methods?

The research described in this thesis aims to answer these three research questions.

1.1 Document structure

In this section I describe the content of each chapter, and highlight the thought process that connects the chapters. Figure 1.1 shows the dependencies between key topics.

Chapter 2 (Literature review) establishes the theoretical basis for this research project, drawing inspiration from computer science, philosophy, and biology. A brief history of AI is presented, and the definition of "intelligence" is considered. A summary of different forms of ALife follows, and its relation to AI is introduced. Dennett's Tower of Generate-and-Test is discussed in more detail, focusing on the adaptive advantages of Popperian creatures over those at lower levels in the tower. Evolution and its relationship to both biological and artificial life is presented, followed by gene expression and the benefits of sexual reproduction for the adaptability of an organism. The growing importance of data

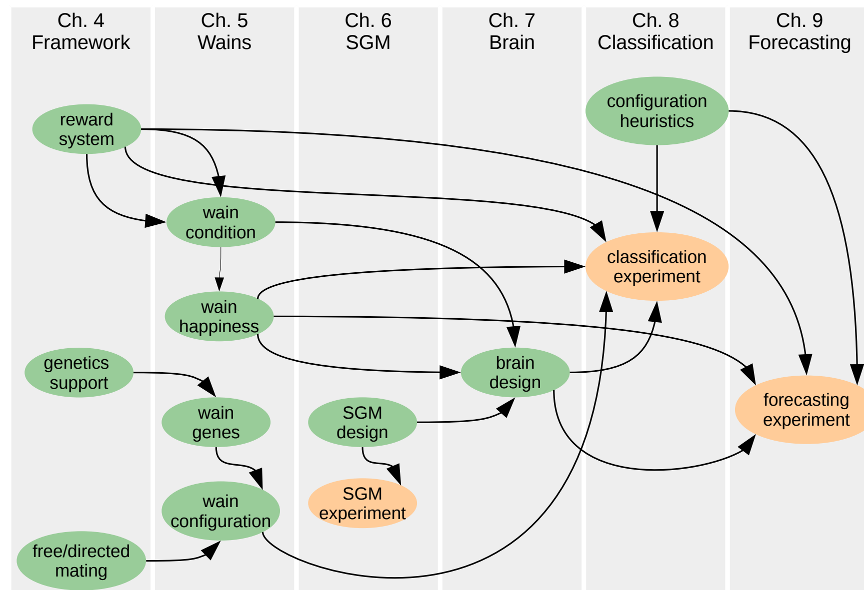

Figure 1.1: Key topic dependencies. An arrow from one topic to another indicates that the second topic requires an understanding of the first topic.

mining is discussed, along with the role of ALife in data mining. One particular data mining tool, the Self-Organising Map (SOM), is presented, setting the stage for a version of this algorithm adapted for ALife as part of this research project. Wains are then described in more detail. Since Automated Speech Recognition (ASR) is used in one of the experiments described in later chapters, the fundamental concepts are introduced here. Functional programming concepts such as *monads*, the Domain-Specific Embedded Language (DSEL) and *datatype-generic programming*, used in this research project; are explained. Finally, the data sets which were used in experiments for this project are introduced.

Chapter 3 (Approach) describes the reasoning behind my choice to make wains Popperian creatures, and to apply them to classification and forecasting rather than other data mining tasks.

In preparation for the experiments conducted as part of this research project, several improvements were made to the Créatúr framework. These improvements were not strictly required to answer the research questions, but they made the framework much easier to use, both by the author and future researchers. One of the most significant improvements is a mechanism that allows any data type to be used as a gene, with automatic support for for encoding, decoding, recombination, and expression. Chapter 4 (Improving the Créatúr framework) describes issues commonly faced by developers of ALife, and shows how the new framework addresses them.

Extensive changes were made to the wains as part of this research; discussion of those changes is split over the next three chapters. Chapter 5 (Improving the wain) describes the non-brain-related improvements. One of the most significant improvements is an extensible architecture allowing wains to customised to work with different types of data.

The core of the wain brain is a modified SOM. Chapter 6 (The Self-generating model) describes how and why the SOM was adapted for use in wains, producing the Self-Generating Model (SGM). The SGM could be suitable for a variety of intelligent data mining ALife agents, and as a standalone classifier. A series of experiments comparing the performance of the SOM and SGM is presented. Although the experiments do not answer any of the original research questions, they demonstrate the advantages of the SGM.

Chapter 7 (Improving the brain) describes how the wain brain was redesigned to make wains Popperian creatures, with the hope that the resulting increase in cognitive power would allow them to perform data mining tasks. The new brain is built upon two SGMs.

In Chapter 8 (Classification with wains), a series of experiments designed to answer research questions 1 (are Popperian wains better decision-makers?) and 2 (can they perform classification?) is presented.

Chapter 9 (Forecasting with wains) presents a series of experiments designed to answer research question 3 (can Popperian wains perform forecasting?).

Chapter 10 (Conclusions) provides a synopsis of the thesis, presents some conclusions about this research project, and proposes future directions for research continuing on from, or inspired by, this project.

A **Glossary**, list of **Acronyms**, and **Bibliography** are provided. Appendix A provides a short introduction to Haskell Syntax. Appendix B discusses the methodology used in configuring experiments.

1.2 Major contributions

The major contributions of this work are summarised below.

- A proposed approach for creating artificial life that is smart enough to perform data mining; namely, instead of designing agents specifically for data mining, design them with general-purpose intelligence, with brains that place them at the Popperian (or higher) level in Dennett's Tower of Generate-and-Test.

- A design for a Popperian ALife brain. This design allows an agent to generate hypotheses about the scenario it is facing, consider the actions available to it, predict the outcome of each action for each hypothesis, and

choose the action that is likely to produce the best outcome.

- A re-design and re-implementation of wains to give them Popperian-level AI and allow them to perform classification and forecasting, with an accuracy that is comparable to traditional methods.

- A re-design and re-implementation of Créatúr, a library and framework for creating ALife and running ALife experiments. Any data type can be used as a gene; it will inherit default mechanisms for gene encoding, decoding, recombination, and expression. Créatúr supports both sexual and asexual reproduction, with a flexible language to control recombination and mutation.

- The SGM, a modified SOM that was adapted for use in intelligent data mining ALife agents. By sacrificing topology-preservation, the SGM requires fewer calculations, making it faster than a SOM of comparable size. The SGM achieves a higher accuracy more quickly than the SOM, which could allow an agent to make good survival decisions with less training. With greater *model stability* (the ability of a model to continue to match patterns it was created in response to, while adjusting to match new patterns) and fewer *wasted models* (models that will not be used to classify future patterns), the SGM could be a useful component for implementing intelligent agents, and for other clustering or *classification* applications.

1.3 Conventions used in this thesis

When discussing a range of possible values for a variable, the notation $[a, b]$ indicates a closed interval (one that contains the endpoints), i.e., $\{x : a <= x <= b\}$.

In this thesis, a `wain` is sometimes referred to using the more generic term "agent". This is done in statements that could apply to both `wains` and other types of agents. For example, the brain design described in Chapter 7 could be used in agents other than `wains`. Thus, that chapter frequently refers to "agents".

An individual representation of a pattern that an agent has observed in the environment is called a "model". the collection of all such representations that the agent has formed is called a "model set". There is one exception to this convention: SGM stands for "Self-generating Model", where "model" in this case refers to a model of the input pattern space.

Chapter 2

Literature review

As shown in Figure 2.1, this research project incorporates concepts from computer science, philosophy, and biology. This chapter introduces those concepts, describes the application domains to be focused upon, establishes the theoretical basis for this research, and describes the data sets used in the experiments performed as part of this research.

2.1 Artificial Intelligence

Artificial Intelligence (AI) refers to programs or machines which exhibit intelligent behaviour. Unfortunately, the word "intelligent" is difficult to define satisfactorily. The definitions below will illustrate some common themes.[1]

> "Ability to learn or having learned to adjust oneself to the environment." S. S. Colvin, quoted by Sternberg [7, p. 8]

[1] For more definitions of "intelligence", see Legg and Hutter [6], Sternberg [7, p. 8], and Rapaport [8].

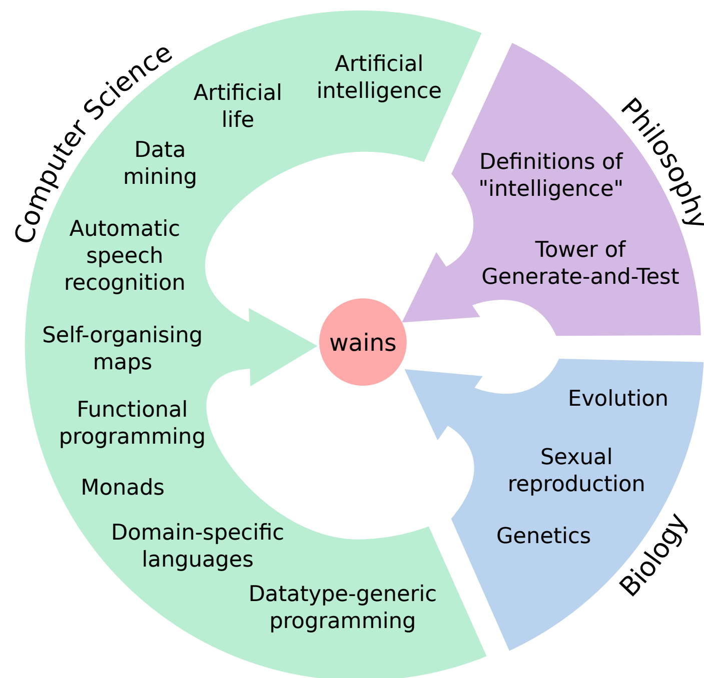

Figure 2.1: Topic map

"The aggregate or global capacity of the individual to act purposefully, think rationally, and deal effectively with his environment." Wechsler and Matarazzo [9, p. 79]

"Part of the internal environment that shows through at the interface between person and external environment as a function of cognitive task demands." R. E. Snow, quoted by Legg and Hutter [6]

"The power of good responses from the point of view of truth or facts." E. L. Thorndike, quoted in [7, p. 8]

"Behaviour that we call intelligent behaviour when we observe it in human beings." Slagle [10, p. 1]

The above definitions illustrate some important aspects of intelligence: learning, adapting to the environment, acting purposefully to achieve a goal, and providing logical responses given one's knowledge.

In the 1940s, a number of important developments laid the foundation for AI. McCulloch and Pitts [11] showed that any computable function could be represented by a network of artificial neurons. Hebb [12, Ch. 4] described a way that these artificial neural networks might learn.

Now computer scientists could begin to take seriously the idea of implementing human-level intelligence in a machine. Turing [13] proposed the now famous *Turing Test* which states that a computer could be said to think if a human interrogator could not distinguish between it and another human through typed conversation. Turing [13] concluded that by the end of the 20th century, a computer would be built that could pass the test.

The birth of AI as a field of research happened in the summer of 1956, with a workshop at Dartmouth College in New Hampshire [14, p. 18]. The event was organised by John McCarthy, who coined the phrase "artificial intelligence" [15]. Other attendees included Marvin Minsky (a well-known cognitive scientist who later founded the MIT Artificial Intelligence Lab in 1959 with McCarthy [16]), and Claude Shannon (who founded information theory). The conference proposal stated "This study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it." [17]

The Dartmouth conference ushered in a period of rapid progress in AI. Samuel [18] developed a program to play checkers (draughts) which learned from experience. The STUDENT program, developed by Bobrow [19], solved algebra

problems written in a subset of English. Winograd [20] developed the SHRDLU program, which could carry on a natural dialog with a human about a world of blocks. Evans' ANALOGY program could solve geometric analogy problems of the type found in IQ tests [21, Section 6.1].

These events fostered a feeling of optimism in the AI community. Simon [22, p. 96] wrote in 1965 "machines will be capable, within twenty years, of doing any work a man can do". Two years later, Minsky [23] wrote "Within a generation, I am convinced, few components of intellect will remain outside the machine's realm – the problems of creating 'artificial intelligence' will be substantially solved."

In the 1970s cracks began to appear in the AI foundation. Tasks such as recognising a face, or navigating a room without bumping into obstacles turned out to be far more difficult than anticipated. This turnabout became known as *Moravec's paradox*: "it is comparatively easy to make computers exhibit adult level performance on intelligence tests or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility" [24]. In 1972 Karp [25] showed that many problems in computer science are NP-complete, that is, computationally intractable. The time required to solve such problems using known algorithms increases rapidly with the size of the inputs, suggesting that many of the known AI solutions would never scale into useful systems. These and similar setbacks led to a lack of funding which characterised the "AI Winter" of the 1970s [21, p. 477, 14, p. 21][26, p. 154ff].

Eventually AI techniques became fashionable again, although typically under other names. In the 1980s, corporations around the world became interested in *expert systems*, applications that use a knowledge base of human expertise to

make decisions [27, 14, p. 9]. In the 1990s, improvements in *data mining* (extracting insight from data) made interest in this field increase dramatically [28, p. 6]. From the 1990s onward, AI played a crucial role in computer games: producing intelligent behaviour on the part of game characters not controlled by the player. The reduction in cost of computer hardware predicted by Moore's Law allowed previously infeasible projects to be tackled. IBM Watson, which won on the quiz show "Jeopardy" in 2011, had access to four terabytes of data [29]; this amount of storage would have been unthinkable decades earlier.

To an outsider, the history of AI may look like a long string of failures and disappointing results. There are two reasons for this. Firstly, whenever an AI research project is successful, it usually becomes a new scientific or commercial specialty with a new name [30]. This happened, for example, with robotics, expert systems, automatic theorem proving, machine vision, knowledge engineering and computational linguistics [30]. This leaves the field of AI with little to claim but unsolved problems, and solutions in "toy domains" that do not scale to the real world.

The second reason for the apparent lack of success is the "AI Effect". As mentioned earlier, observers generally choose to call an AI system intelligent if its behaviour would be considered intelligent when performed by a human (Slagle's definition). However, as soon as the mechanism by which the AI system achieves its goals is explained, the observer narrows the definition of intelligence to exclude the behaviour in question. As Kahn [31] wrote, "Every time we figure out a piece of it, it stops being magical; we say, 'Oh, that's just a computation'. We used to joke that AI means 'almost implemented'." More succinctly, Hofstadter [32, p. 601] quotes *Tesler's Theorem*: "AI is whatever hasn't

been done yet".

2.2 Artificial Life

Artificial Life (ALife) is a field which attempts to create life-like behaviour using software, hardware, biochemistry or other media; this thesis focuses on software. Whereas biology is the study of "life-as-we-know-it", ALife is the study of "life-as-it-could-be" [33]. ALife is not only used as a simplified model of biological life and ecosystems; it is also increasingly applied to real-world problems as diverse as data mining [34], music composition [35], and management of dam operations in multi-reservoir river systems [36].

Automata of various kinds (for example, the cuckoo clock) existed long before the computer age, however they performed a predetermined sequence of actions, or responded in a predetermined way to inputs. With the advent of computers, it became possible to create far more flexible and life-like systems. One of the earliest such systems was developed by John von Neumann as a result of his work on self-replicating machines. Von Neumann [37] had shown that such a machine was possible, but had not suggested an implementation [38, p. 2]. It is believed that the mathematician Stanislaw Ulam suggested the use of what are now known as *cellular automata* to von Neumann [38, p. 3].

A cellular automaton consists of a grid of cells. At the initial step, each cell is in one of a finite number of states, such as on or off. At each successive step, the cells in the grid are assigned a (possibly new) state based on a predetermined rule. Depending on the rules, patterns can form that persist over long periods of time or interact in complex and interesting ways. The most famous cellular

automaton is called The Game of Life; it was developed by John Conway [39].

In 1987, Christopher Langton organised the first ALife conference, "Workshop on the Synthesis and Simulation of Living Systems" [40]. Since then, the field has developed in a variety of directions. ALife has been used to model group motion as observed in nature, such as flocking, swarming, or schooling. An early example is Reynold's Boids [41]. There are also ALife projects to simulate biological organisms. For example, OpenWorm simulates the roundworm *Caenorhabditis elegans* at the cellular level [42].

Another common form of ALife consists of small computer programs which can replicate, recombine, and mutate. There does not appear to be a commonly-used term for this type of ALife; in this thesis they will be referred to as *Instruction-based ALife*. Often there is no fitness function for this type of agent; there is only survival or death. Instruction-based ALife agents include Tierra [43, 44] Avida [45], Framsticks [46], BREVE [47], and Darwinbots [48].

Some ALife implementations use some sort of AI to allow the agent to make decisions. This can be a neural net, as in Noble Ape [49, 50], PolyWorld [51, 52, 53, 54], Creatures [55, 56] (no relation to Créatúr), or Critterding [57].

2.3 Evolution

The recipe for evolution is simple; it requires the following ingredients [58]:

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 how an element's features interact with its environment.

All of the complexity and variation of biological life arises from this recipe. 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. The work of Bernard et al. [59] suggests that evolution leads to a higher fitness level for the agent than other forms of adaptation. Evolution is often used in ALife.

The field of , founded by D. T. Campbell, is based on the idea that cognition should be studied from the perspective of evolutionary theory. [60] Campbell applied the above recipe, which he called "blind variation and selective retention" to creative thought, describing how thoughts undergo a process of selection within a single organism. [61] The next section presents a framework for analysing cognition within the framework of evolution.

2.4 Dennett's Tower of Generate-and-Test

The behaviour of most biological animals is controlled by the organ called the *brain*. By analogy, any mechanism which controls the behaviour of an AI or ALife agent is typically called a "brain". Dennett [3] used the term in this looser sense when he proposed a framework for ranking brain designs, which he called the *Tower of Generate-and-Test*. As illustrated in Figure 2.2, each floor in the tower represents an important increase in cognitive power. (In this section, the description of the tower is taken from Dennett [3, 4, p. 83-98, 5, p .258-262]; the assignment of various ALife species to levels in the tower is based on this

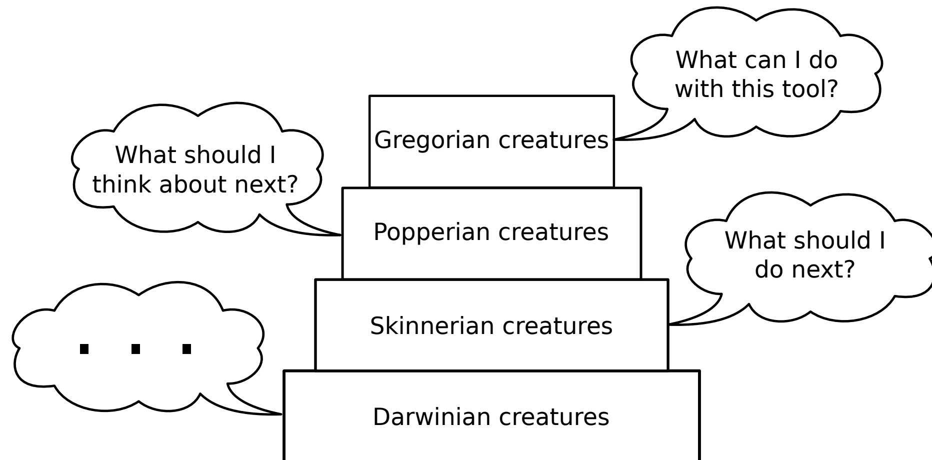

Figure 2.2: Dennet's Tower of Generate-and-Test

author's interpretation of Dennett's framework.)

The ground floor contains what Dennett calls *Darwinian creatures.* These organisms have a fixed design; only the best designs survive. They can only adapt to the environment through recombination and mutation of genes. Instruction-based ALife agents are Darwinian, as are simple ALife species used to model group motion (such as Reynold's Boids).

The next floor contains organisms where some aspect of their design is adjusted by events that occur during their lifetimes. Dennett calls these *Skinnerian creatures* after the psychologist B.F. Skinner because they learn through *operant conditioning.* Skinner saw operant conditioning as a sort of "next step" beyond evolution, saying "Where inherited behaviour leaves off, the inherited modifiability of the process of conditioning takes over." [62]

In operant conditioning, actions that are rewarded are reinforced, and are more likely to be repeated in future on similar occasions. This is a second mechanism (in addition to genetics) by which organisms can adapt to the environment. ALife agents with some form of AI are usually Skinnerian creatures. (For exam-

ple, Noble Ape, PolyWorld, Creatures, and Critterding are all Skinnerian). However, this mechanism is a trial-and-error approach; a purely Skinnerian creature is likely to try out a number of stupid moves before stumbling upon the smart moves.

A still better system would "weed out" stupid moves without needing to experience the consequences. Organisms on the next floor of the tower have a sort of *inner environment* where they can "try out" an action and predict the results. Thus, they have three mechanisms for adapting to the environment. Dennett, quoting the philosopher Sir Karl Popper, says that this mechanism "permits our hypotheses to die in our stead" and calls the inhabitants of this floor *Popperian creatures*. When a Popperian creature takes an action, it observes the difference (if any) between the predicted and actual results, and makes appropriate adjustments to its models. Thus, Popperian creatures have a third mechanism (in addition to genetics and operant conditioning) by which they can adapt to the environment.

Dennett points out that most biological species are Popperian creatures, including the pigeons that were the focus of much of Skinner's research. He concludes that if there are any purely Skinnerian creatures, they would be simple invertebrates. The difference in cognitive power between Skinnerian and Popperian creatures is important.

> [It is as if] Skinnerian creatures ask themselves, "What do I do next?" and haven't a clue how to answer until they have taken some hard knocks. Popperian creatures make a big advance by asking themselves, "What should I think about next?" before they ask them-

selves, “What should I do next?” [4, p.98]

Organisms on the next floor exhibit some form of tool use. (The word “tool” should be interpreted in the broadest sense; one of the most important tools humans have are words.) Thus, they have four mechanisms for adapting to the environment. Dennett calls these organisms *Gregorian creatures* after the psychologist Richard Gregory, who observed that designed artefacts such as scissors are a form of intelligence frozen from the past that can be applied now. [63, p. 311-314]

How might we identify the level to which an organism belongs, if we do not know the inner workings of its mind, or if indeed it has a mind? We could first test whether or not it has any ability to learn; if not, we can assume it is Darwinian (assuming of course that it evolves). If we observe the organism using tools, we know that it is Gregorian. The more difficult task is to distinguish between Popperian creatures and Skinnerian creatures. One possibility is to train the organism to take a certain action in response to a stimulus. If the organism gives the same response to a similar but not identical stimulus, we might assume it is Popperian. However, we must ensure that the test stimulus is not so similar to the training stimulus that mere operant conditioning would cause the organism to give the correct response. In practice, it may not be easy to determine the appropriate level of similarity between the training stimulus and the test stimulus.

The Tower of Generate-and-Test has frequently been used to represent levels of cognition in nature. [64, p. 1-9, 65, p. 153-157, 66, p. 9-12] Many researchers have used it to evaluate ALife agents. [67, 68, p. 238] Some have proposed ex-

tending the tower with new levels beyond Gregorian. [69] Some have been inspired by Dennett's tower to develop similar scales for other characteristics such as volition. [70].

While the Tower of Generate-and-Test is useful for categorising both biological and artificial *agents*, applying it to *algorithms* is problematic. An algorithm for unsupervised learning might be considered Popperian, but unless it takes some sort of action based on what it learns, this seems a poor fit. In fact, we might imagine using the same learning algorithm as a component in a Skinnerian, Popperian, or even a Gregorian brain.

2.5 Reproduction

In biology, reproduction occurs in two forms. (Information in this section comes from Beukeboom and Perrin [71].) In asexual reproduction, a single parent produces offspring, and the offspring inherit traits from that parent. A common example of this is *fission*, where an organism such as a bacterium divides, producing two offspring which replace the parent.

In the second form of reproduction, sexual reproduction, two parents produce offspring, and the offspring inherit a mixture of traits from both parents. Typically, organisms which reproduce sexually are *diploid*: their cells (apart from the sex cells) contain two sets of genetic information. Each parent contributes a *gamete* (sex cell) which is *haploid*; these fuse during fertilisation to form a diploid cell. In *isogamous* organisms, the gametes are of similar size and form. Even so, fertilisation can only occur between two gametes of different *mating types.* (Some species have hundreds of mating types.)

Anisogamous organisms, on the other hand, produce different types of gametes. Smaller gametes are called *sperm cells*; the organisms which produce these are defined to be male. Larger gametes are called *egg cells*; they are produced by female organisms. Males and females typically have differences in size and shape beyond differences in their sex organs, this is called *sexual dimorphism*.

Reproduction for ALife agents is usually modelled on either asexual reproduction, or (diploid) sexual reproduction. The latter approach may benefit ALife by encouraging a diverse gene pool, preserving partial solutions that may be useful as the environment changes [72, 73]. However, if the population is divided into male and female agents, the number of mating opportunities is halved, and ALife populations are usually small due to processor limitations. For this reason, many ALife implementations use a form of asexual reproduction, but with two parents (e.g., Tierra [43, 44], PolyWorld [51], and Creatures [55]). Some of the offspring's genes are taken from one parent; some from the other.

2.6 Gene expression

In biology, a gene is the fundamental unit of heredity. For organisms which reproduce sexually, *gene expression* is the mechanism that determines the *phenotype* (the observable traits of the organism) from the *genotype* (genetic makeup) [74]. Genes can have multiple *alleles* (forms). Most multi-cellular organisms are diploid; they have two sets of chromosomes, one from each parent. Such an organism will have *two* strands of genetic information. When corresponding genes from the two sets are not identical, the resulting phenotype will depend

on the *dominance* relationship between the two alleles. One allele may be *dominant* (expressed in the agent) and the other *recessive* (not expressed), or some sort of blending may occur when the gene is expressed. Similarly, an ALife species which uses sexual reproduction needs a way to determine the characteristics of an agent from the two strands of genetic information.

2.7 Data mining

The explosion in online activity, the falling cost of storing data, and improvements in technology for processing data have led to the phenomenon called *big data* [75, p. 2]. Sciences such as astronomy and genomics were the first to experience this phenomenon, but it is now widespread [76, p. 6]. Mayer-Schönberger and Cukier provide the following definition:

> Big data refers to things that one can do at a large scale that cannot be done at a smaller one, to extract new insights or new forms of value, in ways that change markets, organisations, the relationship between citizens and governments, and more. [76, p. 6]

The term “pattern” refers to any structure, configuration, grouping of characteristics, or set of relationships that can be found in multiple places in the data.

Extracting those insights is the focus of data mining. Data mining is the process of exploring data to discover interesting and useful patterns [77, p. 33, 78, p. 7]. In contrast with an ordinary database search or query, where the key features and relationships are known; in data mining, they have to be discovered [79, p. 5]. Gorunescu [79] offers the following overlapping definitions:

- The automatic search of patterns in huge databases, using computational techniques from statistics, machine learning and pattern recognition;
- The non-trivial extraction of implicit, previously unknown and potentially useful information from data;
- The science of extracting useful information from large datasets or databases;
- The automatic or semi-automatic exploration and analysis of large quantities of data, in order to discover meaningful patterns;
- The automatic discovery process of information. The identification of patterns and relationships 'hidden' in data. [79, p. 5]

Gorunescu [79] also distinguishes between an ordinary database search or query, where the relevant factors and relationships are known, and data mining, where they are not; the first item on the list above should be interpreted with that context in mind. The goal of data mining can be *descriptive* (e.g., to model and understand the data) or *predictive* (using some of the variables to predict other variables), or both [79, p. 5].

The terms data mining and Knowledge Discovery from Data (KDD) are sometimes used interchangeably, but it can be useful to draw a distinction. Suh [80] considers data mining to be the *knowledge generation* step in the overall KDD process, occurring after pre-processing (cleaning, converting, and otherwise preparing the data) and before post-processing (synthesising the knowledge into information that can be used for decision-making) [80, p. 4f]. Maimon

and Rokach [81] describe data mining as "the core of the KDD process, involving the inferring of algorithms that explore the data, develop the model, and discover previously unknown patterns." [81, p. 1]

Data mining includes a wide range of tasks, such as those listed below.

- *Landscape mining* explores the data to find the space of possible inferences (the data's "landscape") and to identify interesting patterns before leaping in with more traditional data analysis tools [82].
- *Classification* assigns objects to predefined categories based on the attributes of the objects [83].
- *Cluster analysis* (also known as unsupervised classification or exploratory data analysis) partitions items into a set of clusters such that objects within a cluster have similar characteristics, and objects in different clusters have dissimilar characteristics [83, 84].
- *Anomaly detection*, the discovery of unusual data values, can occur as a side-effect of cluster analysis.
- *Prediction* (also known as forecasting) estimates future (or unknown) data based on present data and past trends, validating hypotheses [83, 85, p. 4].
- *Regression* identifies functions which map data objects to prediction variables [83, 85, p. 4].
- *Modelling* produces a (typically simpler) representation of the data that captures important features and relationships. Such models can be used for classification, prediction, and to provide insight about the data.

- *Visualisation* makes insights understandable by humans [83].

As new techniques for analysing data are discovered, the nature of data mining is changing. Menzies categorises the data mining community as moving from *algorithm mining* (tuning parameters in data mining algorithms) to landscape mining (exploring the shape of the decision space) [82].

> Algorithm mining is a "leap before your look" approach in which researchers throw algorithms at data and then see what comes out. A second approach is the "look before you leap" option—mining the data to find the space of possible inferences before leaping in with the learners. This is the data's "landscape". [82]

2.8 Agent-based approaches to data mining

ALife may be a useful tool for exploring these data landscapes; it has been used in a variety of ways for data mining. Techniques modelled on insect behaviour are common because "*at some level of description* it is possible to explain complex collective behaviour by assuming that insects are relatively simple interacting entities" (emphasis in the original) [86]. Ant Colony Optimisation (ACO) has been used for supervised classification; is modelled on the behaviour of ants finding a trail between their colony and a source of food [87, section 1.5, 88, 89]. Data clustering techniques have also been modelled on the sorting behaviour of ants [89]. Insects are not the only biological model used. When a predator encounters prey, it must decide whether to attack or continue searching for better or easier prey. This technique has been used to reduce the dimensionality of clustered data [90].

Cao argues persuasively that there is a synergy between autonomous software agents (ALife) and data mining. Both streams of research face challenges such as distributed, parallel, and adaptive learning. Both require applications which can understand and represent the interactions between components in their domain [34, Ch. 1]. By providing a complex “environment”, big data may encourage greater complexity and intelligence in agents.

> Ecologists have long recognised that the complexity of an organism’s behaviour is related to the environment it must “solve”. [91]

2.9 The SOM algorithm

The Self-Organising Map (SOM) provides a way to represent high-dimensional data in fewer dimensions (typically two), while preserving the topology of the input data [92]. A SOM is a set of models associated with nodes in a regular grid. (The term *node* can refer to either the physical location of a model, or the grid co-ordinates of a model.) Patterns that are similar to each other in the high-dimensional space are typically mapped to models that are near each other on the grid. (There are exceptions to this topology-preserving property, however; see Villmann et al. [93]).

In addition to topology preservation, a SOM has benefits that make it useful for ALife and intelligent agents. It is easy to understand and implement. The SOM models can be inspected directly, which makes it easier to debug problems with the implementation or the learning function. After a SOM has been trained, labels can be assigned to the nodes to allow it to be used for classification. It can also be used to cluster data; a U-matrix (whose elements are the Euclidean

distance between neighbouring cells) will have high values at the cluster edges [94].

The SOM has an established place in the data mining tool set, especially for clustering and classification. It has also been used, sometimes with modifications, in ALife [95, 96] and artificial intelligence [97, 98].

SOM training (see Algorithm 1) is unsupervised. The elements (patterns) of the input data are typically numeric vectors, but they can be any data type so long as we can define a measurement of similarity between two patterns, and a method to make one pattern more similar to another, by an arbitrary amount. The SOM models are arranged on a (typically two-dimensional) grid of fixed size. The models must be initialised.

Algorithm 1 SOM algorithm [92].

For each input pattern,

1. Compare the input pattern to all models in the SOM. The *node* with the model that is most similar to the input pattern is called the *winning node*.
2. The winning node's model is adjusted to make it slightly more similar to the input pattern. The amount of adjustment is determined by the *learning rate*, which typically decays over time.
3. The models of all nodes within a given radius of the winning node are also adjusted to make them slightly more similar to the input pattern, by an amount which is smaller the further the node is from the winning node.

Step 3 ensures that as additional input patterns are received, nodes that are physically close respond to similar patterns in the input data. Thus, the resulting grid preserves the topology of the original high-dimensional data. SOMs therefore "translate *data similarities* into *spatial relationships*" (emphasis in the original) [99].

The traditional SOM has been adapted and extended in many ways. Common modifications include using grids in non-euclidean spaces [99], dynamically increasing the size of the grid [100], replacing the grid with a hierarchical arrangement of nodes [101], and combining with principal component analysis [102]. There is extensive literature on SOMs with two or more of these modifications [103, 104, 105, 106, 107, 108, 109, 110].

2.10 Wains

Wains are an ALife species with artificial intelligence. They live in, and subsist on, data; for them, finding patterns in data is a survival problem. In order to stay alive, they must discover patterns in the data, build models of those patterns, classify new data based on the models, decide how to respond to data, and adapt to changes in the patternicity of the data. (Except where noted, the information in this section is taken from de Buitléir, Russell, and Daly [2] and de Buitléir [1].)

Wains are diploid, but have only one sex. This may confer some of the advantages of sexual reproduction, without reducing the number of mating opportunities. Genes affect the appearance, configuration or capability of a wain. Each wain gene has multiple alleles (forms). If two corresponding alleles are not identical, the characteristics of the child will depend on the relationship between the alleles. One allele may be dominant and the other recessive, or the resulting wain may have a blending of the traits encoded for by the alleles

The version of wains and Créatúr documented in de Buitléir, Russell, and Daly [2] and de Buitléir [1] will hereinafter be referred to as the *original implementation*. The version that incorporates the changes described in later chapters

of this thesis will be referred to as the *new implementation.* The original implementation will be briefly described here; detailed discussion will be deferred to Chapters 5 and 7, when it is contrasted with the new implementation.

In the original implementation, a wain's brain contained a *classifier* and a *decider.* The classifier was a modified SOM; it identified patterns in sensory inputs that a wain received during its lifetime. The SOM was modified by omitting Step 3 in Algorithm 1. This sped up the algorithm, but the topology of the input data was no longer preserved. The decider was a decision matrix with adjustable weights. A wain's response to a situation was chosen by weighted random selection; the weights were taken from the row of the decision matrix corresponding to the pattern identified by the classifier. Based on the outcome (positive or negative), the weights would be adjusted. This is a form of operant conditioning, making the original wains Skinnerian creatures.

The brain also implemented a feature inspired by the theory of *Neural Darwinism*, which proposed dynamic selection between neuronal groups [111]. SOM models (which might be considered analogous to neuronal groups) competed with each other. Periodically, the least useful model was erased (by setting it to random noise) so that it could learn a new pattern. (The total number of models remained constant.) However, experiments showed that this feature was not useful; evolution lengthened the erasure cycle to the point where a wain was unlikely to experience any erasure during its lifetime.

The original wains were tested with handwritten numerals taken from the *MNIST* database [112], (which will be described further in Section 2.13.1). This was not strictly a classification task, however. When encountering an object (a numeral or another wain), the agent could choose to try to eat it, play with it,

flirt with it, or ignore it. A numeral was either "fun" (would reduce boredom when played with), or "boring" (no effect on boredom). Also, numerals were either "nutritious" (would provide energy when eaten), or "poisonous" (would deduct energy when eaten).

When encountering a numeral, the decision matrix was used to estimate the desirability of each possible action, and a response was chosen by weighted random selection. Only the model that best fit the input pattern was used for decision-making. It was possible for the least desirable action to be chosen, so the wain would occasionally take risks. Since wains were asked to choose from a small set of actions rather than to classify the numerals, it is difficult to say how accurate their mental models of numerals were. For example, not eating an "edible" numeral, which wains did approximately 30% of the time, is not necessarily a mistake; the wain might already have maximal energy or benefit more from playing with it. However, eating a "poisonous" numeral is clearly a mistake, one which the wains made nearly 10% of the time. Attempting to mate with a numeral rather than another wain is another clear mistake, wains did this approximately 4% of the time. Clearly there is scope for improving wains' decision-making ability.

The appearance of a wain in the original implementation was a 28x28 grey-scale image, which was *genetically determined*. (A genetically determined value is one that is specified by an agent's genes, can be different for each agent, can be inherited by its children, and is subject to evolutionary pressures.) Wains in the starter population had the image of an X as their appearance; subsequently, several *mutations* arose producing wains with different appearances. When a wain encountered an object, the appearance of that object was presented to the

wain's senses, along with information about the wain's current state. It could then choose to eat, attempt to mate with, or play with the object. When a child was born, it remained until maturity with the parent who initiated the mating.

Experiments demonstrated that a population of wains can indeed discover patterns, make survival decisions based on those patterns, and adapt to changes in the patternicity of their data environment. Not only did individual agents learn to make better decisions during their lifetime, but evolution made changes to the brain that improved the decision-making ability of the agents. Evolution made their brains more efficient, by reducing the number of patterns that the SOM stored, without affecting an agent's ability to identify sufficient food to survive. A series of changes was made to the environment; they adapted quickly to these challenges, primarily by modifying their learning rates through evolution. Wains have also been applied to the task of speech recognition, by having them identify audio samples of spoken numerals [113, 114].

Wains were developed and tested using a framework called Créatúr, a reusable software framework for automating ALife experiments [1, Ch. 5]. Créatúr provided a daemon, an event scheduler, and a log facility. It allocated processing resources to agents, provided persistence of statistics and data between runs, deducted energy from agents to simulate metabolic requirements, and removed dead agents from the population and archived them.

In developing the wains, De Buitléir [1, p. 66] used the strategy outlined below.

- 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 (everything the agent does, even just living, must have a cost).
- Protect the young while they learn.
- Use diploid reproduction.
- Provide a means for agents to estimate degrees of kinship.

Several directions for future research were identified, including improving the wain's decision-making process, and implementing cultural transmission (allowing children to learn by observing the actions of their parents, and adults to learn by observing their peers).

2.11 Automatic speech recognition

The value of a new data mining technique can be demonstrated by applying it to a common data classification task, and comparing its performance with traditional techniques. One such "benchmark" task is Automated Speech Recognition (ASR), which is the process of converting an acoustic signal (spoken language) into the corresponding sequence of words. The primary goal of ASR is to allow humans to interact more naturally with computers. Part of the research described in this thesis involves applying wains to an ASR task.

The temporal nature of speech creates special challenges for ASR [115]. One challenge is varying dimensionality. In most types of machine learning, all data inputs are the same size. However, speech is elastic; even when the same speaker utters the same word twice, the resulting acoustic data will have differing lengths. Another challenge is identifying word boundaries in continuous speech. For this reason, accuracy is lower when recognising continuous speech as opposed to isolated words [116, p. 7]. For more information about the challenges of ASR, and a survey of the methods used, see O'Shaughnessy [117].

ASR systems rarely operate on the raw waveform data. It is preferable to find some features of the audio signal that are characteristic of a particular utterance, producing representations that are more suitable for ASR. For example, two samples of the word "three" should have similar representations. even for different speakers. Conversely, a sample of the word "three" should have a different representation from one of the word "six", even for the same speaker. Producing such a representation is called *feature extraction*.

Although an audio signal is constantly changing, dividing the signal into short frames allows us to treat the signal as if it were constant during each frame. There are a variety of methods for extracting features from a frame; one common technique is to represent the signal using mel-frequency cepstral coefficients (MFCCs). This involves taking the Fourier transform, converting to a mel scale (a scale based on human perception of pitch distance), taking the log of the energy for each mel frequency, and applying a discrete cosine transform (DCT). The amplitudes of the resulting spectrum are the MFCCs, that is, the feature vector for that frame. Davis and Mermelstein [118] describe this technique in more detail.

One widely used ASR technique is the Hidden Markov Model (HMM) [119]. A *Markov process* is a stochastic (random) process where the next state of the system is conditional on the current state, and independent of the past history of the system. A *Markov model* is a model of such a process. If the state is only partially observable, it is called a *hidden Markov model*. In ASR, the observable part of the state is the audio signal, and the hidden part is the text. The hidden Markov model toolkit (HTK) provides the ability to construct and manipulate HMMs [120]. HTK is widely used for speech recognition research.

2.12 Functional programming concepts

Functional programming is a paradigm that treats expressions as mathematical functions, and avoids side effects of computation. Wains and Créatúr were developed in *Haskell* [121] which is a purely functional programming language. Named after the logician Haskell Curry, the language uses strong static typing, with type inference (automatic deduction of the data type of an expression), and lazy evaluation (delaying the evaluation of an expression until its value is needed) [122]. Appendix A provides a brief introduction to Haskell syntax, which may be useful to consult when reading Chapter 4.

Functional programming languages like Haskell support powerful concepts and techniques that are likely to be unfamiliar to programmers accustomed to procedural languages such as Java or C. Some of these concepts and techniques are discussed below.

2.12.1 Domain-specific languages

A Domain-Specific Language (DSL) is a special-purpose language tailored to meet the needs of a limited domain. As "little languages" [123], they do not include all of the features provided in a general-purpose language like Haskell. Instead, they "trade generality for expressiveness in a limited domain" [124].

Fowler identifies two main reasons for using DSLs. First, they can improve the programmer's productivity because programs in the target domain are "easier to understand and therefore quicker to write, quicker to modify, and less likely to breed bugs". The second reason is that because DSLs are smaller and targeted to the problem domain, they make it easier for non-programmers to understand the code [125].

Designing and implementing a language from scratch is difficult, but the process can be simplified by inheriting the infrastructure of the "container" language (Haskell), tailoring it to meet the needs of the domain [126, 127, 128]. A language implemented in this way is called a Domain-Specific Embedded Language (DSEL). The new version of Créatúr developed as part of this research project uses DSELs.

2.12.2 Monads

Monads (the term comes from category theory) "provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism." [129]. The new version of Créatúr developed as part of this research project uses monads in all of these roles.

Some monads can be loosely described as containers for values. For example,

the Haskell list monad (`[]`) contains a sequence of zero or more values (which might be called a vector or array in some programming languages). The `Maybe` monad either contains `Just` a value, or it contains `Nothing`. The `Either` monad contains one of two possible value types, constructed using `Left` or `Right`. (Typically `Left` right represents some sort of error, while `Right` represents a normal value.) Some monads (e.g. `IO`) are better thought of as contexts for computation, rather than as containers.

However, since a monad defines a small set of operations that can be used within it, it is essentially a DSEL. Hudak calls monads used in this way "modular monadic interpreters" because they allow different language features to be isolated, given context-specific interpretations, and combined like "building blocks" [126].

2.12.3 Datatype-generic programming

Generic programming is programming that references types to be specified later. The actual implementation is automatically generated when the types are finally specified. The Haskell 98 standard [121] included some support for generic programming, in the form of derived instances, but only for six typeclasses. The Glasgow Haskell Compiler (GHC) provided support for five more typeclasses as part of the *Scrap Your Boilerplate* system [130, 131, 132].

GHC version 7.2 added support for *datatype-generic programming* as proposed by Magalhães et al. [133]. The new version of Créatúr developed as part of this research project uses this feature to minimise the amount of code that users of the framework must write. This lightweight and portable approach al-

lows the programmer to specify how to derive arbitrary class instances. The key is that the "generic" type is represented at run-time using a *sum-of-products* representation, which involves the following types:

- `U1` Unit, used for constructors without arguments
- `K1` Constants, additional parameters and recursion
- `M1` Meta-information (constructor names, etc.)
- `:+:` Sum, which encodes choices between constructors
- `:*:` Product, which encodes multiple arguments to constructors

As a result of this approach, the programmer usually only needs to write implementations for a set of base types, plus an implementation for each of the representation types above. Finally, the end user simply declares their type to be an instance of the desired type (using the `DeriveGeneric` pragma).

2.13 Data sets

This section describes the data sets used as part of the research described in this thesis.

2.13.1 MNIST

The MNIST database is a collection of images of hand-written numerals from Census Bureau employees and high-school students [112]. The training set contains 60,000 images, while the test set contains 10,000 images. All images are

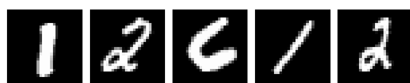

Figure 2.3: Sample images from the MNIST database [112].

28x28 pixels, and are grey-scale as a result of anti-aliasing. The centre of pixel mass of the numeral has been placed in the centre of the image. Sample images are shown in Figure 2.3.

2.13.2 TI46

The *TI46* speech database is a corpus of 46 isolated spoken words recorded for both male and female speakers. The corpus is intended for the evaluation of ASR products [134]. Among the words in the corpus were the numerals “zero” through “nine”. The training set contains 1,594 samples of spoken numerals; the test set contains 2,541 samples.

2.13.3 ISP traffic

Cortez et al. [135] provided Internet traffic data (in bits) from a private Internet Service Provider (ISP) with centres in 11 European cities. One of the datasets (referred to as *A5M* in Cortez et al. [136]), is a univariate time series; transatlantic link traffic was measured at five-minute intervals, from 6:57 a.m. on 7 June to 11:17 a.m. on 31 July 2005, for a total of 14,772 data points. There were no missing values.

2.13.4 New York City weather data

The Zonination weather database contains weather data for 24 international cities [137]. The New York City data contains daily weather records from 1 July 1948 through 31 December 2015. As shown in Table 2.1, the records for 95 days are missing. The total number of records is 24,560.

Table 2.1: Missing records in New York City weather data

start	stop	days
2000-02-23	2000-05-02	70
2000-06-01	2000-06-08	8
2000-08-12	2000-08-21	10
2000-08-23	2000-08-26	4
2000-08-28	2000-08-30	3
	total missing days	95

The data is a multivariate time series; each record contains the information listed below. Some records have missing values.

- Date
- Maximum, mean, and minimum temperature (°F)
- Maximum, mean, and minimum dew point (°F)
- Maximum, mean, and minimum humidity
- Maximum, mean, and minimum sea level pressure (inches)
- Maximum, mean, and minimum visibility (miles)
- Maximum and mean wind speed (miles/hour)

- Maximum gust speed (miles/hour)
- Precipitation (inches)
- Cloud cover (an integer in the interval $[0, 8]$)
- Events ("", "Fog", "Fog-Rain", "Fog-Rain-Hail-Thunderstorm", "Fog-Rain-Snow", "Fog-Rain-Snow-Thunderstorm", "Fog-Rain-Thunderstorm", "Fog-Snow", "Fog-Snow-Thunderstorm", "Fog-Thunderstorm", "Rain", "Rain-Snow", "Rain-Snow-Thunderstorm", "Rain-Thunderstorm", "Snow", "Thunderstorm" or "Tornado")
- Wind direction (degrees)
- City (always "New York City (USA)")
- Season ("Spring", "Summer", "Autumn" or "Winter")

2.14 Summary

The ideas and techniques discussed in this literature review are summarised below, with references to the section in which the topic was discussed.

Artificial Intelligence (AI) (Section 2.1) refers to programs which exhibit intelligent behaviour. Intelligence is difficult to define, but key aspects include learning, adapting to the environment, acting purposefully to achieve a goal, and providing logical responses given one's knowledge. Although AI has had a rocky history, it has made practical contributions to fields such as robotics, expert systems and machine vision [30].

Artificial Life (ALife) (Section 2.2) attempts to create life-like behaviour using software, hardware, biochemistry or other media. Some ALife implementations use some sort of AI to allow the agent to make decisions.

Dennett's Tower of Generate-and-Test (Section 2.4) is a framework for ranking brain designs, which can be applied to artificial as well as biological life. Each floor in the tower represents an important increase in cognitive power. Darwinian creatures have a fixed design; they can only adapt to the environment through recombination and mutation of genes. Skinnerian creatures learn through operant conditioning; actions that are rewarded are reinforced and are more likely to be repeated in similar situations. Popperian creatures have the ability to predict the results of some actions, allowing them to weed out stupid moves without needing to experience the consequences. Gregorian creatures exhibit some form of tool use. [3, 4, p. 83-98, 5, p .258-262]

Although evolution (Section 2.3) is normally associated with biological organisms, it can apply in other contexts, including ALife, provided that the conditions are met: there must be a continuing abundance of different agents, the ability to create new agents by copying genetic information from existing agents, and competition so that the number of agents in the population with a specific trait depends on how the trait interacts with the environment [58]. Evolution may lead to a higher fitness level for agents than other forms of adaptation [59].

In biological organisms, gene expression (Section 2.6) describes how the genetic information of an organism gives rise to its physical traits. Most multicellular organisms have two sets of chromosomes, one from each parent. When corresponding genes from the two sets are not identical, one allele (form) may be dominant (expressed in the agent) and the other recessive (not expressed),

or some sort of blending may occur. Sexual reproduction may benefit ALife by encouraging a diverse gene pool, preserving partial solutions that may be useful as the environment changes [72, 73]. However, it halves the number of mating opportunities, and ALife populations are usually small due to processor limitations.

Data mining (Section 2.7) refers to extracting insight from data by discovering interesting and useful patterns. It encompasses a variety of tasks, including classification (assigning objects to predefined categories) [83] and prediction (estimating future (or unknown) data based on present data and past trends [83, 85, p. 4]. Software ALife is sometimes used for data mining.

The Self-Organising Map (SOM) (Section 2.9) provides a way to represent high-dimensional data in fewer dimensions, while preserving the topology of the input data [92]. The SOM has an established place in the data mining tool set, especially for clustering and classification. It has also been used, sometimes with modifications, in ALife [95, 96] and artificial intelligence [97, 98].

Wains (Section 2.10) are an ALife species with artificial intelligence. They live in, and subsist on, data; for them, finding patterns in data is a survival problem. Wains have two sets of chromosomes, but have only one sex. This may confer some of the advantages of sexual reproduction, without reducing the number of mating opportunities. Genes affect the appearance, configuration or capability of a wain. Wains were developed and tested using a framework called Créatúr, a reusable software framework for automating ALife experiments. Several directions for future research were identified, including improving the wain's decision-making process, and implementing cultural transmission.

Automated Speech Recognition (ASR) (Section 2.11) is the process of converting an acoustic signal (spoken language) into the corresponding sequence of words. The hidden Markov model toolkit (HTK) [120] is widely used for ASR research, and is a useful "benchmark" task for evaluating data mining techniques.

Haskell [121] (Section 2.12) is a purely functional programming language. Functional programming is a paradigm that treats expressions as mathematical functions, and avoids side effects of computation. Wains and Créatúr were developed in Haskell.

A Domain-Specific Language (DSL) (Section 2.12.1) is a special-purpose language tailored to meet the needs of a limited domain. DSLs can help the programmer to write and debug code more quickly, and make it easier for non-programmers to understand the code [125]. If the DSL is implemented withing a "container" language, it is called a Domain-Specific Embedded Language (DSEL). The new version of Créatúr developed as part of this research project uses DSELs.

Monads (Section 2.12.2) "provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism." [129]. The new version of Créatúr developed as part of this research project uses monads in all of these roles.

Generic programming (Section 2.12.3) references types to be specified later; the actual implementation is automatically generated when the types are finally specified. The Glasgow Haskell Compiler (GHC) provides support for datatype-generic programming as proposed by Magalhães et al. [133]. The new version of Créatúr developed as part of this research project uses this feature to minimise the amount of code that users of the framework must write.

The research described in this thesis uses the following data sets: a) The MNIST database (Section 2.13.1), a collection of images of hand-written numerals. b) The TI46 speech database (Section 2.13.2), a corpus of isolated spoken words recorded for both male and female speakers. c) The Cortez et al. [135] Internet traffic data (Section 2.13.3) from a private ISP with centres in 11 European cities. d) The Zonination weather database (Section 2.13.4), which contains weather data for 24 international cities [137].

Chapter 3

Approach

Recall the research questions presented in Chapter 1:

> **Research Question 1**: Will giving wains a mechanism to predict the outcomes of possible actions, and to choose the action with the best predicted outcome, make them better decision-makers?

> **Research Question 2**: Can Popperian wains learn to classify data with accuracy and speed comparable to traditional classification methods?

> **Research Question 3**: Can Popperian wains learn to forecast future values in a data stream, with accuracy and speed comparable to traditional forecasting methods?

This chapter describes the approach used to answer these questions.

De Buitléir, Russell, and Daly [2] recommended improving the wain's decision-making process, and implementing cultural transmission of information, both

from parent to offspring, and between adults. As discussed in Section 2.10, wains in the original implementation used a decision matrix with adjustable weights to determine their response to each stimulus. Actions that were rewarded are more likely to be repeated. This is operant conditioning; the original wains were Skinnerian creatures. If wains were given a way to mentally test an action and predict the results, they would become Popperian creatures, with the improvement in cognitive power that entails. This would require a complete redesign of the decision-making process.

Based on the literature review undertaken (see Chapter 2), no ALife species with Popperian-level AI has been used for data mining, so it is not obvious how wains should be used for this purpose. Recall from Section 2.7 that data mining includes the following tasks:

- landscape mining
- classification
- cluster analysis
- prediction
- regression
- modelling
- visualisation

Wains do not have the ability to perform regression. Neither can they be used for visualisation, except to the extent that their mental models can be examined. Landscape mining is a rather broad term; at present it is not clear how

wains might contribute in this area. In the original implementation of wains, the modified SOM allowed them to build a set of models representing the environment, as an aid to decision-making. Thus, at the individual level they performed cluster analysis and modelling. However, the desired output of data mining is a single, unified view of the data, not hundreds of conflicting views. It should be possible to have a population of wains co-operate to build a single, unified set of models, but designing a suitable reward system could prove challenging.

The original wains also performed classification, but to internal categories (unique to each individual wain) rather than to human categories. However, this seems to be a promising area. With better decision-making, the ability to be *taught* rather than merely learning through trial and error, and a suitable reward system, wains might be able to perform true classification. And if wains could become Popperian creatures with the ability to predict the outcome of their actions, they might also be used to predict data trends. These two areas, classification and prediction, seem to be the logical choice for using wains as data mining tools, and were the focus of this research project. To discover if wains could be truly useful for data mining, the decision was made to test them with complex data, and with data in a variety of domains, and to compare the results with traditional data mining techniques.

It is important to note that wains were not designed to do data mining, either in the original implementation or the new implementation. Instead, they were designed to be Popperian creatures, and to live in an environment of data. In order to get wains to do data mining, we frame it as a survival problem as recommended by de Buitléir [1, p. 60].

Chapter 4

Improving the Créatúr framework

In preparation for the experiments described in later chapters, several improvements were made to the Créatúr framework. These improvements were not strictly required to answer the research questions, but they made the framework much easier to use, both by the author and future researchers. This chapter describes the improvements to the framework. Some Haskell code snippets (e.g., examples of how to use the framework) are presented. However, the examples will be explained in context, so it is not necessary to know Haskell to read this chapter. (If more information is needed, Appendix A provides a brief introduction to Haskell syntax.)

The framework code had originally been combined with the code implementing the wains; it is now re-factored into a separate, reusable package called `creatur`. (Wains are now implemented in separate packages, to be discussed in Chapter 5.) The re-factored framework can support a variety of types of agents. To illustrate this, three sample agents were developed: a `Rock` which does not reproduce, a `Plant` which reproduces asexually, and a `Bug` which reproduces

sexually. A tutorial[1] is available that illustrates the use of Créatúr with these sample agents, both individually and in combination.

The new implementation of Créatúr adds features that make it easier to run experiments. It maintains a cache in memory of agents in the current population to minimise the number of times an agent is read from storage, reducing I/O time. Also, Créatúr now allows the user to define constraints on statistics that must hold after a certain number of rounds; if these constraints are not satisfied, the experiment will halt.

At the start of an experiment, the agents are not likely to be very good at the assigned tasks. They need help staying alive, or else they will die before they have time to learn. However, the environment should become harsher over time to ensure sufficient selection pressure to drive continued improvement. In the original implementation, the user had to set up a generous initial reward system, monitor the progress of the agents, and manually adjust the reward system to make rewards smaller or more difficult to get as the agents began learning. The monitoring and adjustment could be tedious.

The new implementation of Créatúr provides a mechanism which eliminates the need to adjust the reward system. It ensures that agents have a continuing incentive to learn, and helps to keep the variation in population size to approximately $\pm 50\%$. This "balancing" feature gives energy to agents, or deducts energy from them, as needed to satisfy two constraints. The primary constraint is that the average energy of agents should not exceed a user-defined threshold. An agent's energy is typically restricted to the range $(0, 1)$. If the average energy level is high, it is likely that many agents have maximal energy. Those

[1] https://github.com/mhwombat/creatur-examples/raw/master/Tutorial.pdf

agents will not gain any benefit from further energy rewards, and thus have little incentive to learn. Keeping the average energy below a certain threshold helps to ensure that most wains are motivated to continue learning.

If the primary constraint has been met, a secondary constraint is applied: the total energy of all agents at the beginning of each round should equal the total energy at the beginning of the experiment. This conservation of total system energy helps to ensure that agents compete for energy rewards. Effectively, this makes the environment gentle at the beginning of the experiment, gradually becoming harsher as the agents master the task, forcing them to compete for resources.

The most significant improvement to Créatúr is the support for automatic genetic encoding and decoding. In the original Créatúr and wain implementation, all genetic information was encoded using a custom scheme. Adding a gene to an agent's *genome* required that the programmer write code to encode and decode the new gene, and to apply any dominance relationship between alleles. The new `creatur` provides this functionality automatically for any Haskell type, so the programmer does not need to write encode and decode functions. This makes it easier to add new genes. The rest of this chapter will describe this feature.

4.1 Artificial Life genetics and recombination

Consider the agent below. Associated with the agent is a name, a flower colour, a current energy level, and some genetic information.

```
data Plant = Plant
```

```
  {
    plantName :: String,
    plantFlowerColour :: FlowerColour,
    plantEnergy :: Int,
    plantGenome :: [Bool]
  }

data FlowerColour = Red | Orange | Yellow | Violet | Blue
```

This is of course a very simple example. There is only one genetic trait, `plantFlowerColour`; it is specified by the `plantGenome`, which is encoded as a sequence of `Bool`s. (The field `plantEnergy` is not genetic; it is set to the same initial value for all `Plant`s at "birth".)

Our `Plant` type has only *one* strand of genetic material; this illustrates a common approach [138, p. 10f] in evolutionary computation that will be referred to as *simplified sexual reproduction* hereafter. During reproduction, the strands from two parents are recombined to produce two new strands. Two offspring can be created from the new strands. Alternatively, one strand may be chosen at random to create a child, and the other strand discarded. In either case, each parent contributes approximately half of its genetic information to the offspring.

Compare the definition of `Plant` with the following definition. This agent, called `Bug`, uses an approach that more closely models sexual reproduction in biology.

```
data Bug = Bug
  {
```

```
    bugName :: String,
    bugColour :: BugColour,
    bugSpots :: [BugColour],
    bugSex :: Sex,
    bugEnergy :: Int,
    bugGenome :: ([Word8],[Word8])
  }

data BugColour = Green | Purple | Red | Brown | Orange | Pink | Blue

data Sex = Male | Female
```

This agent has three genetic traits: a base colour, `bugColour`, one or more coloured spots, `bugSpots`, and sex `bugSex`. More importantly, there are *two* strands of genetic information, represented by a *tuple* containing two sequences of `Word8`s. During reproduction, the two strands from one parent are recombined to produce two new strands. One of those strands is chosen at random to become that parent's contribution to the child's genome. This is analogous to the production of a gamete in biology. The process is repeated for the other parent. Thus the child has two strands of genetic information, one contributed by each parent. As before, each parent contributes approximately half of its genetic information to the offspring.

Although there are differences in the details, the task of implementing either style of reproduction is very similar. The programmer must design a genome, implement recombination of genetic information, support occasional mutation

of genes, provide a means to encode a set of traits into a strand of genetic information, provide a means to decode strands of genetic information to determine the corresponding traits, and implement the construction of an agent from the genome.

The researcher may not care about the precise design of the genome, or its implementation, only requiring that it behaves in a way that supports evolution. Specifically, the genome and the recombination technique must be designed to ensure that offspring are similar to their parents (except in the case of mutation). A straightforward conversion of numeric values to binary is not a good approach; an agent with, say, 18 legs (10010) and one with 20 legs (10100) could produce a child with 31 legs (11111) – not very similar to either parent!

So designing, implementing, and testing a genome is not trivial. Are there tools that can make this easier? The rest of this chapter describes how Créatúr uses Haskell features such as monads, DSELs, and datatype-generic programming to address genetics and reproduction.

4.2 Gene encoding

The Créatúr library provides tools to develop an encoding scheme for a gene or an entire organism. The `Genetic` class provides the functions for encoding and decoding. (A Haskell class defines an interface for which there can be multiple implementations. In that sense, it is similar to a Java interface). There are multiple modules that implement this interface. By simply changing the `import` statement, the user can change the base type used for the encoded genes. This makes it easy for the user to benchmark different types to determine, for exam-

ple, whether [Word8] (i.e., a vector whose elements are 8-bit bytes) or [Word16] will be more efficient in a given application.

The implementation of Genetic is shown below [2]. The function put writes a gene to a sequence; get reads the next gene in a sequence. Créatúr also provides Reader and Writer monads for operating on an encoded gene sequence. These will be discussed in more detail in Section 4.6.

```
class Genetic g where
  put :: g -> Writer ()
  get :: Reader (Either [String] g)
```

Datatype-generic programming allows Créatúr to automatically generate instances for put and get. The details of how to use datatype-generic programming are described by Magalhães et al. [133] and on the Haskell wiki [140]. Here is a summary of the steps taken to allow implementations of the Genetic class to be automatically generated.

- Implement Genetic for a set of base types Bool, Char, Word8 and Word16, along with types of the form [a], Maybe a, (a, b) and Either a b, where a and b are themselves instances of Genetic.
- Create a new class, GGenetic, which handles encoding and decode the sum-of-products representation of a value.
- Implement GGenetic for each of the types used in the sum-of-products representation.

[2] Readers familiar with Haskell may wish to consult de Buitléir et al. [139] for a discussion of why the Créatúr library uses multiple modules implementing the same interface, rather than, for example, multi-parameter typeclasses.

- Provide a default implementation of put and get in the Genetic class that simply invokes the corresponding methods in the GGenetic class.

As a result, the end user can automatically create an instance of Genetic for any type without writing an implementation for put or get, as long as the type is constructed using only the supported base types. For example, we can modify the FlowerColour type to use the automatically-generated genetic encoding scheme by using the language pragma DeriveGeneric, importing GHC.Generics, and declaring FlowerColour to be an instance of Genetic. Now get and put can be used with the FlowerColour type.

```
{-# LANGUAGE DeriveGeneric #-}
...
import ALife.Creatur.Genetics.BRGCBool
import GHC.Generics
...

data FlowerColour = Red | Orange | Yellow | Violet | Blue
  deriving Generic
instance Genetic FlowerColour
```

There are five variants of Genetic. The one in ALife.Creatur.Genetics.Code.BRGCBool encodes genes to produce a sequence of Bools. This is practical when the genes of an agent have a small set of possible values. If an agent has genes with a larger number of possible values, it may be more efficient (i.e., require fewer bits) to store their genetic information as a string of numbers. ALife.Creatur.Genetics.Code.BRGCWord8 encodes genes to

produce a string of 8-bit bytes. Similarly, `ALife.Creatur.Genetics.Code.BRGCWord16` uses 16-bit words, `ALife.Creatur.Genetics.Code.BRGCWord32` uses 32-bit words, and `ALife.Creatur.Genetics.Code.BRGCWord64` uses 64-bit words.

Straightforward binary encoding of numeric genes is problematic. Performing *crossover* (discussed in 4.4) on the binary values 1000 and 0111 could easily yield 1111 or 0000. So parents with 8 legs and 7 legs could have offspring with 15 legs or 0 legs, very different from their parents. To avoid this problem, all three implementations encode integral and character values using a Binary-Reflected Gray Code (BRGC). A Gray code maps values to codes in a way that guarantees that the codes for two consecutive values will differ by only one bit [141]. This feature is useful for encoding genes because the result of a crossover operation will be similar to the inputs. This helps to ensure that offspring are similar to their parents, as any radical changes from one generation to the next are the result of mutation alone.

4.3 Reproduction

Recall that in our `Plant` example, each agent has a *single* strand of genetic information. During reproduction, the strands from two parents are recombined, creating genetic information for potential offspring. Thus, each parent contributes approximately half of its genetic information to the offspring. The recombination process will be discussed in Section 4.4.

Créatúr provides the `Reproductive` class (shown below) in the `ALife.Creatur.Genetics.Reproduction.SimplifiedSexual` module for this pur-

pose. This class can be used with either BRGCBool, BRGCWord8, BRGCWord16, BRGCWord32 or BRGCWord64, and contains three functions. The function recombine recombines the genetic information from two potential parent agents, as discussed above. The user must provide the implementation for recombine using a DSEL which will be described in Section 4.4. The function build constructs an agent from a strand of genetic information, if it is possible to do so (i.e. if the genes translate to a valid agent). The user must provide an implementation of this function as well; this is discussed in Section 4.6. Finally, the makeOffspring function takes two agents and attempts to produce offspring. A default implementation is provided, which calls recombine to create a genome for the child and calls build to construct the child.

```
class Reproductive a where
  type Strand a
  recombine :: RandomGen r => a -> a -> Rand r (Strand a)
  build :: AgentId -> Strand a -> Either [String] a
  makeOffspring :: RandomGen r
    => a -> a -> AgentId -> Rand r (Either [String] a)
```

In our Bug example, each agent has *two* strands of genetic information. During reproduction, the two strands from one parent are recombined to produce two new strands. (The recombination process will be discussed in Section 4.4.) One of these strands is chosen at random to become that parent's contribution to the child's genome. This is analogous to the production of a gamete (ovum or sperm) in biology. The process is repeated for the other parent. Thus the child has two strands of genetic information, one contributed by each parent.

As before, each parent contributes approximately half of its genetic information to the offspring.

Créatúr provides a class for this, also called Reproductive, in the ALife.Creatur.Genetics.Reproduction.Sexual module. As before, this class can be used with either of the encoding methods described in Section 4.2, and contains three functions. The produceGamete function recombines the twin strands of genetic information from two potential parents, using the technique described above. The user must provide the implementation for recombine using the DSEL described in Section 4.4. The function build constructs an agent from two strand of genetic information, if possible. The user must provide an implementation of this function; this will be discussed in Section 4.6.

Finally, the makeOffspring function takes two agents and attempts to produce offspring. A default implementation is provided, which calls produceGamete to produce a single strand of genetic information from each parent, pairs the two strands to create a genome for the child, and calls build to construct the child.

The Reproductive API from the ALife.Creatur.Genetics.Reproduction.Sexual module is shown below.

```
class Reproductive a where
  type Strand a
  produceGamete :: RandomGen r => a -> Rand r (Strand a)
  build :: AgentId -> (Strand a, Strand a) -> Either [String] a
  makeOffspring :: RandomGen r
    => a -> a -> AgentId -> Rand r (Either [String] a)
```

4.4 Gene recombination

As described in Section 4.3, reproduction of both `Plants` and `Bugs` involve shuffling a pair of sequences to produce two new pairs, and possibly discarding one of the sequences. Additionally, occasional random mutations are allowed. The `ALife.Creatur.Genetics.Recombination` module in the Créatúr library provides a DSEL for genetic recombination. These operations can be applied with specified probabilities and combined in various ways. Two common operations are crossover and *cut-and-splice*. In crossover (Figure 4.1), a single crossover point is chosen. All data beyond that point is swapped between strings. In cut-and-splice (Figure 4.2), two points are chosen, one on each string. This generally results in two strings of unequal length.

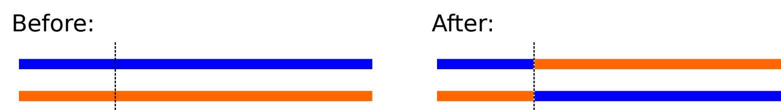

Figure 4.1: Crossover

Figure 4.2: Cut-and-splice

Here's a sample program that might be used to shuffle two sequences of genetic material. (The numbers 0.1, 0.01, and 0.001 are used for illustration.)

```
withProbability 0.1 randomCrossover (xs, ys) >>=
withProbability 0.01 randomCutAndSplice >>=
withProbability 0.001 mutatePairedLists >>=
randomOneOfPair
```

To illustrate how this program would work, suppose this program acted on the following pair of sequences:

```
([A,A,A,A,A,A,A,A,A,A],[C,C,C,C,C,C,C,C,C,C])
```

The first line of the program has a 10% probability (0.1) of performing a simple crossover at a random location, perhaps resulting in:

```
([A,A,A,A,A,A,A,C,C,C],[C,C,C,C,C,C,C,A,A,A])
```

The second line of the program has a 1% probability (0.01) of performing a cut-and-splice, perhaps resulting in:

```
([A,A,A,A,C,A,A,A],[C,C,C,C,C,C,A,A,A,C,C,C])
```

The third line of the program has a 0.1% probability (0.001) of mutating one or both sequences, perhaps resulting in

```
([T,A,A,A,C,A,A,A],[C,C,C,C,C,C,A,A,C,C,C,C])
```

After the first three operations, we have two new sequences. In this example, we only want one of the sequences, so the final line randomly chooses one.

To perform more than one crossover, the operation can simply be repeated as shown below. (The numbers 0.1, and 0.08 are used for illustration.)

```
withProbability 0.1 randomCrossover (xs, ys) >>=
withProbability 0.08 randomCrossover (xs, ys)
```

Alternatively, we can choose the number of crossover operations at random. The function `repeatWithProbability` performs an operation a random number of times, such that the probability of repeating the operation `n` times is p^n. (The number 0.1 is used for illustration.)

```
repeatWithProbability 0.1 randomCrossover (xs, ys)
```

Table 4.1 contains the full list of available operators.

4.5 Gene expression

The `Diploid` class, in the module `ALife.Creatur.Genetics.Diploid`, represents paired genes or paired instructions for building an agent. `Diploid` (shown below) contains the function `express`. Given two possible forms of a gene or gene sequence, `express` takes into account any dominance relationship, and returns a gene representing the result. Créatúr uses datatype-generic programming (discussed in Section 4.2) to provide a default implementation of `Diploid`, including `express`.

```
class Diploid g where
  express :: g -> g -> g
```

Default implementations of `Diploid` are provided for the following types: `Bool`, `Char`, `Double`, `Int`, `Word`, `Word8`, `Word16`, `Word32`, and `Word64`, along with sequences, tuples, and sums or products of any types that themselves implement `Genetic`. In practice, this means that the user can often create an instance of `Diploid` without writing an implementation for `express`.

In the default implementation of `express` "small" is dominant over "large". Small numeric values are dominant over larger ones. If arrays are of different lengths, the result will be as long as the shorter array, as illustrated below.

```
express [1,2,3,4] [5,6,7,8,9]  →  [1,2,3,4]
```

Table 4.1: The Recombination DSEL.

function and description
`crossover :: Int -> ([a], [a]) -> ([a], [a])` Cuts the list `xs` at position `n`, cuts the list `ys` at position `m`, swaps the ends, splices them, and returns the modified pair. The result will be `(xs[0..n-1]++ys[m..], ys[0..m-1]++xs[n..])`
`cutAndSplice :: Int -> Int -> ([a], [a]) -> ([a], [a])` Cuts both the lists `xs` and `ys` at position `n`, swaps the ends, splices them, and returns the modified pair. This is equivalent to `cutAndSplice n n (xs,ys)`.
`mutateList :: (Random n, RandomGen g) => [n] -> Rand g [n]` Mutates a random element in the list `xs`, and returns the modified list.

`mutatePairedLists`
`:: (Random n, RandomGen g) => ([n], [n]) -> Rand g ([n], [n])`
Randomly chooses `xs` or `ys`, mutates a random element in that list, and returns the modified list.

`randomOneOfList :: RandomGen g => [a] -> Rand g a` Randomly returns one element from the list `xs`.
`randomOneOfPair :: RandomGen g => (a, a) -> Rand g a` Randomly returns `x` or `y`.

`randomCrossover`
`:: RandomGen g => ([a], [a]) -> Rand g ([a], [a])`
Same as `crossover`, except that `n` is chosen at random.

`randomCutAndSplice`
`:: RandomGen g => ([a], [a]) -> Rand g ([a], [a])`
Same as cutAndSplice, except that `n` and `m` are chosen at random.

`withProbability`
`:: RandomGen g => Double -> (b -> Rand g b) -> b -> Rand g b`
Either applies op to `x` (with probability p) and returns the result, or returns the unmodified `x` (with probability `p-1`).

`repeatWithProbability`
`:: RandomGen g => Double -> (b -> Rand g b) -> b -> Rand g b`
Applies op to `x` random number of times. The probability of applying op `n` times is p^n.

Consider the following type, which has three constructors: one that takes a Boolean parameter, one that takes an integer parameter, and a recursive version that takes both a Boolean and an integer, as well as an array whose elements are of the same type as its parent.

```
data MyType = MyTypeA Bool | MyTypeB Int
  | MyTypeC Bool Int [MyType] deriving (Show, Generic)
instance Diploid MyType
```

Here are some examples of how `express` operates.

```
express (MyTypeA True) (MyTypeA False)  →  MyTypeA True
express (MyTypeB 2048) (MyTypeB 36)     →  MyTypeB 36
```

When a type has multiple constructors, the constructors that appear earlier in the definition are dominant over those that appear later. For example:

```
express (MyTypeA True) (MyTypeB 7)          → MyTypeA True
express (MyTypeB 4) (MyTypeC True 66 []) → MyTypeB 4
```

Even with complex data structures, the implementation should just "do the right thing".

```
express
  (MyTypeC False 789 [MyTypeA True, MyTypeB 33,
    MyTypeC True 12 []])
  (MyTypeC True 987 [MyTypeA False, MyTypeB 11,
    MyTypeC True 3 []])
  → MyTypeC True 789
    [MyTypeA True, MyTypeB 11, MyTypeC True 3 []]
```

Given a numeric type, it would seem that the logical way to express two values is to average them. So why use the smaller value instead? In the author's experience with ALife, numeric genes usually control the resources used by an agent. Examples include a gene which specifies the number of neural connections in the agent's brain, or a gene which controls the age at which offspring become mature and are no longer dependent on a parent. Choosing the smaller number helps to ensure that agents use resources efficiently. Of course, a different dominance rule can be used by writing a custom implementation of `express`.

4.6 Constructing an agent from its genome

This section demonstrates how monads are used to create tools for constructing agents. As mentioned in Section 4.3, implementations of the class `Reproductive` must implement the function `build`, which constructs an agent from a genome, if the genome is valid. To see how this is done, recall the definition of `Plant` from Section 4.1.

```
data Plant = Plant
  {
    plantName :: String,
    plantFlowerColour :: FlowerColour,
    plantEnergy :: Int,
    plantGenome :: [Bool]
  }
```

```
data FlowerColour = Red | Orange | Yellow | Violet | Blue
```

To create a plant, we need to determine the flower colour from the genome, and set the ID and energy. The `BRGCBool`, `BRGCWord8` and `BRGCWord16` modules define a monad called `Reader` (unrelated to `Control.Monad.Reader`), which provides functions for decoding a strand of genetic information. Thus, the `Reader` monad is a DSEL for reading genomes; this language is defined in Table 4.2.

Table 4.2: The Reader DSEL.

function and description
`get :: Reader (Either [String] g)` Reads the next gene. If it can be decoded, returns the decoded value. Otherwise, returns a list of error messages.
`getWithDefault :: g -> Reader g` Reads the next gene. If it can be decoded, returns the decoded value. Otherwise, returns the default value
`copy :: Reader Sequence` Return the entire genome.
`consumed :: Reader Sequence` Return the portion of the genome that has been read (by `get` or `getWithDefault`).

We can write a `buildPlant` method using this DSEL. The function will take a `String` (a unique identifier of the plant to be created), and it will return a program that runs in the `Reader` monad. That program will return a either a list of `String`s containing error messages, or a plant. Thus, the type signature for the `buildPlant` function is:

```
buildPlant :: String -> Reader (Either [String] Plant)
```

Now to write the program. First, each plant needs a copy of its genome in order to produce offspring; we can use the copy function to obtain this. Next, we determine the colour of the plant. We could use the method get, which returns a Maybe value containing the next gene in a sequence. But consider that our sequence of Bools may not be a valid code for any colour. If an error occurs, we could treat the mutation as non-viable and return Nothing. However, in this example, we wish to create a plant no matter what errors are in the genome, so we will use getWithDefault, with Red as the default value. All plants start life with an energy of 10. Here is the program:

```
buildPlant name = do
  g <- copy
  colour <- getWithDefault Red
  return . Right $ Plant name colour 10 g
```

Now, buildPlant is a function that returns a program that runs in the Reader monad. How do we run that program? ALife.Creatur.Genetics.BRGCBool, ALife.Creatur.Genetics.BRGCWord8 and ALife.Creatur.Genetics.BRGCWord16 provide a function for this purpose, called runReader. Now we have everything we need to declare Plant to be an instance of Reproductive.

```
instance Reproductive Plant where
  type Base Plant = Sequence
  recombine a b =
    withProbability 0.1
        randomCrossover (plantGenome a, plantGenome b) >>=
```

```
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runReader (buildPlant name)
```

Recall the definition of Bug from Section 4.1.

```
data Bug = Bug
  {
    bugName :: String,
    bugColour :: BugColour,
    bugSpots :: [BugColour],
    bugSex :: Sex,
    bugEnergy :: Int,
    bugGenome :: ([Word8],[Word8])
  }

data BugColour = Green | Purple | Red | Brown | Orange | Pink | Blue

data Sex = Male | Female
```

It has have two strands of genetic information which determine the bug's traits. The `BRGCBool`, `BRGCWord8` and `BRGCWord16` modules define a monad called `DiploidReader` for this situation. The `DiploidReader` monad is also DSEL; this language is defined in Table 4.3.

Our `buildBug` method will take a `String` (a unique identifier), and it will return a program that runs in the `DiploidReader` monad. The implementation

Table 4.3: The DiploidReader DSEL.

function and description

`getAndExpress`
`:: (Genetic g, Diploid g) => DiploidReader (Either [String] g)`
Reads the next pair of genes from twin strands of genetic information. If the genome can be decoded, takes into account any dominance relationship and returns the decoded value. Otherwise, returns a list of error messages.

`getAndExpressWithDefault`
`:: (Genetic g, Diploid g) => g -> DiploidReader g`
Reads the next pair of genes from twin strands of genetic information. If the genome can be decoded, takes into account any dominance relationship and returns the decoded value. Otherwise, returns the default value

`copy2 :: DiploidReader DiploidSequence` Returns the entire genome (both strands).
`consumed2 :: DiploidReader DiploidSequence` Returns the portion of each strand that has been read (by `get` or `getWithDefault`).

is similar to `buildPlant`, except that the single-strand operations have been replaced with versions that work with both strands.

```
buildBug :: String -> DiploidReader (Either [String] Bug)
buildBug name = do
  sex <- getAndExpress
  colour <- getAndExpress
  spots <- getAndExpress
  g <- copy2
  return
   $ Bug name <$> sex <*> colour <*> spots <*> pure 10 <*> pure g
```

The `runDiploidReader` function runs a program written in the `DiploidReader` DSEL and returns the result. Now we can implement `Reproductive`.

```
instance Reproductive Bug where
  type Base Bug = Sequence
  produceGamete a =
    repeatWithProbability 0.1 randomCrossover (bugGenome a) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runDiploidReader (buildBug False name)
```

The `BRGCBool`, `BRGCWord8` and `BRGCWord16` modules also define a monad called `Writer`, used for encoding genetic information. This is useful for generating an initial population. The `Writer` DSEL consists of one function, put, which writes a gene to a sequence.

One approach to creating an initial population is to feed random strings of genetic information into the function that builds the agent, but instruct it to keep only as much of the sequence as it needs to build a complete agent. The functions `consumed` (from the Reader DSEL) and `consumed2` (from the DiploidReader DSEL) are useful here. For example, we can modify the `buildBug` method from Section 4.6 to accept a Boolean that tells it whether or not to discard the unread portion of the sequences.

```
buildBug :: Bool -> String -> DiploidReader (Either [String] Bug)
buildBug truncateGenome name = do
  sex <- getAndExpress
  colour <- getAndExpress
  spots <- getAndExpress
  g <- if truncateGenome then consumed2 else copy2
  return $
    Bug name <$> sex <*> colour <*> spots <*> pure 10 <*> pure g
```

4.7 Limitations

Evolution in the Créatúr framework is limited by the data types and build functions defined by the user. For example, the `Plant` type allows the flower colour to be red, orange, yellow, violet or blue. If the initial population of `Plants` contained only red and yellow flowers, the species could evolve orange, violet and blue flowers. However, it could not evolve flowers or other colours, or flowers with spots. Similarly, `Plants` could not evolve legs or sexual dimorphism. The `buildPlant` function expects a single strand of genetic material, so `Plants`

could not evolve into diploid organisms. The `Bug` species cannot evolve wings or tails because those fields do not exist in the `Bug` type, and they cannot evolve a different ploidy because it is not supported by the `buildBug` function.

In addition to recombination and mutation, there are other types of genetic changes that occur in biology. *Horizontal gene transfer* is the mechanism where genetic material is transferred directly from one organism to another instead of vertically from parent to offspring. It is a common mechanism for the development of antibiotic resistance [142]. There are three types of horizontal gene transfer: *transduction*, where foreign genetic material is introduced by a virus, *conjugation*, where genetic material is transferred between organisms in direct contact, and *transformation*, where an organism incorporates naked genetic material from its surroundings [142]. Although Créatúr does not implement these mechanisms, the user could implement them using the monads provided.

4.8 Summary

Re-factoring the Créatúr framework and wains code into re-usable packages makes it easier to create new agent types and experiments. Agent caching can help to reduce I/O time by reducing the number of times an agent must be read from storage. Users can define constraints on statistics which must be satisfied or the experiment will halt; this can help the user identify problems more quickly and avoid wasting processing resources on experiments which are not progressing satisfactorily. This is particularly useful as experiments may require multiple days to run.

The automatic population "balancing" feature eliminates the need to repeat-

edly adjust the experiment configuration as agents learn the assigned task. Chapters 8 and 9 will present a series of experiments; in each case, a single configuration was used throughout the duration of the experiment.

Créatúr supports both sexual and asexual reproduction, with a flexible DSEL to control recombination and mutation. It was straightforward to use the datatype-generic programming feature of GHC to specify how to derive instances of `Genetic` and `Diploid`. The user has it even easier; they can simply declare their custom types to be instances of these classes, taking advantage of the default implementation provided by Créatúr.

Each of the DSELs developed for Créatúr required only a small set of operations; it was easy to embed them in Haskell. This avoids the need to design a language and write a parser for it. The user does not have to learn a "new" language, and rather than being restricted to the semantics of the DSEL, the user has access to all the features of Haskell, if needed. Finally, using monads for the `Reader`, `DiploidReader` and `Writer` DSELs allowed us to isolate the stateful computations required to read and write genes.

The full source code for Créatúr is available on GitHub [143]; a tutorial is also provided [144].

Chapter 5

Improving the wain

This chapter describes the new wain implementation, which is at the core of the research described in this thesis, and discusses some of the changes that have been made since the original implementation.

5.1 Architecture

The wain implementation was originally combined with the framework; the code has now been re-factored into a set of packages which communicate through APIs, making it easier to extend and maintain. In the original implementation, wains could only interact with grey-scale images, and other wains. The new implementation includes custom wains for working with two data formats, audio and numeric vectors. A generic wain implementation is also available, upon which new custom wains can be based.

Figure 5.1 shows the architecture for a typical experiment using wains. Suppose *xxxx* indicates the data format used in the experiment (e.g., vector, image,

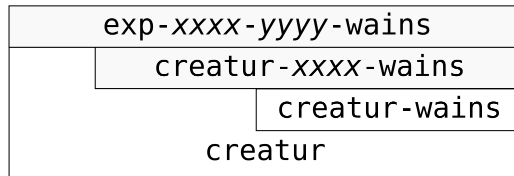

Figure 5.1: Architecture for a typical experiment.

or audio), and *yyyy* indicates for the desired behaviour of the wains (e.g. classifying, forecasting, or clustering). We will examine each component in the diagram, beginning at the bottom and moving upward.

The `creatur` package (Figure 5.1) provides the framework, and an Application Programming Interface (API) of common functions which are used by other packages. The functionality provided by this package has already been discussed in Chapter 4.

The `creatur-wains` package (Figure 5.1) provides a generic wain implementation, and an API for common functions that a specific wain implementation may need. It uses the automatic genetic encoding and decoding provided by the `creatur` package.

The package `creatur-`*xxxx*`-wains` (Figure 5.1) is a placeholder for a package that implements a wain able to read *xxxx* data, and choose from a predefined repertoire of actions. Typically such a package would use the automatic genetic encoding and decoding provided by `creatur`. The currently available packages that can be used in this slot in the architecture are listed below.

creatur-audio-wains A wain that interacts with MFCC feature vectors for audio samples.

creatur-image-wains A wain that interacts with grey-scale images.

creatur-uivector-wains A wain that interacts with numeric vectors, where each vector element is in the interval $[0, 1]$.

The package `exp-`*xxxx*`-`*yyyy*`-wains` (Figure 5.1) is a placeholder for a package that sets up the experiment according to a configuration file, reads the data and presents it to the wains, defines the repertoire of actions that a wain can take, and determines the reward system. Typically it would use tools for managing a daemon, task scheduling, logging, and managing persistence from the package `creatur`, and the wain implemented in the package `creatur-`*xxxx*`-wains`. The currently available packages that can be used in this slot in the architecture are listed below.

exp-audio-id-wains Experiment to identify samples of spoken numerals.

exp-image-id-wains Experiment to identify images of handwritten numerals.

exp-uivector-prediction-wains Experiment to forecast the next value of a variable, given historical values of the variable of interest, and the historical values of related variables (optional).

A typical ecosystem for wain experiments contains two types of objects, wains and data. Using the Créatúr API, the user writes a simple daemon which reads the current list of wains, queues the wains in random order, and processes the queue, giving each wain a CPU turn. A different random order is used to ensure that no wain has an unfair advantage. For example, the pool of energy rewards available for each CPU turn may be fixed, and could be exhausted before the turn is finished. If they were always processed in the same order, wains near the end of the list could receive less energy on average.

The daemon then selects one or more objects, where each object may be a data object or another wain, and presents them to the wain whose CPU turn it is, prompting it to select a response from a predefined set of actions. The daemon runs the action, determines the appropriate rewards for the selected action and gives them to the wain. The daemon also applies the metabolism cost (which will be discussed in Section 5.3), and returns the wain to the pool. The daemon is also responsible for housekeeping tasks such as separating children from parents when the children reach maturity, and removing wains from the population when they exceed the maximum age set by the user.

5.2 Condition

The factors related to the wain's state of health and contentment are called its *condition*. Like biological animals, wains try to act in ways that will improve or at least maintain their condition. This section defines those factors, and discusses how wains evaluate their condition.

Wains have an *energy level* in the interval $[0, 1]$. They gain or lose energy as a result of the reward system, which is unique to the experiment. For example, a wain might be rewarded for accurately identifying a pattern. If a wain's energy falls below zero, it dies.

Wains also have a *boredom level* and a *passion level*, each in the interval $[0, 1]$. The wain's boredom level is set to one at birth. Depending on the reward system, boredom might be set to zero as a reward for novelty-seeking behaviour. After the reward, the passion level will increase steadily until the next reward (up to a maximum of one). The wain's passion level is set to one at birth, and set

to zero when it mates. After mating, the passion level will increase steadily until the next mating (up to a maximum of one). In the original implementation, the rate at which the passion and boredom levels increased was fixed, and identical for all wains. In the new implementation, these rates are genetically determined. Collectively, the wain's energy level, passion level, boredom level, and whether or not it is currently rearing a child, constitute its condition.

Wains seek to maximise their *happiness.* In the original implementation, happiness was given by a weighted sum of the wain's energy level, its passion level, and its boredom; the weights were fixed. Happiness in the new implementation also takes into account a wain's parental status, which might encourage wains to rear children for longer, counterbalancing the drive to rear children quickly (caused by the higher metabolism cost a wain pays when rearing children). The weights are now genetically determined, which allows evolution to optimise the equation for survival. The new happiness equation is

$$happiness = w_e e + w_p(1 - p) + w_b(1 - b) + w_l l, \tag{5.1}$$

where e is the wain's energy level; p its passion level; b its boredom level; l is 1 if the wain is currently rearing a child, 0 otherwise; and w_e, w_p, w_b, w_l are genetically determined weights. The weights are normalised so that the happiness lies in the interval $[0, 1]$.

Because the weights are genetically determined, each agent can have a different definition of happiness, which need not relate to the agent's fitness (in Darwinian terms). An agent could have weights that cause it to be happier when it is less fit. However, such an agent would tend to further reduce its fitness as it

seeks to maximise its happiness, and would likely die young and produce fewer offspring. Over many generations, we would expect agents that a sensible definition of happiness (i.e., where the weights cause them to be happier when they are fit) to dominate the population. Thus, in the long run, happiness should be somewhat correlated to fitness.

The original implementation used the boredom field to encourage wains to more fully explore their environment, by rewarding play with a decrease in boredom. However, it is not required for all experiments. When not required, it can simply be ignored; the boredom level will remain constant throughout a wain's life, and thus will not affect its behaviour.

Wains are rewarded (with energy, a reduction in boredom, or a reduction in passion) when they perform desirable behaviours. This increases their happiness, which encourages them to repeat the behaviour. For example, a wain that makes an accurate prediction might be given an energy reward. In the original implementation, the reward system was fixed. In the new implementation, the reward system is customised for each experiment.

It should be noted that terms such as "happiness", "boredom', "passion', and "metabolism" (introduced in the next section) are intended only as convenient mnemonics for some important numeric values. Wains do not feel happiness, boredom or passion; they do not have a biological metabolism. In 1976, McDermott [145] was critical of this sort of "wishful mnemonic" because it might imply that the concept which inspired the mnemonic (such as happiness or boredom) was well-understood and in some sense *solved*. However, a modern scientific audience is more familiar with the limitations of AI, and is therefore less likely to read more into these terms than is intended. Having warned the reader not

to infer too much meaning into these terms, the author feels justified in using them for their mnemonic value.

5.3 Metabolism

Everything a wain does, even just being alive, should have an energy cost. (This drives evolution by ensuring that unfit individuals are unlikely to survive long or produce many children.) Wains lose energy at regular intervals; this is called the *metabolism tax*. In the original implementation, the metabolism tax was a fixed calculation based on the wain's brain complexity and sensory capacity. In the new implementation, the calculation of the metabolism tax can be customised for each experiment, and is deducted every time a wain has a CPU turn. (The design of the metabolism tax calculation for one set of experiments is discussed in Section 9.1.1.) As a rule of thumb, it is convenient to scale the tax according to one or more of the most scarce resources used by wains. Typically this is the CPU usage (for which the number of classifier models may be good proxy).

5.4 Appearance

Wains have one external sensory input, used to recognise both data objects and other wains. A wain's appearance is simply data; it is in the same format as the data used in the experiment. Thus, the appearance of a wain that interacts with images is also an image; the appearance of a wain that works with audio samples is an audio sample. Typically, the initial population of wains generated for an experiment will all have an identical appearance, one that is easily distin-

guished from ordinary data objects. For example, if the data consists of images of handwritten numerals, an image of an 'X' might be used for the appearance of wains in the initial population.

A wain's appearance is genetically determined. Over time, recombination and mutation can cause the appearance of wains to diverge from that of the initial population. This could allow wains to estimate how genetically close another wain is, by observing the similarity of its appearance. In theory, this could allow wains to choose whether or not to co-operate with others based on kinship. If sub-populations emerge with strong genetic differences such that offspring from cross-mating would no longer be viable, or at least not fertile, wains would likely be able to differentiate between their own kind and other by their appearance. Speciation occurs when previously interbreeding populations no longer (or only rarely) produce fertile offspring. This might occur due to genetic, behavioural, or anatomical differences, or geographic separation. In the case of software-based ALife such as wains, the first two causes of speciation are most clearly applicable. However, some sort of simulated anatomy or geography could be implemented; this might also lead to speciation.

5.5 Brain

The brain is responsible for choosing appropriate actions in response to the objects that the wain encounters. To do this, it creates an internal set of models representing the types of objects the wain has seen. These models are fluid; they become broader or narrower according to the inputs a wain receives during its lifetime.

In the original implementation, the repertoire of available actions was fixed. In the new implementation, this is customised for each experiment. For example, if a wain encounters a data object, it might attempt to mate with it (which makes sense if the object is a wain) or classify it (which makes sense if the object is a data object).

The brain has been completely re-designed since the original implementation. One of the most interesting changes is that the brain now predicts how happy the wain will be after each possible response. The new brain design will be presented in Chapter 7.

5.6 Genetics and reproduction

In the original implementation, the wain genome was encoded as a series of building instructions. Adding a new gene would require writing a custom encoder and decoder for that gene. The new implementation uses the encoding tools described in Chapter 4. When encoded, the wain genome consists of a sequence of `Word8`s. Most genes use the default encoding scheme and the default gene expression rules.

Depending on the needs of the experiment, mating typically occurs in one of two ways: *free mating* or *directed mating*. In free mating, when a wain encounters another wain, the first wain can choose to flirt, at which point it will pay the *flirtation tax* specified in the experiment's configuration. The flirtation tax could help to encourage wains to choose the most suitable partners rather than flirting indiscriminately. If the first wain is not currently rearing a child, mating will occur. Free mating gives wains control over when to mate, and with

whom. Since flirtation and child-rearing cost energy, wains have an incentive to recognise others of their kind. A wain's appearance is genetic and subject to mutation, differences in appearance are likely to be correlated with other genetic differences. As a result, wains could learn to distinguish close relatives from distant relatives or unrelated wains, which could encourage the formation of family groups or tribes.

The new implementation provides an alternative strategy, directed mating, in which wains are randomly allowed to mate with a frequency specified in the experiment's configuration. Directed mating can be useful for simpler experiments where co-operation and other social behaviours are not needed. Since the wains do not have control over mating, there is no need for a flirtation tax. Again, if the first wain is not currently rearing a child, mating will occur.

As discussed in Section 2.10, wains are diploid (they have two strands of genetic information), but they only have one sex (any wain can mate with any other wain). The child's genes are assembled using the sexual reproduction approach discussed in Section 4.3. Each parent contributes approximately half of the child's genetic material, using the algorithm which was discussed in Section 4.4. The frequency of crossover, cut-and-splice, and mutation were 10%, 1%, and 0.1%, respectively. In the original implementation, these values led to rapid evolution and good diversity in the gene pool.)

The genetic strands contributed by each parent are paired, and the dominance rules are applied to determine the traits to be expressed in the child. If possible, a new wain (the child) is constructed from the result. (Due to mutation or crossover, it is possible for a gene sequence to end prematurely or not encode a valid sequence of alleles.) Each parent transfers some energy to the child; the

amount transferred is determined by the parent's genes.

5.7 Child-rearing

When a child is born, the parent who initiated the mating (by flirting) becomes the child's *carer*. The child remains with the carer until it is mature; the age of maturity is genetically determined. During this time, the child shares in any energy rewards or penalties earned by the carer (apart from metabolism costs, discussed in Section 5.3). The fraction of the energy reward or penalty given to the child is also genetically determined.

A child receives the same sensory inputs as the carer, allowing it to build a mental set of models representing its environment. In the original implementation, children did not learn to make decisions. In the new implementation, a child observes its carer's actions, and learns decision rules based on the assumption that the carer's action in response to each situation is appropriate. This form of learning uses the *imprinting* mechanism which will be described in Section 7.4.

5.8 Summary

The wain code has now been re-factored into a set of reusable packages which communicate through APIs. Wains can now be used to perform multiple tasks with a variety of data formats, and support is provided for future extensions to handle new tasks and data formats. Wains now have a more flexible concept of "happiness". Available actions and reward systems can be customised for each

experiment. Finally, the wain code takes advantage of the improved Créatúr framework for reproduction.

The improvements to the brain will be discussed in Chapter 7.

Chapter 6

The Self-generating model

In support of the brain improvements planned for wains, the SOM was modified to create the Self-Generating Model (SGM). This chapter describes the motivation for those changes, and the design and testing of the SGM.

The requirements for a classifier used in intelligent data mining ALife agents are rather different than for more common applications. For example, in recognising handwritten or spoken numerals, it is not necessary to preserve the topology of the input data set. (We may not be interested in knowing whether a particular '3' is more similar to an '8' or a '6'.) In the original implementation of wains, a small modification was made to the SOM to improve performance. By updating only the winning node, the topology-preserving ability of the SOM was sacrificed in favour of speed.

If we dispense with topology preservation, what is the cost? Consider that in addition to a SOM-like classifier, the brain of an intelligent data mining ALife agent might include a mechanism that uses the information provided by the classifier to determine what response to take. This is the approach used in the

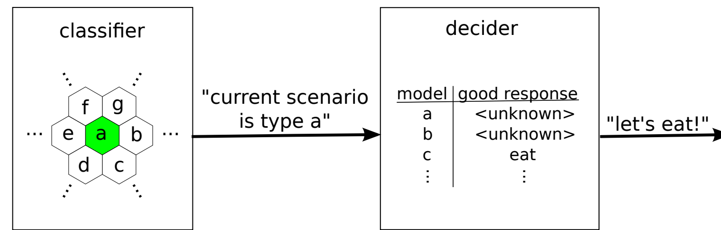

Figure 6.1: Decision-making using a classifier that preserves topology.

original implementation; the mechanism was called the decider. Suppose that the classifier assigns the label a to the current scenario, and the decider does not know a good response to a. If the classifier preserves the topology of the input data, the decider can look for the nearest neighbour of a for which it *does* know a good response, and choose that (see Figure 6.1). If a good response to a neighbour of a is likely to be a good response to a, this tactic could benefit the agent's survival.

However, there may be other ways to achieve the same result. The classifier could report the similarity of the scenario to *all* models, including the model labelled a. (This information is calculated anyway as part of the SOM algorithm.) Without needing to know anything about the topology used by the classifier, the decider can look for known responses to models that are similar to the scenario, and choose a response that is known to be good for a similar model (see Figure 6.2). Thus, we can sacrifice topology preservation in favour of other goals, introduced below.

One advantage of the SOM for intelligent agents is that the models can be extracted from the classifier, making it easier to understand how an agent perceives the object, and evaluate any decisions the agent makes in response. This is a desirable feature to keep. In a traditional Artificial Neural Network (ANN),

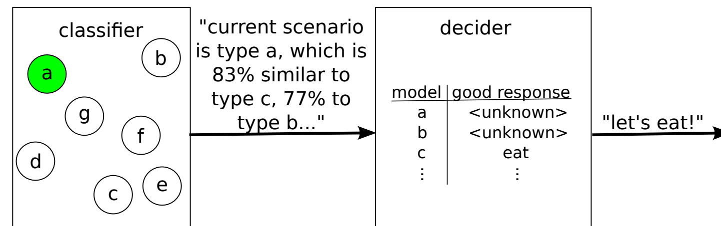

Figure 6.2: Decision-making using a classifier that does not preserve topology.

it can be difficult or even impossible to analyse why the net makes certain classifications.

Many SOM modifications are motivated by a desire for greater accuracy in classifying; however, this may not be necessary for some agent implementations. In a multi-agent system one can ask the same question of multiple agents, each with a different set of lifetime experiences, to get independent opinions. By averaging the responses, a "wisdom of the crowd" effect could produce greater accuracy than a single agent could achieve. Thus, increasing accuracy was not a goal for this project.

However, **early accuracy** is an important goal. Agents continue to learn throughout their lives, but they cannot wait until they have a full, final set of models to begin learning rules for survival. They need to be able to "hit the ground running". An agent should have a useful, if small, set of models early in life; this will allow it to experiment with possible responses to objects in their environment, and to learn from the results.

Another goal in adapting the SOM was **model stability**. Agents make decisions based on the patterns that they encounter, and the mental categories (node labels) associated with the patterns. Agents experiment by trying different actions in response to each cluster of patterns. Through trial and error,

each agent develops rules that select the appropriate action to take in response to each pattern cluster. If models change to such a degree that they no longer match patterns that they used to match, agents may need to "unlearn" existing rules and replace them with new ones.

As will be shown in Section 6.3, SOM models can be very unstable. This can make it more difficult for agents to learn appropriate responses for their environment. For example, suppose the environment has both edible and poisonous berries; and an agent can distinguish between them by some characteristic such as colour. We would expect an agent to develop at least one model that matches edible berries but not poisonous ones. The agent has a better chance of surviving if it learns to eat objects that match this model. Now imagine that the model changes so much that it now matches the poisonous berries. The eating response that the agent has learned is now dangerous. In order to survive, the agent must "unlearn" the eating response and learn a more appropriate action. Alternatively, if the model changes from matching poisonous berries to edible ones, the agent may have ruled out eating anything that matches this model, and may never try eating the edible berries. This could be disastrous if there is only one source of food.

Once those goals are met, there are additional features that would be desirable in a modified SOM. It would be advantageous to have a **faster algorithm**; this can be achieved if we **minimise** what the author calls **wasted models**. Models that will not be used to classify future patterns are wasted; the computational effort to create and update those models is unnecessary. This is especially important when working with population of agents, each with their own SOMs, as any inefficiencies in the algorithm would be amplified.

Note that model stability and model usage are independent concepts. A model can be stable (continuing to be a good match for the stimulus it was created in response to) but wasted (is not the winning node at any point during the testing or classification phase).

Finally, the modified SOM should be suitable for a variety of data mining applications. Thus, it should be a **generic algorithm**, not one that was tailored to a specific type of data such as images or audio samples.

Why modify the SOM, when other classifier algorithms are available that are also capable of *unsupervised learning*? It is often impractical for an agent to keep a copy of every data input it has encountered during its life; fortunately the SOM only requires that we keep the models. Contrast this with an algorithm such as k-means which requires that we re-calculate the centroid at each step, accessing all of the data seen previously [84]. Particle Swarm Optimisation (PSO) similarly iterates over all the data, making it unsuitable for this application [146].

Learning Classifier Systems (LCSs) [147, Section 3.9] learn the best action to take in response to a set of conditions. As such, the LCS might be suitable as a replacement for *both* the classification and decision-making components in a wain (to be discussed in Section 6.4). However, it seems overkill to replace just the classifier with an LCS. Finally, as mentioned earlier, SOM models can be inspected directly. A trained neural network stores what it has learned as weights [84]; making it difficult to extract the models.

6.1 Self-Generating Model

To satisfy these goals, the basic SOM algorithm has been adapted to produce the SGM algorithm (see Algorithm 2). The SGM can be initially empty, or it can be initialised with a set of (possibly random) models. Step 2, adjusting the winning node, has been modified to allow the classifier to grow as needed and produce models that are useful as soon as they are created. In addition, Step 3 of the SOM algorithm, adjusting models in the neighbourhood of the winning node, has been eliminated in an attempt to improve performance and minimise wasted models. The *difference threshold* helps to ensure that models do not change too much during the lifetime of the SGM, providing model stability Like the SOM, the SGM design is generic; it has not been tailored to a specific kind of data.

Algorithm 2 SGM algorithm.

For each input pattern,

1. Compare the input pattern to all models in the SGM. The node with the model that is most similar to the input pattern is called the winning node.
2. If the difference between the input pattern and the winning node's model is greater than the difference threshold, and the SGM is not at capacity (number of models $<$ maximum), a new model is created that is identical to the input pattern. Otherwise, the winning node's model is adjusted to make it slightly more similar to the input pattern. The amount of adjustment is determined by the learning rate, which typically decays over time.

Note that while SGM can grow (add new models), it can never contract (remove existing models). As discussed in Section 2.10, it was found that the ability to remove models did not confer an evolutionary advantage on the original wains.

6.2 Experimental set-up

The experiments described in this chapter used the MNIST database (presented in Section 2.13.1). The database images were used without modification. For all experiments, the SOM used the learning function given by Equation 6.1,

$$f(d,t) = re^{-\frac{d^2}{2w^2}}, \tag{6.1}$$

where

$$r = r_0 \left(\frac{r_f}{r_0}\right)^a, \quad w = w_0 \left(\frac{w_f}{w_0}\right)^a, \quad \text{and} \quad a = \frac{t}{t_f}.$$

The parameter r_0 is the initial learning rate, r_f is the learning rate at time t_f, w_0 is the radius of the initial neighbourhood, and w_f is the radius of the neighbourhood at time t_f. For the winning node, $d = 0$, and Equation 6.1 reduces to Equation 6.2,

$$f(t) = r = r_0 \left(\frac{r_f}{r_0}\right)^a. \tag{6.2}$$

Equation 6.2 was used as the learning function for the SGM in all experiments. Thus, at all times the learning rate of the SGM matches the learning rate of the winning node in the SOM. This permits a fairer comparison of the SOM and the SGM.

The Mean of Absolute Differences (MAD) was used as a measure of difference between two images. The absolute difference between each pair of corresponding pixels in the two images is calculated and the mean taken, to obtain a number in the interval [0, 1], where 0 indicates the images are identical and 1 indicates that they are maximally dissimilar. As all the images in the MNIST database have the same size, viewing direction (normal to the plane of the image,

from above), and comparable intensity, the MAD is an appropriate difference metric.

The models for each SOM were initialised with images containing random low pixel values similar to the background of the MNIST images. Each SGM was initially empty, having no nodes or models.

Once a classifier has been trained, the nodes must be labelled with the numeral represented by the associated model before the classifier can be used for testing. To do this, the number of times each node was the winning node for each numeral during the training phase was counted. The node was then labelled with the numeral it most often matched.

6.2.1 Experiment 1: Early accuracy

Recall that agents cannot wait until they have a full, final set of models to begin learning appropriate responses. This experiment determines how long it takes to develop a useful, if small, set of models. This experiment used a SOM and SGM of similar size. After 25 training images, chosen at random, had been presented to a classifier, its accuracy was tested on the entire test set, presented in random order. This process was repeated with successively greater amounts of training (50 images, then 75, 100, 150, 200, 250, 300, 400, and finally 500 images).

Table 6.1 shows the configuration of the classifiers for this experiment. The values r_0 and r_f were chosen so that the learning rate would start at maximum and be near zero by the end of training. The values w_0 and w_f were chosen through experimentation. The value of t_f is the total number of training images (even though only a small number of images was actually presented).

Table 6.1: Configuration of SOM and SGM in Experiment 1.

variable	SOM	SGM
node count	100	96
grid type	rectangular	unconnected nodes
r_0	1	1
r_f	$1{\times}10^{-4}$	$1{\times}10^{-4}$
w_0	2	not applicable
w_f	$1{\times}10^{-4}$	not applicable
t_f	60000	60000
difference threshold	not applicable	0.165

Recall that if the difference between an input pattern and the winning node's model is greater than the difference threshold, and the SGM is not at capacity, a new model will be created. Once capacity is reached, the SGM will always update the most similar model, which increases the chance of a model eventually representing a different numeral than it was created for. For a fair comparison, it is important that the SGM and SOM be of similar size. However, an SGM may not create the maximum number of models. In order to maximise model stability, a SGM with a maximum capacity much larger than necessary (2000 models) was used, allowing the difference threshold to indirectly control the number of models created.

To find an appropriate value for this difference threshold, a random set of 500 images was chosen, and the MAD measured between all pairs of images. The number 500 was chosen because it would result in enough comparisons (250,000) to provide the accuracy needed, while being small enough to be calculated in a few hours. The results are shown in Table 6.2. Experimenting with the values between the two means showed that a threshold of 0.165 resulted in the SGM

creating 96 models, which was useful for comparison with the 100 models in the SOM.

Table 6.2: Analysis of MAD between MNIST images, based on a sample of 500 images. The first column contains the *mean of the mean* absolute difference; the second, the *standard deviation of the mean.*

	mean	std. dev.
same numeral	0.135	0.0436
different numerals	0.171	0.0374

6.2.2 Experiment 2: Full training run

To compare the overall accuracy of the SOM and SGM, the 60,000 images in the MNIST training set were presented, in random order, to a small and large SOM, and a small and large SGM. Next, the 10,000 images in the test set were presented, again in random order, to the SOM and the SGM for classification. This permitted a comparison of the accuracy, speed, model stability and number of *wasted models* for the two classifiers.

Table 6.3 shows the configuration of the classifiers for this experiment. Preliminary trials showed that the accuracy of both the SOM and the SGM depends strongly on the number of models, weakly on r_0, and very weakly on the other configuration parameters. Therefore, for this experiment the classifier size was allowed to vary while r_0 and r_f were kept constant. The values r_0 and r_f were chosen so that the learning rate would start high and be near zero by the end of training. The values w_0 and w_f were chosen through experimentation. The value of t_f is the number of training images.

Table 6.3: Configuration of SOM and SGM in Experiment 2.

variable	SOM	SGM
grid size	4×4, 6×6, 8×8, 10×10, 15×15, 20×20, 25×25, 30×30, 35×35, 40×40, 45×45, 70×70	initially empty, grows as needed
grid type	rectangular	unconnected nodes
r_0	0.1	0.1
r_f	1×10^{-4}	1×10^{-4}
w_0	2	not applicable
w_f	1×10^{-4}	not applicable
t_f	60000	60000
difference threshold	not applicable	0.09, 0.1, 0.105, 0.11, 0.115, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2, 0.21

6.3 Results and interpretation

6.3.1 Experiment 1: Early accuracy

Figure 6.3 compares the accuracy of the SOM and SGM during the early part of training. The SGM reaches a usable level of accuracy (i.e., sufficient for an agent to base decisions on) faster than the SOM.

6.3.2 Experiment 2: Full training run

Figures 6.4 and 6.5 show one pair of small (10x10) classifier models after all of the training images have been presented to the small classifiers. From Figure 6.4 we can see that many of the models are blurry combinations of more than one numeral. The topology of the input data has been partially preserved; models of the same numeral tend to be near each other.

There are four shaded models in Figure 6.4. They were not winning nodes

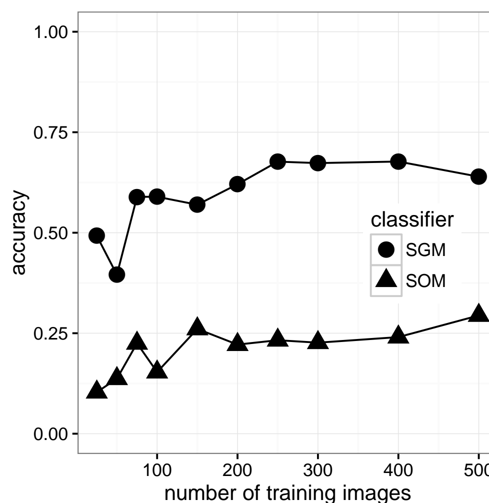

Figure 6.3: Early accuracy comparison

at any point during testing, were not used to classify testing images and are counted as wasted. An unmatched model could be assigned the same label as was assigned to a majority of its neighbours. However, this would result in the left pair of unmatched models (shaded) being assigned labels for the numeral '1', even though they are clearly better matches for '0'! The right pair of unmatched models are very ambiguous; it may be better not to use them.

Figure 6.5 shows the small SGM after training. We can see that the topology has not been preserved. Unfortunately, there are still many ambiguous models (models that would likely match two or more different numerals), perhaps due to the small size of the classifier.

Figure 6.6 compares model stability for the SOM and SGM. To measure this, the first numeral matched by each model was noted. (In the case of an SGM, this is the numeral the model was created in response to.) This initial match was compared to the numeral used to label the model's node (at the end of training). If the numerals were the same, the model was counted as stable. The SGM

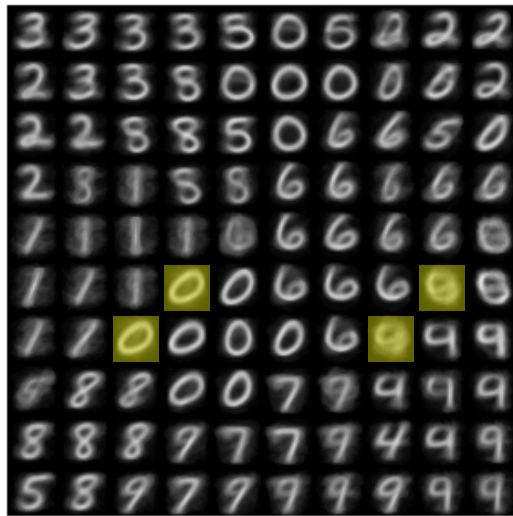

Figure 6.4: Small SOM after all 60,000 training images have been presented. Models are arranged in a grid. Wasted models are shaded.

Figure 6.5: Small SGM after all 60,000 training images have been presented. Models are unconnected; they are shown here in the order they were created. There were no wasted models.

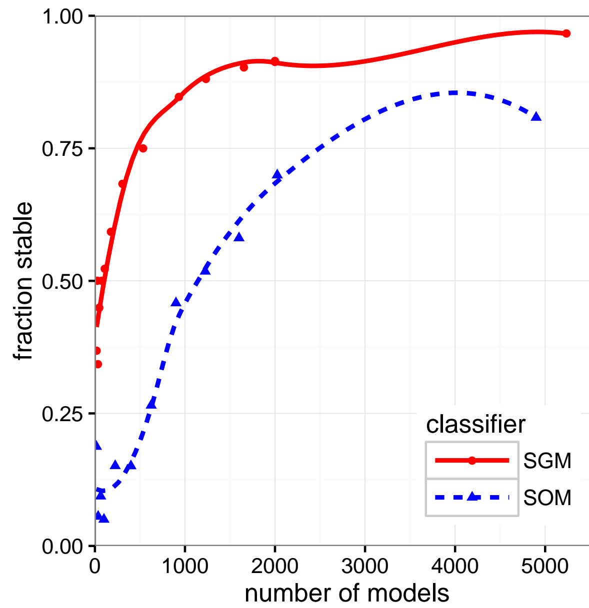

Figure 6.6: Model stability. Larger values are better. The lines show a loess (local polynomial regression) data fit.

consistently achieved higher model stability.

Figure 6.7 compares model usage. A model is counted as used if it was the winning node at any point during testing, otherwise it is considered wasted. The SGM used more of its models, reducing the problem of wasted models.

Figure 6.8 shows the time required for training and testing the SOM and SGM. For all but the smallest classifiers, the SGM is considerably faster than the SOM. The primary reason for the reduction in processing time is presumably that the SGM only updates the winning node's model during training, while the SOM also updates models in the neighbourhood of the winning node. In addition, the SGM has fewer models during the early part of training, and therefore does not need to make as many comparisons as the SOM does.

Figure 6.9 compares the accuracy of the classifiers. The accuracy is the number of times that an image was correctly identified, divided by the total number

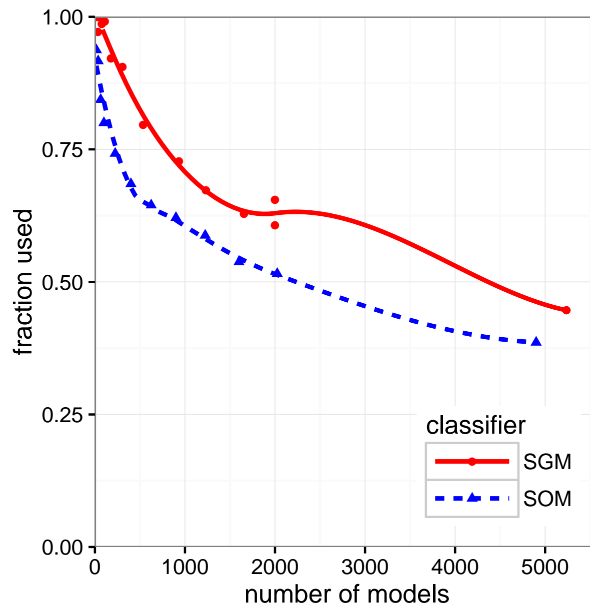

Figure 6.7: Model usage. Larger values are better. The lines show a loess (local polynomial regression) data fit.

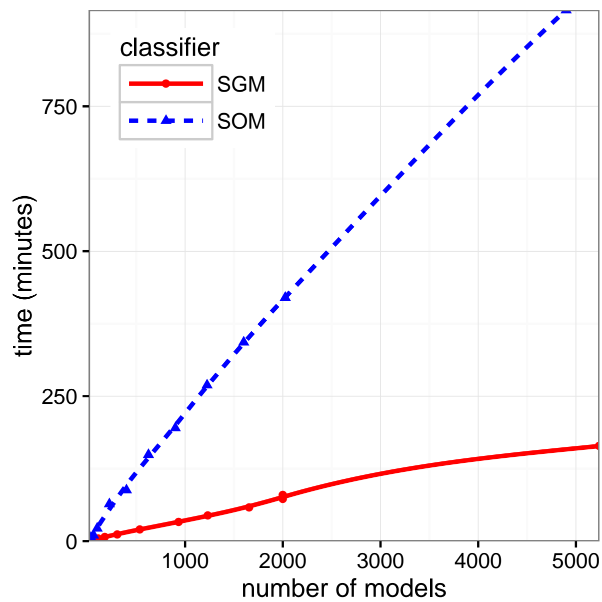

Figure 6.8: Processing time. Smaller values are better. The lines show a loess (local polynomial regression) data fit.

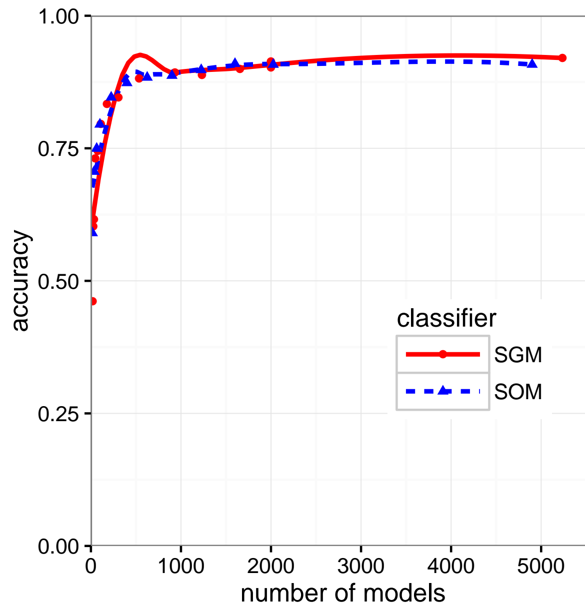

Figure 6.9: Accuracy. Larger values are better. The lines show a loess (local polynomial regression) data fit.

of images. The accuracy of the two methods appears to be comparable. For all but the smallest SOMs, a small fraction of the nodes were not winning nodes at any point during training. These could have been labelled to match the majority of their neighbours, or to match the most similar neighbour. However, there were not enough to significantly impact the accuracy, so they were counted as correct answers.

6.4 Summary

The SGM is a new version of the SOM that was adapted for use in intelligent data mining ALife agents. Although the SGM sacrifices topology-preservation, the two classifiers were equally accurate at identifying handwritten numerals. The SGM also achieves a higher accuracy more quickly, which could allow an

agent to make good survival decisions with less training. Model stability was higher in the SGM, and there were fewer wasted models, making it faster than the SOM. The SGM could be a useful component for implementing intelligent agents, and for other clustering or classification applications.

Chapter 7 will present a new brain design for wains, based on the SGM. The code and results for the experiments described in this chapter are open access[1].

[1] See http://dx.doi.org/10.5281/zenodo.45039 (som) and http://dx.doi.org/10.5281/zenodo.45040 (exp-som-comparison)

Chapter 7

Improving the brain

The most significant change to the `wains` was the redesign of their brains. The primary goal was to make them Popperian creatures with the ability to predict the outcome of their actions (rather than Skinnerian creatures learning only through blind trial-and-error), with the hope that the corresponding increase in cognitive power would allow `wains` to perform useful data mining tasks. This chapter describes the new brain design; the testing of the new design is discussed in Chapters 8 and 9.

7.1 Seeking inspiration

Is there an existing Popperian brain architecture that can be adapted for `wains`? As discussed in Section 2.4, ALife species are typically Darwinian or Skinnerian. An ALife species that is a neuron-for-neuron simulation of a biological organism (such as OpenWorm) might possibly be Popperian. However, such detailed simulations are too slow for this project. (For example, 10 seconds of simulated time

for OpenWorm requires a few days of processing on a dedicated GPU [148].) It is desirable that an experiment with a population of hundreds of data mining agents should real-time run on an ordinary computer; neuron-level simulations are not fast enough.

What about a higher-level model of cognition in a higher animal? Models of human-level (or near-human) general intelligence do exist (e.g., the Blue Brain Project [149], BECCA [150], Nexting [151], SNePS [152], Novamente [153]); These are likely Popperian or even Gregorian. However, they use sophisticated models of knowledge representation, belief representation, reasoning, and/or planning; again they are too slow for this project.

Unfortunately, a search of the literature did not find any purely software-based forms of ALife that were fast enough to run multi-agent, real-time experiments and would qualify as Popperian.

7.2 Making decisions

In the absence of a suitable model of a Popperian brain, it was necessary to create one. The first step was to define a simple algorithm for decision-making. The algorithm is outlined below.

1. Compare the objects the agent currently senses with its internal models, determining how similar they are. (If the agent does not already have a similar model, it should create one.)

2. Develop one or more hypotheses about the kind of situation the agent is currently facing. Depending on how well the situation matches the

agent's mental models, it may be advisable to consider more than one possibility. For example, when presented with a berry, the agent should consider the possibility that is poisonous as well as the possibility that it is nutritious.

3. Select the most likely hypotheses to consider. It would take too long for an agent to evaluate the effect of every possible action in every possible scenario when a decision is required; it is better to filter the list of hypotheses before making predictions. In the previous example, the berry might be nutritious or poisonous, but it is probably safe to disregard the possibility that it is an attacking enemy.

4. Predict the effect of possible actions on the agent's condition. For example, the agent might predict that eating the berry will make it 47% less hungry. This step, together with the next, make the agents Popperian.

5. Taking into account the agent's current condition, evaluate how the agent's happiness will be affected by the predicted changes to its condition. For example, eating the berry will only make the agent happier if it is currently hungry.

6. Choose the action that will result in the greatest happiness.

The new brain architecture was designed around this six-step process.

7.3 The SGM and genetics

As discussed in Chapter 6, the SGM has greater model stability than the SOM, and fewer wasted models, making it faster. For these reasons, The SGM was chosen to be the main building block in the new brain design.

The learning function was discussed in Section 6.1 (Equation 6.2). The parameters r_0, r_f and t_f are determined by the parent's genes, as is the difference threshold and the maximum number of models an SGM can contain, and the initial models (if any). Finally, an SGM needs to be able to measure the difference between an input pattern and each of its models, and to tweak a model to more closely match an input pattern. This is accomplished using a custom component for each data type and experiment. This component, called a *tweaker*, may also have genetically determined parameters.

The SGM will allow an agent to build a set of models representing its environment and to classify (Step 1 in the decision process) the patterns it encounters according to its internal models. However, that is only part of the brain's job. In order to make good decisions, an agent also needs the ability to predict the outcome of an action (Step 4), so that it can choose the action that is expected to produce the best outcome. (This ability to make predictions is essential if we want the agents to be Popperian creatures.) How could this prediction-making feature be implemented? With another SGM! The role of this second SGM is to model the outcomes of possible actions in a variety of situations. Section 7.4 will discuss the role of the two SGMs in more detail.

7.4 A new brain architecture

In the new implementation, the brain has three components: a *classifier*, a *muser*, and a *predictor*. This structure is fixed; however, evolution can fine-tune operating parameters such as the learning rate. The classifier maintains models of patterns encountered, the muser generates possible responses to situations, and the predictor maintains models of responses selected and their outcomes.

Both the classifier and predictor are implemented using an SGM that is initially empty. The process by which the brain makes decisions is illustrated in Figure 7.1. When one or more patterns are presented to the agent, the classifier produces a *signature*, a vector whose elements indicate how similar each input pattern is to each classifier model, and reports this to the brain. For example, suppose the agent encounters two objects and has three classifier models. The signature might be as shown in Table 7.1. This shows that the best matches for objects #1 and #2 are classifier models A and C, respectively.

Table 7.1: Example of a signature

	classifier model		
object	A	B	C
1	**0.1**	0.2	0.6
2	0.9	0.8	**0.3**

For each object and model, the brain estimates a *p-score*, which might be interpreted as an estimate of the probability that the object actually belongs to the category represented by the model. This p-score will be used to weight possible responses. It could be estimated as $1 - d$, where d is the difference between the object and the model. However, preliminary experiments showed

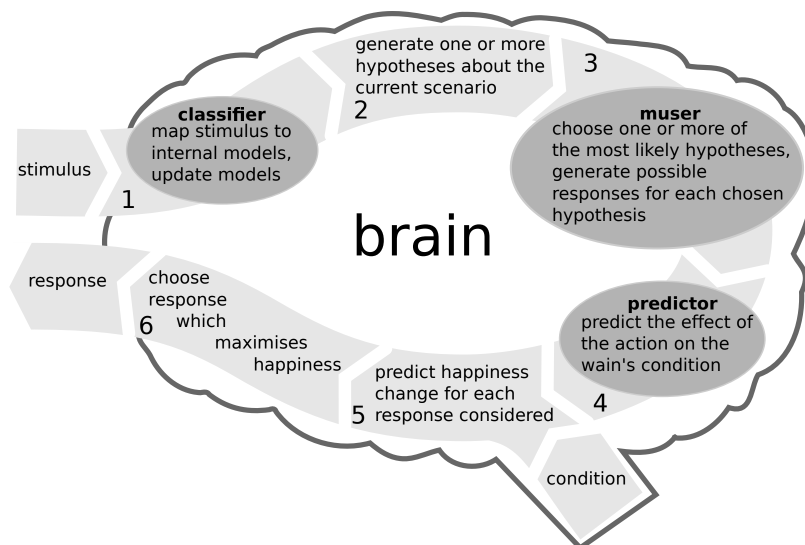

Figure 7.1: The decision-making process. The numbers refer to the steps in the algorithm described in Section 7.2.

that this results in models that are weak matches being given too much weight in decision-making. For example, consider Figure 7.2. The points A and B represent likely matches for the object; the possibility that the object belongs to one of these categories should be weighted strongly when making decisions. With this function, the estimated probabilities $p(A)$ and $p(B)$ should be, and are, high. However, the point C represents a much poorer match. Given that there are two very good matches, the brain should not give much consideration to the possibility that the object belongs to category C. Thus, $p(C)$ seems high.

As an alternative, consider the function

$$1 - e^{1-\frac{1}{1-(1-d)^s}} \tag{7.1}$$

where s is an adjustable parameter that controls the steepness of the curve. This function was constructed to have the shape shown in Figure 7.3. Note that the

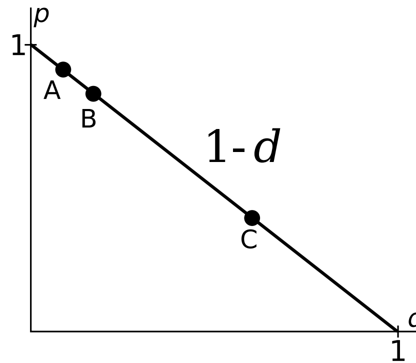

Figure 7.2: A simple p-score function.

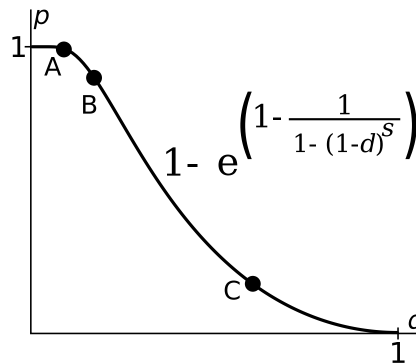

Figure 7.3: A more sophisticated p-score function.

estimated probabilities $p(A)$ and $p(B)$ are still high, but $p(C)$ has been reduced. Higher values of s make the brain more likely to disregard poor matches for the object when making decisions; thus s is called the *strictness*. This is the approach used by the brain in the new implementation. Table 7.2 shows the result of applying Equation 7.1 with $s = 2$ to the signature in Table 7.1. The probabilities for each object are then normalised so that the total of the probabilities for each model is one, as shown in Table 7.3.

The brain generates *hypotheses* by considering each possible combination of object and model. The p-score for each hypothesis is the product of the individual object-model probabilities. Table 7.4 shows an example of the hypotheses

Table 7.2: Unnormalised probabilities for signature in Table 7.1.

	classifier model		
object	A	B	C
1	**0.99**	0.83	0.17
2	0.01	0.04	**0.62**

Table 7.3: Normalised probabilities from Table 7.2. As a result of rounding, the sum of the probabilities for an object may not exactly equal one.

	classifier model		
object	A	B	C
1	**0.50**	0.42	0.09
2	0.02	0.06	**0.92**

that would be generated for the signature in Table 7.1. According to the table, the most likely hypothesis is that the agent has encountered objects matching classifier models A and C. However, the next most likely hypothesis is that the objects match classifier models B and C; it may be worthwhile to consider that possibility as well.

Next, the muser chooses one or more of the most likely hypotheses, and generates a set of responses to evaluate. The number of hypotheses chosen is genetically determined, and is called the agent's *depth*. (Mnemonic: An agent that considers many hypotheses before making a decision might be called a "deep thinker").

The predictor then estimates how each proposed response will affect each aspect of the agent's condition (energy, passion, boredom, and litter size). It does this by selecting the response model that best matches the proposed response, and returning the condition changes predicted by that model, adjusted according

Table 7.4: Hypotheses based on probabilities in Table 7.3. As a result of rounding, the sum of the probabilities for all hypotheses may not equal one.

hypothesis	p-score
object 1 is A, object 2 is A	1%
object 1 is A, object 2 is B	3%
object 1 is A, object 2 is C	**46%**
object 1 is B, object 2 is A	1%
object 1 is B, object 2 is B	3%
object 1 is B, object 2 is C	39%
object 1 is C, object 2 is A	0%
object 1 is C, object 2 is B	1%
object 1 is C, object 2 is C	8%

to the p-score for the hypothesis. If no response model is sufficiently similar, a new model may be created (according to Algorithm 2). The new implementation even provides support for measuring how similar two actions are. For example, if an agent has a model for walking in response to a given situation, but no model for running in that same situation, it can infer that the result of running will be more similar to that of walking, than that of standing still. This allows the agents to filter out unwise actions ("stupid moves") without needing to try them, an important advantage for Popperian creatures.

The brain combines the agent's current condition with the predicted changes, and calculates the resulting happiness change, according to Equation 5.1. The brain chooses the action that is predicted to have the most favourable (most positive or least negative) effect on happiness. After the agent has received any rewards or penalties as a result of that action, the predictor adjusts its models according to the actual change in happiness.

By considering more than one hypothesis, the agent can employ more subtle

reasoning. It can base its actions not only on what scenario it thinks it is facing, but also on how confident it is, and what is likely to happen if the agent is wrong. For example, suppose the agent considers two hypotheses, where the estimated payoff (happiness increase) is given by Table 7.5. If the agent is reasonably confident that the more likely hypothesis is actually true, the best response is action #1. Otherwise, it may be worth the gamble to go for action #2, in hope of the large payoff.

Table 7.5: Sample payoff matrix.

	payoff if more likely hypothesis is true	payoff if less likely hypothesis is true
action #1	medium	small
action #2	small	large

The brain can also learn as a result of imprinting, which is a shortcut where the agent is shown one or more patterns and an action, and concludes that taking the action in a similar situation would alter its energy, boredom level, passion level, and litter size by genetically determined amounts called *imprint outcomes*. If the brain has already learned the response (either through imprinting or through experience), subsequent imprintings will strengthen the response models at genetically determined rates called *reinforcement deltas*. Imprinting can be used to allow children to learn by observing their parents, or for adults to learn by observing other adults. Although this feature was originally intended to allow agents to learn from each other, it can also be used by the operator to train agents (i.e., one library function can be invoked for both purposes).

The new brain architecture supports analysis of a wain's knowledge, deci-

sion rules, and learning. Patterns of objects in its environment are represented as SGM models, which are in the same form as the objects themselves. (For example, if the objects are images, the models are images that can be viewed, if the objects are audio samples, the models are samples that can be listened to, etc.) Logging can be configured to include detailed information about every step of the decision-making process documented in Section 7.2, including similarity of input patterns to existing models, creation of new models, formation and ranking of hypotheses, and the predicted effect of actions on the wain's condition and happiness.

7.5 Summary

The new brain design has three components, a classifier to build a set of models representing patterns in the environment, a muser to generate possible responses, and a predictor to predict the outcome of actions. This allows the wain to base its actions on what scenario it thinks it is facing, how confident it is, the predicted outcome of the action assuming the wain's assessment is correct, and the predicted outcome based on alternative assessments. This ability to hypothesise scenarios and to predict the outcome of an action based on similar actions in similar situations, makes the new wains Popperian creatures. Wains can ask themselves "what should I think about next" in the sense that they can decide which hypotheses to consider and which potential actions to evaluate. The new brain allows wains to be taught behaviour patterns by their parents, other wains, or through a formal training session run by a human experimenter.

The testing of the new brain design is discussed in Chapters 8 and 9.

Chapter 8

Classification with wains

As discussed in Chapter 3, it was decided that wains would be tested at both classification and prediction tasks, to evaluate their potential as data mining tools. This chapter describes a series of experiments designed to test the ability of wains to classify data. (Chapter 9 will address their ability to perform prediction.) These experiments will answer the first two research questions, repeated below.

> **Research Question 1:** Will giving wains a mechanism to predict the outcomes of possible actions, and to choose the action with the best predicted outcome, make them better decision-makers?

> **Research Question 2:** Can Popperian wains learn to classify data with accuracy and speed comparable to traditional classification methods?

Recall from Section 7.4 that a wain maintains a set of internal models for the range of objects that it has encountered. These internal models need not (and

usually do not) map directly to human categories. Based on the resemblance between a stimulus and its internal models, the wain chooses, from a predefined set, the response that it predicts will lead to the greatest happiness. Then how can we get a wain to perform classification? By making the set of available responses be classifications! (For example, one action that a wain can choose is "assign the object to category A", another is "assign the object to category B", and so on.) Using the wain as a classifier also exercises its ability to make good decisions.

How can we know if the new brain design is an improvement over the original design? Recall from Section 2.10 that the original wains were asked to choose from a small set of actions rather than to classify the images of handwritten numerals. Therefore, it is difficult to say how accurate they would have been at a true classification task. However, wains ate "poisonous" numerals nearly 10% of the time, and attempted to mate with numerals approximately 4% of the time. Not eating an "edible" numeral is not necessarily a mistake, but in the absence of any other information we might estimate that half of the occasions on which they did this were mistakes. Examination of the decisions made by some randomly selected individual wains as recorded in the program logs indicate that this is a conservative estimate; i.e., the actual rate is probably higher. Finally, wains chose not to eat "edible" numerals 30% of the time; so it was estimated that if the original wains were classifying handwritten numerals, their error rate would have been approximately 30%, as shown below.

15%	$\frac{30\%}{2}$ of "edible" numerals not eaten when hungry
10%	"poisonous" numerals eaten
4%	numerals mated with
$\approx$ 30%	total estimated mistakes

If the new brain design was found to be significantly more accurate than this, it was considered an improvement over the original design. Furthermore, the accuracy of the new brain was compared against conventional classification techniques. Experiments were designed to test the accuracy of a single wain as a classifier in two different domains, image and audio data. Further classification experiments involving populations of wains were planned; however, as will be shown, individual wains (in the new implementation) were quite accurate. Thus, the population experiments were not needed. However, experiments involving populations of wains are discussed in Chapter 9.

8.1 Experimental set-up

Figure 8.1 shows the architecture for these experiments. The `creatur-image-wains` package customises the generic wain implementation to interact with grey-scale images. The `creatur-audio-wains` package customises them to work with audio samples. The packages `exp-image-id-wains` and `exp-audio-id-wains` drive the experiments.

Although these experiments use an individual wain rather than a population, the wain is trained using the same mechanism that allows wains to learn from one another.

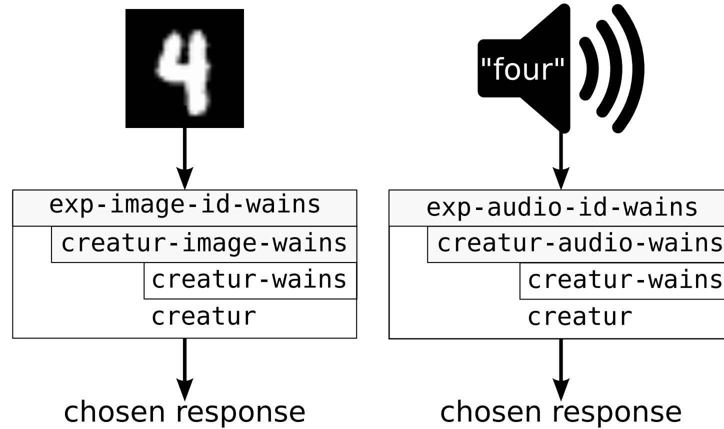

Figure 8.1: Architecture for classifying images and audio samples.

8.1.1 Classifying images

Images from the MNIST database (described in Section 2.13.1) were used without modification. The images were presented to the agent as a sequence of integers. Each element of the sequence was a number from 0 to 255, indicating the intensity of the pixel. The agent was not given any information about the geometry of the image. For example, it did not know that in a 28x28 image, the 29th pixel is immediately below the first pixel.

The brain was configured to use the MAD as a measure of difference between an input image and the classifier models. This is calculated by taking the absolute difference between each pair of corresponding pixels, and then taking the mean to obtain a number in the interval [0, 1], where 0 indicates the images are identical and 1 indicates that they are maximally dissimilar. All the images in the MNIST database have the same size, viewing direction (normal to the plane of the image, from above), and comparable intensity, so the MAD is an appropriate difference metric.

To evaluate the performance of the brain at handwriting recognition, it was

compared with a traditional classifier. Other classification techniques can achieve better accuracy at handwriting recognition than the SOM, for example, support vector machines [154] and traditional *neural networks* [155]. However, the wain's new brain design is partly based on modified SOMs (as discussed in chapter Section 7). For this reason, the SOM was used as the benchmark.

The learning functions used for the SOM and SGM were previously introduced in Section 6.2. As discussed in that section, the learning function of the SGM (see Equation 6.2) matches the learning function of the winning node (where d = 0) in the SOM (see Equation 6.1). This permits a fairer comparison of the SOM and the SGM.

Table 8.1 shows the configuration of the two classifiers. The values r_0 and r_f were chosen so that the learning rate would start high and be near zero by the end of training. The value w_0 was chosen so that the width would be approximately one-tenth of the width and length of the grid. (Since there are ten numerals, this would minimise the risk that a model would simultaneously be trained to more than one numeral.) The value w_f was chosen so that the neighbourhood width would be near zero by the end of training. The value of t_f is the number of training images. To determine the difference threshold, a range of values was tested near the mean difference between images of the same numeral; the one that resulted in the best accuracy was chosen.

8.1.2 Classifying audio samples

The MFCC feature vectors (discussed in Section 2.11) were extracted from the samples in the TI46 corpus (introduced in Section 2.13.2) by Flynn [156] using

Table 8.1: Configuration for working with MNIST images

variable	SOM	brain
final node count	1024	956
grid type	rectangular	unconnected nodes
classifier r_0	1	1
classifier r_f	1×10^{-15}	1×10^{-15}
classifier w_0	3	not applicable
classifier w_f	1×10^{-7}	not applicable
classifier t_f	60000	60000
classifier threshold	not applicable	0.12
predictor r_0	not applicable	1×10^{-9}
predictor r_f	not applicable	1×10^{-10}
predictor t_f	60000	60000
predictor threshold	not applicable	0.1

the HCopy tool provided as part of the Hidden Markov Model Toolkit (HTK) [120]. Each frame (feature vector) has 13 static coefficients, 13 velocity coefficients (first derivatives), and 13 acceleration coefficients (second derivatives) for a total of 39 coefficients. First order pre-emphasis was applied using a coefficient of 0.97. There were 23 *filterbank* channels (which show how much energy exists in each frequency region), and 22 cepstral *liftering* (inverse filtering) coefficients. (Liftering is filtering performed on an inverse Fourier transform of a signal.) The frame rate used was 10 ms with a 25ms *Hamming window* (a mathematical function). These configuration parameters were selected by Flynn [156] to maximise the accuracy with which the Hidden Markov Model (HMM) identified the end-pointed samples. The feature vectors for each audio sample were concatenated, in time order, and presented to the brain as a sequence of double-precision floats.

The brain was configured to use the square of the Euclidean distance as a

measure of difference between an input sample and the classifier models (The Euclidean distance is a common distance metric for numeric vectors.). The length of samples differs, so the resulting number of vectors in each sample differs as well. However, brains require that all input patterns have the same length. Therefore, the agent was configured to "stretch" or "compress" the samples as needed so they all have the same number of vectors. Stretching is achieved by duplicating vectors; as illustrated in Figure 8.2. The duplications were distributed as evenly throughout the pattern as possible.

Table 8.2: Examples of stretching, illustrated using a character string

original pattern	desired length	resulting pattern
abcdefghi	9	abcdefghi
	10	abcdeefghi
	11	abcddefgghi
	12	abccdeefgghi
	13	abbcddeffghhi
	14	abbccddeffghhi

The algorithm for compressing samples is straightforward. First, calculate the differences between each consecutive pair of vectors. Second, find the vector with the smallest change from the previous one, and drop it. These two steps are repeated until the sample is of the desired length.

Table 8.3 shows the configuration of the brain. The values r_0 and r_f were chosen so that the learning rate would start high and be near zero by the end of training. The values w_0 and w_f, and the number of vectors, were determined empirically. The value of t_f is the number of training images. To determine the difference threshold, a range of values was tested near the mean difference

between samples of the same numeral; the one that resulted in the best HMM accuracy was chosen.

Table 8.3: Configuration of brain for working with audio samples.

variable	as-is samples	end-pointed samples
classifier r_0	0.1	0.1
classifier r_f	0.001	0.001
classifier t_f	1594	1594
difference threshold	0.00018	0.00018
predictor r_0	0.1	0.1
predictor r_f	0.001	0.001
predictor t_f	1594	1594
num. vectors	159	154

As discussed in Section 2.11, the HMM is widely used for ASR, so this was a logical choice as a benchmark to compare the wain's performance against. The HMM-based classifier was implemented by Flynn [156] using the HTK Speech Recognition Toolkit [120]. The classifier uses ten whole word HMMs (one for each numeral), each of which has three states, with each state having three Gaussian mixtures.

End-pointing is the process of removing silence from the beginning and end of an audio sample, in order to simplify the classification task. The short-term energy for each frame is calculated as the sum of the absolute values of the sample amplitudes in the frame. End-pointing is performed by determining whether or not the short-term energy of successive frames is above a defined threshold (to determine the start of the utterance) or below a defined threshold (to determine the end of the utterance). For example, to get the start point, look for three consecutive frames with energy exceeding the threshold; the first frame of the

three is assumed to be the start of the utterance.

For working with non-endpointed samples, the classifier developed by Flynn [156] uses two additional models to represent pauses in speech, "sil" and "sp". The "sil" model has three states and each state has six mixtures. The "sp" model has a single state.

8.1.3 Training and testing

The general procedure for working with either images or audio samples is the same. In both cases, the training data set and the test data set are distinct; the standard training and test sets were used for both the MNIST and TI46 data. First, the training patterns were presented in random order to the agent, along with the correct identification. This was done using imprinting, as described at the end of Section 7.4.

Next, the test patterns were presented to the agent, again in random order. As each pattern was presented, the agent responded with an identification. For a fair comparison with the SOM or HMM, it was necessary to prevent learning during the testing phase. To achieve this, after each response from the wain, it was restored to the state it had at the end of the training (imprinting) phase. The wain's condition and happiness never actually change, and it is never given an opportunity to reflect on the outcome of its decisions. Thus, it continues to expect an increase in happiness, and to take that into account when making decisions.

8.2 Results and interpretation

Table 8.4 compares the image classification performance of the brain with that of the SOM. The accuracy of both methods is comparable. Furthermore, the overall error rate of the new brain design is 15%, significantly better than the estimated 30% for the original design. Training and testing the brain required less than half the time of the SOM. Both the SOM and the brain were implemented in the same language (Haskell). The reduction in processing time occurs primarily because the SGM only updates one model during training, while the SOM updates the models in the neighbourhood of the winning node.

Table 8.4: Comparison of image classification results.

classifier	SOM	brain
no. models	1024	941
numeral	accuracy	
0	0.952	0.9408
1	0.970	0.9736
2	0.837	0.9109
3	0.835	0.8634
4	0.725	0.6609
5	0.739	0.8341
6	0.967	0.9415
7	0.873	0.7772
8	0.753	0.7956
9	0.834	0.7929
all	0.853	0.8508
time	6273s	2514s

Table 8.5 compares the audio classification performance of the brain with that of the HMM. The accuracy of both methods is comparable, however, the brain is significantly slower. The difference in speed may be an implementation

artefact. For example, the HTK is implemented in C and, since it has been in use since 1989, has likely been tuned for greater performance, while the wain was written in Haskell, has not been tuned, and is not parallel. The HTK is specialised for a single purpose (running the HMM), while the wain is designed to handle a variety of data mining and decision-making tasks; this may account for much of the difference in speed. The brain was slightly more accurate when working with the as-is data than with the end-pointed data. The compression algorithm has the side-effect of removing some of the silence from the beginning and end of the sample, thus an extra end-pointing step is not required.

Table 8.5: Comparison of audio classification results.

data type	as-is		end-pointed	
classifier	HMM	brain	HMM	brain
word	accuracy			
"zero"	1.0000	1.0000	1.0000	0.9840
"one"	1.0000	0.9882	1.0000	0.9922
"two"	1.0000	1.0000	1.0000	1.0000
"three"	1.0000	0.9881	1.0000	0.9961
"four"	1.0000	1.0000	1.0000	0.9961
"five"	1.0000	1.0000	1.0000	0.9961
"six"	0.9961	0.9961	1.0000	1.0000
"seven"	1.0000	0.9922	1.0000	0.9961
"eight"	1.0000	1.0000	1.0000	0.9883
"nine"	0.9881	0.9763	1.0000	0.9802
all	**0.9984**	**0.9941**	**1.0000**	**0.9929**
time	$<$1m	14m	$<$1m	12m

8.3 Summary

A wain with the new brain design was applied to two classification tasks: handwritten numeral recognition and spoken numeral recognition. In the experiment with handwritten numerals, the overall error rate of the new brain design was significantly better than for earlier experiments with the original design. Thus, the answer to research question 1 is that giving wains a mechanism to predict the outcomes of possible actions, and to choose the action with the best predicted outcome, does indeed made them better decision-makers.

When working with both handwritten and spoken numerals, the accuracy of the wains was comparable to more traditional classifiers. These results suggest that wains could be useful as a general-purpose classifier, applied to a variety of domains. In the experiment with handwritten numerals, the wain was faster than the SOM. However, the wain was much slower than the HMM at recognising spoken numerals. Thus, the answer to research question 2 is that Popperian wains can learn to classify data with accuracy comparable to traditional classification methods. However, whether or not the speed of the wains will be comparable with traditional methods depends on the type of data involved and the methods available for that domain.

Why should anyone be interested in a new classifier that is no more accurate than traditional classifiers, and for audio, is significantly slower? One advantage is that the new brain design is not just a classifier; it also *makes decisions* by choosing the action that leads to the best predicted outcome. In the experiments described in this chapter, the only available actions were to choose a classification; however, other types of actions could also be performed. An-

other advantage to the new design is its generality; it could be used in domains where custom classifiers have not yet been developed.

As this is a new approach to pattern recognition and decision-making, there is scope for improvement. Accuracy might be improved by choosing more sophisticated distance metrics. For images, the MAD could be replaced with a metric that takes into account a pixel's neighbours. This might allow it to cope better with writing that is heavily slanted, or is thinner or thicker than typical writing. For audio samples, a variable frame rate analysis such as that suggested by Cerf and Compernolle [157] could be used. The run-time of the software is dominated by the comparisons between models, so performance could also be improved by choosing a different distance metric.

Although a single wain was used in these experiments, wains were designed to be used in a population. The configuration parameters are genetic, so it is possible to have a population of wains with varying configurations. Awarding energy for more accurate classifications would encourage evolution to find a range of suitable configurations. wains have the ability to teach their young, as well as other adults, so each generation can augment the species' knowledge. A population of wains with slightly different configurations, and different life experiences, could give independent opinions on a classification.

The code and results for the experiments presented in this chapter are open access[1]

[1] See http://dx.doi.org/10.5281/zenodo.46981 (exp-image-id-wains), http://dx.doi.org/10.5281/zenodo.46980 (exp-audio-id-wains), http://dx.doi.org/10.5281/zenodo.46989 (creatur-image-wains), http://dx.doi.org/10.5281/zenodo.46992 (creatur-audio-wains), http://dx.doi.org/10.5281/zenodo.46987 (creatur-wains), http://dx.doi.org/10.5281/zenodo.46994 (creatur), and http://dx.doi.org/10.5281/zenodo.45039 (som).

Chapter 9

Forecasting with wains

In Chapter 8, it was shown that wains can be used for one data mining task (classification). The next step was to apply their powers of prediction to a second data mining task: forecasting. Experiments were designed involving univariate and multivariate time series data in two different domains, ISP traffic and weather. These experiments will answer the third research question.

> **Research Question 3**: Can Popperian wains learn to forecast future values in a data stream, with accuracy and speed comparable to traditional forecasting methods?

9.1 Experimental set-up

Figure 9.1 shows the architecture used for these experiments. The `creatur-uivector-wains` package customises the generic wain implementation to interact with numeric vectors, where each vector element is in the interval $[0, 1]$. The package `exp-uivector-prediction-wains` drives the experiments.

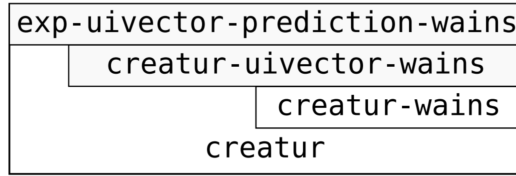

Figure 9.1: Architecture for predicting stream data.

9.1.1 Predicting ISP traffic

The values in the ISP traffic data (discussed in Section 2.13.3) were scaled to lie in the interval $[0, 1]$. There were no missing values to handle. This experiment used unsupervised learning (i.e., there was no training period.) At each time step, a vector containing the current value and the delta from the previous value (set to zero for the first time step) was presented to each wain in the population. Wains were then asked to predict the next value. No information about the time of day was provided.

A metabolism tax was deducted from each wain according to Equation 9.1,

$$e_{metabolism} = c_{metabolism} + f_{metabolism} \, n_c \tag{9.1}$$

where $c_{metabolism}$ and $f_{metabolism}$ are configurable constants for the experiment, and n_c is the wain's current number of classifier models. Children do not pay a metabolism tax until they are mature. Instead, the carer pays a fraction of the child's metabolism tax according to Equation 9.2

$$e^{child}_{metabolism} = f_{child} \, e_{metabolism} \tag{9.2}$$

where f_{child} is a configurable constant for the experiment, and $e_{metabolism}$ is the

metabolism tax that the child would pay if it were an adult, as given by Equation 9.1.

Beginning with the second round, wains were given an energy reward for their predictions according to Equation 9.3,

$$e_p = [f_{prediction}(1 - |x_{actual} - x_{predicted}|)]^{k_{prediction}} \quad (9.3)$$

where $f_{prediction}$ and $k_{prediction}$ are configurable constants for the experiment, $x_{predicted}$ is the prediction (for the current round) that the wain made in the previous round, and x_{actual} is the actual value for the current round.

The experiment used the directed mating approach described in Section 5.6; the frequency of flirting opportunities is f_{flirt}.

Table 9.1 shows the configuration parameters for this experiment. Preliminary trials (for this and other data mining experiments) showed that an average metabolic cost of approximately 0.1 per wain per turn is typically high enough to drive rapid learning, but not so high that the first generation dies before it has an opportunity to learn the task. This should be balanced with a maximum reward of 0.1 per wain per turn, so that successful wains can live long lives, but must remain competitive. The values of $f_{prediction}$, $c_{metabolism}$, $f_{metabolism}$ and f_{child} were chosen to satisfy this rule of thumb. (Appendix B discusses the need for a heuristic approach to configuration, and outlines one approach.) The value of f_{flirt} was chosen to approximately balance the birth and death rates. The initial population size has been found empirically to provide sufficient diversity in the gene pool. The maximum lifetime was chosen to ensure that a wain could live long enough to rear several children.

Table 9.1: Configuration for predicting ISP traffic.

variable	value
$f_{prediction}$	0.1
$k_{prediction}$	16
$c_{metabolism}$	-0.1
$f_{metabolism}$	-0.0001
f_{child}	0.1
f_{flirt}	0.1
initialPopSize	100
maximum lifetime	2000

The genetic make-up of the initial population is shown in Table 9.2. These settings were chosen by following the heuristic approach presented in Appendix B.

A set of traditional forecasting techniques was chosen for comparison. The statistical programming language *R* was used to run the selected techniques. The first, labelled "naive", simply used the current value as the estimate for the next value. Next, Simple Moving Average (SMA) was used. SMA calculates the arithmetic mean over the most recent n observations, where n is called the *window width*. Weighted Moving Average (WMA), which calculates a weighted average of the most recent n observations using a linear weighting, was also used. Both SMA and WMA were calculated using an assortment of window widths.

The next technique used was Auto-Regressive Integrated Moving Average (ARIMA), which predicts future values as a weighted sum of one or more recent values, plus a weighted sum of one or more recent values of the errors. (For more information about ARIMA, see Coghlan [158]). The R function `auto.arima` was used to determine the parameters; it recommended a non-seasonal model,

Table 9.2: Gene pool of initial population for predicting ISP traffic.

variable	value
devotion	random value in [0.0, 0.3]
maturityRange	random value in [50,300]
Δp	random value in [0.001, 0.1]
Δb	not used
depth	random value in [1, 3]
strictness	random value in [2, 50]
imprint outcomes	four random values in [0.1, 1]
reinforcement deltas	four random values in [0.001, 0.1]
classifier max size	random value in [10, 50]
classifier r_0	random value in [0.1, 1]
classifier r_f	random value in [0.0001, 0.001]
classifier t_f	random value in [1000,5000]
classifier threshold	random value in [0.01, 0.05]
predictor max size	random value in [10, 100]
predictor r_0	random value in [0.1, 0.3]
predictor r_f	random value in [0.00001, 0.001]
predictor t_f	random value in [1000,5000]
predictor threshold	random value in [0.05, 0.2]

`ARIMA(0,0,5)`. In this notation, p is the order (number of time lags) of the autoregressive model, d is the degree of differencing, and q is the order of the moving-average model. For each prediction, a new ARIMA model was created using the previous data values.

Finally, *Holt-Winters* with exponential smoothing was used. This technique models the level of a variable, its trend, and seasonal effects. (For more information about Holt-Winters, see Coghlan [158]). “Seasons” of one hour and one day were tried, as well as a non-seasonal model.

9.1.2 Predicting weather

The NYC weather data (discussed in Section 2.13.4 included four non-numeric fields, date, events, city and season, which required special consideration. The date field was not likely to be relevant to predicting the weather, except insofar as it would indicate the season, so it was discarded. (The records are in date order, which provided a means to detect trends over time.) The city field was identical for all records, and therefore not useful, so it was also discarded.

The events field might have been useful to the wains, and a numeric coding scheme could have been used. However, this would require special handling when measuring the difference between two models (what should the numeric difference between “Fog” and “Tornado” be?), and when adjusting a model to make it more similar to an input pattern (if the input pattern has “Snow”, the model has “Rain”, and the learning rate is currently 0.15, what should the new value of the field be?). For convenience, the event field was discarded.

The season field suffers from the same difficulty as the events field, so it was

replaced with a field for ordinal "day of year" [0, 366], which might allow wains to observe seasonal patterns. Additional fields were discarded because of missing values: maximum, mean, and minimum visibility, gust speed, precipitation and cloud cover. This left 16 numeric fields, which were then scaled to lie in the interval [0, 1]. Finally, two of the 24,560 records were also discarded because they contained null values for one or more fields, leaving 24,558.

The remaining fields were expected to contain more than enough patterns and correlations for the test. The focus of this experiment was to see how well wains could learn a complex data set in situations where little prior domain knowledge (such as the existence of weather seasons) exists. The NYC weather data is merely being used as an example of complex data set to allow the comparison of different classification systems; the intention is not to replace existing weather forecasting systems. All classification systems tested were presented with the same data; dropping some fields might have some impact on *absolute* accuracy, but should not affect the accuracy of one classification system *relative to* another.

A metabolism tax was deducted from each wains according to Equation 9.1 and Equation 9.2. In each round after the first, wains were given an energy reward for their predictions according to Equation 9.3. The experiment used the directed mating approach described in Section 5.6.

Table 9.3 shows the configuration parameters for this experiment. The values of $f_{prediction}$, $c_{metabolism}$, $f_{metabolism}$ and f_{child} were chosen to satisfy the 0.1 metabolism cost and reward rule of thumb introduced in Section 9.1.1. The value of f_{flirt} was chosen to approximately balance the birth and death rates. The initial population size has been found empirically to provide sufficient diversity in

the gene pool. The maximum lifetime was chosen to ensure that a `wain` could live long enough to rear several children.

Table 9.3: Configuration for predicting next day's high temperature.

variable	value
$f_{prediction}$	0.1
$k_{prediction}$	16
$c_{metabolism}$	-0.09
$f_{metabolism}$	-0.0001
f_{child}	0.1
f_{flirt}	0.1
initialPopSize	100
maximum lifetime	5000

The genetic make-up of the initial population is shown in Table 9.4. These settings were chosen by following the heuristic approach presented in Appendix B.

A set of traditional forecasting techniques, using logical but not necessarily optimal configuration parameters, was chosen for comparison. Again, the statistical programming language R was used to run the selected techniques. The "naive" technique simply used the current value as the estimate for the next value. The next techniques were SMA and WMA, each with an assortment of window widths.

As the weather data is multivariate, ARIMA and Holt-Winters could not be used. Instead, Vector Auto-Regression (VAR) was used. This technique models linear interdependencies among multiple time series. (For more information about VAR, see Pfaff and Stigler [159]). For each prediction, a new VAR model was created using the previous data values.

Table 9.4: Gene pool of initial population for predicting next day's high temperature.

variable	value
devotion	random value in [0.0, 0.3]
maturityRange	random value in [100,300]
Δp	random value in [0.001, 0.1]
Δb	not used
depth	random value in [1, 5]
strictness	random value in [2, 200]
imprint outcomes	four random values in [0.1, 1]
reinforcement deltas	four random values in [0.001, 0.1]
classifier max size	random value in [10, 50]
classifier r_0	random value in [0.1, 1]
classifier r_f	random value in [0.0001, 0.001]
classifier t_f	random value in [1000,5000]
classifier threshold	random value in [0.01, 0.05]
predictor max size	random value in [100, 1000]
predictor r_0	random value in [0.1, 0.3]
predictor r_f	random value in [0.0001, 0.001]
predictor t_f	random value in [1000, 5000]
predictor threshold	random value in [0.1, 0.16]

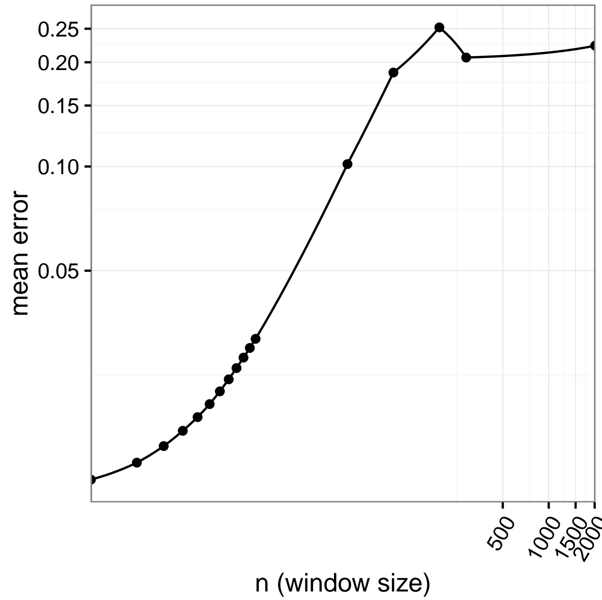

Figure 9.2: SMA forecasting of ISP traffic with different window widths. A log-log scale is used to make it easier to identify the point where the minimum error occurs.

9.2 Results and interpretation

As shown in Figure 9.2 and Figure 9.3, the best results were obtained with window widths set to 1 (a single five-minute interval), which is identical to the "naive" approach.

Of the traditional forecasting techniques tried, the non-seasonal Holt-Winters model gave the best results, as shown in Figure 9.4[1]. The wains were able to match Holt-Winters after they had seen approximately 2000 values. Since SMA(n=1) and WMA(n=1) are identical to the "naive" approach, SMA(n=12)

[1]Cortez et al. [136] experimented with forecasting the same ISP traffic data set. It is difficult to compare the results described above with theirs, because they used 2/3 of the A5M data to train the models and the remaining 1/3 to evaluate their accuracy. They tested an extensive range of different ARIMA models, and found that the X-12-ARIMA package gave better results than Holt-Winters. An Neural Network Ensemble (NNE) with 5 different networks was also used, where the average of the individual network predictions became the NNE prediction. The NNE gave slightly better accuracy than ARIMA. [136]

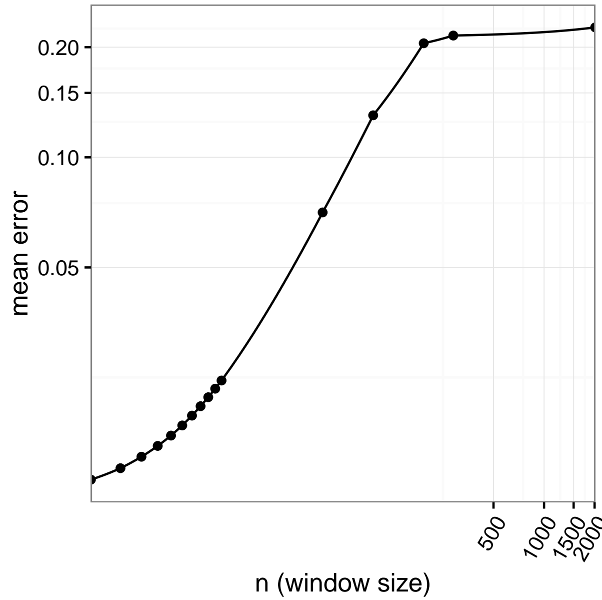

Figure 9.3: WMA forecasting ISP traffic with different window widths. A log-log scale is used to make it easier to identify the point where the minimum error occurs.

WMA(n=12), a window width of one hour, are included in Figure 9.4.

The 2000-value delay is believed to be due to the time needed for early generations to learn the basic patternicity of the data and pass it on to their offspring. The first generation gave birth in the first few rounds, so they had not acquired much knowledge to pass onto the second generation; effectively, the first and second generations were learning the data patterns from scratch. The third generation was the first to be able to build upon the knowledge of their ancestors, and they did not dominate the population until about time t =1000 (i.e., when 1000 values had been presented). These wains could then complete the task of learning the more basic patterns in the data. By t =2000, the population would be dominated by fourth and fifth-generation wains, who knew the basic patterns by the time they were reared.

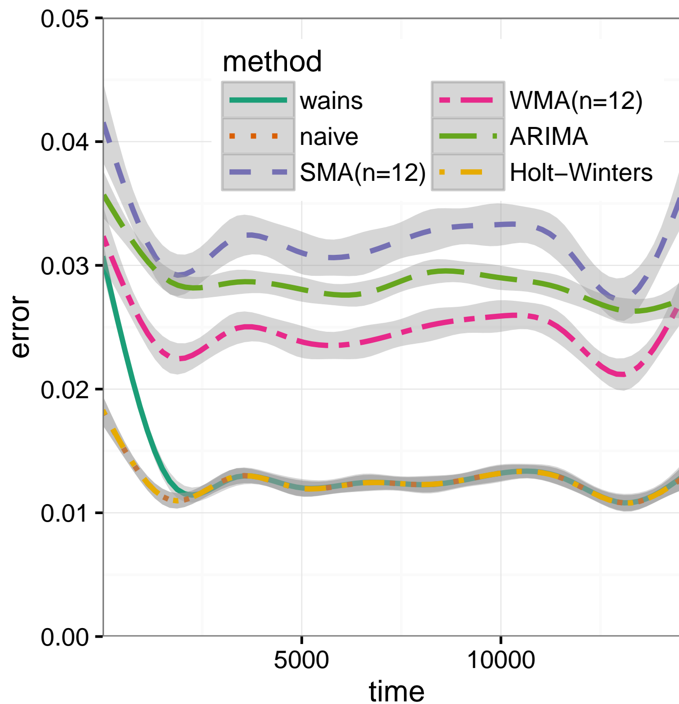

Figure 9.4: Error in ISP traffic prediction.

The naive, SMA, WMA, and ARIMA implementations required less than 10 minutes to run, Holt-Winters took approximately four hours, and the wains took approximately 20 hours. This means that wains were able to generate each prediction in under 5 seconds, on average. Considering that the ISP traffic was measured at 5-minute intervals, wains would likely be fast enough for real-time prediction of ISP traffic. The slower speed of wains may be due to differences in the implementation rather than differences in the algorithm. The R libraries used were coded in C and have likely been tuned for greater performance, while the wain was written in Haskell, has not been tuned, and is not parallel.

For the weather data, as shown in Figure 9.5 and Figure 9.6, SMA and WMA gave the best results with window width set to one, which is identical to the "naive" approach.

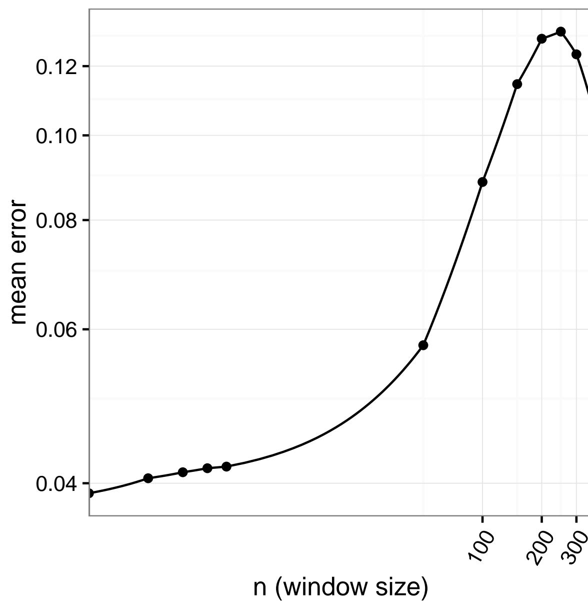

Figure 9.5: SMA forecasting of next day's high temperature with different window widths. A log-log scale is used to make it easier to identify the point where the minimum error occurs.

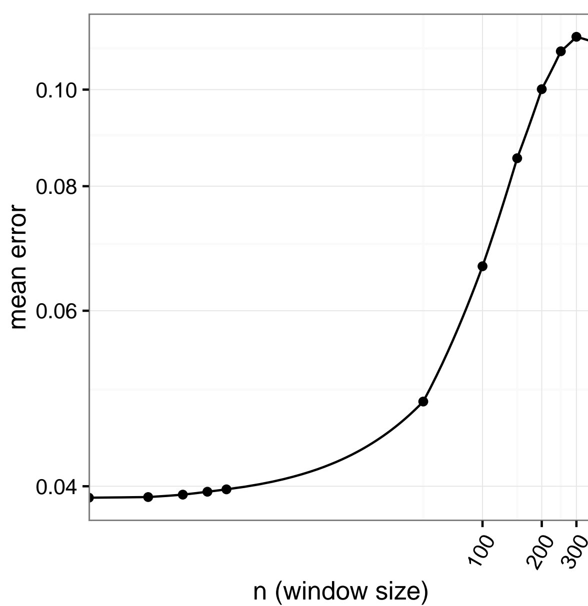

Figure 9.6: WMA forecasting of next day's high temperature with different window widths. A log-log scale is used to make it easier to identify the point where the minimum error occurs.

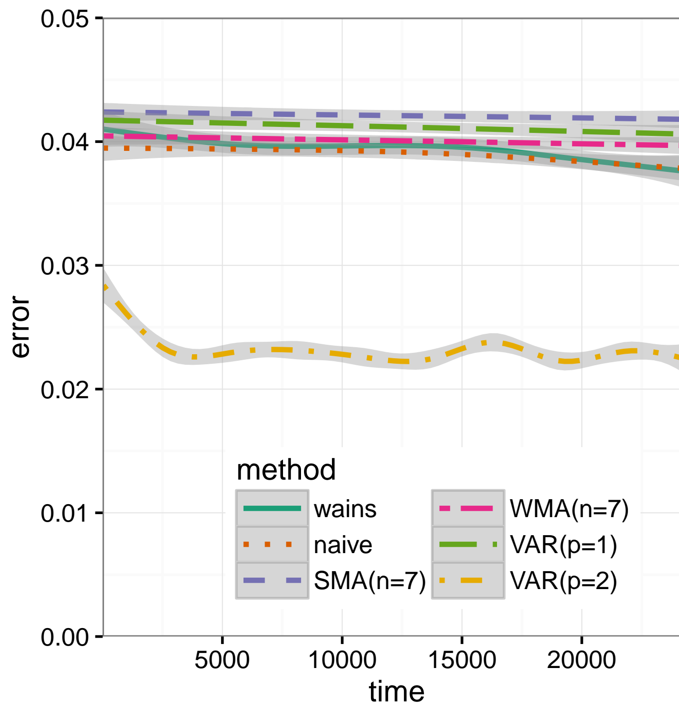

Figure 9.7: Error in next day's high temperature prediction.

Of the traditional forecasting techniques tried, VAR(p=2) gave the best results by far, as shown in Figure 9.7. Since SMA(n=1) and WMA(n=1) are identical to the "naive" approach, SMA(n=7) and WMA(n=7) (a window width of one week) are included in Figure 9.7. The wains were able to match or exceed the accuracy of most of the other techniques after viewing approximately 5000 records. (Since the weather data is multivariate, it may have taken more generations for the population to learn the basic patterns in the data.) However, the wains were not able to match the accuracy of VAR(p=2). The error rate for wains shows a downward trend; it is possible that the accuracy gap between them and VAR(p=2) would have narrowed if there had been more data. It is also possible that a larger population would be more accurate.

The naive, SMA, and WMA implementations required less than 15 minutes

to run, both VAR runs took approximately six hours, and the wains took approximately two days. This means that wains were able to generate each prediction in just over 7 seconds, on average. The weather was measured at daily intervals, so wains would likely be fast enough for real-time prediction in this scenario. Again, the slower speed of wains may be due to differences in the implementation rather than differences in the algorithm. The R libraries used were coded in C and have likely been tuned for greater performance, while the wain was written in Haskell, has not been tuned, and is not parallel.

9.3 Summary

Populations of wains were exposed to a stream of time series data, and tasked to predict future values. The results were compared with traditional mathematical forecasting algorithms (using logical but not necessarily optional configuration parameters). When working with univariate ISP traffic data, the wains quickly exceeded or matched the accuracy of the traditional algorithms. With the multivariate weather data, wains took longer to achieve accuracy comparable to most of the traditional algorithms (possibly due to the greater complexity of multivariate data), and they were unable to match the accuracy of VAR(p=2). Wains showed a continuing trend of increasing accuracy over time; it is possible that the accuracy gap between them and VAR(p=2) would have narrowed if there had been more data. It is also possible that a larger population would be more accurate.

The wains were much slower than traditional forecasting methods, but they were fast enough to keep up with a daily real-time data stream. The traditional

algorithms are each fine-tuned for a single purpose (prediction), while the wain is designed to handle a variety of data mining and decision-making tasks; this may account for much of the difference in speed.

Although wains were only asked to predict the next value in the time series, they could be asked to predict values further in the future. More research is needed to determine how the accuracy would drop off as predictions were made further into the future. As with the experiments in Chapter 8, these forecasting experiments demonstrate the decision-making ability of wains. In this case, the only available actions were to select a prediction from a list of possible predictions, other types of actions could be selected.

In conclusion, the answer to research question 3 is that Popperian wains can learn to forecast future values in a data stream. Their accuracy is comparable to that of traditional forecasting methods in some cases; however, they are likely to be somewhat slower. If faster and equally accurate techniques are available, could there be any advantage to using wains for forecasting? Consider that it is sometimes difficult to know, a priori, which of the traditional techniques, and with what configuration, will give the most accurate forecasts. It is also possible for the patternicity of data to change so much that the technique/configuration that works best at one point in time will not be the best choice later. Because wains evolve, we might expect them to be able to adapt to changes in the patternicity of the data. More research is needed to discover if this is indeed the case. Also consider that wains have the innate ability to make decisions; most forecasting algorithms do not. Section 10.3.1 will present some ideas on how this ability might be used.

The code and results for the experiments described in this chapter are open

access[2]

[2] See https://doi.org/10.5281/zenodo.212847 (exp-uivector-prediction-wains), https://doi.org/10.5281/zenodo.212912 (creatur-uivector-wains), https://doi.org/10.5281/zenodo.212889 (creatur-wains) and https://doi.org/10.5281/zenodo.212879 (creatur).

Chapter 10

Conclusions

10.1 Synopsis

Wains are an ALife species with artificial intelligence that live in, and subsist on, data. For them, finding patterns in data is a survival problem. De Buitléir, Russell, and Daly [2] showed that wains can discover patterns, make survival decisions based upon those patterns, and adapt to changes in the patternicity of their data environment. Could wains could be improved to make them useful data mining tools? Answering this question was the objective of this research project.

In order for wains to be useful data miners, they needed better decision-making skills. The approach chosen was to give wains the ability to predict the outcome of their actions, making them Popperian creatures in Dennett's Tower of Generate-and-Test.

In preparation for the experiments, several improvements were made to the Créatúr framework, with the goal of making it easier to use, both by the au-

thor and future researchers. The code was refactored into re-usable packages to make it easier to create new agent types and experiments. Agent caching was added to reduce I/O time. A mechanism was provided for the user to define conditions under which an experiment should halt, in order to allow problems to be identified more quickly, and to avoid wasting processing resources on experiments which are not progressing satisfactorily. An automatic population "balancing" feature was added, which eliminates the need to repeatedly adjust the experiment configuration to make the environment harsher as agents learn the assigned task.

Support for artificial genetics was also added to the Créatúr framework. Arbitrary user-defined data types can now be automatically encoded, decoded, recombined, and expressed. This reduces the amount of code that must be written by the user, and makes it easy to add new genetic traits to an agent. The framework now supports both sexual and asexual reproduction, with a flexible Domain-Specific Embedded Language (DSEL) to control recombination and mutation.

A number of improvements were made to the wain implementation to make them more suitable for a variety of tasks. The code was re-factored into a set of reusable packages which communicate through APIs. This makes it easier to add support for new data types, define new tasks, extend the capabilities of wains, and create new experiments with custom reward systems. The new implementation takes advantage of the improved Créatúr framework for reproduction. The new wains can perform multiple kinds of tasks (including classification, forecasting, and decision-making), and work with a variety of data types and formats (including grey-scale images, audio samples, and numeric vectors).

The most important improvement to wains was the redesign of the brain. The new brain is based upon the SGM, a modified SOM that was adapted for use in intelligent data mining ALife agents. By sacrificing topology-preservation, the SGM requires fewer calculations, making it faster than a SOM of comparable size. When tested at identifying handwritten numerals, the SGM and SOM were found to be equally accurate. The SGM achieves a higher accuracy more quickly, which could allow an agent to make good survival decisions with less training. Model stability (the ability of a model to continue to match patterns it was created in response to, while adjusting to match new patterns) was higher in the SGM. There were fewer wasted models (models that will not be used to classify future patterns), reducing unnecessary computation. The SGM could be a useful component for implementing intelligent agents, and for other clustering or classification applications.

The new brain design has three components: a classifier to build a set of models representing patterns in the environment, a muser to generate possible responses, and a predictor to predict the outcome of actions. This allows the wain to base its actions on what scenario it thinks it is facing, how confident it is, the predicted outcome of the action assuming the wain's assessment is correct, and the predicted outcome based on alternative assessments. The new design also allows wains to be taught behaviour patterns by their parents, other wains, or through a formal training session run by a human experimenter.

By allowing a wain to generate hypotheses about the scenario it is facing, consider the actions available to it, and predict the outcome of each action for each hypothesis, the new brain design makes them Popperian creatures. This promotion to a new level in the tower represents an increase in cognitive power.

A series of experiments tested the new wains on their ability to classify data and predict future values, using complex data from a variety of domains, and comparing their accuracy against traditional data mining techniques. First, a single wain was applied to two classification tasks: handwritten numeral recognition and spoken numeral recognition. In both cases, the wain's accuracy was comparable to more traditional classifiers. These results suggest that wains could be useful as a general-purpose classifier, applied to a variety of domains.

Next, populations of wains were exposed to a stream of time series data, and tasked to predict future values. When working with univariate ISP traffic data, the wains quickly exceeded or matched the accuracy of the traditional algorithms. With the multivariate weather data, wains took longer to achieve accuracy comparable to most of the traditional algorithms (possibly due to the greater complexity of multivariate data), and they were unable to match the accuracy of the best algorithm tested. However, wains showed a continuing trend of increasing accuracy over time, so it is possible that the accuracy gap between them and the best algorithm would have narrowed if there had been more data. It is also possible that a larger population would be more accurate.

The wains were much slower than traditional forecasting methods, but they were fast enough to keep up with a daily real-time data stream. The traditional algorithms are each fine-tuned for a single purpose (prediction), while the wain is designed to handle a variety of data mining and decision-making tasks; this may account for much of the difference in speed.

It is sometimes difficult to know, a priori, which of the traditional techniques, and with what configuration, will give the most accurate forecasts for a given data set. Furthermore, the patternicity of data may change so much that the

technique and configuration that works best at one point in time will not be the best choice later. Because wains evolve, we might expect them to be able to adapt to changes in the patternicity of the data.

10.2 Conclusions

It is now possible to answer all of the research questions raised in Chapter 1.

> **Research Question 1**: Will giving wains a mechanism to predict the outcomes of possible actions, and to choose the action with the best predicted outcome, make them better decision-makers?

In the experiment with handwritten numerals, the overall error rate for the new Popperian wains was significantly better than for earlier experiments with the original wains. So the answer to this question is yes.

> **Research Question 2**: Can Popperian wains learn to classify data with accuracy and speed comparable to traditional classification methods?

In the experiments with handwritten and spoken numerals, the accuracy of Popperian wains was comparable to traditional classification methods. However, whether or not the speed of the wains will be comparable with traditional methods depends on the type of data involved and the methods available for that domain.

> **Research Question 3**: Can Popperian wains learn to forecast future values in a data stream, with accuracy and speed comparable to traditional forecasting methods?

When working with univariate ISP traffic data, the wains quickly exceeded or matched the accuracy of the traditional algorithms. With the multivariate weather data, wains took longer to achieve accuracy comparable to most of the traditional algorithms (possibly due to the greater complexity of multivariate data), and they were unable to match the accuracy of one algorithm. Wains are generally slower than traditional methods; however, they are likely to be fast enough for real-time forecasting.

The experiments with grey-scale images, audio samples, ISP traffic data, and weather data indicate that wains might be used in a variety of domains. Custom data mining algorithms have been developed for some domains; suitably tweaked, they are likely to out-perform wains. Therefore, wains may be most useful in domains where custom algorithms are not available.

Based on the literature review undertaken (see Chapter 2), this is the first time an ALife species with Popperian-level AI has been applied to data mining. Popperian creatures have the ability to predict the outcome of their actions; this same ability could be used to predict future data values. For this reason, and because of their greater cognitive power, the author believes that Popperian ALife agents are better suited to data mining than are Skinnerian creatures. This is a new direction, but a promising one, as shown by the experimental results presented in this thesis.

10.3 Future directions

This section proposes future directions for research continuing on from, or inspired by, this research project. Any of the directions could be pursued inde-

pendently of the others.

10.3.1 Making decisions and managing systems

Wains have the innate ability to make decisions; most forecasting algorithms do not. Forecasting does not exist in a vacuum; it is done to support decisions, which are typically made by humans. It could be possible for wains to handle some of the decision-making. For example, consider a system where, if a variable exceeds a pre-determined threshold, a quality of service agreement will be breached. The system operator might set an alarm to be triggered when the variable reaches some lower value, in hopes that he or she will have time to diagnose and prevent a problem before it occurs. However, it is not obvious what the alarm value should be. Too high, and the operator will not be able to prevent problems from occurring. Too low, and there will be frequent "false alarms", which the operator will soon start to ignore.

Suppose that wains were introduced into such a system. They could forecast future values of the variable in question, consider how far over the threshold the value is likely to be, and take into account how confident they are in their forecasts. If a two-thirds majority of wains vote in favour, the operator would be notified of a potential problem. If the operator takes action, the wains that voted to notify the operator would be rewarded, and those that voted against might be penalised. If the operator takes no action, then the rewards and penalties would be reversed. The wains might, over time, settle on the appropriate value at which the alarm should be triggered.

We can also imagine the wains observing the corrective actions that the

operator takes, and begin to take those actions automatically when appropriate, or at least recommend those actions to the operator. Such a system would be partially or fully autonomous.

10.3.2 Improved configurations

The eventual success of wains depends greatly on the genetic makeup of the original population. Evolution cannot select for genes that are not in the gene pool. Mutation introduces new genes, but since it is random, most of the genes it introduces will be harmful rather than beneficial. Thus, mutation has a small effect on wain success. If an experiment is poorly configured, particularly as regards the initial gene pool, wains may not achieve optimum accuracy. They may even die out before they learn the task. The ideal gene pool would be large and highly varied, but an experiment with a very large population is impractical, it will take too long to run.

Observing how the genes evolve during the course of an experiment does provide useful feedback. The next time the experiment is run, the gene pool can be adjusted to include more variations of favourable genes, and fewer variations of unfavourable genes. However, this risks stranding the population on islands of local extrema.

It might be worthwhile to identify more extensive heuristics for configuring experiments. One way to achieve this would be to run many experiments using a variety of configurations to see which ones perform best. Another way would be to run a few experiments using very large populations with very diverse gene pools. Working with larger data sets might also help to identify these rules of

thumb. For example, running forecasting experiments with much longer time series data sets might show a further increase in accuracy; the resulting gene pool could be analysed to determine which genes are favourable. These rules of thumb might be applied to other domains, speeding up the learning process.

10.3.3 Smarter agents

Giving wains or other agents the ability to develop and use simple tools (such as a simple form of language) would promote them to Gregorian creatures. The associated increase in cognitive power might make them better data miners. For forecasting, agents might be given traditional forecasting algorithms as tools; the task of the agents would be to judge the reliability of the tools, and to combine the predictions from the various tools into a single, unified prediction.

Glossary

agent In this thesis, the term “agent” refers to a program that has goals, and makes decisions about how to achieve those goals. 15–17, 25–27, 32, 36, 37, 40–43, 52, 53, 59–62, 64, 66–69, 72, 75, 80, 82, 96–100, 103, 106, 112, 114, 115, 117, 120–122, 127, 130, 132, 155, 156, 162, 184

algorithm mining Tuning parameters in data mining algorithms to achieve suitable results. 35

allele One of the possible forms that a given gene may take. 31, 32, 39, 53, 61, 93

anisogamous Describes an organism which produces either egg cells or sperm cells, which have different size and form. 30

anomaly detection The identification of portions of a data set which do not follow typical patterns. 34

appearance The sensory data that is received by a wain when it encounters another wain. 39, 41, 54, 90, 91

Application Programming Interface A set of data types, subroutines, and tools for building software and applications. 85, 86, 177

Artificial Intelligence The use by machines of techniques that mimic intelligent behaviour in humans or other animals. 4, 10, 12, 16, 18, 20–23, 25–27, 51, 52, 57, 89, 159, 177

Artificial Life Refers to systems which mimic some aspects of biological life. 4, 10–16, 23–27, 29, 31, 32, 36, 37, 39, 42, 52–54, 57, 91, 96, 113, 114, 154, 159, 177

Artificial Neural Network A computer model based on biological neural networks. 97, 177

asexual reproduction A method of reproduction where the genome for an individual consists of a single strand of genetic information. Crossover, cut-and-splice, and mutation operations are performed on two parent genomes to produce two child genomes. 30, 31

Auto-Regressive Integrated Moving Average A forecasting technique. 140, 142, 144, 146, 148, 177

Automated Speech Recognition The process of converting an acoustic signal (spoken language) into the corresponding sequence of words. 13, 43–45, 49, 54, 131, 177

big data Data sets so large that traditional data analysis tools are either too slow or too cumbersome to use on them. 32, 36

Binary-Reflected Gray Code A scheme that maps values to codes in a way that guarantees that the codes for two consecutive values will differ by only one bit. Named after the developer, Frank Gray. 67, 177

carer The parent which rears a child produce by mating. 94, 138

cellular automaton A grid of cells where each cell is in one of a set of finite states; at each time step the cells follow predefined rules for changing states. 24

classification Assigning objects to predefined categories based on the attributes of the objects. 16, 34, 35, 37, 40, 43, 53, 57, 58, 100, 105, 112, 124–126, 128, 131, 133, 135, 136, 156, 157

classifier In mathematics and neural networks, a machine learning program. In wains, the component of the brain that builds a set of models representing the types of objects that it encounters. 40, 90, 96, 97, 100, 101, 103, 105–107, 109, 117, 120, 123, 125–128, 130, 131, 135, 136, 138, 156, 157

cluster analysis Partitioning objects into a set of clusters such that objects withing a cluster have similar characteristics, and objects in different clusters have dissimilar characteristics. 34, 35, 57, 58

condition A wain's energy level, passion level, boredom level, and whether or not it is currently rearing a child. 87, 88, 120, 121, 132

conjugation The mechanism where genetic material is transferred between organisms in direct contact. 82

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. 67, 70–72, 93

Créatúr A software framework for automating ALife experiments. 10, 14, 25, 39, 42, 45–48, 54, 55, 59–61, 64, 65, 68–70, 72, 83, 86, 154, 155, 172

cut-and-splice See *crossover*. 70, 71, 93

daemon A computer program that runs in the background and does not require user interaction. 42, 86, 87

Darwinian creature An organism that can adapt to the environment through recombination and mutation of genes. 11, 26, 27, 29, 113

data mining The process of discovering interesting and useful patterns in data. The term "data mining" largely overlaps with KDD. 4, 11, 13–15, 22, 23, 32–37, 53, 54, 57, 58, 96, 100, 113, 124, 137, 139, 157, 159

datatype-generic programming An implementation of generic programming for Haskell. 13, 48, 55, 64, 65, 72, 83

decider In the original wain design, the component of the brain that chooses the wain's next action in response to a stimulus. 40, 97

depth The number of hypotheses that a wain considers before making a decision. 120

descriptive data mining The process of modelling and understanding data. 33

difference threshold An SGM will not create a new model unless the difference between its input and all of its models exceeds the difference threshold. 101, 104, 128, 131

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. 30, 31, 39, 93

directed mating A mating system where wains are randomly assigned mating opportunities, with a predetermined frequency. 92, 93, 139, 143

Domain-Specific Embedded Language A DSL that is implemented as an extension of the language in which it is implemented. 13, 46, 47, 54, 64, 68–70, 76, 77, 79–81, 83, 155, 177

Domain-Specific Language A special-purpose language tailored to meet the needs of a limited domain. 46, 54, 177

dominance An effect where a child inherits two different versions of a gene, and one gene is expressed while the other has no effect. 32, 61, 72, 75, 79, 93

dominant Refers to an allele which has an effect even if it is only in one of the genetic strands inherited by the organism. 32, 39, 53, 74

egg cell The larger gamete produced by an anisogamous organism. 31

expert system An application that uses a knowledge base of human expertise to make decisions. 22

feature extraction Finding features of the audio signal that are characteristic of a particular utterance, producing representations that are more suitable for ASR. 44

filterbank A set of frequency filters. 129

fitness function A function that indicates how close a solution is to meeting a particular goal. 25

flirtation tax An amount of energy deducted from wains each time they flirt. 92, 93

free mating A mating system where wains decide when to mate, and with whom. 92

functional programming A programming paradigm that treats expressions as mathematical functions, and avoids side effects of computation. 45, 54, 181

gamete In biology, a sex cell. In Créatúr, a sequence of genes donated by one parent. 30, 31, 63, 69

gene A unit of heredity. 14, 16, 26, 31, 32, 39, 41, 53, 61, 64–68, 72, 75–77, 79, 80, 83, 92–94, 116, 139, 144, 184

gene expression The mechanism that determines the phenotype of an organism from its genotype. 31, 92

generic programming Programming that references types to be specified later. 47, 55

genetically determined Refers to a trait that is specified by an agent's genes, can be different for each agent, is inherited by offspring, and is subject to evolutionary pressures. 41, 88, 91, 94, 116, 120, 122

genome The set of genes for an organism. 61, 63, 64, 68, 69, 75–77, 79, 92

genotype A sequence of genes. Compare with *phenotype*. 31

Graphics Porcessing Unit A processor that is designed to handle graphics operations. Also known as a "graphics card". 114, 177

Gregorian creature An organism that can use tools, including "mind tools" like words. 29, 30, 114, 162

Hamming window A mathematical function commonly used in spectral analysis. 129

haploid An organism or cell having only one sequence of genes. 30

happiness A metric that summarises a wain's condition. Wains are motivated to make decisions that will maximise their happiness. 88, 89, 121, 125, 132

Haskell A purely functional programming language named after the logician Haskell Curry. 45–47, 54, 59, 61, 64, 65, 83, 180

Hidden Markov Model A widely-used ASR technique. 45, 129, 131, 132, 134, 135, 178

Holt-Winters A forecasting technique. 142, 144, 146, 148

horizontal gene transfer the mechanism where genetic material is transferred directly from one organism to another instead of vertically from parent to offspring. 82

hypothesis The set of classifier labels proposed for the objects encountered by a wain. 119–122

imprint outcomes The energy, boredom level, passion level, and litter size used to create a new response model during imprinting. 122

imprinting A technique where a wain is shown one or more patterns and an action, and concludes that taking the action in a similar situation would optimise its condition, maximising its happiness. 94, 122

instruction-based ALife Small computer programs which can replicate, recombine, mutate, and evolve. 24–26

isogamous Describes an organism which produces gametes of a single size and form. 30

Knowledge Discovery from Data The overall process of extracting knowledge from data. Data mining is a step in this process. 34, 178

landscape mining Exploring a data set to find the space of possible inferences and identify interesting patterns. 34, 35, 57

learning rate Controls how much adjustment classifier models are adjusted during the learning phase. 38, 42, 101–103, 105, 117, 128, 130, 142

liftering Filtering on a cepstrum. 129

mating type Fertilisation can only occur between gametes of different mating types. 30

Mean of Absolute Differences A measure of difference between two images. 102–104, 127, 136, 178

Mel-Frequency Cepstral Coefficient A type of feature vector used in ASR. 44, 85, 129, 178

metabolism tax An amount of energy deducted from wains at each CPU turn. 90, 138, 139, 143

MNIST A database of handwritten digits, prepared by Yann LeCun and Corinna Cortes. 40, 49, 55, 102, 103, 105, 127, 132

modelling Producing a (typically simpler) representation of a data set that captures important features and relationships. 35, 57, 58

monad In functional programming, a structure that represents computations. 13, 46, 47, 54, 55, 64, 65, 75–77, 79, 80, 83

muser A component of a wain's brain that generates possible responses to situations. 117, 120, 123, 156

mutation In ALife, randomly altering a bit in a gene sequence. 41, 64, 67, 77, 91, 93

Neural Darwinism A theory proposed by Gerald Edelman which states that connections in the brain undergo a type of natural selection. 40

neural network See artificial neural network. 128

Neural Network Ensemble A technique where multiple neural networks are trained simultaneously. 146, 178

neuron A nerve cell, or a node in an ANN. 20

new implementation The implementation of wains and Créatúr that incorporates the changes described in this thesis. 39, 40, 58, 60, 84, 88–90, 92–95, 116, 117, 119, 121, 123, 125, 126, 133, 135, 155, 158

node In mathematics, a vertex in a graph. In neural networks, another term for an artificial neuron. 38, 101, 103, 107

operant conditioning A method of learning where actions that are rewarded are reinforced, and likely to be repeated in future on similar occasions. 27–29, 40, 57

original implementation The version of wains and Créatúr documented in [2, 1]. 39–41, 57, 58, 60, 84, 88–90, 92–94, 96, 97, 125, 126, 133, 135, 158

p-score A value that controls how much a wain's should rely on a hypothesis when making a decision. 117, 119, 121

persistence In computer science, the ability for data to be preserved between executions of a program. 42, 86

phenotype The physical characteristics of an organism. Compare with *genotype*. 31, 32

Popperian creature An organism that can evaluate possible actions and make choices based on that evaluation. 11, 12, 14, 16, 28–30, 57, 58, 113–116,

121, 123, 154, 156, 159

prediction See *predictive data mining*. 35, 53, 57, 58, 89, 124, 137, 139, 140, 142–144, 146, 148, 149, 151, 152, 157, 160

predictive data mining Using some variables in a data set to predict others. 33

predictor A component of a wain's brain that maintains a model of the space of responses selected, and their outcomes. 117, 120, 121, 123, 156

R A statistical programming language. 140, 144

recessive Refers to an allele which only has an effect if it is in both genetic strands inherited by the organism. 32, 39, 53

regression Identifying functions which map data objects to prediction variables. 35, 57

reinforcement deltas Control the rate at which energy, boredom level, passion level, and litter size in an existing response model is strengthened during imprinting. 122

Self-Generating Model A modified version of the SOM. 14–17, 96, 101–107, 109, 111, 112, 116, 117, 123, 128, 133, 156, 178, 185

Self-Organising Map A technique for representing high-dimensional data in fewer dimensions while preserving the topology of the input data. 13, 14, 16, 37, 38, 40, 42, 53, 58, 96–107, 109, 111, 112, 116, 128, 129, 132, 133, 135, 156, 178

sexual dimorphism The condition where organisms of the same species exhibit differences according to whether they are male or female. 31

sexual reproduction A method of reproduction where the genome for an individual consists of two strands of paired genetic information. Crossover, cut-and-splice, and mutation operations are performed on two parent genomes to produce two child genomes. 30–32, 39, 54, 62, 93

signature In the context of a classfier, a vector whose elements indicate how similar each input pattern is to each classifier model. 117

Simple Moving Average A forecasting technique. 140, 144, 146–149, 151, 178

simplified sexual reproduction In this thesis, the term "simplified sexual reproduction" is used for an artificial method of reproduction where the genome for an individual consists of a single strand of genetic information. This technique is not known to occur in nature, but is commonly used in ALife applications. Crossover, cut-and-splice, and mutation operations are performed on two parent genomes to produce two child genome, of which one is often discarded. 62

Skinnerian creature Organisms that can adapt to the environment through operant conditioning. 11, 27–30, 40, 57, 113, 159

sperm cell The smaller gamete produced by an anisogamous organism. 30

stable model A classifier model that, despite adjustments, continues to be a good representation of the stimulus it was created in response to. . Compare with *unstable model*. 16, 98, 100, 101, 104, 105, 107, 109, 112, 116, 156,

175

strictness Higher values of strictness cause the brain to give less consideration to hypotheses that are not the most likely. 119

TI46 A database of audio samples of spoken numerals. 49, 55, 132

Tower of Generate-and-Test A framework developed by Dennett for ranking brain designs. 11, 12, 15, 26, 29, 30

transduction The mechanism where foreign genetic material is introduced into a cell by a virus. 82

transformation The mechanismwhere an organism incorporates naked genetic material from its surroundings. 82

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. 63, 72

tweaker A component of an SGM that can measure the difference between an input pattern and each of its models, and tweak a model to more closely match an input pattern. 116

unstable model A classifier model that is adjusted so much that it is no longer a good representation of the stimulus it was created in response to. Compare with *stable model*. 99, 175

unsupervised learning A training method in which the desired responses (target values) for the input vectors in the training set are not known. 100, 138

used model A classifier model that is the winning node for at least one input pattern during testing or classification. Compare with wasted model. 100, 109, 176

Vector Auto-Regression A forecasting technique. 144, 149, 151, 178

visualisation Making insights about data understandable by humans. 35, 57

wain An artificial lifeform in the Créatúr habitat. 4, 10–15, 17, 39–43, 45, 53, 54, 56–59, 61, 82, 84–94, 96, 100, 101, 112, 113, 123–126, 128, 131, 132, 134–139, 142–144, 146–149, 151, 152, 154–162, 172, 183

wasted model A classifier model that goes unused during testing or classification. Compare with *used model*. 16, 99, 100, 105, 107–109, 112, 116, 156, 176

Weighted Moving Average A forecasting technique. 140, 144, 147–149, 151, 178

window width The number of values used to calculate a moving average. 140

winning node In a SOM, the node whose weight vector is most similar to the input pattern. 38, 96, 100–104, 106, 109, 111, 128, 133

Acronyms

AI Artificial Intelligence. 4, 10, 12, 16, 18, 20–23, 25–27, 39, 51–53, 57, 89, 154, 159

ALife Artificial Life. 4, 10–16, 23–27, 29, 31, 32, 36, 37, 39, 42, 52–54, 57, 91, 96, 113, 114, 154, 159

ANN Artificial Neural Network. 20, 97

API Application Programming Interface. 84–86, 94, 155

ARIMA Auto-Regressive Integrated Moving Average. 140, 142, 144, 146, 148

ASR Automated Speech Recognition. 13, 43–45, 49, 54, 131

BRGC Binary-Reflected Gray Code. 67

DSEL Domain-Specific Embedded Language. 13, 46, 47, 54, 64, 68–70, 76, 77, 79–81, 83, 155

DSL Domain-Specific Language. 46, 54

GPU Graphics Porcessing Unit. 114

HMM Hidden Markov Model. 45, 129, 131, 132, 134, 135

ISP Internet Service Provider. 49, 55, 137, 151, 157, 159

KDD Knowledge Discovery from Data. 34

MAD Mean of Absolute Differences. 102–104, 127, 136

MFCC Mel-Frequency Cepstral Coefficient. 44, 85, 129

NNE Neural Network Ensemble. 146

SGM Self-Generating Model. 14–17, 96, 101–107, 109, 111, 112, 116, 117, 123, 128, 133, 156, 185

SMA Simple Moving Average. 140, 144, 146–149, 151

SOM Self-Organising Map. 13, 14, 16, 37, 38, 40, 42, 53, 58, 96–107, 109, 111, 112, 116, 128, 129, 132, 133, 135, 156

VAR Vector Auto-Regression. 144, 149, 151

WMA Weighted Moving Average. 140, 144, 147–149, 151

Bibliography

[1] Amy de Buitléir. "Evolving Pattern-seeking Artificial Life with Créatúr". MSc Thesis. Athlone Institute of Technology, 2011.

[2] Amy de Buitléir, Michael Russell, and Mark Daly. "Wains: A pattern-seeking artificial life species". In: *Artificial Life* 18.4 (2012), pp. 399–423. DOI: 10.1162/artl_a_00074.

[3] D. C. Dennett. "Why the Law of Effect will not Go Away". In: *Journal for the Theory of Social Behaviour* 5.2 (1975), pp. 169–188. ISSN: 1468-5914. DOI: 10.1111/j.1468-5914.1975.tb00350.x. URL: http://dx.doi.org/10.1111/j.1468-5914.1975.tb00350.x.

[4] Daniel Dennett. *Kinds of Minds: The Origins of Consciousness.* London: Phoenix, 1997. ISBN: 0-7538-0043-8.

[5] D.C. Dennett. *Darwin's Dangerous Idea: Evolution and the Meanings of Life.* Penguin UK, 1996. ISBN: 9780141949253.

[6] Shane Legg and Marcus Hutter. "Universal intelligence: A definition of machine intelligence". In: *Minds and Machines* 17.4 (2007), pp. 391–444. URL: http://arxiv.org/abs/0712.3329.

[7] R.J. Sternberg. *Handbook of Intelligence*. Cambridge University Press, 2000. ISBN: 9780521596480. URL: https://books.google.ie/books?id=YnBGMpIMfJ0C.

[8] William J. Rapaport. *Some Definitions of Artificial Intelligence*. Sept. 19, 1985. URL: http://www.cse.buffalo.edu/~rapaport/definitions.of.ai.html.

[9] D. Wechsler and J.D. Matarazzo. *Wechsler's Measurement and Appraisal of Adult Intelligence*. Williams & Wilkins, 1972. ISBN: 9780195022964. URL: https://books.google.ie/books?id=rJJ9AAAAMAAJ.

[10] J.R. Slagle. *Artificial Intelligence: The Heuristic Programming Approach*. McGraw-Hill series in systems science. McGraw-Hill, 1971. URL: https://books.google.ie/books?id=p7YlAAAAMAAJ.

[11] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[12] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. New York: Wiley, June 15, 1949. ISBN: 0-8058-4300-0.

[13] A. Turing. "Computing machinery and intelligence". In: *Mind* 59 (1950), pp. 433–460. URL: http://links.jstor.org/sici?sici=0026-4423(195010)2:59:236%3C433:CMAI%3E2.0.CO;2-5.

[14] K. Frankish and W.M. Ramsey. *The Cambridge Handbook of Artificial Intelligence*. Cambridge University Press, 2014. ISBN: 9780521871426. URL: https://books.google.ie/books?id=RYOYAwAAQBAJ.

[15] Elaine Woo. "John McCarthy dies at 84; the father of artificial intelligence". In: *Los Angeles Times* (Oct. 27, 2011). URL: http://www.latimes.com/local/obituaries/la-me-john-mccarthy-20111027-story.html.

[16] Will Knight. "What Marvin Minsky Still Means for AI". In: *MIT Technology Review* (Jan. 26, 2016). URL: https://www.technologyreview.com/s/546116/what-marvin-minsky-still-means-for-ai/.

[17] J. McCarthy et al. *A Proposal For The Dartmouth Summer Research Project on Artificial Intelligence*. 1955. URL: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html.

[18] Arthur L Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[19] Daniel G Bobrow. "Natural language input for a computer problem solving system". In: AI memo (1964). URL: http://dspace.mit.edu/handle/1721.1/5922.

[20] Terry Winograd. *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*. Tech. rep. Mit AI technical report 235, MIT, 1971. URL: http://hci.stanford.edu/winograd/shrdlu/AITR-235.pdf.

[21] N.J. Nilsson. *The Quest for Artificial Intelligence*. Cambridge University Press, 2009. ISBN: 9781139642828. URL: https://books.google.ie/books?id=nUJdAAAAQBAJ.

[22] H.A. Simon. *The shape of automation for men and management*. Harper & Row, 1965. URL: https://books.google.ie/books?id=ZX-aAAAAIAAJ.

[23] M.L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall series in automatic computation. Prentice-Hall, 1967. URL: https://books.google.ie/books?id=CURDAAAAIAAJ.

[24] H. Moravec. *Mind Children: The Future of Robot and Human Intelligence*. Harvard University Press, 1988. ISBN: 9780674576186. URL: https://books.google.ie/books?id=56mb7XuSx3QC.

[25] Richard M Karp. "Reducibility among combinatorial problems". In: *50 Years of Integer Programming 1958-2008*. Springer, 2010, pp. 219–241.

[26] P. Warren, J. Davies, and D. Brown. *ICT Futures: Delivering Pervasive, Real-time and Secure Services*. Wiley, 2008. ISBN: 9780470758663. URL: https://books.google.ie/books?id=uXMs6qStr4QC.

[27] William J. Rapaport. *Definition of: expert system*. 2016. URL: http://www.pcmag.com/encyclopedia/term/42865/expert-system.

[28] G.S. Linoff and M.J.A. Berry. *Data Mining Techniques: For Marketing, Sales, and Customer Relationship Management*. IT Pro. Wiley, 2011. ISBN: 9781118087459. URL: https://books.google.ie/books?id=AyQfVTDJypUC.

[29] Joab Jackson. *IBM Watson Vanquishes Human Jeopardy Foes*. Feb. 16, 2011. URL: http://www.pcworld.com/article/219893/ibm_watson_vanquishes_human_jeopardy_foes.html.

[30] Marvin Minsky. "The age of intelligent machines: thoughts about artificial intelligence". In: *KurzweilAI. net (en línea) http://www. kurzweilai. net/meme/frame. html* (1990). URL: http://www.universelle-automation.de/1991_Boston.pdf.

[31] Jennifer Kahn. "It's Alive". In: *Wired* (Mar. 2002). URL: http://www.wired.com/wired/archive/10.03/everywhere.html.

[32] D. R. Hofstadter. *Gödel, Escher, Bach: an eternal golden braid.* New York: Vintage Books, 1980. ISBN: 0-394-74502-7.

[33] C. G. Langton. "Artificial Life". In: *Artificial life: the proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, held September, 1987, in Los Alamos, New Mexico.* Ed. by C. G. Langton. Addison-Wesley, Redwood City, CA, 1989, pp. 1–48.

[34] Longbing Cao. *Data Mining and Multi-agent Integration.* Boston: Springer-Verlag US, 2009. ISBN: 978-1-4419-0522-2. URL: http://dx.doi.org/10.1007/978-1-4419-0522-2.

[35] Eduardo Reck Miranda. *A-life for music : music and computer models of living systems.* Middleton, Wis.: A-R Editions, 2011. ISBN: 978-0-89579-673-8. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=9780895796738.

[36] Tibebe Dessalegne and JohnW. Nicklow. "Artificial Life Algorithm for Management of Multi-reservoir River Systems". English. In: *Water Resources Management* 26.5 (2012), pp. 1125–1141. ISSN: 0920-4741. DOI: 10.1007/s11269-011-9950-7. URL: http://dx.doi.org/10.1007/s11269-011-9950-7.

[37] John Von Neumann. “The general and logical theory of automata”. In: *Cerebral mechanisms in behavior* 1.41 (1951), pp. 1–2. URL: http://old.nbu.bg/cogs/events/2002/materials/Jeff/L1R2_automat.PDF.

[38] J.L. Schiff. *Cellular Automata: A Discrete View of the World*. Wiley Series in Discrete Mathematics & Optimization. Wiley, 2011. ISBN: 9781118030639. URL: https://books.google.ie/books?id=uXJC2C2sRbIC.

[39] Martin Gardner. “Mathematical Games: The fantastic combinations of John Conway’s new solitaire game ”life””. In: *Scientific American* 223 (1970), pp. 120–123. URL: https://web.archive.org/web/20090603015231/http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm.

[40] Christopher G. Langton, ed. *Artificial life: the proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems, held September, 1987, in Los Alamos, New Mexico*. Vol. VI. Santa Fé Institute studies in the sciences of complexity. Addison-Wesley, 1989. ISBN: 978-0-201-09346-9. URL: http://books.google.ie/books?id=1T9RAAAAMAAJ.

[41] Craig W Reynolds. “Flocks, herds and schools: A distributed behavioral model”. In: *ACM SIGGRAPH computer graphics* 21.4 (1987), pp. 25–34. URL: http://www.macs.hw.ac.uk/~dwcorne/Teaching/Craig%20Reynolds%20Flocks,%20Herds,%20and%20Schools%20A%20Distributed%20Behavioral%20Model.htm.

[42] Balázs Szigeti et al. “OpenWorm: an open-science approach to modeling Caenorhabditis elegans”. In: *Frontiers in Computational Neuroscience* 8

(2014), p. 137. ISSN: 1662-5188. DOI: 10.3389/fncom.2014.00137. URL: http://journal.frontiersin.org/article/10.3389/fncom.2014.00137.

[43] Thomas S. Ray. "An Evolutionary Approach to Synthetic Biology: Zen and the Art of Creating Life." In: *Artificial Life* 1.1-2 (1994), pp. 179–210. URL: http://dblp.uni-trier.de/db/journals/alife/alife1.html#Ray94.

[44] Thomas S. Ray. "An approach to the Synthesis of Life". In: *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity.* Ed. by C. Langton, C. Taylor, J. D. Farmer, S. Rasmussen. Vol. XI. Redwood City, CA: Addison-Wesley, 1991, pp. 371–408.

[45] Chris Adami and C. Titus Brown. "Evolutionary Learning in the 2D Artificial Life System "Avida"". In: *Artificial life IV.* Vol. 1194. cite arxiv:adap-org/9405003Comment: 5 p., postscript with figures (unpack with uufiles), to appear in the Proc. of "Artificial Life IV", MIT Press. MIT Press, Cambridge, MA, 1994, pp. 377–381. URL: http://arxiv.org/abs/adap-org/9405003.

[46] Maciej Komosinski and Szymon Ulatowski. "Framsticks-artificial life". In: *ECML'98 Demonstration and Poster Papers, Chemnitzer Informatik Berichte* (1998), pp. 7–9.

[47] Jon Klein. "Breve: a 3d environment for the simulation of decentralized systems and artificial life". In: *Proceedings of the eighth international conference on Artificial life.* 2003, pp. 329–334.

[48] Unknown. *Darwinbots (website)*. 2016. URL: http://wiki.darwinbots.com/index.php?title=Main_Page.

[49] Tom Barbalet. "The Mind of the Noble Ape in Three Simulations". English. In: *Origins of Mind*. Ed. by Liz Swan. Vol. 8. Biosemiotics. Springer Netherlands, 2013, pp. 383–397. ISBN: 978-94-007-5418-8. DOI: 10.1007/978-94-007-5419-5_20. URL: http://dx.doi.org/10.1007/978-94-007-5419-5_20.

[50] Thomas S. Barbalet. "Noble Ape's Cognitive Simulation: From Agar to Dreaming and Beyond". In: *Nature-Inspired Informatics for Intelligent Applications and Knowledge Discovery: Implications in Business, Science, and Engineering*. IGI Global, 2010, pp. 242–258. ISBN: 978-1-60566-705-8. DOI: 10.4018/978-1-60566-705-8.ch010. URL: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-60566-705-8.ch010.

[51] Larry Yaeger. "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*. Ed. by Christopher G. Langton. Los Alamos, New Mexico: Addison-Wesley, 1993, pp. 263–298. URL: http://www.beanblossom.in.us/larryy/polyworld.html.

[52] L Yaeger and 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*. Ed. by Luis Mateus Rocha et al. Cambridge, MA: MIT Press, 2006, pp. 330–336.

[53] L. Yaeger, V. Griffith, and O. Sporns. "Passive and driven trends in the evolution of complexity". In: *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*. Ed. by S. Bullock et al. Cambridge, MA: MIT Press, 2008, pp. 725–732. URL: http://alifexi.alife.org/papers/ALIFExi%5C_pp725-732.pdf.

[54] Larry S. Yaeger. "How evolution guides complexity". In: *HFSP Journal* 3.5 (2009), p. 328. ISSN: 19552068. DOI: 10.2976/1.3233712. URL: http://www.tandfonline.com/doi/pdf/10.2976/1.3233712.

[55] Stephen Grand and Dave Cliff. "Creatures: Entertainment Software Agents with Artificial Life". In: *Autonomous Agents and Multi-Agent Systems* 1.1 (1997), pp. 39–57.

[56] Steve Grand. *Creation: Life and how to make it*. London: Phoenix, 2001. ISBN: 0-7538-1277-0.

[57] Bob Winckelmans. *Critterding*. 2013. URL: http://critterding.sourceforge.net/.

[58] Daniel C. Dennett. *Consciousness explained*. Penguin, 1993. ISBN: 978-0-14-012867-3. URL: http://www.worldcat.org/isbn/9780140128673.

[59] Yvonne Bernard et al. "Self-organisation and evolution for trust-adaptive grid computing agents". In: *Evolution, Complexity and Artificial Life*. Springer, 2014, pp. 209–224. URL: ftp://aliceford.ce.unipr.it/pub/cagnoni/LW/Bernard.pdf.

[60] Nathalie Gontier. "Evolutionary Epistemology". In: *Internet Encyclopedia of Philosophy*. Accessed Wed Aug 16 17:56:48 IST 2017.

[61] Donald T. Campbell. "Blind variation and selective retentions in creative thought as in other knowledge processes." In: *Psychological Review* 67.6 (1960), pp. 380–400. DOI: 10.1037/h0040373. URL: https://doi.org/10.1037/h0040373.

[62] B.F. Skinner. *Science And Human Behavior*. Free Press, 2012. ISBN: 9781476716152. URL: https://books.google.ie/books?id=QcbJInkd%5C_iMC.

[63] R.L. Gregory. *Mind in Science: A History of Explanations in Psychology and Physics*. Peregrine book : psychology, philosophy. Weidenfeld & Nicolson, 1981. ISBN: 9780297778257. URL: https://books.google.ie/books?id=K5V-AAAAMAAJ.

[64] C.J. Barnard. *Animal Behaviour: Mechanism, Development, Function and Evolution*. Pearson Education, 2004. ISBN: 9780130899361. URL: https://books.google.ie/books?id=di5%5C_OFxTwf4C.

[65] L. Swan, R. Gordon, and J. Seckbach. *Origin(s) of Design in Nature: A Fresh, Interdisciplinary Look at How Design Emerges in Complex Systems, Especially Life*. Cellular Origin, Life in Extreme Habitats and Astrobiology. Springer Netherlands, 2012. ISBN: 9789400741560. URL: https://books.google.ie/books?id=2jb3PL6Yn%5C_oC.

[66] S. Khurshid. *Knowledge Processing Creativity and Politics: A Political Theory Based on the Evolutionary Theory*. AuthorHouse, 2006. ISBN: 9781425907464. URL: https://books.google.ie/books?id=MYhpvryxmbsC.

[67] Alan FT Winfield. "You really need to know what your bot (s) are thinking about you". In: (2010).

[68] J. Pitt. *The Computer After Me: Awareness and Self-Awareness in Autonomic Systems.* World Scientific Publishing Company, 2014. ISBN: 9781783264193. URL: https://books.google.ie/books?id=29O3CgAAQBAJ.

[69] Tibor Solymosi. "We Deweyan Creatures". In: *Pragmatism Today* 7.1 (2016), pp. 41–59.

[70] George L Chadderdon. "Assessing machine volition: An ordinal scale for rating artificial and natural systems". In: *Adaptive Behavior* 16.4 (2008), pp. 246–263.

[71] Leo Beukeboom and Nicolas Perrin. *The Evolution of Sex Determination.* OUP Oxford, 2014. ISBN: 978-0191631405. URL: https://www.amazon.com/Evolution-Sex-Determination-Leo-Beukeboom-ebook/dp/B00LW6IAE4?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=B00LW6IAE4.

[72] Raffaele Calabretta et al. "Two is better than one: A diploid genotype for neural networks". In: *Neural Processing Letters* 4.3 (1996), pp. 149–155.

[73] R. E. Smith and D. E. Goldberg. "Diploidy and Dominance in Artificial Genetic Search". In: *Complex Systems* 6.3 (1992). R. E. Smith and D. E. Goldberg, "Diploidy and Dominance in Artificial Genetic Search", Complex Systems, Vol. 6, pp. 251-285, 1992., pp. 251–285.

[74] Richard Lewontin. “The Genotype/Phenotype Distinction”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2011. 2011.

[75] The Financial Times. *Decoding Big Data*. Penguin UK, 2013. URL: http://books.google.ie/books?id=SOx16gJmrEQC&printsec=frontcover&dq=Decoding+Big+Data+financial+times&hl=ga&sa=X&ei=A-XDU9e-LuLA7AaH5IDYAg&redir_esc=y#v=onepage&q=Decoding%20Big%20Data%20financial%20times&f=false.

[76] Viktor Mayer-Schönberger and Kenneth Cukier. *Big data a revolution that will transform how we live, work, and think*. 2013. URL: http://oclc-marc.ebrary.com/Doc?id=10659211.

[77] Jiawei Han and Micheline Kamber. *Data mining: Concepts and techniques*. Burlington, MA: Elsevier, 2011. ISBN: 978-0-12-381479-1. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=9780123814791.

[78] I. H. Witten, Eibe Frank, and Mark A. Hall. *Data mining: Practical machine learning tools and techniques*. Burlington, MA: Morgan Kaufmann, 2011. ISBN: 978-0-12-374856-0. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=9780123748560.

[79] Florin Gorunescu. *Data mining concepts, models and techniques*. 2011. URL: http://site.ebrary.com/id/10454853.

[80] Sang C. Suh. *Practical applications of data mining*. Sudbury, Mass.: Jones & Bartlett Learning, 2012. ISBN: 978-0-7637-8587-1. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=9780763785871.

[81] Oded Z. Maimon and Lior Rokach. *Data mining and knowledge discovery handbook*. New York: Springer, 2010. ISBN: 978-0-387-09822-7. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=9780387098227.

[82] Tim Menzies. "Beyond Data Mining". In: *IEEE Software* 30.3 (2013), p. 92. ISSN: 0740-7459. DOI: http://doi.ieeecomputersociety.org/10.1109/MS.2013.49.

[83] Michael Goebel and Le Gruenwald. "A survey of data mining and knowledge discovery software tools". In: *ACM SIGKDD Explorations Newsletter* 1.1 (1999), pp. 20–33. URL: http://www.lcb.uu.se/users/janko/data/goebel_sigkddexp99.pdf.

[84] Rui Xu and II Wunsch, D. "Survey of clustering algorithms". In: *Neural Networks, IEEE Transactions on* 16.3 (May 2005), pp. 645–678. ISSN: 1045-9227. DOI: 10.1109/TNN.2005.845141. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1427769.

[85] Usama M Fayyad. *Advances in Knowledge Discovery and Data Mining (American Association for Artificial Intelligence)*. MIT Press, 1996. ISBN: 978-0-262-56097-9. URL: http://www.amazon.co.uk/dp/0262560976.

[86] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: From natural to artificial systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford: Oxford University Press, 1999. ISBN: 978-0-19-513159-8. URL: http://books.google.ie/books?id=PvTDhzqMr7cC.

[87] Ajith Abraham, Crina Grosan, and Vitorino Ramos. *Swarm intelligence in data mining*. Berlin; New York: Springer, 2006. ISBN: 978-3-540-34956-3. URL: http://site.ebrary.com/id/10157825.

[88] Marco Dorigo et al. *Positive Feedback as a Search Strategy*. Tech. rep. Technical Report No. 91-016, Politecnico di Milano, Italy, 1991.

[89] David Martens, Bart Baesens, and Tom Fawcett. "Editorial survey: Swarm intelligence for data mining". In: *Machine Learning* 82.1 (2011). 10.1007/s10994-010-5216-5, pp. 1–42. ISSN: 0885-6125. URL: http://dx.doi.org/10.1007/s10994-010-5216-5.

[90] Luis Felipe Giraldo, Fernando Lozano, and Nicanor Quijano. "Foraging theory for dimensionality reduction of clustered data". In: *Machine Learning* 82.1 (Jan. 2011), pp. 71–90. ISSN: 0885-6125. DOI: 10.1007/s10994-009-5156-0. URL: http://dx.doi.org/10.1007/s10994-009-5156-0.

[91] Richard K. Belew. "Artificial Life: A Constructive Lower Bound for Artificial Intelligence". In: *IEEE Intelligent Systems* 6 (1991), pp. 8–15. ISSN: 0885-9000. DOI: http://doi.ieeecomputersociety.org/10.1109/64.73812.

[92] Teuvo Kohonen. *Self-organizing maps*. 3rd. Springer series in information sciences, 30. Berlin: Springer, Dec. 28, 2001. ISBN: 978-3-540-67921-9. URL: http://www.worldcat.org/isbn/3540679219.

[93] T. Villmann et al. "Topology preservation in self-organizing feature maps: exact definition and measurement". In: *Neural Networks, IEEE Transactions on* 8.2 (Mar. 1997), pp. 256–266. ISSN: 1045-9227. DOI: 10.1109/72.

557663. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=557663.

[94] Alfred Ultsch and H. Peter Siemon. “Kohonen’s Self Organizing Feature Maps for Exploratory Data Analysis”. In: *Proceedings of the International Neural Network Conference (INNC-90), Paris, France, July 9-13, 1990 1. Dordrecht, Netherlands*. Ed. by Bernard Widrow and Bernard Angeniol. Vol. 1. Dordrecht, Netherlands: Kluwer Academic Press, 1990, pp. 305–308. URL: http://www.uni-marburg.de/fb12/datenbionik/pdf/pubs/1990/UltschSiemon90.

[95] Robert Saunders and John S Gero. “Artificial creativity: A synthetic approach to the study of creative behaviour”. In: *Computational and Cognitive Models of Creative Design V, Key Centre of Design Computing and Cognition, University of Sydney, Sydney* (2001), pp. 113–139.

[96] João M Martins and Eduardo R Miranda. “A connectionist architecture for the evolution of rhythms”. In: *Applications of Evolutionary Computing*. Springer, 2006, pp. 696–706.

[97] Thomas Riga, Angelo Cangelosi, and Alberto Greco. “Symbol grounding transfer with hybrid self-organizing/supervised neural networks”. In: *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*. Vol. 4. IEEE. 2004, pp. 2865–2869.

[98] Rob Saunders et al. “Curious whispers: an embodied artificial creative system”. In: *International conference on computational creativity*. 2010, pp. 7–9.

[99] Helge Ritter. "Self-organizing maps on non-euclidean spaces". In: *Kohonen maps* 73 (1999), pp. 97–110.

[100] Damminda Alahakoon, Saman K. Halgamuge, and Bala Srinivasan. "Dynamic self-organizing maps with controlled growth for knowledge discovery". In: *Neural Networks, IEEE Transactions on* 11.3 (May 2000), pp. 601–614. ISSN: 1045-9227. DOI: 10.1109/72.846732. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=846732.

[101] Pasi Koikkalainen and Erkki Oja. "Self-organizing hierarchical feature maps". In: *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. Vol. 2. June 1990, pp. 279–284. DOI: 10.1109/IJCNN.1990.137727.

[102] Teuvo Kohonen. "The adaptive-subspace SOM (ASSOM) and its use for the implementation of invariant feature detection". In: *Proc. ICANN*. Vol. 95. 1995, pp. 3–10.

[103] Luana Bezerra Batista, Herman Martins Gomes, and Raul Fernandes Herbster. "Application of growing hierarchical self-organizing map in handwritten digit recognition". In: *Proceedings of 16th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*. 2003, pp. 1539–1545.

[104] Hubert Cecotti and A.bdel Belaïd. "Rejection strategy for convolutional neural network by adaptive topology applied to handwritten digits recognition". In: *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*. Vol. 2. Aug. 2005, pp. 765–769. DOI: 10.

1109/ICDAR.2005.200. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1575648.

[105] Ehsan Mohebi and Adil Bagirov. "A convolutional recursive modified Self Organizing Map for handwritten digits recognition". In: *Neural Networks* 60 (2014), pp. 104–118. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2014.08.001. URL: http://www.sciencedirect.com/science/article/pii/S0893608014001968.

[106] Jörg Ontrup and Helge Ritter. "A hierarchically growing hyperbolic self-organizing map for rapid structuring of large data sets". In: *Proceedings of the 5th Workshop on Self-Organizing Maps, Paris (France).* 2005.

[107] Jussi Pakkanen. "The Evolving Tree, a new kind of self-organizing neural network". In: *proceedings of the Workshop on Self-Organizing Maps.* Vol. 3. Citeseer. 2003, pp. 311–316.

[108] Andreas Rauber, Dieter Merkl, and Michael Dittenbach. "The growing hierarchical self-organizing map: exploratory analysis of high-dimensional data". In: *Neural Networks, IEEE Transactions on* 13.6 (Nov. 2002), pp. 1331–1341. ISSN: 1045-9227. DOI: 10.1109/TNN.2002.804221.

[109] Hamed Shah-Hosseini. "Binary tree time adaptive self-organizing map". In: *Neurocomputing* 74.11 (2011). Adaptive Incremental Learning in Neural NetworksLearning Algorithm and Mathematic Modelling Selected papers from the International Conference on Neural Information Processing 2009 (ICONIP 2009)ICONIP 2009, pp. 1823–1839. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2010.07.037. URL: http://www.sciencedirect.com/science/article/pii/S0925231211000786.

[110] Huicheng Zheng et al. "Learning nonlinear manifolds based on mixtures of localized linear manifolds under a self-organizing framework". In: *Neurocomputing* 72.13–15 (2009). Hybrid Learning Machines (HAIS 2007) / Recent Developments in Natural Computation (ICNC 2007), pp. 3318–3330. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2009.01.008. URL: http://www.sciencedirect.com/science/article/pii/S0925231209000605.

[111] Gerald M. Edelman. *Neural Darwinism : the theory of neuronal group selection*. Basic Books, 1987. ISBN: 978-0-465-04934-9. URL: http://www.worldcat.org/isbn/9780465049349.

[112] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[113] Rosemary T. Salaja, Ronan Flynn, and Michael Russell. "Automatic speech recognition using artificial life". In: *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communities Technologies (ISSC 2014/CIICT 2014)*. Institution of Engineering and Technology (IET), 2014. DOI: 10.1049/cp.2014.0665. URL: http://dx.doi.org/10.1049/cp.2014.0665.

[114] Rosemary T. Salaja, Ronan Flynn, and Michael Russell. "Evaluation of wains as a classifier for automatic speech recognition". In: *Signals and Systems Conference (ISSC), 2015 26th Irish*. June 2015, pp. 1–6. DOI: 10.1109/ISSC.2015.7163770. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7163770.

[115] Li Deng and Xiao Li. "Machine Learning Paradigms for Speech Recognition: An Overview". In: *Audio, Speech, and Language Processing, IEEE*

Transactions on 21.5 (May 2013), pp. 1060–1089. ISSN: 1558-7916. DOI: 10.1109/TASL.2013.2244083. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6423821.

[116] K. F. Lee. *Automatic Speech Recognition: The Development of the SPHINX System*. The Springer International Series in Engineering and Computer Science. Springer US, 2012. ISBN: 978-1-4615-3650-5. URL: https://books.google.ie/books?id=KVwFCAAAQBAJ.

[117] Douglas O'Shaughnessy. "Invited paper: Automatic speech recognition: History, methods and challenges". In: *Pattern Recognition* 41.10 (2008), pp. 2965–2979. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2008.05.008. URL: http://www.sciencedirect.com/science/article/pii/S0031320308001799.

[118] Steven B Davis and Paul Mermelstein. "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences". In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 28.4 (1980), pp. 357–366.

[119] Lawrence R Rabiner. "A tutorial on hidden Markov models and selected applications in speech recognition". In: *Proceedings of the IEEE* 77.2 (1989), pp. 257–286.

[120] Steve Young et al. *The HTK Book*. HTK Version 3.4. Cambridge University Engineering Department, 2006.

[121] Simon Peyton-Jones. *Haskell 98 language and libraries : the revised report*. Cambridge: Cambridge University Press, 2003. ISBN: 978-0-521-82614-3. URL: http://www.worldcat.org/isbn/9780521826143.

[122] *Haskell.* 2016. URL: https://www.haskell.org/ (visited on 09/22/2016).

[123] Jon Bentley. "Programming Pearls: Little Languages". In: *Commun. ACM* 29.8 (Aug. 1986), pp. 711–721. ISSN: 0001-0782. DOI: 10.1145/6424.315691. URL: http://doi.acm.org/10.1145/6424.315691.

[124] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-Specific Languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344. URL: http://dblp.uni-trier.de/db/journals/csur/csur37.html#MernikHS05.

[125] Martin Fowler. *Domain-specific languages.* Upper Saddle River, N.J.: Addison-Wesley, 2011. ISBN: 978-0-13-210754-9.

[126] Paul Hudak. "Building domain-specific embedded languages". In: *ACM Computing Surveys (CSUR)* 28.4es (1996), p. 196.

[127] Paul Hudak. "Domain-specific languages". In: *Handbook of Programming Languages* 3 (1997), pp. 39–60.

[128] Paul Hudak. "Modular domain specific languages and tools". In: *Software Reuse, 1998. Proceedings. Fifth International Conference on.* IEEE. 1998, pp. 134–142.

[129] Philip Wadler. "Monads for functional programming". English. In: *Advanced Functional Programming.* Vol. 925. Lecture Notes in Computer Science. Springer, 1995, pp. 24–52. ISBN: 978-3-540-59451-2. DOI: 10.1007/3-540-59451-5_2. URL: http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf.

[130] Ralf Lämmel and Simon Peyton Jones. "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming". In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. Ed. by Zhong Shao and Peter Lee. TLDI '03. New Orleans, Louisiana, USA: ACM, 2003, pp. 26–37. ISBN: 1-58113-649-8. DOI: 10.1145/604174.604179. URL: http://doi.acm.org/10.1145/604174.604179.

[131] Ralf Lämmel and Simon Peyton Jones. "Scrap More Boilerplate: Reflection, Zips, and Generalised Casts". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ICFP '04. Snow Bird, UT, USA: ACM, 2004, pp. 244–255. ISBN: 1-58113-905-5. DOI: 10.1145/1016850.1016883. URL: http://doi.acm.org/10.1145/1016850.1016883.

[132] Ralf Lämmel and Simon Peyton Jones. "Scrap Your Boilerplate with Class: Extensible Generic Functions". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: ACM, 2005, pp. 204–215. ISBN: 1-59593-064-7. DOI: 10.1145/1086365.1086391. URL: http://doi.acm.org/10.1145/1086365.1086391.

[133] José Pedro Magalhães et al. "A Generic Deriving Mechanism for Haskell". In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, 2010, pp. 37–48. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863529. URL: http://doi.acm.org/10.1145/1863523.1863529.

[134] Mark Liberman et al. *TI 46-Word LDC93S9*. Philadelphia, 1991. URL: https://catalog.ldc.upenn.edu/docs/LDC93S9/ti46.readme.html.

[135] Paulo Cortez et al. "Internet traffic data (in bits) from a private ISP with centres in 11 European cities. The data corresponds to a transatlantic link and was collected from 06:57 hours on 7 June to 11:17 hours on 31 July 2005. Data collected at five minute intervals." In: (2014). URL: https://datamarket.com/data/set/232n/internet-traffic-data-in-bits-from-a-private-isp-with-centres-in-11-european-cities-the-data-corresponds-to-a-transatlantic-link-and-was-collected-from-0657-hours-on-7-june-to-1117-hours-on-31-july-2005-data-collected-at-five-minute-intervals#!ds=232n&display=line.

[136] Paulo Cortez et al. "Internet traffic forecasting using neural networks". In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. IEEE. 2006, pp. 2635–2642. URL: http://www3.dsi.uminho.pt/pcortez/0715.pdf.

[137] Github user "zonination". *weather-intl: Weather for 24 International Cities*. 2016. URL: https://github.com/zonination/weather-intl.

[138] Melanie Mitchell. *An introduction to genetic algorithms*. 2. repr. New Delhi: Prentice Hall of India, 2002. ISBN: 978-81-203-1358-3.

[139] Amy de Buitléir et al. "A Functional Approach to Sex: Reproduction in the Créatúr Framework". English. In: *Trends in Functional Programming: 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*. Ed. by Jurriaan Hage and

Jay McCarthy. Vol. 8843. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 68–83. ISBN: 978-3-319-14674-4. DOI: 10.1007/978-3-319-14675-1_5.

[140] Wiki contributors. *GHC.Generics.* Wiki page. 2013. URL: http://www.haskell.org/haskellwiki/GHC.Generics.

[141] F. Gray. *Pulse code communication.* US Patent 2,632,058. 1953. URL: http://www.google.com/patents/US2632058.

[142] C. Gyles and P. Boerlin. "Horizontally Transferred Genetic Elements and Their Role in Pathogenesis of Bacterial Disease". In: *Veterinary Pathology* 51.2 (2014). PMID: 24318976, pp. 328–340. DOI: 10.1177/0300985813511131. eprint: http://dx.doi.org/10.1177/0300985813511131. URL: http://dx.doi.org/10.1177/0300985813511131.

[143] Amy de Buitléir. *Créatúr GitHub.* GitHub repository. 2014. URL: https://github.com/mhwombat/creatur.

[144] Amy de Buitléir. *Créatúr Tutorial.* 2014. URL: https://github.com/mhwombat/creatur-examples/raw/master/Tutorial.pdf.

[145] Drew McDermott. "Artificial intelligence meets natural stupidity". In: *ACM SIGART Bulletin* 57 (1976), pp. 4–9.

[146] DW Van der Merwe and Andries Petrus Engelbrecht. "Data clustering using particle swarm optimization". In: *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on.* Vol. 1. IEEE. 2003, pp. 215–220.

[147] Jason Brownlee. *Clever algorithms : nature-inspired programming recipes.* [Place of publication not identified]: Lulu, 2011. ISBN: 978-1-4467-8506-5. URL: http://www.worldcat.org/search?qt=worldcat_org_all&q=9781446785065.

[148] Andrey Palyanov. *Sibernetic: Simulation of C. elegans crawling*. In a comment, Palyanov says "About 10 seconds of simulated physical time took a few days of parallel OpenCL-based calculations on a Radeon R290X GPU (see sibernetic.org for algorithm details)". 2016. URL: https://www.youtube.com/watch?v=J_wG5PfDIoU.

[149] Henry Markram. "The blue brain project". In: *Nature Reviews Neuroscience* 7.2 (2006), pp. 153–160.

[150] Brandon Rohrer. "An implemented architecture for feature creation and general reinforcement learning". In: *Workshop on Self-Programming in AGI Systems, Fourth International Conference on Artificial General Intelligence*. 2011.

[151] Akshay Vashist and Shoshana Loeb. "Attention Focusing Model for Nexting Based on Learning and Reasoning." In: *BICA*. 2010, pp. 170–174.

[152] Stuart C Shapiro et al. "Metacognition in sneps". In: *AI Magazine* 28.1 (2007), p. 17.

[153] Moshe Looks, Ben Goertzel, and Cassio Pennachin. "Novamente: An integrative architecture for general intelligence". In: *AAAI fall symposium, achieving human-level intelligence*. 2004.

[154] Dennis Decoste and Bernhard Schölkopf. "Training invariant support vector machines". In: *Machine learning* 46.1-3 (2002), pp. 161–190.

[155] Dan Claudiu Cireşan et al. "Deep, Big, Simple Neural Nets for Handwritten Digit Recognition". In: *Neural Computation* 22.12 (Dec. 2010), pp. 3207–3220. DOI: 10.1162/neco_a_00052. URL: http://dx.doi.org/10.1162/NECO_a_00052.

[156] Ronan Flynn. *Models and test scripts (email).* Dec. 2015.

[157] Philippe Le Cerf and Dirk Van Compernolle. "A new variable frame analysis method for speech recognition". In: *Signal Processing Letters, IEEE* 1.12 (1994), pp. 185–187.

[158] Avril Coghlan. *A Little Book of R For Time Series.* July 2016. URL: https://media.readthedocs.org/pdf/a-little-book-of-r-for-time-series/latest/a-little-book-of-r-for-time-series.pdf.

[159] Bernhard Pfaff and Matthieu Stigler. *Package 'vars'.* Feb. 2015. URL: https://cran.r-project.org/web/packages/vars/vars.pdf.

Appendices

Appendix A

An introduction to Haskell

Note: The information in this appendix has been taken verbatim from my MSc thesis [1].

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`.

Appendix B

A heuristic approach to configuring experiments with wains

A typical wain experiment (such as the one in Chapter 9) uses approximately 25 configuration parameters which are important to the success of the experiment. These parameters can interact in subtle ways. For example, suppose a population is not learning the task. This might be because the rewards are too low, providing little incentive for the agents to learn. If the reward is then increased and the agents start to learn, they will probably need more brain power in the form of more classifier and predictor models. If the metabolism cost is based on the number of models created, the cost may then be too high, and the agents may die before they can raise offspring.

Because of these potential interactions, a parameter cannot be optimised in isolation. Yet it is impractical to test a wide variety of configurations (combinations of parameter settings) because each experiment may take hours or days to run. For example if only five values were tested for each parameter, there would

be 5^{25} or approximately 3×10^{17} configurations to test! Therefore, one strategy is to guess some suitable parameter values (guided by earlier, similar experiments, when available), run an experiment until a problem is discovered, and then tweak the values as needed to correct the problem. The strategy outlined below was found to be useful during the research described in this thesis.

1. Choose metabolism-related parameters such that the average metabolic cost is approximately 0.1 per agent per turn. The metabolism cost will typically depend on the resources required by the agent's brain (e.g., number of classifier models) which won't be completely known until some agents master the task.

2. Choose reward-related parameters such that the maximum possible reward balances the average metabolic cost. Together with step 1, this ensures that if the population has mastered the task, agents that make more than an occasional mistake will be eliminated.

3. If there is a cost for flirting, choose it to approximately balance the birth and death rates.

4. An initial population size of 100 to 500 will typically provide sufficient diversity in the gene pool. Larger populations may yield better results, but will run more slowly. Short trial runs with a population size of 25 to 50 may be used initially to identify problems in the configuration.

5. Choose a maximum lifetime which is long enough to allow an agent to rear two or three children. (How long this takes will not be completely known until a few generations master the task.)

6. If possible, find acceptable (not necessarily optimal) values for classifier parameters by experimenting with a standalone SGM. Then define the ranges for the classifier genes to include those values, with a margin on either side to allow evolution to optimise them.

7. Set the ranges for genetic parameters wider than you think is necessary. A range that is too narrow may exclude the optimal value for that gene, in which case it will only appear in the population as a result of mutation. A range that is too wide will typically be corrected by evolution as genes that lead to unfit agents disappear from the population.

8. Since the configuration parameters can interact with each other, change only one parameter at a time.

9. If the agents do not live long enough to reproduce, check that the rewards and metabolism costs are balanced as described in steps 1 and 2.

10. If the agents stop mating, or do not mate frequently enough, try lowering the cost of flirting.

11. If agents frequently die before the first child is reared, the cost of raising a child may be too high. Try decreasing the lower limit of the devotion gene range.

12. If one generation has partial success at the task, but their children show no improvement, the agents may not be rearing their children long enough. Try raising the upper limit of the maturity gene range.