Gaining Perspective with an Evolutionary Cognitive Architecture for Intelligent Agents

A dissertation submitted in satisfaction of the requirements for the degree

Master of Science in Computer Science

David Griffin Jones

Supervised by Dr. Geoff Nitschke

Department of Computer Science, 2022

I would like to acknowledge a special thanks to my supervisor Dr. Geoff Nitschke for his insight, patience, understanding and support that has made the completion of this thesis possible.

Contents

Abstract 5

Chapter 1 Introduction 6

1.1 Problem Statement 10

1.2 Research Goals & Hypothesis 13

1.3 Contributions 15

1.4 Limitations and Scope 17

Chapter 2 Literature Review 18

2.1 Artificial Intelligence 18

2.1.1 Aspects of Intelligence 19

2.1.2 Cognitive Architectures 25

2.1.3 Learning Approaches 29

2.1.4 Intelligent Agents 32

2.1.5 Intelligent Agents in Mars Exploration 36

2.1.6 Reinforcement Learning 40

2.1.7 Summary and Recap: Artificial Intelligence 44

2.2 Metaheuristics 48

2.2.1 Evolutionary Algorithms 52

2.2.2 Evolutionary Selection Methods 54

2.2.3 Multi-Objective Evolutionary Algorithms 56

2.2.4 Hyperparameter Optimization 59

2.2.5 REVAC 61

2.2.6 Summary and Recap: Metaheuristics 62
2.3 Artificial Neural Networks 64
2.3.1 Convolutional Neural Networks 67
2.3.2 Gradient Descent 68
2.3.3 Neuroevolution 70
2.3.4 NEAT 71
2.3.5 HyperNEAT 74
2.3.6 DeepNEAT 74
2.3.7 Spiking Neural Networks 76
2.3.8 Summary and Recap: Artificial Neural Networks 80
2.4 Gaining Perspective 83
2.4.1 Hierarchical Learning 86
2.4.2 Meta Learning Shared Hierarchies 87
2.4.3 Attention 90
2.4.4 Summary and Recap: Gaining Perspective 93
2.5 General Discussion of Literature 95
Chapter 3 Methods 103
3.1 Module One: DNE 107
3.2 Module Two: REVAC 112
3.3 Module Three: MLSH 113
3.4 Module Four: Attention Unit 115
3.5 Module Five: SNNs 117
3.6 Module Six: Exploration 118
3.7 Summary and Recap: Methods 119

Chapter 4 Experiments and Results 121
4.1 Experiments 121
4.1.1 Test Functions 122
4.1.2 B-Suite Environments 126
4.1.3 Mars Rover Test Environments 131
4.1.4 Experimentation Process and Data Gathering 133
4.2 Results 139
4.3 Summary and Recap: Experiments and Results 144
Chapter 5 Discussion 146
5.1 Overview 146
5.2 Discussion and Analysis of Results 147
Chapter 6 Conclusions 161
5.1 Potential Future Research 169
References 171
Appendix 181
A: Brain Evolver Software Structure 181
A.1 Environment Interface 181
A.2 Graphics Engine 183
A.3 File System 185
A.4 Performance 185
A.5 Parameters and Settings 187
A.6 Graphical User Interface 192

Abstract

This thesis targets the boundary surrounding the creation of strong AI using AutoML *(Automatic Machine Learning)* through the development of a general cognitive architecture called *Brain Evolver*. To do this, the notion of what intelligence is in the context of machines and how it can practically be applied to physical intelligent agents is explored. Some core components that make up what a potentially strong AI system must possess are identified and outlined as *basic task completion*, *exploration*, *scalability*, *noise reduction*, *generalization*, *memory*, and *credit-assignment*. A wide set of tests that target these components are used to test the general capabilities of *Brain Evolver* as well as some more high-level tests that abstractly simulate *space rover mission* tasks. The notion of *perspective* and how it pertains to solving problems using appropriate levels of generalisation and historical information without explicitly storing all memory is also a subtle focus. *Brain Evolver* was developed using hypothetical reasoning from the literature reviewed and uses a modular design. All modules are implemented with evolutionary approaches and include *Deep Neural Evolution*, *Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters*, *Meta Learning Shared Hierarchies*, *Attention*, *Spiking Neural Networks*, and *Guided Epsilon Exploration* (a novel method). The relevance of these components in different combinations are analysed in the varying contexts of each test environment in order to gain insights and contribute to the body of evolutionary research targeted towards general problem solvers. The predictions made regarding the effect each module would have on each type of task proved to be unreliable and the program struggled with efficiency. However, *Brain Evolver* was still able to successfully and adequately solve all but one of the test environments in a completely autonomous way.

Chapter 1

Introduction

Throughout modern history, the progress of technology and development of machines has shaped and defined our society (Nadikattu, 2016). Since Charles Babbage's development of the *Analytical Engine* in the mid-1830s and Alan Turing's conception of the *Universal Computing Machine* an entire century later to the devices of today, information processing has grown to become an integral part of our lives, economy, and creation of other new technologies (Mahoney, 1988; Yudowsky, 2008; Nadikattu, 2016).

One particular fascination of many researchers is the concept of the *thinking machine*; the idea that machines can compute information in a similar capacity to a human (Oke, 2008). This idea gives rise to the field of *Artificial Intelligence* (AI). AI stems from advancements in *cognitive science* (the study of intelligent thought) with the realization that our own intelligence is actually rather computational in nature and can be broken down into definable sub-processes and patterns (Clarke and Sternberg, 1986; Cooper *et al.*, 1996; Oke, 2008; Holland and Gamez, 2009). AI therefore encompasses the area of research pertaining to the creation of machines that behave in a way that we perceive as intelligent (Minsky, 1961; Yudowsky, 2008).

Most AI models of today are highly concerned with the practicality of their applications and their ability to solve very specific problems. This makes these solutions unviable for different and varying tasks and are hence referred to as implementations of *weak AI* (Holland and Gamez, 2009; Brinton and Atm, 2017). However, the elusive goal of creating a *generalised* AI that is competitive with our

own brains in all respects has not been lost on AI researchers. This is known as *true AI* or *strong AI* and would be capable of advanced human-like reasoning with adaptive problem-solving skills (Yudowsky, 2008; Holland and Gamez, 2009; Nadikattu, 2016).

One way of approaching AI research is by copying intelligent processes found in nature. This, as well as the study of the human brain and its application to methods in AI, is called the *cognitive modelling* approach (Cooper *et al.*, 1996; Holland and Gamez, 2009). Any theoretical model pertaining to how intelligence and its various components work in relation to each other is called a *cognitive architecture*. One of the main goals of a cognitive architecture is to define a general computer model that has the ability to produce strong AI for any complex environment given to it (Cooper *et al.*, 1996; Langley, Choi and Shapiro, 2004; Laird, 2008).

A particularly significant challenge that AI solutions face is in their ability to demonstrate a high-level sense of *perspective* and generality (Long *et al.*, 2007; Holland and Gamez, 2009; Belani, Vuković and Car, 2019). The Oxford Dictionary definition of "perspective" is given as *"the ability to think about problems and decisions in a reasonable way without making them seem worse or more important than they really are"* (Perspective, 2020). This is something that humans can easily achieve by efficiently utilizing our cumulative experience when making decisions. We can generalise solutions to a problem by looking at the problem as a whole and maintaining a high-level understanding of it (Clarke and Sternberg, 1986). To truly be competitive with humans, AI solutions must adequately address this "perspective problem".

A key issue around the idea of perspective is *temporal* generality which deals with making decisions based on information that spans across time. This pertains to the concept of *memory* and how to effectively manage it (Vezhnevets *et al.*, 2017; Fujimoto, Meger and Precup, 2018; Ke *et al.*, 2018; Bhandarkar *et al.*, 2019).

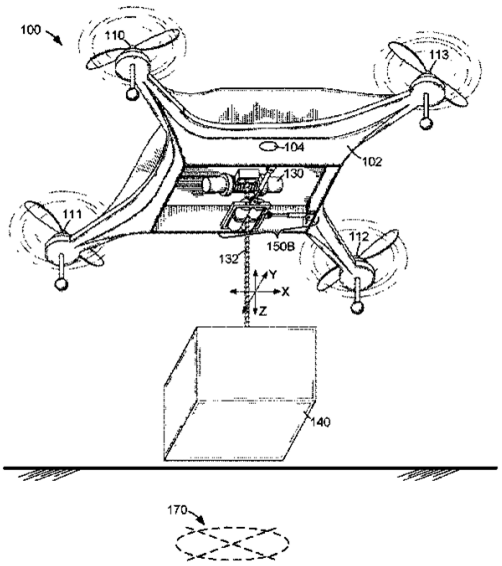


Figure 1

An early diagram of Amazon's autonomous delivery drone created for a patent that was filed in 2015 (Daniel Buchmueller, 2017).

Furthermore, it is also important to consider *what* information in memory must be used when making a decision. Recalling an entire history of events for a single decision can sometimes become infeasible. Humans address this by only remembering what is relevant for their current task (Clarke and Sternberg, 1986). This is known as the *credit-assignment* problem and is at the heart of the perspective problem. Effectively dealing with credit-assignment can become very challenging when relevant information is dispersed over temporally distant horizons and is an issue that AI research is currently grappling with (Frans *et al.*, 2018; Ke *et al.*, 2018; Bhandarkar *et al.*, 2019; Tavanaei *et al.*, 2019; Osband *et al.*, 2020).

The merging of AI and physical machines gives way to the creation of *automated agents*. Examples of some simple automated agents can be seen in the automation of industries (such as production lines) and the use of unmanned vehicles and drones as seen in *Figure 1* (Tambe *et al.*, 1995; Long *et al.*, 2007; Qiu *et al.*, 2020; Badue *et al.*, 2021). Agents that exhibit their own autonomy in an

environment are called *intelligent agents* (*see chapter 2.1.4*) and are the next step beyond remote controlled or pre-programmed agents (Tambe *et al.*, 1995; Long *et al.*, 2007).

One application of intelligent agents is in the use of *rovers* for space exploration *(see chapter 2.1.5)*. A *space rover* is a vehicle that is able to explore the surface of an extra-terrestrial body such as a moon or planet (Sandra May, 2021). Space rovers are remotely controlled and cannot be operated in real-time due to the delay of the speed of light across the vast distances of space. Effective use of autonomy can therefore significantly increase the efficiency of a mission as the rover can act without having to wait for commands. As the number of autonomous decisions increase, there is a greater demand on a rover's system to be able to handle unpredictable, dynamic, and time-dependent scenarios. Therefore, demonstrating a sense of perspective to an objective or set of objectives can drastically improve the capabilities of a rover's autonomy and progression of a mission (Maimone, Leger and Biesiadecki, 2007; Joyeux, Schwendner and Roehr, 2014).

A prominent approach to training intelligent agents is with the use of *Reinforcement Learning* (RL) *(see chapter 2.1.6)*. RL is inspired by how biological organisms naturally learn about their surroundings. It allows agents to learn how to act rationally in an environment by balancing *exploration* of what is not known and *exploitation* of what is already known. An agent perceives input from its environment which it uses to decide what action to take. The agent then receives a positive or negative feedback from its environment which facilitates its learning process. This also allows an agent to learn in an environment for itself even if we do not know much about the environment. The objective feedback function given by an environment is known as the *reward function* (Barto and Sutton, 1999; Tokic, 2010; Fujimoto, Meger and Precup, 2018).

Another approach to training intelligent agents is through the use of *Evolutionary Algorithms* (EAs). An EA is a heuristic-based algorithm inspired by the process of evolution as seen in nature (Olague, 2016). EAs are an approach to solving problems that avoid the use of *gradient-descent* based methods *(see chapter 2.3.2)*. They are therefore less susceptible to the short-comings of gradient-descent which include getting stuck in locally optimal solutions or struggling to deal with multiple objectives. On the other hand, EAs have their own potential short-comings as they are often slow, do not guarantee optimality in their results, and the reasoning behind their solutions can be difficult to analyse and understand (Branke, Kaußler and Schmeck, 2001; Ruder, 2016).

1.1 Problem Statement

As technology advances, there is an ever-growing need for intelligent agents to complete more challenging tasks in more complex environments (Schatten, 1995; Tambe *et al.*, 1995; Belani, Vuković and Car, 2019). These tasks often entail multiple objectives that must be tackled in a dynamic, time-sensitive way. A simple example of this can be seen in how a self-driving car must achieve the goal of reaching its destination while doing it in an efficient yet hazardless manner. Its environment is unpredictable and always changing and previously observed moving objects that are temporarily hidden from view must still be considered when making new decisions (Badue *et al.*, 2021).

Most approaches to solving problems like this involve trying to manually identify possible scenarios with more manageable sub-goals that are trained separately before being put together, as seen in *Figure 2* (Helman, 1986). The problem with this is that often one cannot account and train an agent for every conceivable

situation, even if it is able to be general within a sub-goal (Langley, Choi and Shapiro, 2004; Long *et al.*, 2007; Badue *et al.*, 2021). There is hence a need for a more general approach or cognitive architecture for building robust intelligent agents, regardless of their target environment. This reduces the human influence in an agent's creation and leaves a system to find general and complete solutions for itself. The result of this is to limit the potential for mistakes or development of rigid architectures (due to human design errors) and is a step towards a general AI problem solver. The process of automating tasks (which are often repetitive and time-consuming) behind the development of AI systems and their application to problems is called *Automated Machine Learning* (AutoML) (Hutter, 2014). In practice, this general approach may initially yield suboptimal results, however, the progress of general AI research may eventually lead to greater results than what were previously possible (Cooper *et al.*, 1996; Langley, Choi and Shapiro, 2004; Laird, 2008).

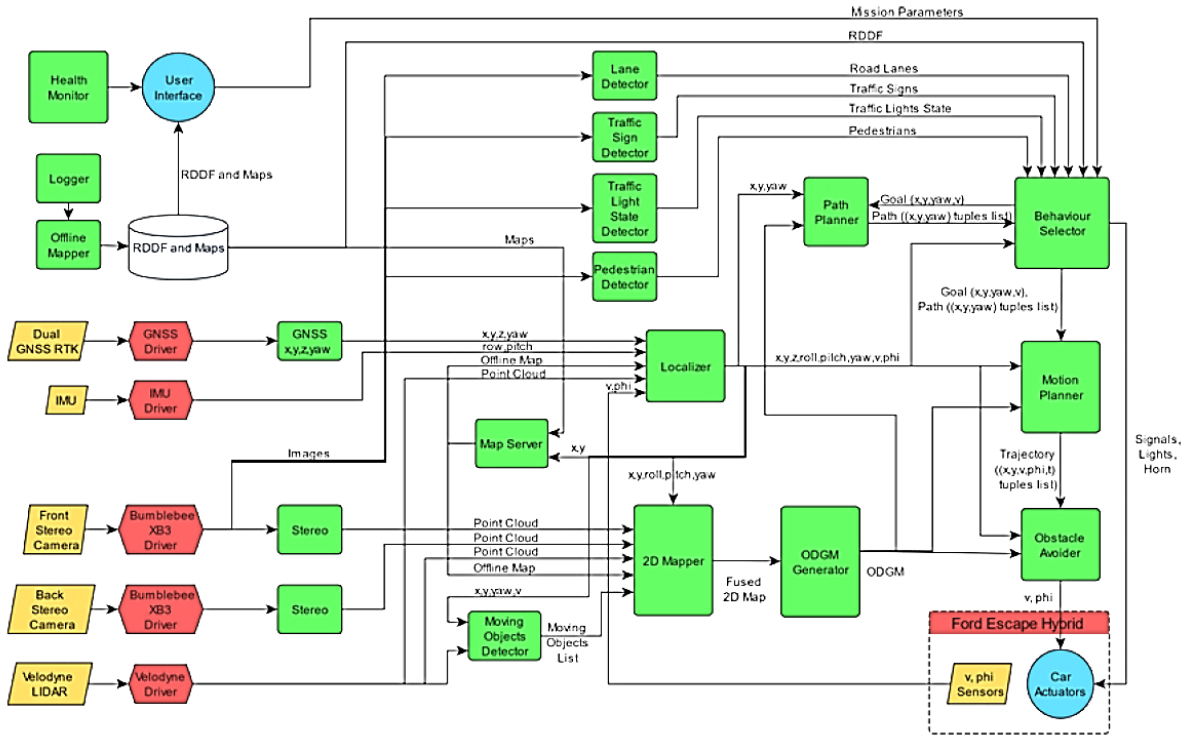


Figure 2

Diagram of how different software modules combine in the IARA self-driving car (Badue *et al.*, 2021).

Most intelligent agent systems struggle at approaching tasks with perspective. In this context, perspective abstractly refers to the challenge of temporal generality, credit-assignment, and adaptability to environmental changes (Fujimoto, Meger and Precup, 2018; Ke *et al.*, 2018; Osband *et al.*, 2020). Furthermore, manually dividing an agent's objective (or objectives) into sub-goals may be suboptimal as better results could potentially be achieved by letting a system learn to decide sub-goals for itself through intelligently balancing exploration and exploitation of its environment (Tokic, 2010; Frans *et al.*, 2018; Gupta *et al.*, 2018).

Another potential limitation of many intelligent agent systems is that they are often trained using *gradient-descent* based methods *(see chapter 2.3.2)*. While gradient descent methods can sometimes be fast and accurate for certain problems, they often guide towards a single solution and struggle to account for

adaptability and intelligent exploration (Ruder, 2016). An alternative approach to learning that does not use gradient descent is to use EAs (Branke, Kaußler and Schmeck, 2001).

Ultimately, the problem identified by this thesis is that there is a broad-based lack of research targeted towards general problem solvers and general AI. Furthermore, most current insights and advancements are quite isolated to their current goal of research. This is not a problem in itself, however, little research targets general AI from a wider perspective or the analysis of multiple topical ideas and their interaction under one system.

1.2 Research Goals & Hypothesis

The main goal of this research is to use evolutionary methods with RL to develop a cognitive architecture that provides a general model for the creation of intelligent agents that can behave sufficiently well in any potential environment. The architecture is presented in the form of a program called *Brain Evolver* (BE) which is structured in a modular way so that the user has full control over all of its parameters. The intention is to have a single system that can be used by other people to create intelligent agent models regardless of their target environment. It is hence important that BE is built so that it can be used as tool for further research

The purpose of BE is to take a step towards general intelligence, or strong, true AI. In order to do this, the first sub-goal of this thesis is to explore the idea of what it means to be intelligent from a cognitive modelling perspective. The aim

is to break the idea of intelligence down into identifiable components so that they can be directly addressed by BE's architecture.

One component of intelligence that many current intelligent agents struggle with is generality across time and the credit-assignment problem as encapsulated by the perspective problem (Ke *et al.*, 2018; Osband *et al.*, 2020). A subtle focus and additional sub-goal of this thesis and the implementation of BE is hence to explore solutions to these memory and recall-based challenges without explicitly storing every piece of information ever received.

In order to effectively test BE's ability to create intelligent agents that are able to perform sufficiently well in different types of environments, the program includes a set of focused tests that target specific types of general problems (Osband *et al.*, 2020). Beyond this, tests are conducted on environments that abstractly represent the kinds of problems an agent would confront in a real-world scenario. In this case, the use of intelligent agents in space-rover exploration is explored by constructing environments that cover some of the basic tasks that these agents may encounter.

BE's cognitive architecture is built from the following components: *Deep Neuroevolution* (DNE), *Attention Matrices*, *Meta Learning Shared Hierarchies* (MLSH), *Spiking Neural Networks* (SNNs) and *Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters* (REVAC). All components are implemented using evolutionary methods. The details regarding the implementation of these components, what they are, and the reasons behind their selection are discussed throughout the rest of this thesis.

The hypothesis proposed for this thesis is that, using evolutionary RL, the culmination of all the aforementioned components should be sufficient as a cognitive architecture for BE to produce intelligent agents that can behave sufficiently well in any type of environment. The focus is hence on generality

rather than performance. A subsequent goal of this hypothesis is to then find out if all the components are actually necessary and how much of an impact using them in different combinations has on the performance of the agents created for different environments.

1.3 Contributions

The main contribution of this thesis is the creation of a cognitive architecture (BE) that is focused on the general development of intelligent agents. BE makes use of evolutionary approaches and its contribution is hence also extended to include the further study and analysis of evolutionary methods as a means to address the elusive goal of a general AI problem solver.

BE's effectiveness at training robust intelligent agents with minimal human intervention (AutoML) is demonstrated by evaluating its ability to create intelligent agents on a wide set of disparate environments. These test environments broadly target different components of intelligence. The analyses of these components and the AI methods used to address them are hence also contributions of this thesis. The AI methods used (DNE, Attention, MLSH, SNNs and REVAC) are all implemented using evolutionary approaches. The individual analyses of applying evolution to each of the methods are also valid contributions, particularly as there is little research on the use of Attention, MLSH or SNNs using evolutionary methods.

The decision to use these particular AI methods as sub-components of BE's architecture was not definitive and there were many other approaches that could have been used instead or in addition. However, whether optimal or not, a model

for BE was settled upon based on hypothetical reasoning. It can therefore be said that the exploration of BE's architecture in its current state and the success or failure of its components only serves to assist other future studies in the same field.

An example of a real-world task environment that would benefit from a generally intelligent solution with an improved sense of perspective is in the use of space-rover missions (Bresina *et al.*, 2003; Bresina and Morris, 2007). This thesis hence continues to explore the potential advantages of using BE in such an environment by running further tests on a set of simulated tasks that abstractly represent what a space rover may encounter.

The intention of BE is *not* to create intelligent agents that are able to achieve extremely accurate, cutting-edge results in highly complex, large-scale environments. Rather, the tests run on BE serve as small-scale proofs of concepts and an opportunity to further study what it means to be generally intelligent. For this reason, BE and its components are analysed and compared using different internal configurations (rather than being compared to other state-of-the-art methods). The contribution of this is an evaluation of how useful each component's model is (individually or in conjunction with other components) in the context of creating generally intelligent agents.

1.4 Limitations and Scope

The scope of this thesis is quite broad as it explores the expansive topic of creating a general AI problem solver. The intelligent agents created by BE are meant to represent how real robotic agents may operate in real environments,

however, due to physical, financial, and time limitations, no tests will be done with actual robotic agents. The tests conducted only involve simulated agents and target certain types of sub-tasks that agents might be required to do in a real-world situation.

Due to scope constraints, BE will not be tested in comparison to other potentially more common or state-of-the-art methods. As already mentioned, the intention of BE is not to produce better results than other implementations but rather to develop a cognitive architecture in the pursuit of studying general intelligence. The testing environments for BE are highly focused and simplified since they only test specific criteria. Furthermore, the model complexities of the agents created are also scaled to be very small. This means that environments only operate for a short number of time-steps in a limited virtual space and the size of the *Neural Networks (see chapter 2.2)* evolved remains small. This keeps processing times down as there are a large number of tests.

The tests conducted for this thesis are implemented on 15 different environments that have been chosen to target specific kinds of fundamental tasks that reflect different aspects of generally intelligent behaviour. A further four environments are also used to test certain kinds of tasks an intelligent agent operating as Mars rover may encounter for a total of 19 test environments. Each set of tests is run using different configurations of BE's internal architecture in order to isolate the efficacy of each of the individual components and their ability to work in conjunction with other components. The details regarding the testing environments and the tests themselves are outlined in *chapter 4.1.*

Chapter 2

Literature Review

2.1 Artificial Intelligence

Artificial Intelligence (AI) is the broad study of machines or computer systems that are able to perform tasks and solve problems in a way that seems intelligent by our own human standards (Minsky, 1961; Oke, 2008; Holland and Gamez, 2009). The term was coined by John McCarthy in 1955 and has since grown to become one of the most highlighted fields of research in the 21st century (McCarthy *et al.*, 2006; Nadikattu, 2016). Research into AI can be divided into many subfields. These include the theoretical and mathematical concepts behind AI, practical uses and social implications of AI, and philosophical questions that AI raises (Minsky, 1961; Helman, 1986; Armstrong, 2004; Yudowsky, 2008; Nadikattu, 2016).

In today's age, the practical realization of AI predominantly manifests as *Machine Learning* (ML). ML is a subcategory of AI that focuses on the creation of systems that learn to improve their behaviour over time at doing a certain task (with access to relevant training data) while still being able to generalize about the problem at hand. Since ML is targeted at doing specific tasks, it is classified as weak AI. It is hence more useful to refer to general intelligence as AI instead of ML (Donald Michie, 1968; Stevens and Soller, 2005; Holland and Gamez, 2009; Mnih *et al.*, 2013; Brinton and Atm, 2017).

2.1.1 Aspects of Intelligence

One particular goal of many researchers is to develop true, general AI. However, as society's confidence in what computers are capable of increases, our standard for what is considered to be true AI shifts. When *Deep Blue* beat the World Chess Champion of 1997 *Garry Kasparov*, many people considered what had been achieved to be true AI until the algorithm employed was better understood and the standard for true AI was pushed back (Yudowsky, 2008; Nadikattu, 2016; Hassabis, 2017). Another common metric for true AI is the *Turing Test* for which an AI passes if a human is unable to distinguish an interaction with the AI from a real human. A further extension of the Turing Test sometimes includes the goal of creating AI that is physically indistinguishable from humans. However, most research is not concerned with the anthropomorphic development of machines but rather AI's practical uses through the development of ML (Long *et al.*, 2007; Oke, 2008; Holland and Gamez, 2009).

With today's advancements in AI research, we recognize that there are many aspects to strong, true AI beyond being good at a board game or fooling a human with convincing natural language processing. Intelligence can manifest in a multitude of different ways that are sometimes not obvious to us as our own understanding of what intelligence is comes from a human perspective (Minsky, 1961; Holland and Gamez, 2009; Mnih *et al.*, 2013). It is hence useful to explore what it actually means to be generally intelligent. This allows certain goals to be framed that can be used to define a general cognitive architecture *(see chapter 2.1.2)* or AI solutions able to solve problems that are useful to us (Cooper *et al.*, 1996).

The question of what makes something truly intelligent or even what the notion of intelligence actually is has challenged humans ever since ancient Greek philosophers such as *Plato* tried to reason about our own existence in life

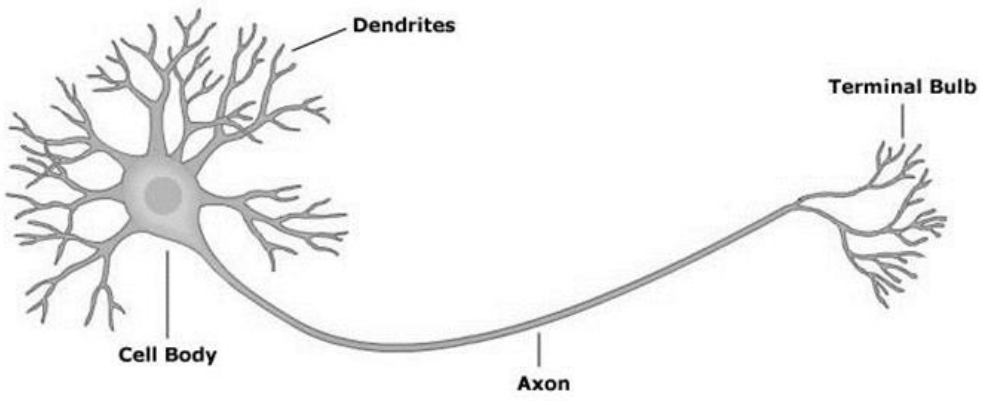


Figure 3

A diagram of a neural connection found in the brain (Taylor, 2017) *which inspired the design of Artificial Neural Networks (see chapter 2.3)*

(Armstrong, 2004). If a machine can solve a given problem efficiently but it is only following a set algorithm, is it really intelligent? Even the concept of consciousness and its potential relevance to AI is an important question that needs to be asked in order to begin to tackle the elusive idea of general intelligence. It may be that the emergence of consciousness or self-awareness is imperative to perform extremely advanced tasks as it implies introspection and internal evaluation (Oke, 2008; Omohundro, 2008; Holland and Gamez, 2009). However, since the concept of consciousness is one that is so philosophically controversial, its practical use in our current understanding of general AI and the research of this thesis is not particularly relevant.

The best example of true intelligence that we know of is exhibited by human beings, so naturally AI research often aims to copy the human mind through the cognitive modelling approach (Clarke and Sternberg, 1986; Cooper *et al.*, 1996; Yudowsky, 2008). Beyond simply looking at our brains, many AI approaches use other processes that occur in nature as their inspiration. Two examples of this are in the development of RL methods as well as many *metaheuristics (see chapter*

2.2) such as EAs. Another example can be seen in the attempt to replicate the systems in our own brains *(see chapter 2.3)* as seen in *Figure 3* (Taylor, 2017).

A further important question centres around the notion of rationality. The goal of many AI systems implemented today is to always act rationally. Acting rationally means that the best output is always given in every scenario to maximise an AI's solution to its task at hand. However, the idea of rationality implies that there exists some over-arching, objectively optimal goal (Omohundro, 2008; Gupta *et al.*, 2018). Intelligent life in nature does not conform to this idea as even the emergence of life itself can be argued to be arbitrary and without goal or purpose. Hence, many biologically intelligent organisms often do things that seem irrational. It is this irrational behaviour that is in fact very important to the learning process as it allows for the *exploration* of the *problem space (see chapter 2.1.3)* of an organism's environment (Minsky, 1961; Stevens and Soller, 2005; Omohundro, 2008; Tokic, 2010; Olague, 2016). Irrationality can be represented as stochasticity, which implies randomly deviating from what is currently believed to be optimal with some probability in order to find new possible solutions or goals (Gupta *et al.*, 2018).

Exploration of one's environment can be represented in humans as *curiosity* (Minsky, 1961). There is no obvious reason why we would be interested in learning about things like mathematics from an evolutionary perspective, however, in the long term our curiosity has driven the development of modern society and hence put us at the top of the food chain. Curiosity also allows us to learn through experience. By trying different actions, we determine what is dangerous and what is useful. This is how RL learning shapes our behaviour *(see chapter 2.1.6)* (Turney, Whitley and Anderson, 1996; McLeod, 2007; Jabri *et al.*, 2019). However, there are other aspects to human knowledge that we know intuitively. Among many other things, some examples of this include breathing, responding to pain or hunger, an innate fear of things like fire and other creatures

and a natural propensity to believe what our parents say as children. These skills are useful as if we had to learn that predators are dangerous by first experiencing being attacked, we would rarely live to tell the tale. This common theme in biologically intelligent creatures can be referred to as *instinct* (Turney, Whitley and Anderson, 1996; Omohundro, 2008).

While the concept of intelligence can be seen as being quite abstract or informally defined, the attempt of cognitive science to break intelligence down into definable parts helps guide research into general AI (Cooper *et al.*, 1996; Langley, Choi and Shapiro, 2004). It is evident from the distinction between the concepts of curiosity and instinct that there is an allusion to two separate aspects of intelligence; one being the ability to learn new things through exploration and exploitation using RL, and the other being the development of set and pre-trained knowledge which comes about in nature through the *evolution (see chapter 2.2)* of cognitive behaviours. The emergent intelligent behaviours that these learning methods naturally produce in humans can be further broken down into a set of core components. This is not to say that these components are supposed to be a universally perfect summation of what it means to be intelligent, but rather that they signify useful goals that a general cognitive architecture may need to address.

One theory of intelligence is the *Triarchic Theory of Intelligence* which was proposed by *Sternberg* in 1984 and divides general intelligence into analytical, creative, and practical intelligence. *Sternberg* defined intelligence as the *"mental activity directed toward purposive adaptation to, selection, and shaping of real-world environments relevant to one's life"* (Clarke and Sternberg, 1986). On the other hand, *Thurstone* breaks intelligence into memory, numerical ability, perceptual speed, reasoning, spatial visualization, and verbal comprehension (Thurstone, 1962). While these analyses are useful to some degree, they are more applicable to biological entities and less applicable to practical implementations

of AI. It is hard to identify achievable goals in *Sternberg's* analysis and *Thurstone* uses many metrics that are trivial for digital machines such as numerical ability and perceptual speed. Furthermore, neither analysis considers the ability to be general.

In order to bridge the gap between biological, human intelligence and practical, general AI, *Minsky* identifies five aspects of intelligence. These include the ability to search for possible solutions, learn over time, recognize patterns, plan, and use inductive reasoning (Minsky, 1961). The concept of intelligent agents is also useful when dealing with the idea of intelligent machines *(see chapter 2.1.4)*. An intelligent agent allows for an AI to express itself in an environment by receiving input before acting in the environment using some output (Tambe *et al.*, 1995; Langley, Choi and Shapiro, 2004; Long *et al.*, 2007). *Osband* offers a more detailed analysis of intelligence in machines by looking at how an AI performs through the use of intelligent agents. The analysis targets definable metrics that can be used for specifically evaluating AI in intelligent agents before going on to outline seven core components of intelligence that an effective AI must be capable of (Osband *et al.*, 2020). The details of these core components are given below.

- **Basic Task Completion:** The ability to solve any type of basic task given valid and accurate input (Osband *et al.*, 2020).

- **Exploration:** The ability to intelligently try different unknown solutions to potentially find better solutions. This entails being able to weigh up the possible benefits of exploring (with the possibility of not finding anything of value) against exploiting what is currently known (Gupta *et al.*, 2018; Osband *et al.*, 2020).

- **Scalability:** Scalability is the idea that as the complexity of a task or size of an environment increases, an AI's performance should remain

sufficiently accurate compared to when it learned to solve the task on a small scale (Lozano, Molina and Herrera, 2011; Osband *et al.*, 2020).

- **Noise Reduction:** Noise reduction is the ability to deal with random, superfluous, and inaccurate input data by filtering out what is not needed and using what is relevant to the problem at hand. This is especially important when it comes to real-world environments where information is rarely precise or clean (Jakobi, Husbands and Harvey, 1995; Osband *et al.*, 2020).

- **Generalization:** A significant factor that differentiates AI from other solutions is that AI is able to generalise. This means that AI can learn rules that govern an environment or problem and can act rationally given input from the same problem even if it has never explicitly seen or been programmed to deal with that exact input configuration (Minsky, 1961; Osband *et al.*, 2020).

- **Memory:** Memory is the ability to effectively store and recall past actions and events. Memory can become difficult when dealing with long periods of time or large amounts of information as knowing what is important to remember and what can be forgotten is challenging. A further aspect of memory that can be explored is *temporal generality* and the ability to abstractly represent events in memory rather than storing every single detail (Cooper *et al.*, 1996; Osband *et al.*, 2020).

- **Credit-Assignment:** Credit-assignment is the ability to attribute importance to certain inputs or actions across time. It is the ability to understand what specific actions in the past led to a current positive or negative state (Ke *et al.*, 2018; Osband *et al.*, 2020).

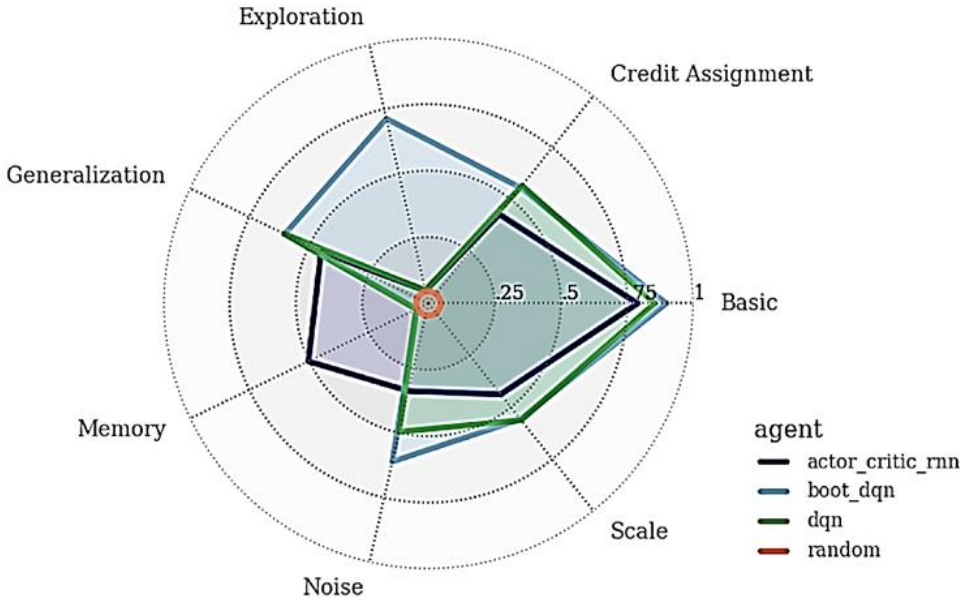


Figure 4

Diagram of the performance of different intelligent agent methods in the seven core capabilities of intelligence (Osband *et al.*, 2020).

2.1.2 Cognitive Architectures

The purpose of defining what makes something intelligent and what it means in the context of machines is relevant in developing a systematic, general approach to building AI systems. The construction of a unified theory of cognition or structure of intelligence is referred to as a cognitive architecture (Cooper *et al.*, 1996; Langley, Choi and Shapiro, 2004). The benefit of a cognitive architecture is in its ability to provide a general framework for intelligence so that useful AI solutions can be created regardless of the target environment or problem. This removes the need for humans to have to redesign separate AI approaches to

individual problems (AutoML). Furthermore, a reliable cognitive architecture also mitigates the potential for mistakes introduced by independent designs.

Practically, the implementation of a particular cognitive architecture to software is informal as the process of coding requires assumptions about the expression of certain cognitive processes. These assumptions are not robust in the same way a mathematical proof is and any attempt to build a cognitive architecture is a hypothetical claim to its effectiveness that must be tested and evaluated (Cooper *et al.*, 1996). Hence, a cognitive architecture in the context of computer science is not only a theory about cognition but also a theory about how cognition may be ported to machines.

Michie offers a simplistic but fundamental look at how intelligence might be expressed in the context of machines and introduces the idea of *memo functions*. Although he never refers to his idea as a cognitive architecture, its general principle around how AI works aligns with the basic principles of many AI solutions today. The idea is based around the simple ability to learn through experience as in the definition of ML. If a machine can do this, then it is considered intelligent. This is extended to categorise all decisions as having a rule processing part and a generalized look-up stack. Experiences that occur often take priority in the stack and things that hardly happen are eventually forgotten. Every time input is received, the stack is searched and if nothing is found, a new experience is stored at the top by receiving feedback from processing the input using currently known rules (Donald Michie, 1968). Although this analysis may seem over-simplified, it identifies a few main points that define a machine as being intelligent. These include the ability to learn through trial and error (exploration), utilize (exploit) memories of generalized experiences, and prioritise important memories.

One of the most popular long-term cognitive architecture projects that is still being developed today is called *Soar* and was first created in 1983 by *John Laird*.

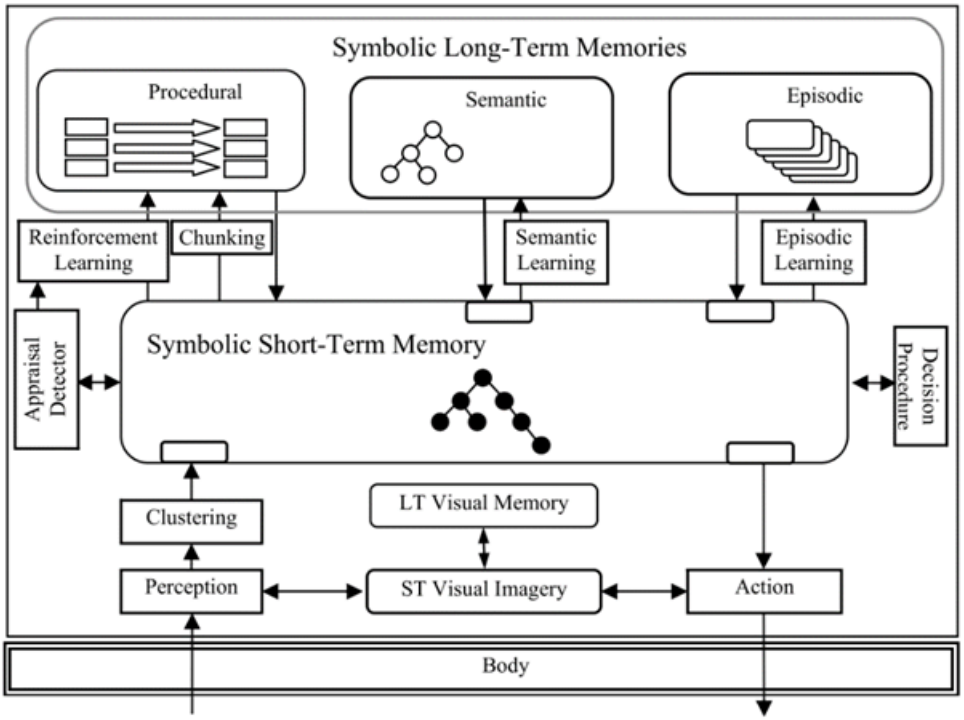


Figure 5

Diagram representing the general structure of the Soar cognitive architecture (Laird, 2008).

The goal of Soar is to develop a structured system of all the computational components that make up any generally intelligent agent. This is essentially what this thesis is trying to achieve so it is useful to look at Soar's own architecture (Laird, 2008).

Soar's over-arching structure works by utilizing what is currently known in order to define goals for itself so that a global goal can eventually be obtained. At each step of the way, if Soar does not have sufficient knowledge or experience to find a solution to reach its current goal, it recursively creates hierarchies of sub-goals using any information available until it is able to proceed (Cooper *et al.*, 1996). The process of breaking down a global task into sub-goals is known as *problem decomposition* (Helman, 1986). Soar's approach to problem-solving is based on the assumption of the *Problem Space Hypothesis (see chapter 2.1.3)* which asserts that any search for a solution to a goal can be expressed as a search through a set

of possible states known as the *problem space*. This is a fundamental idea that underpins most modern AI and ML research (Newell, 1990).

Soar differentiates between procedural, working, semantic and episodic memory and maintains the idea that knowledge is stored through a structured representation of symbols. Procedural memory is knowledge about *how* to do things and working memory is contextual information of recent events in an environment. Semantic memory is a very long-term structure of essentially fact-like knowledge while episodic memory keeps track of working memory in a time-related stream. It then uses the combination of these four sources of information when making decisions about how to act within a goal or whether to create a new goal (Cooper *et al.*, 1996; Laird, 2008). This system of knowledge is then built up using RL *(see chapter 2.1.6)* which is Soar's primary method of learning (Laird, 2008). An overview of Soar's architecture can be seen in *Figure 5*.

Another prominent modern cognitive architecture is called ACT-R (Adaptive Control of Thought - Rational). Unlike Soar which is more concerned with how to apply AI to intelligent agents (machines), ACT-R is more concerned with directly replicating the cognitive functions of the human brain. The goal of ACT-R is to reduce human cognition down to irreducible functions (Scerri, 2006). ACT-R identifies knowledge as either being declarative (explicit) or procedural (implicit). Declarative knowledge is conscious long-term memory while procedural knowledge is unconscious long-term memory. Like Soar, ACT-R also proposes that knowledge is purely symbolic (Squire and Dede, 2015). This is in opposition to another cognitive architecture called CLARION (Connectionist Learning with Adaptive Rule Induction On-line) which proposes that knowledge is emergently represented through connections between nodes of structured information (Cooper *et al.*, 1996; Scerri, 2006; Oke, 2008). An example of a connectionist approach can be seen in the development of *Artificial Neural Networks (see chapter 2.3)* (Agatonovic-Kustrin and Beresford, 2000).

2.1.3 Learning Approaches

It is important to note that there is a difference between the resultant behaviours of certain cognitive components that make an entity intelligent and the processes that led to the acquisition of the entity's knowledge (through learning). Furthermore, it is evident that there is a disparity between approaching knowledge and intelligent behaviour from a biological point of view and the practical application of these concepts to machines. It is therefore necessary to define certain broadly categorical ways that machines can go about developing knowledge.

The practical objective of AI is to find or learn the optimal solution to a given problem or task. The ability of a machine to learn over time (as in the definition of ML) is what separates AI from hard-coded solutions. There are four main categories of approaches to learning. These approaches are defined as being *Supervised*, *Unsupervised*, *Semi-Supervised* or RL based (Stevens and Soller, 2005; Jabri *et al.*, 2019).

Supervised learning relies on training an AI with the use of examples in the form of labelled data while unsupervised learning is when a machine is trained using unlabelled data. Unsupervised learning is useful when humans do not know much about the problem or what to look for in the data, whereas supervised learning requires humans to understand the learning environment and hence, in some way, take on the role of the AI's teacher. Semi-supervised learning is simply when only some of the data is labelled (Jabri *et al.*, 2019). The details regarding the last learning approach listed (RL) can be seen in *chapter 2.1.6*.

As already defined, AI learns by striving to find the optimal set of outputs to give for any set of inputs in an environment. In order to better understand this, we expand on the *Problem Space Hypothesis* outlined in *chapter 2.1.2*. The relation between an AI's inputs and outputs is simply some function that can be described

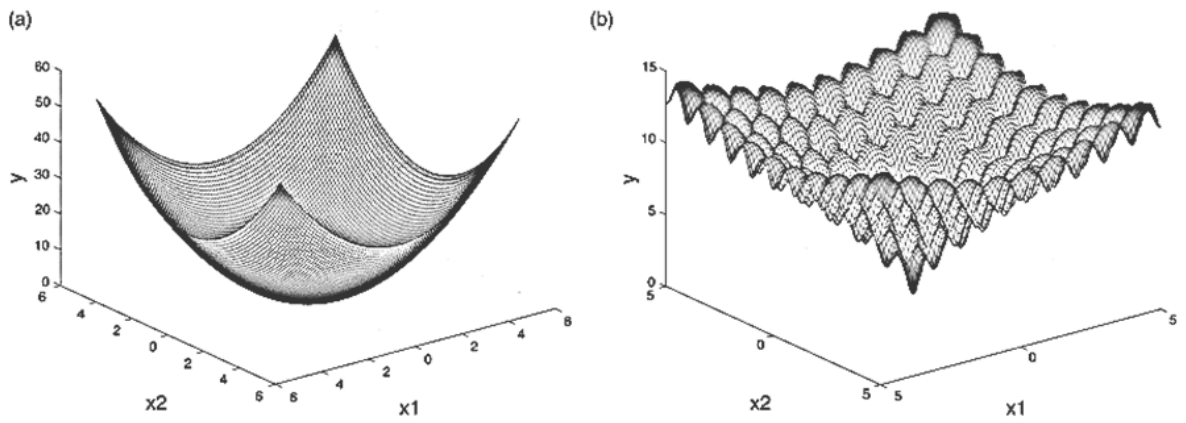


Figure 6

The problem space surfaces of two common test functions; the Sphere Function (left) and the Ackley Function (right) (Mirfenderesgi and Mousavi, 2015) *which were also used in the testing phase of this thesis (see chapter 3.8.1).*

mathematically. The learning process can therefore be seen as simply being function optimization. Considering this, it is useful to think of a task or problem as an *x*-dimensional surface, known as the *problem space* (Goldberg and Holland, 1988; Scerri, 2006; Brinton and Atm, 2017). If the number of input dimensions for a given environment is *x*, and the number of output dimensions is *y*, then for every combination of inputs and outputs there is some level of behavioural optimality *z* which represents the surface of the problem space. The dimensionality of a single-objective problem space is hence $x + y + 1$ (Brinton and Atm, 2017).

Finding a function that maps inputs *x* to outputs *y* such that *z* has a maximal or minimal (depending on the problem) value out of all possible *z* values for *x* is called finding a *globally optimal* solution. If a function maps *x* to *y* with an optimal *z* value for all *y* values in the vicinity of *z*'s location in the problem space, then we have found a *locally optimal* solution as there may still be a greater *z* value for the set of inputs *x* elsewhere in the problem space (Dietterich, 2000; Mishra, 2011; Gobeyn *et al.*, 2019).

Each combination of inputs can be thought of as a state. If the set of possible input states are discrete and not too extensive, then finding the optimal function may be trivial. However, this is not the case for sets that have a high dimensionality with complex, continuous problem spaces (Goldberg and Holland, 1988). This issue is known as the *curse of dimensionality* which essentially says that as the number of dimensions in the problem space increases, the volume increases so exponentially fast that the availability of data to fill it becomes sparse, requiring ever higher levels of generalisation to act optimally with unseen input (Poggio *et al.*, 2017).

One response to solving problem spaces with high dimensionality is to use what is called *Deep Learning* (DL). DL is an approach to AI that entails creating models that are capable of learning multiple levels of hierarchies and structures in data. This approach may also assist in the navigation of an unlabelled problem space (unsupervised learning). Due to the nature of real-world environments, practical applications of AI often require DL approaches in order to perform sufficiently well (Minsky, 1961; Poggio *et al.*, 2017; Tavanaei *et al.*, 2019; Gupta, 2020).

An important concept when dealing with DL is the idea of a *latent space*. The word "latent" means "hidden" and a latent space can be thought of as an abstract representation of the characteristics of some data (Tiu, 2020). Latent spaces have the ability to show generality with regards to some information by reducing the dimensionality of the information or compressing it in some way (Shen *et al.*, 2019). An analogy of how latent spaces store information can be shown in the way humans express knowledge. If we describe the appearance of another person, we use abstract terms like "height" or "hair colour" to convey meaning. Someone else can then use this information to reconstruct a mental image of what this person may look like in their mind. Latent spaces work similarly as they are able to hold information about the relevant features of some data. These features can then be used in what is called *feature extraction* to generalize about what the important

aspects of a problem are as well as improve processing times due to the dimensional reduction or compression of data (Pan, Kwok and Yang, 2008; Shen *et al.*, 2019; Tiu, 2020).

2.1.4 Intelligent Agents

Since we have defined some ways that knowledge and intelligent behaviour can be represented and obtained, it now becomes useful to have some paradigm to express these concepts. The acquisition of AI is useful if it can be demonstrated by a machine, but a concept in AI has little practical use if it remains just a collection of theoretical and mathematical ideas.

This makes way for the introduction of *intelligent agents.* Intelligent agent*s* are systems that are capable of autonomous action in an environment. Agents are said to be autonomous if their behaviour is dictated by their own experience. Intelligent agents possess the ability to learn and adapt and are entities that allow an AI to be implemented in an environment (Tambe *et al.*, 1995; Bansall, 2019).

The effective implementation of intelligent agents has the potential to drastically improve many aspects of modern society as well as assist in further scientific advancements (Yudowsky, 2008; Nadikattu, 2016). There is a continuing demand for intelligent agents to tackle increasingly challenging tasks (Belani, Vuković and Car, 2019). Some examples of uses of intelligent agents in real-world environments that are becoming more prominent include (but are not limited to) the following list.

Figure 7

Robotic agent for firefighting and clearing debris (Matrossov *et al.*, 1992).

- *Space rover missions* (Bresina *et al.*, 2003; Bresina and Morris, 2007; Maimone, Leger and Biesiadecki, 2007; Yliniemi, Agogino and Tumer, 2014; Boukas *et al.*, 2017)
- *Self-driving cars* (Badue *et al.*, 2021)
- *Automated medical procedures and operations* (Schatten, 1995; Agatonovic-Kustrin and Beresford, 2000)
- *Unmanned military drones* (Long *et al.*, 2007; Qiu *et al.*, 2020)
- *Automated farming* (Bryndin, 2020)
- *Exploration drones* (Boukas *et al.*, 2017)
- *Automated completion of hazardous tasks* (Matrossov *et al.*, 1992)
- *Automated delivery drones* (Daniel Buchmueller, 2017)
- *Manufacturing processes* (Tambe *et al.*, 1995)
- *Entertainment* (Tambe *et al.*, 1995)
- *Education and training* (Tambe *et al.*, 1995)
- *Automated maintenance of infrastructure* (Joyeux, Schwendner and Roehr, 2014)
- *Pollution control* (Oke, 2008)

Agents perceive their environment by gathering input data through the use of *sensors* and perform actions with the use of *actuators* such as a robotic arm or wheels on a car (Daniel Buchmueller, 2017; Badue *et al.*, 2021). Agents that operate in physical environments through the use of robotics are always bound by the dimension of time and hence must consider actions that may have long-term, delayed rewards (Barto and Sutton, 1999; Gupta *et al.*, 2018; Ohnishi *et al.*, 2019). Other potential requirements of intelligent agents that operate in real-world environments involve the ability to adapt to change and the ability to create and pursue goals (Cooper *et al.*, 1996). An agent must therefore learn to map functions from its histories of perceptions to actions and potential rewards (Barto and Sutton, 1999; Scerri, 2006; Mnih *et al.*, 2013). It is also important to note that agents need not only be applied to robotics as many problems have environments that can be simulated or may only exist digitally or conceptually (Schatten, 1995; Tambe *et al.*, 1995).

Agents need to know information about their environment and there are two ways that this can happen. Agents are either able to observe (know) everything about their environment at once (known as *fully observable environments*) or they are only able to partially observe their environment. In *partially observable environments*, agents interact with their surroundings and keep track of what they have perceived and what has happened (Oke, 2008; Ohnishi *et al.*, 2019). This updates an internal model (known as a *belief state*) of what the agent thinks about their environment (Badue *et al.*, 2021). Agents also have a set of variables representing the agent's own *internal state*. By considering an agent's current belief state and internal state, the agent can update its actions or current goal based on what needs to be achieved (Ke *et al.*, 2018).

Some simple approaches to agents include *Finite State Machines* and *Bayesian Networks*. Finite state machines simply map every state an agent is in with regards to its environment to a predefined action. Bayesian Networks work

similarly but operate on probabilities that certain states will occur given certain actions which then directs the agent to choose the action that yields the highest probable return (Long *et al.*, 2007; Oke, 2008; Badue *et al.*, 2021). The state an agent is in is simply the combination of internal and environmental variables that exist at a particular time.

Agents can be categorized based on their perceived level of intelligence or capabilities. Agents that are able to learn from their past experiences and hence act more optimally over time are known as *learning agents* (Belani, Vuković and Car, 2019). More information relating to learning agents can be seen in *chapter 2.1.6*, however, further categorizing agents yields the following four general groups listed below.

- **Simple Reflex Agents:** Agents only act based on their current input (Belani, Vuković and Car, 2019). Many agents are reflex agents but some learning agents also have the ability to obtain memory through experience.

- **Model-Based Agents:** Agents act by finding a rule to follow that matches the conditions of the current situation. These agents keep track of an internal state-based map of their perception history (Belani, Vuković and Car, 2019).

- **Goal-Based Agents:** Agents that are able to make decisions based on their current desires and goals. This makes the agent more flexible as its desires and goals can change based on what is needed (Belani, Vuković and Car, 2019).

- **Utility-Based Agents:** Sometimes taking the quickest route to a goal may not be the best choice as there may be other factors that affect an agent's internal state. When there are multiple possible alternative actions that

may still lead to the desired goal, a utility agent chooses actions that take these factors into consideration (Belani, Vuković and Car, 2019).

2.1.5 Intelligent Agents in Mars Exploration

Since the goal of this thesis is to produce a general cognitive architecture for the development of intelligent agents, it seems prudent to demonstrate any agents created on at least one practical example of an abstracted real-world scenario, which in this case is use of intelligent agents in space rovers for Mars exploration. One of the current forefronts of human technological advancement is space exploration and an important aspect of space exploration includes the implementation of intelligent agents through the use of space rovers. Space rovers are remotely operated machines that enable the exploration of other worlds and are necessary due to the fact that humans cannot feasibly accompany most space missions, particularly distant ones (Sandra May, 2021). It is inefficient to directly control a rover due to the time delays in the speed of light over space, therefore, delegating tasks and certain decision-making responsibilities to a rover itself can drastically improve the likelihood of a mission's success (Yliniemi, Agogino and Tumer, 2014). This means that rovers must operate as intelligent agents with varying levels of autonomy. Additionally, following the rise of companies such as *SpaceX*, there has been a recent growing interest in the exploration of our neighbouring planet Mars and, as a consequence, research into intelligent agents as Mars rovers (Zheng, 2020).

In the case of Mars rover missions, it takes a signal up to twenty minutes to reach Mars one-way. Furthermore, humans are usually only able to interact with a rover when it is on the side of the planet facing Earth. This means that for half a Mars

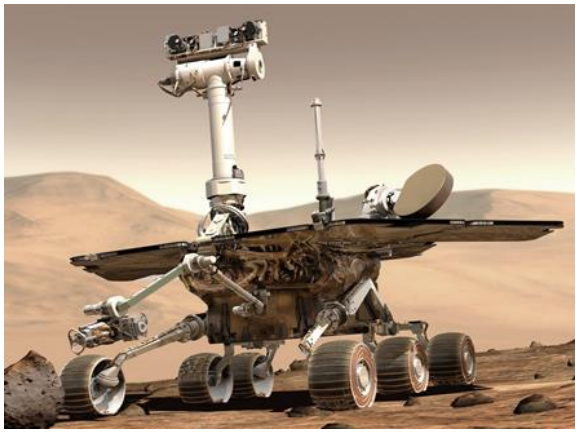


Figure 8

A computer-generated image of a NASA space rover exploring the surface of the planet Mars (Sandra May, 2021).

sol (a planet's single solar day) contact is not an option which is time that automation could save (Maimone, Leger and Biesiadecki, 2007).

Arguably the most significant rover mission to Mars was conducted by NASA and saw the launch of the *Mars Exploration Rovers* (MER) *Spirit* and *Opportunity* in 2003. The system used by these rovers was built from multiple components that worked in a hierarchical way. These components were divided up in order to tackle certain aspects of one of the MER rover's current tasks. Some of these components handled more common ML tasks such as image and pattern recognition as well as locomotive adjustments for efficient movement. Other AI driven components that handled goal-based tasks such as navigation, obstacle avoidance and sample gathering were hierarchically at a higher level as they relied on lower level systems (such as identifying objects or moving while compensating for drag or slip) to operate effectively (Bresina *et al.*, 2003; Bresina and Morris, 2007; Maimone, Leger and Biesiadecki, 2007).

Most high-level autonomous planning that MER implemented pertained to navigation and task selection. These are the kinds of tasks that would usually need to be delegated to mission control and are hence bigger, more

consequential decisions that often require a human-level sense of *perspective*. Furthermore, the development of these high-level problem-solving skills may assist in other space rover missions. Some navigational-based tasks that a rover may encounter (depending on the mission) include gathering sample data, collecting and returning resources to a base location, avoiding dangers (such as rocks, loose sand or sheltering from bad weather) and being aware of power usage. Since navigation is an important aspect to high-level autonomy in space rovers, it is worth analysing how MER implemented its autonomous navigation system (Bresina *et al.*, 2003; Cheng *et al.*, 2004; Bresina and Morris, 2007).

Over the course of the MER mission lifespan, multiple software updates were introduced that gave the rovers increased levels of autonomy. These were predominantly seen in the development of the rovers' navigation through the categorization of *drive modes*. Firstly, the *directed* drive mode would predefine a rover's path without any ability to adapt while the *visual odometry* mode introduced positional awareness. The *terrain assessment* mode was able to look for obstacles while the *local path selection mode* combined the benefits of all modes over short distances. Lastly, the *global path selection mode* was able to provide autonomous navigation over long distances by utilizing other drive modes in a hierarchical way (Cheng *et al.*, 2004; Maimone, Leger and Biesiadecki, 2007).

The local path selection mode worked by converting the rover's internal model of its environment (based on visual sensory input) into a 10 x 10 meters^2 grid world with each block being roughly 0.2 meters^2. This method was called GESTALT *(Grid-based Estimation of Surface Traversability Applied to Local Terrain)*. The global path selection mode built upon this by using a larger grid of 50 x 50 meters^2 with each block being roughly 0.4 meters^2 and greatly improved the rovers' navigational efficiency (Maimone, Leger and Biesiadecki, 2007). The global path selection mode also used the *Field D** planner algorithm to guide its

movement. The Field D* algorithm simply finds a path from a starting point to a goal point in a 2D grid-world before converting the grid-based path to an optimally smooth gradient that can be executed in a real-world environment (Ferguson and Stentz, 2007).

Despite the addition of local and global path selection, a significant portion of the important decisions and actions were still issued by the team back on Earth (Maimone, Leger and Biesiadecki, 2007). NASA then introduced a system called MAPGEN (*Mixed Initiative Activity Plan Generator*) which assisted scientists on Earth in plotting out planned routines for the rovers that could then be autonomously executed in a somewhat intelligent manner. MAPGEN ensured that all goals and actions taken would be as low-risk as possible and within any constraints specified by the user. It could hierarchically organise and optimise sequences of sub-goals and hypothesise about what may happen should a rover execute a plan in a certain way. Despite this, MAPGEN still remained heavily reliant on the user and its behaviour was more rigid than anything one may consider to resemble truly intelligent or general AI (Bresina *et al.*, 2003; Bresina and Morris, 2007). This is exemplified by the fact that *Bresina* and *Morris* identified three major issues in MAPGEN. They pointed out that the plan commands were very broken up and required too many safety confirmations. Furthermore, all high-level planning initiatives had to be directed by the user, and the system lacked a sense of its own ability to be aware of the intent of the user's plan (Bresina and Morris, 2007).

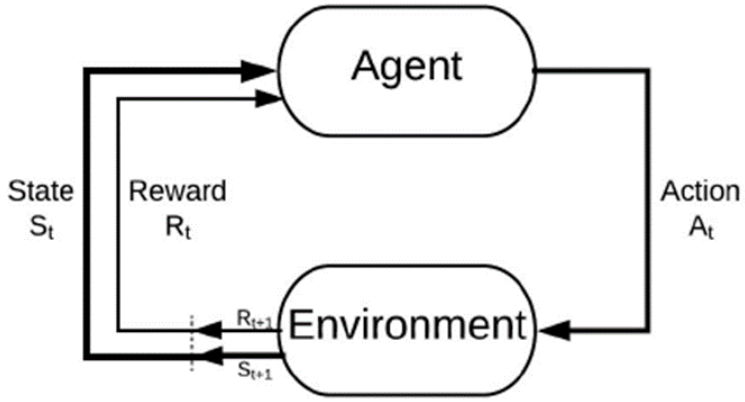


Figure 9

The basic components of RL as applied to intelligent agents (Shyalika, 2019).

2.1.6 Reinforcement Learning

Intelligent agents that make use of AI begin to move away from the rigidity of predefined algorithmic implementations. However, this introduces a certain level of risk as one cannot be perfectly sure exactly how an agent may respond to a certain situation. It is therefore important to ensure that an agent is able to learn how to effectively act in a rational manner in its environment. Expanding on the learning approaches discussed in *chapter 2.1.3*, a common method of learning that lends well to intelligent agents (particularly those operating in environments that we do not know everything about) is the method of *Reinforcement Learning* (RL).

RL is based off how living things learn through feedback by interacting with an environment through a *reward function* (Barto and Sutton, 1999). An agent that learns using RL gathers observations by exploring its environment and then learns through *Skinnerian conditioning* which dictates that positive feedback is

received for doing something desirable and negative feedback is received for doing something undesirable. An agent then uses this information to update its internal state (belief state) of the environment that it is in (McLeod, 2007).

Broadly speaking, RL systems can learn in two different ways. One way is known as a *model-based* approach and the other is a *model-free* approach. In a model-based approach, the goal of an RL algorithm is to learn how to map environmental states to actions to maximise rewards over time. In a model-free approach, a system bypasses attempting to learn a specific model for an environment and instead tries to learn general rules that it can use. In practice, many RL systems have a combination of both approaches (Mnih *et al.*, 2013; Belani, Vuković and Car, 2019).

Many RL systems follow what is called a *Markov Decision Process* (MDP). An MDP uses the *Markov Property* which states that *"the future is independent of the past given the present"*. This essentially means that the past events leading up to a current state are encoded in the current state itself, and hence, all that is needed to make an optimal decision for the future is the current state (Hunt, 2010).

One of the issues with dynamic and changing environments (which is often the case in real-world environments) is that the current state that an agent is in might be related to a sequence of environmental variables stretching back over a period of time. Using this data can often provide more context or *perspective* to an agent's decision. In this case, the current state in the MDP can be represented by all the sensor inputs over that window of time (Oke, 2008; Gupta *et al.*, 2018; Ke *et al.*, 2018).

An MDP considers four aspects. These include the *states* that encode the environment configuration, *actions* made by the agent, *transition functions* that map how the environment changes under a given action, *rewards* fed back to the agent, and a *discount factor* γ for future rewards (Minsky, 1961; Watkins and

Dayan, 1992; Hunt, 2010). A *policy* is a particular mapping of states to actions while the optimal policy is the action taken from a particular state that has the highest expected long-term reward. The expected reward after following a policy is known as the *value function* (V) and the expected reward after acting optimally is known as V* which can be applied recursively (Dietterich, 2000; Mnih *et al.*, 2013; Frans *et al.*, 2018). *Bellman's Principle of Optimality* (BPO) says that an agent should make an action that yields V*. This works well in a deterministic, fully observable environment, but like many real-world scenarios, this is not always the case (Helman, 1986).

When an agent first enters an environment, it may not know anything about it. The agent may also only be able to observe part of the environment at once and would hence be using a *Partially Observable Markov Decision Process* (POMDP) (Oke, 2008; Ohnishi *et al.*, 2019). The agent must therefore explore and learn about its environment before it can start to act optimally (Gupta *et al.*, 2018). One of the problems with the BPO is that, if an agent only knows about one optimal reward in an environment, it will always act so as to achieve that reward and may miss out on a potentially greater reward (Gupta *et al.*, 2018). This is a fundamental issue surrounding the idea of exploration vs exploitation. Another problem is that agents in large environments can only practically look a certain number of states ahead when considering expected future rewards. This is called having a *finite horizon* (Kocsis and Szepesvari, 2006).

An agent is said to act rationally when it always acts in a way to achieve the best expected outcome, such as in the case of the BPO. However, acting randomly or irrationally adds an element of exploration (Gupta *et al.*, 2018). *Monte Carlo methods* take random actions (known as a *Bandit Based* approach) to previously unexplored states in order to find more optimal solutions. This notion of random exploration is useful when not much is known about the underlying dynamics of an environment (Kocsis and Szepesvari, 2006).

Q-Learning (QL) makes use of Monte Carlo methods and introduces a new function known as the *Q-Function* which finds the *Q-Value* for any given state. The Q-Value is the expected return from starting in a particular state and following some action before acting optimally from the resulting next state (Watkins and Dayan, 1992; Ohnishi *et al.*, 2019). The initial action need not be optimal. QL uses an *ϵ-greedy* policy which means that at each state, the agent chooses a random action with a probability ϵ, otherwise it acts optimally according to what it knows so far (Tokic, 2010). QL exhibits what is called an *off-policy* which means that the Q-Value is evaluated by calculating the expected future returns based on the assumption that all actions will be greedy. An *on-policy* means that the actual actions of the agent (whether greedy or explorative) are taken into consideration (Fujimoto, Meger and Precup, 2018).

The Q-Function makes use of BPO and is expressed by *Equation (1)*. The calculation of the Q-Value takes in two parameters, the current state *s* and an action *a*. *R(s,a)* is the reward for state *s* and action *a* while *P(s,a,s')* is the probability of moving to state *s'* given action *a* and the current state *s*.

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} \left(P(s,a,s') \max_{a'} Q(s',a') \right) \qquad (1)$$

QL can take a lot of computational time when Q-Values are calculated over long horizons. One solution to this is to build a generalized understanding of the problem space and learn policies that act upon this latent representation. This can be done through *Deep Q-Learning* (DQL) (Mnih *et al.*, 2013; Ohnishi *et al.*, 2019). An example of DQL is a *Deep Q-Network* (DQN) where an *Artificial Neural Network (see chapter 2.3)* is used to take in all sensory information defining an agent's current state before learning to map this information to an expected return for each possible action (Ohnishi *et al.*, 2019).

DQL is good at dealing with high-dimensional inputs from sensors, which is often the case when a state is defined by a time-based window. DQL can be useful in abstracting multiple related states down to a single type of action so that it can behave appropriately when the agent is in an unseen but similar state (Mnih *et al.*, 2013).

Another method of tackling distant horizons that many modern RL methods leverage is to use a *replay buffer* through *experience replay*. In this approach, Q-Values are not evaluated when the occur, but experiences (represented as state, action, reward, and next-state combinations) are instead stored in a memory buffer. Learning can then be implemented by evaluating a random sample of experiences from the buffer at any point which can be very effective when acting and learning are then also interleaved through Monte Carlo methods (Fujimoto, Meger and Precup, 2018). This is an example of how an RL method can make use of episodic memory.

2.1.7 Summary and Recap: Artificial Intelligence

Understanding what makes something generally intelligent is important as it directs how to go about building a general cognitive architecture that allows for the development of multiple potentially strong AI for any environment. This "one solution fits all" (AutoML) approach mitigates humans having to rebuild and debug new AI solutions for every new task.

Mimicking nature has often been a starting point for the development of many approaches to AI. Through the analyses of fundamental intelligent processes in nature, it is evident that biologically intelligent entities all exhibit some common traits. These include learning through experience, exploring through curiosity,

exploiting knowledge that is already known, and the ability to develop fundamental generational knowledge as demonstrated through instinct.

The goal of this thesis is to construct a general cognitive architecture (BE). By following the cognitive modelling approach, these four aspects of biological intelligence need to be directly addressed in the construction of BE. Learning through experience, and the idea of exploration vs exploitation can be summed up as RL while the notion of instinct is representative of cognitive *evolution (see chapter 2.3.3)* (Barto and Sutton, 1999; Tokic, 2010; Miikkulainen and Lehman, 2013). Furthermore, the idea of exploration can also be expressed through Monte Carlo methods which use randomization to try new paths in a problem space (Kocsis and Szepesvari, 2006).

For a more fine-grained analysis of the emergent behaviours that intelligent entities must poses, both *Sternberg* and *Thurstone* attempt to define intelligence as the sum of some components but fail to offer definable targets that can be applied to software (Thurstone, 1962; Clarke and Sternberg, 1986). *Minsky's* analysis is more applicable to AI, however, *Osband's* identification of seven core capabilities that intelligent agents must possess offers the most useful and clear set of targets for BE to address in order to create generally adequate agents (Minsky, 1961; Osband *et al.*, 2020).

The cognitive architectures Soar and ACT-R also promote the effectiveness of reducing cognition down to a set of components. Their approaches to analysing intelligence also draws many parallels to *Minsky's* analysis with regards to learning in a problem space, planning, and using inductive reasoning in order to use structured knowledge in deriving sub-goals to solve global ones. Both ACT-R and Soar also categorize memory (or knowledge) into different components. The common theme differentiates between current and short-term knowledge (working and episodic memory) from long-term and factual knowledge (procedural and semantic memory). ACT-R further defines this knowledge as

being explicit or implicit (Minsky, 1961; Scerri, 2006; Laird, 2008). However, the concept of memory and how knowledge across time relates to each other is a common challenge in AI and is a limiting factor of Q-learning (over distant horizons) as well as the credit-assignment problem (with regards to long-term, delayed rewards). One way of addressing this is to use a replay buffer, however, it is still evident that memory is a key issue that must be addressed by BE (Watkins and Dayan, 1992; Fujimoto, Meger and Precup, 2018; Ke *et al.*, 2018). It is also relevant to note that neither Soar nor ACT-R have yet to explore evolutionary approaches. Furthermore, CLARION introduces the contrasting idea of representing knowledge using connectionist methods as opposed to symbolic ones (Scerri, 2006).

The widespread and complex demands of modern intelligent agents highlight the importance of BE being capable of learning in an unsupervised manner and dealing with POMDPs as humans do not always know all the details about an intelligent agent's target environment. Secondly, due to the potentiality for high-dimensional problem spaces, there is an inevitability that some approach to DL is necessary in the construction of BE (Mnih *et al.*, 2013; Poggio *et al.*, 2017; Ohnishi *et al.*, 2019).

This thesis focuses on Mars rovers as a real-world example of intelligent agents operating in a dynamic, unpredictable and unknown environment. The need for high-level, time-sensitive planning skills and the ability to easily find solutions for new and different tasks (through the use of a cognitive architecture) is very evident in the challenges that faced the MER mission. Even the later software updates that included MAPGEN lacked a sense of autonomous *perspective*. Most of these tasks required some level of goal-driven navigational ability. One useful approach taken by the system in place was to break down sensory input of the environment into a grid-world representation (Bresina and Morris, 2007; Maimone, Leger and Biesiadecki, 2007; Boukas *et al.*, 2017). This abstracts the

information at hand and renders any higher-level decisions easier to make. A similar approach can also be used when simulating tests for potential tasks that a Mars rover may encounter *(see chapter 3.8.3)*.

2.2 Metaheuristics

The strength of AI methods over predefined algorithms is in their ability to learn, adapt, and figure out solutions for themselves, even if we do not know exactly how we might go about implementing a solution ourselves. However, unlike predefined algorithms, AI solutions can be unpredictable and hence unreliable. An AI solution traverses a problem space trying to find the best solution that it can, but there is no guarantee that the best solution it finds will be the globally optimal solution. This alludes to the concept of *heuristics*, as fundamentally, AI approaches are heuristic in nature. A heuristic is a method of finding a solution to a problem that exploits the nature of the problem by taking shortcuts that yield potentially viable solutions in a shorter period of time (Glover, 1986). Alternatively, one could perform a brute-force search of the problem space to find the most optimal solution, but this is usually far from being a feasible solution. Furthermore, the idea of an optimal solution becomes ever more abstract when dealing with strong AI and so it can be difficult to know what to search for.

As previously discussed in *chapter 2.1*, there are multiple paradigms of knowledge acquisition to address when considering how to approach the construction of a general cognitive architecture. Not only do many problem domains often require an understanding of multiple hierarchies of information (as tackled by DL), but the methods used to solve these problems are problems within themselves as their configurations affect their performances. If a cognitive architecture can be applied to any situation using AutoML, it needs to make use of some kind of general problem-solving mechanism that can be applied in a layered way from a fine-grained scope to a course-grained one. These issues are directly tackled by what are called *metaheuristics*. Metaheuristics are simply methods of general

problem-solving and are approaches or frameworks that guide the development of other processes (Voß, 2001; Lozano, Molina and Herrera, 2011).

The resultant solutions produced by a metaheuristic algorithm are hence heuristics themselves, which is to say that they are not guaranteed to be globally optimal solutions, but rather ensure some reliability that they will be at least adequately good (Glover, 1986; Voß, 2001). Metaheuristics are also well suited to finding solutions that would otherwise be missed if only a search for a local optimum was conducted (Glover, 1986).

Since metaheuristics can be seen as a *master strategy* to solving problems, they can be applied in many various situations or problem spaces (Voß, 2001). However, since metaheuristics just approximate good solutions, they are generally only used if there exists no exact feasible solution for the problem (Glover, 1986).

Metaheuristics are often inspired by real processes in nature. A common idea is that individual components can self-organise through interactions to produce a good global solution (Gobeyn *et al.*, 2019; Bryndin, 2020). Some of the most commonly used metaheuristics are EAs (as discussed later in this chapter), however, a few other well-known metaheuristics are listed below.

- **Simulated Annealing (SA)** *(see Figure 10)* is modelled after the way heuristically optimal solutions can be found when moving from high entropy (chaos) to low entropy (order). This process comes from physics where particles in a system settle into more stable states as they go from a high system energy to a low one (Odziemczyk, 2020).

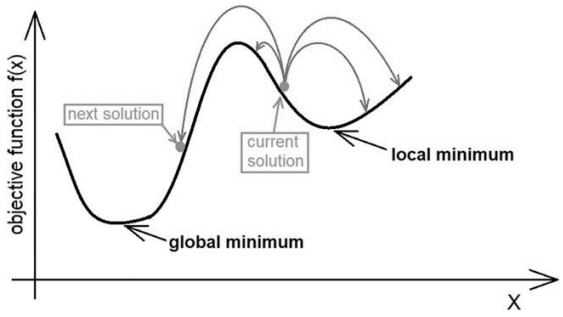


Figure 10

Diagram showing how SA traverses a 1D problem space by progressively making smaller randomized exploratory jumps (Odziemczyk, 2020).

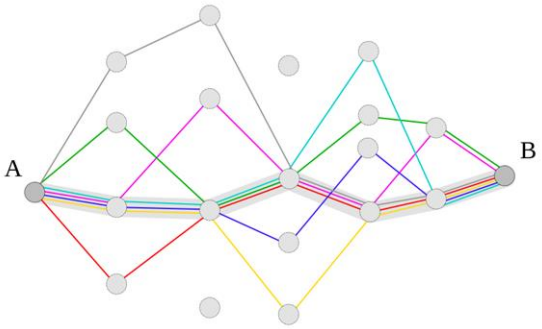


Figure 11

Diagram showing how ACO finds an optimal path from A to B. Different entities (ants) explore in varying directions, but the shortest paths leave the strongest "ant pheromone" trails and hence become more predominantly used (Dréo, 2006).

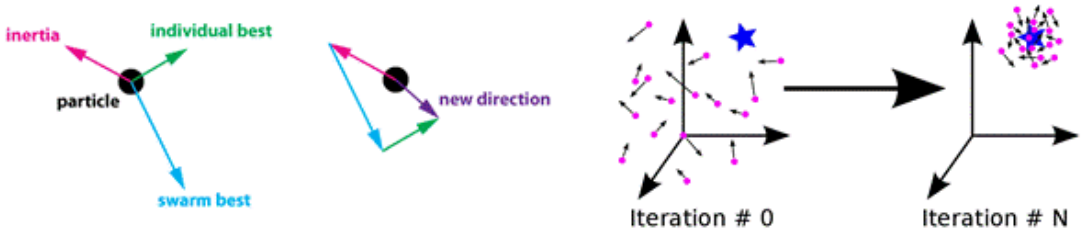


Figure 12

Diagram showing how PSO will converge over time to optimal solutions (right) by adjusting its momentum through a problem space using the locations of the best solutions found so far by the swarm as well as its current inertia (left) (Adyatama, 2019).

- **Ant Colony Optimization (ACO)** *(see Figure 11)* is based on the idea that intelligent behaviour emerges from the coordination of multiple less intelligent entities. It is inspired by the way ants are able to construct complex societies and structures through many simple workers making many simple interactions (Utzle *et al.*, 2010).

- **Particle Swarm Optimization (PSO)** *(see Figure 12)* is centred around the idea that systems can self-organise whilst remaining decentralized. These systems are then able to find emergent, heuristically optimal solutions. An example of this kind of behaviour in nature is in how migrating birds or swimming shoals of fish can seem to move in coordination without explicitly communicating with each other (Mishra, 2011; Mirfenderesgi and Mousavi, 2015).

Naturally, as the dimensionality of a problem space increases, a metaheuristic solution will often take an exponentially long time to find a good solution in accordance with the curse of dimensionality (Poggio *et al.*, 2017). One issue with metaheuristics is that they can be seen as *black-box* approaches. In other words, it is often unclear how or why a metaheuristic algorithm produced a particular solution or even why it may be a good solution (Belani, Vuković and Car, 2019; Gobeyn *et al.*, 2019). The outputs of a solution that a metaheuristic returns may be difficult to interpret and can be given without the programmer even knowing much about the problem it is solving.

2.2.1 *Evolutionary Algorithms*

Arguably some of the most popular metaheuristic approaches are *Evolutionary Algorithms* (EAs). EAs are universal methods for finding solutions to problems that are inspired by the principle of *Darwinian evolution* as seen in nature. EAs are a subset of algorithms in the broader category of *Evolutionary Computing* (EC) which is itself just a classification of metaheuristic approaches (Turney, Whitley and Anderson, 1996; Olague, 2016).

For the process of evolution to work, certain conditions must be met. There must exist a set of potential solutions to a problem or task. This represents a *population* of *individuals* in a single *generation*. Each of these individuals must have varying characteristics which must be passed on through each successive generation. There must also exist a *selection pressure* in which the individuals that possess characteristics that perform better in their environment are more likely to be selected to produce offspring. Such individuals are said to have a higher *fitness*. The process of passing on emerging characteristics due to environmental pressures is called the *Baldwin effect* (Turney, Whitley and Anderson, 1996; Gobeyn *et al.*, 2019).

For each successive generation, new individuals (children) are produced from the parent individuals that have been selected from the previous generation. The traits from the parents of a child must be *recombined* in the child. *Mutation* of characteristics is also important to ensure genetic diversity and exploration of the problem space. Both recombination and mutation are referred to as *genetic operators* (Zelinka, Senkerik and Pluhacek, 2013; Olague, 2016).

A common approach to recombination is to use what are called *crossover* operators. Crossover is a process of recombination whereby the information defining a solution from two or more parents is divided and then reassembled in

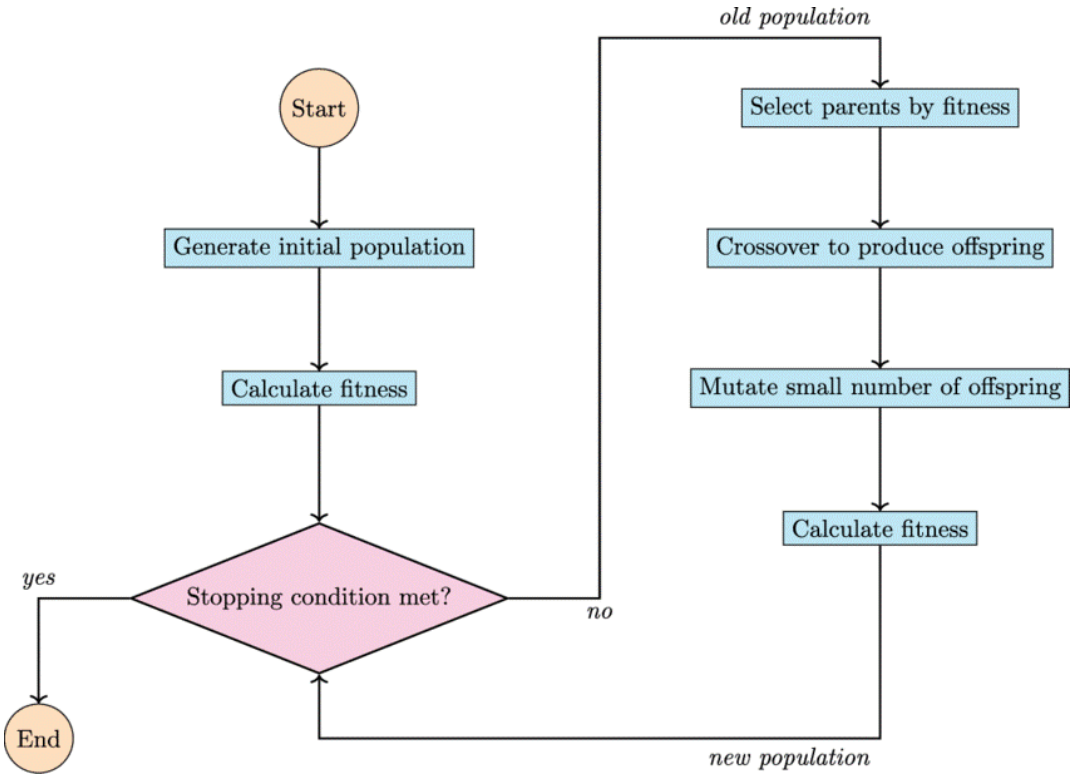


Figure 13

Diagram showing a general overview for an EA (Wilde, Knight and Gillard, 2020).

the resultant child using various methods. Different portions of the data in the child are randomly assigned from each of the parents. Recombination operators mimic the way genes from parent animals or humans are combined in their children in nature (Goldberg and Holland, 1988; Champrasert, Suzuki and Otani, 2009; Olague, 2016).

The expression of an individual can be divided into its *phenotype* and its *genotype*. The phenotype is an individual's physical manifestation and is what the parent selection process operates on. This can be seen in the individual's appearance or its behaviour and response to its environment. The genotype of an individual is its actual genetic encoding and is passed on through reproduction and altered through recombination and mutation (Turney, Whitley and Anderson, 1996; Hadjiivanov and Blair, 2016).

Each individual's varying characteristics (genetic code) are represented by different data structures depending on what kind of EA is being used. The EA also affects the choice of approach to recombination and mutation. Some common types of EAs are listed below (Voß, 2001; Olague, 2016).

- **Genetic Algorithms** represent genotypes using *binary strings* (Goldberg and Holland, 1988).
- **Evolutionary Programming** uses *Finite State Machines* (Olague, 2016).
- **Evolutionary Strategies** represent genotypes using *real-valued strings* (Rubenstein, 1982).
- **Genetic Programming** makes use of *tree structures* (Olague, 2016).

Selection pressure is generated by testing each individual's ability to solve the problem at hand as they all compete to see whose solution performs better. The task or problem is represented by the individual's environment and how well an individual does is defined by the *fitness function* for that environment. After multiple generations, an evolutionary process should result in the adaption of the population to its environment and a general increase in fitness (Jakobi, Husbands and Harvey, 1995; Turney, Whitley and Anderson, 1996; Olague, 2016).

2.2.2 Evolutionary Selection Methods

EAs can either be *generational* or *steady-state*. In a generational EA, all the individuals for each successive generation are newly created children and all parents from the previous generation are replaced. This can also just be done on a significant portion of the population instead of its entirety. In a steady-state EA,

only one child is created each generation which replaces only one parent from the previous generation (Branke, Kaußler and Schmeck, 2001; Olague, 2016).

There are various ways to perform selection. Selection can either be deterministic or stochastic. A *fitness based* deterministic method ranks both parents and children by their fitnesses and then simply carries over the fittest percentage of individuals to the next generation. An *age based* deterministic method allocates a finite number of generations for an individual to exist in its population before it is replaced, regardless of its fitness (Blickle and Thiele, 1996; Eiben and Smith, 2015).

In stochastic selection, parents are only selected to produce offspring based on probability. This probability is defined by the individual's fitness where the fitter an individual is, the more likely it is to be selected. The idea of *elitism* can also be implemented where the fittest individuals from one generation are kept for the next generation unchanged (Eiben and Smith, 2015; Gupta *et al.*, 2018).

Stochastic selection methods have the benefit of promoting exploration of the problem space. If the selection pressure is too high or too deterministic, an EA may quickly converge to a solution and settle on a local optimum whilst missing out on better solutions. Stochasticity is good at mitigating this. Some common approaches to stochastic selection are listed below (Turney, Whitley and Anderson, 1996; Gupta *et al.*, 2018).

- **Linear Selection** ranks individuals based on their fitness and then selects parents with a probability based on their ranking. The highest ranked individuals are more likely to be selected. This prevents very dominant solutions from becoming the pervasively selected parent (Blickle and Thiele, 1996).

- **Exponential Selection** works the same way as linear roulette-wheel except that lower ranking individuals have an exponentially diminishing probability of being selected. The rate of this exponential decay is controlled by adjusting the base of the exponent (Blickle and Thiele, 1996).

- **Proportional Selection** directly assigns the probability of an individual being selected as its fitness over the total fitness of the population (Blickle and Thiele, 1996).

- **Tournament Selection** randomly selects *n* individuals from the population and places them in a tournament pool. The fittest individual from this pool is then selected. The larger *n* is the greater the selection pressure (Blickle and Thiele, 1996).

2.2.3 Multi-Objective Evolutionary Algorithms

EAs have been demonstrated to be strong universal problem-solving heuristics. They have also been shown to be well suited for multi-objective problems as well as being particularly good at being *Pareto-optimal* in such environments (Zitzler and Thiele, 1999). A Pareto-optimal solution means that if an improvement is made to the solution for any one of its objectives, it is accompanied by a degradation of the solution for one or more of any of the other objectives. A solution can also *dominate* another solution if it is no worse than the other solution for all objectives and better than the other solution in at least one objective. The set of all Pareto optimal solutions is called the *Pareto front* and is shown in *Figure 14* (Branke, Kaußler and Schmeck, 2001; Zitzler, Laumanns and Thiele, 2001). The ability of EAs to find any given Pareto front makes them useful

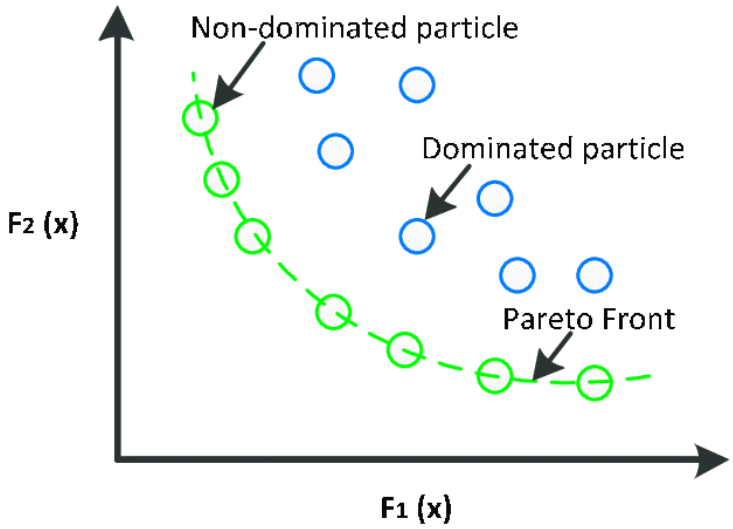


Figure 14

Diagram showing the difference between dominated and non-dominated solutions as well as the Pareto front (Mahesh, Nallagownden and Elamvazuthi, 2016).

tools to use when modelling intelligent agents that have to complete complex, multi-objective tasks, which is often the case in real-world environments (Branke, Kaußler and Schmeck, 2001; Belani, Vuković and Car, 2019).

One of the main challenges of multi-objective EAs is determining how an individual's fitness is evaluated. Another challenge is ensuring that there is sufficient population diversity to cover all possible objective solutions (Zitzler and Thiele, 1999). Diversity begins by randomly generating the initial population for a particular run of an EA. This initial population needs to be sufficiently large and random enough to cover a significant spread of the problem space. Sometimes EAs can get stuck in local optima, however, this can be mitigated by altering certain parameters of the EA such as (but not limited to) decreasing the elitism factor or increasing the mutation rate. Mutation promotes exploration of the problem space and lowering elitism decreases selection pressure (Eiben and Smith, 2015; Olague, 2016; Gupta *et al.*, 2018; Gobeyn *et al.*, 2019).

In multi-objective evolution, there are varying approaches to deal with Pareto-optimality for all different objectives of an environment. A simple method is to aggregate an agent's fitness into a single value by scaling each fitness value for each objective by the importance of the objective before adding them together. However, constructing a good fitness function becomes difficult when not much is known about the problem space or the importance of each sub-objective (Lozano, Molina and Herrera, 2011; Olague, 2016).

Any method that aims to find multiple peaks in a problem space (multiple objectives) while maintaining a global fitness diversity is called *niching*. One method proposed by *Goldberg* called *fitness sharing* scales the fitness of an individual by how close it is to other solutions in the problem space. Good solutions in highly populated areas will have decreased fitnesses compared to worse solutions in less populated areas. This encourages exploration of new regions (Goldberg and Holland, 1988; Gobeyn *et al.*, 2019).

Another approach that utilizes niching is called *speciation*. In speciation, the data defining an individual is represented as a vector or array of vectors. A distance function is then constructed that can be used to calculate a single real valued distance between two different individuals (Stanley and Miikkulainen, 2002). All individuals are assigned to a species based on the distance function before the selection process begins. Each individual is compared to a representative individual from each species (which is commonly chosen at random for each generation) before being assigned to the first species that is closer than some threshold value. If the individual does not fit into any species, a new species is created. During the selection process, each species must select a certain number of parents from its sub-population based on the species' cumulative fitness as a portion of the entire population's cumulative fitness (Stanley and Miikkulainen, 2002; Hadjiivanov and Blair, 2016).

Speciation can be performed dynamically at each generation and can be useful when not much is known about the fitness landscape. Speciation also allows new directions in potential solutions that have not had time to evolve the chance to expand without having to compete with older, more established solutions (Hadjiivanov and Blair, 2016). Some more advanced applications of speciation utilize clustering algorithms such as the *K-means* clustering algorithm to organise individuals into species (Aspinall and Gras, 2010).

Some other niching techniques include *crowding*, *Evaluated Genetic Algorithm* (VEGA) and the *Strength Pareto Evolutionary Algorithm 2* (SPEA2). In crowding, children are paired with parents based on some similarity metric before being evaluated by a *replacement rule* that decides their selection (Eiben and Smith, 2015). VEGA works similarly to speciation and uses a vector containing evaluations for all objectives for a particular individual before splitting the population into sub-populations that each optimize to different parts of the vector (Zitzler and Thiele, 1999). Lastly, SPEA2 works by utilizing an archive of all non-dominated solutions from both the archive and population of the previous generation. The archive is then reduced or filled to maintain its size for each generation (Zitzler, Laumanns and Thiele, 2001).

2.2.4 Hyperparameter Optimization

At the highest level of AutoML problem-solving stands the overall parametrization of an architecture. Any algorithmic approach to AI is defined by its own set of global parameters. In an EA these could be (among many other parameters) the mutation or crossover rate. The parameters that govern the way an algorithm runs are called *hyperparameters*. Any meta-algorithm or

metaheuristic that converges on optimal values for a set of hyperparameters is called a *hyperparameter optimization* (HPO) algorithm (Hutter, 2014). HPO is a useful tool as algorithms (such as EAs) can yield very poor results if they have ineffective hyperparameter settings and potentially very good results if set correctly (Hutter, 2014; Gobeyn *et al.*, 2019). HPO is a sub-problem of AutoML and is an essential component in covering all aspects of a cognitive architecture's hierarchy (Cooper *et al.*, 1996; Hutter, 2014).

HPO can be done with an *offline* or *online* approach. An offline approach (parameter *tuning*) adjusts the hyperparameter values of an algorithm and then runs it to evaluate the performance of the current configuration. Online adaption (parameter *control*) is done at runtime and can be heuristically determined using three different approaches. A *deterministic* approach uses static heuristics to modify parameters over time while an *adaptive* approach uses heuristics that are based on monitoring the current progress of the algorithm being optimized. Lastly, a *self-adaptive* approach includes the hyperparameters to be adjusted into the algorithm itself which the hyperparameters govern (Glover, 1986; Utzle *et al.*, 2010; Eiben and Smith, 2015).

When evaluating a potential hyperparameter configuration, multiple runs should be done. A good configuration should yield a statistically similar result over different runs. There are multiple performance evaluation measures that one can use. Some of the main metrics are listed below (Smit and Eiben, 2010).

- **Best Solution:** The best solution obtained across all runs of the algorithm.
- **Average Evaluations to Solution (AES):** The number of fitness evaluations before a specified minimum highest fitness is reached.
- **Success Rate (SR):** The percentage of fitness evaluations that are above a specified minimum highest fitness.
- **Mean Best Fitness (MBF):** The average best fitness found across all runs of the algorithm.

2.2.5 *REVAC*

Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters (REVAC) is an offline HPO algorithm that utilizes an evolutionary strategy. REVAC is designed to converge rather quickly as computational time is more of an issue in offline HPO algorithms as opposed to online ones (Smit and Eiben, 2010).

REVAC works by initiating a population of potential parameter configurations. These configuration solutions are evaluated by running them *n* times before averaging their evaluated performance (which is usually done using MBF). Parents for the next generation are selected deterministically. A predefined portion of the population with the highest utility values are selected to be parents (Smit and Eiben, 2010).

Children solutions are created by using a multi-parent crossover operator across all the selected parents. Each parameter value assigned to a child is chosen randomly from one of its parents' corresponding parameter values (Smit and Eiben, 2010).

The next step is to implement mutation which is independently applied to each parameter for every child. All parameters are randomly assigned new values within a given range above and below their original values. This range is found by ordering all parents by the specific parameter in question and then finding the lower bound value at the n^{th} lowest neighbour below the parent that holds the child's original value and the upper bound value at the n^{th} highest neighbour. All individuals in a previous generation are replaced by the children for the next generation (generational selection) (Smit and Eiben, 2010).

2.2.6 Summary and Recap: Metaheuristics

Metaheuristics are powerful tools when it comes to general problem-solving. Furthermore, the idea of *meta* problem-solving is also very useful in analysing the development of general intelligence as biologically intelligent entities are able to learn how to solve new problems by developing higher level skills that utilize the assumption that the underlying mechanics of a problem hold true (Hutter, 2014). This concept of meta-learning has similarities with the concept of hierarchical learning which is explored later in *chapter 2.4.1*.

Some of the most popular metaheuristic approaches include EAs. Biologically intelligent entities are themselves emergent through the process of evolution which is a genetic metaheuristic process. Such entities then further learn how to act optimally in their environment through RL during their lifespans. Evolution can also be evident in the development of instinct as evolution does not only apply to a creature's phenotype but also moulds neural structures so that creatures are born with some level of a pre-trained ability to do things (Turney, Whitley and Anderson, 1996; Miikkulainen and Lehman, 2013). Furthermore, due to the ability of metaheuristics to be master strategies, they may prove to be a necessary tool in the construction of general cognitive architectures.

EAs ability to solve multi-objectives is also relevant as intelligent agents operating in complex real-world environments often encounter such situations (Zitzler and Thiele, 1999; Badue *et al.*, 2021). EAs are also good at solving tasks in POMDPs as they are able to explore the problem space in an unsupervised manner (Miikkulainen *et al.*, 2018). However, due to the fact that, by definition, heuristic methods can only guarantee adequately good solutions, they are likely not strong enough on their own to produce convincing general intelligence and may need to be coupled with other learning methods such as RL (as is the case with biologically intelligent entities).

Evolutionary approaches can often take extended amounts of time to converge (Branke, Kaußler and Schmeck, 2001). However, it may be that these longer processing times are necessary for the emergence of strong solutions (as evident in the billions of years of biological evolution) and that algorithmic shortcuts may ultimately limit the potentiality of a solution by encouraging the convergence of local optima. Furthermore, since there are so many parameters that may govern an evolutionary metaheuristic algorithm, the need to find some optimal set of hyperparameters is also prevalent (Hutter, 2014). This optimization of hyperparameters can be extended to be a potentially fundamental requirement of BE's cognitive architecture. HPO, or some kind of meta-metaheuristic may hence be an important step in the development of generally intelligent agents.

REVAC is a strong and fast (relative to other generic EA methods) off-line HPO algorithm. Online adaptive approaches are much faster but if implemented deterministically these methods can often be suboptimal. On the other hand, self-adaptive implementations can drastically increase the dimensionality of the problem space, thus also potentially hindering the evolutionary process (Smit and Eiben, 2010; Utzle *et al.*, 2010).

The process of selection in an EA is also important. According to Blickle and Thiele, stochasticity is crucial and linear, exponential, and tournament selection are good methods to use depending on the problem at hand, but proportional selection seems to perform consistently worse (Blickle and Thiele, 1996). One of the strengths of exponential selection is that it can behave in a similar way to other methods if its parameters are set in a particular manner as well as being able to exert far greater selection pressure than other methods. This makes it a very versatile method of selection.

2.3 Artificial Neural Networks

In the effort to cover all aspects of a comprehensive cognitive architecture, we have explored intelligence and how it can be expressed in machines, outlined different approaches to learning, and highlighted some methods to higher-level problem-solving through the use of metaheuristics, EAs and HPO. Since any problem can essentially be reduced to function-mapping, methods that enable a machine's knowledge to be directly exploited at a lower, fine-grained level of problem-solving need to be explored (Minsky, 1961; Scerri, 2006). By following the cognitive modelling approach, these methods would represent the basic task completion abilities that biological brains are capable of. Arguably the most popular approach to dealing with these kinds of problems involve solutions that use *Artificial Neural Networks* (ANNs).

ANNs, or *Neural Networks* (NNs), are approaches to AI that are based on the way neurons fire signals between each other in biological brains. ANNs are a connectionist approach to processing data and can be thought of as *universal function approximators* which means that they can learn to approximate any function that produces optimal outputs for a set of given inputs in a problem space (Agatonovic-Kustrin and Beresford, 2000; Bre, Gimenez and Fachinotti, 2018).

An ANN contains nodes that simulate neurons. Each node receives a collection of input signals before sending a corresponding output signal based off the sum of its inputs with the use of some function (known as the *activation function*). Some of the most commonly used activation functions are shown in *Figure 15* (Agatonovic-Kustrin and Beresford, 2000; Gupta, 2020).

The nodes in an ANN are linked through weighted connections. The signal fired by a node is sent down one or more connections and its value is multiplied in

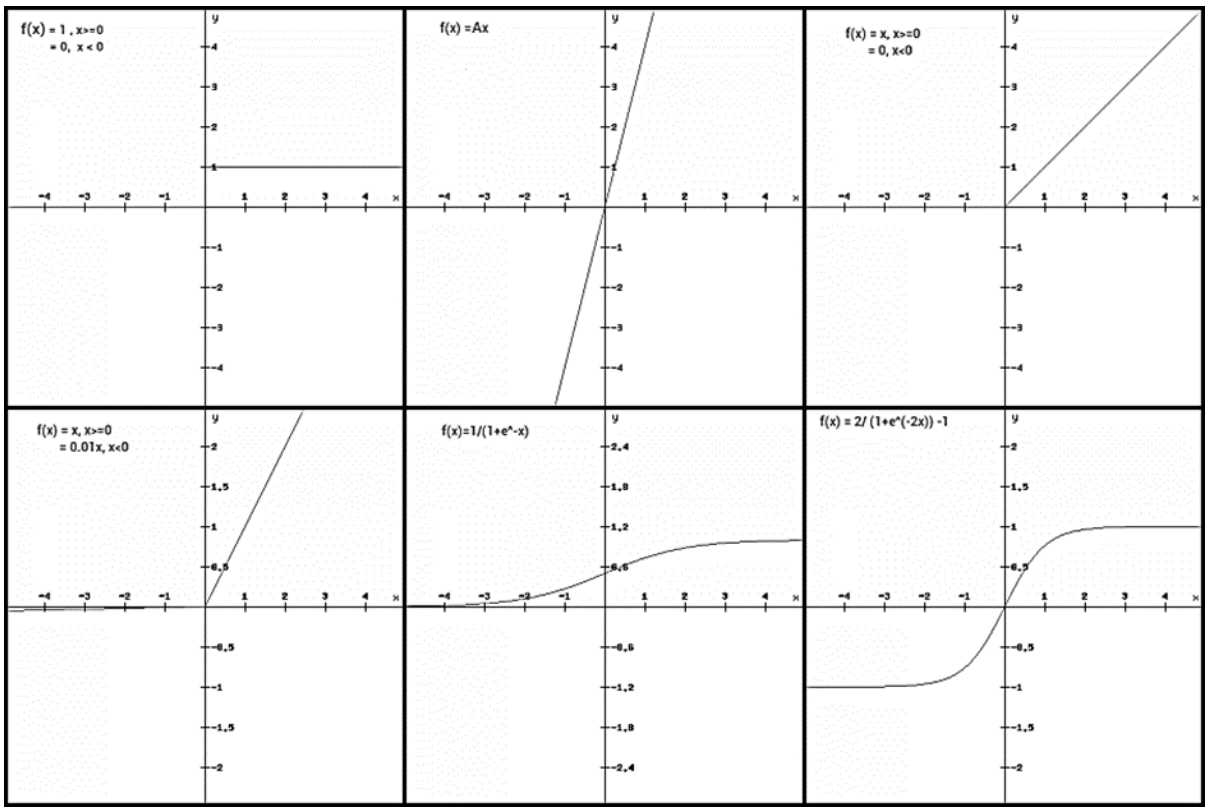


Figure 15

Common activation functions from top-left to bottom-right; Binary, Linear, Rectified Linear Unit (RELU), Leaky RELU, Sigmoid, and TanH (Gupta, 2020).

each connection path by that connection's weight. At first, all weights are randomized, but as the ANN learns it adjusts all its weights so as to perform optimally for the task at hand (Agatonovic-Kustrin and Beresford, 2000). A diagram of a basic ANN can be seen in *Figure 16*.

The nodes of an ANN are structured in layers. The first layer is called the *input* layer and the last layer is the *output* layer while all the layers between are called *hidden* layers. ANNs are good at generalizing about problems once they have been trained and as a result can be used to give good results on new unseen, similar data. ANNs are also good at dealing with noise and are hence suited for real world problems where noise is almost an inevitability (Agatonovic-Kustrin and Beresford, 2000; Bre, Gimenez and Fachinotti, 2018).

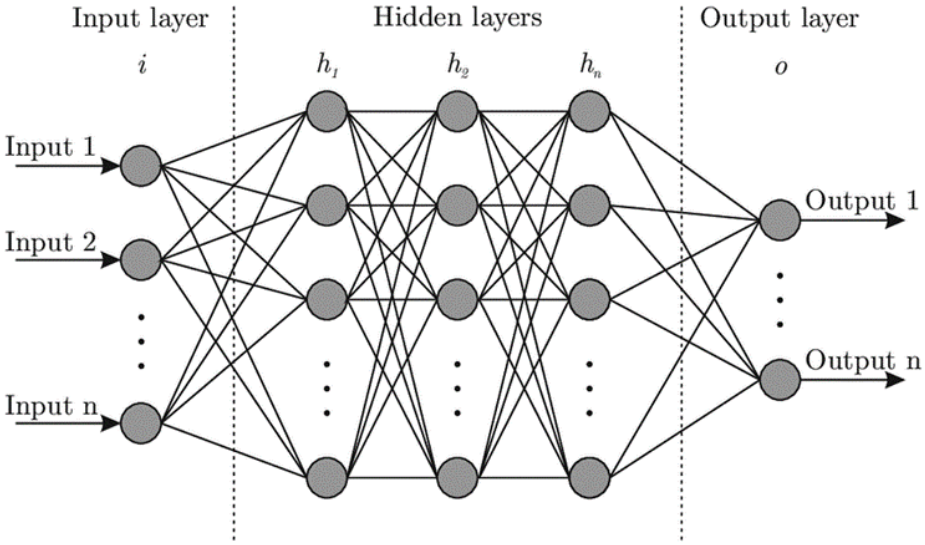


Figure 16

Diagram depicting the structure of a typical FFNN (Bre, Gimenez and Fachinotti, 2018).

ANNs can either be *cyclic* or *acyclic*. A cyclic ANN simply implies that there is at least one connection path within the network of the ANN that can be traced from a node back to itself. Networks that include input from the previous output of the same ANN are called *Recurrent Neural Networks* (RNNs). Acyclic ANNs simply contain no cycles and non-RNN based acyclic ANNs are called *Feed-Forward Neural Networks* (FFNNs) (Bhandarkar *et al.*, 2019). Furthermore, when all neurons in one layer of a network are connected to all neurons of an adjacent layer, the two layers are said to be *fully connected* (Tavanaei *et al.*, 2019).

Most implementations of ANNs today combine the application of ANNs with DL to give way to *Deep Neural Networks* (DNNs). Networks that are not DNNs are called *shallow networks* (Poggio *et al.*, 2017). DNNs cover all networks that consist of more than one hidden layer and are hence able to learn multiple levels of hierarchies and structures in data and is a necessity in solving more complex tasks. This is because the number of hidden neurons defines the upper-limit of a

network's search space and hence the maximum complexity of a task that it can solve (Hadjiivanov and Blair, 2016; Tavanaei *et al.*, 2019).

2.3.1 Convolutional Neural Networks

Building on the concepts introduced by ANNs, *Convolutional Neural Networks* (CNNs) assist in the ability of a network to extract features from data by structurally mapping an *n*-dimensional input to a latent space. CNNs are often used in image classification to extract potential features of an image before sending the feature data to an ANN to be classified (Karpathy and Leung, 2014). A CNN has an architecture that is partitioned into definable layers. The first layer (known as the *convolutional layer*) divides the image into equal regions and applies a filter function to each region. The next layer simply applies an activation function (usually RELU) to each region's output before the *pooling layer* down-samples the image, thus creating features in the newly formed latent space. This cycle can be repeated any number of times before the image is passed on to a fully connected ANN (Vedaldi and Lenc, 2015).

CNNs work well with n-dimensional images, however, many problems can be transformed to be made to look like an image (Karpathy and Leung, 2014). In the case of an agent operating in an environment, the image created would be the agent's current interpretation of the world from its input sensors. The dimensionality of this image can be increased if we include previous sensory data into the input (Mnih *et al.*, 2013; Vedaldi and Lenc, 2015). Another benefit of CNNs is that they can decrease computational times as the latent representation of any input data has a lower dimensionality once compressed and hence does not require as much processing by a fully connected ANN (or any other further

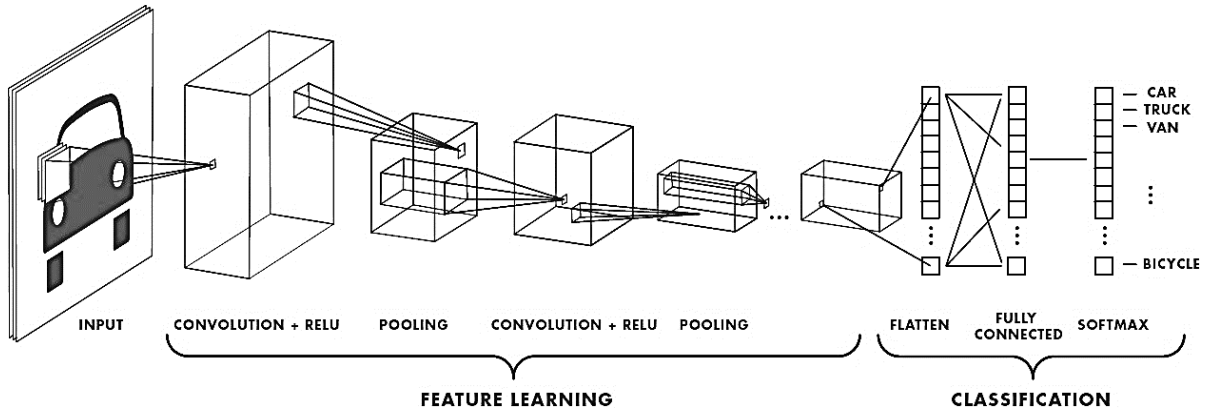


Figure 17

Diagram depicting how a CNN processes and classifies image data (Saha, 2018).

architectural components) to achieve accurate results (Karpathy and Leung, 2014).

2.3.2 Gradient Descent

One method that is often used to train ANNs is through a process called *gradient descent* which is achieved through an *optimization algorithm*. A gradient descent optimization algorithm aims to find a local (or potentially global) minimum of a differentiable function. It is important to note that gradient descent optimization need not only be used in the context of ANNs and can be applied to many other AI approaches since any continuous problem space is differentiable (Ruder, 2016).

There are numerous optimization algorithms and many of them employ different strategies to perform gradient descent without getting stuck in local optima. One of the most well-known and straight-forward gradient descent algorithms is

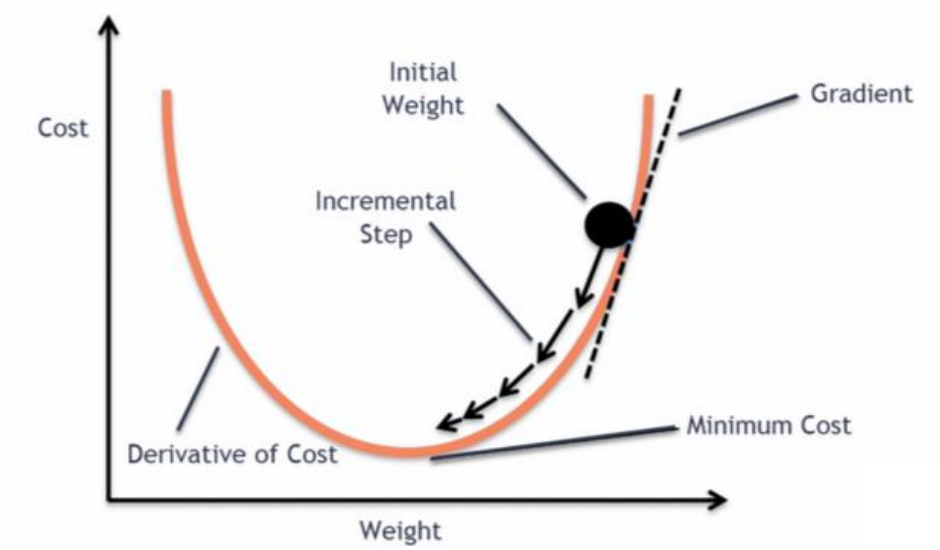


Figure 18

Diagram depicting the fundamental process of gradient descent. Note that "cost" is another term for "loss" (M, 2019).

called *backpropagation* which simply propagates the error difference between an ANN's output and its expected or optimal output (known as the *loss function*) back through the network (Agatonovic-Kustrin and Beresford, 2000; Poggio *et al.*, 2017; Tavanaei *et al.*, 2019; Fil and Chu, 2020). Gradient descent can be mathematically described by *Equation (2)* where w_0 is the current weight of an ANN connection, w_1 is the new weight, and a is the learning rate. A visual depiction of gradient descent is also given by *Figure 18*.

$$w_1 = w_0 - a\frac{\partial loss}{\partial w} \tag{2}$$

Some other commonly used gradient descent optimization algorithms include *Stochastic Gradient Descent*, *Resilient Propagation*, *Root Mean Squared Propagation*, the *Adam* algorithm, and *Adamax*. Each of these methods uses their own techniques to avoid terminating on local optima, such as exploiting stochasticity or the statistical averages of a batch's result to promote exploration of the problem space (Ruder, 2016).

The issues involved with using gradient descent optimization algorithms do not only end at getting stuck in local optima. One of the difficulties associated with error propagation is that it becomes a very challenging task in unsupervised and semi-supervised learning environments since examples of optimal outputs need to be known before-hand (Agatonovic-Kustrin and Beresford, 2000). Furthermore, if the end-goal or resultant behaviour of a sequence of actions is evaluated, it is difficult to attribute what actions contributed negatively or positively to the result. This is often the case in dynamic environments as opposed to problems such as classification tasks (Belani, Vuković and Car, 2019). Multi-objective tasks are also difficult to optimize through gradient descent as the network must usually find a Pareto-optimal solution that may not be the optimal solution for any one of the individual objectives (Zitzler and Thiele, 1999).

2.3.3 Neuroevolution

One way to escape the pitfalls of gradient descent is to leverage the problem-solving abilities of EAs as defined in *chapter 2.2* through the implementation of *neuroevolution*. Neuroevolution is the study of applying EAs to ANNs. Neuroevolution can be applied in two ways: using *direct* or *indirect* encoding. Indirect encoding entails evolving the topology of a network (the number of layers and number of nodes per layer), activation functions used, as well as various hyperparameters governing the network. The process of finding optimal connection weights is then left up to another optimization process such as gradient descent. Direct encoding evolves the network itself from the weights of

the connections to the organisation of the nodes in the layers themselves (Miikkulainen and Lehman, 2013; Hadjiivanov and Blair, 2016).

One simple implementation of neuroevolution is a direct encoding approach called *Conventional Neuroevolution* (CNE). When using CNE, the topology of the network is predefined and only the weights of the network are evolved (Gomez and Miikkulainen, 2006).

2.3.4 NEAT

One of the most popular neuroevolutionary algorithms that uses both indirect and direct encoding to evolve the weights and topology of an ANN is called *Neuro Evolution of Augmenting Topologies* (NEAT). NEAT encodes the details pertaining to an ANN in its genome. There are no longer hidden layers, just nodes in-between the input and output layers that have connections to any number of other nodes (Stanley and Miikkulainen, 2002).

The information stored in a network's genes consists of the data pertaining to all of its connections. This data includes a connection's start node, end node, connection weight, an enabled or disabled flag, and something called an *innovation number*. The innovation number is just a global variable that allows NEAT to keep track of all new connections made. NEAT aims to produce minimal topologies and hence starts with very small networks that consist of exclusively their input and output layers. These networks only grow if an augmenting mutation is evolutionary advantageous. (Stanley and Miikkulainen, 2002).

There are five mutation operators that can be used in NEAT. These mutations consist of:

- adding a new connection with a random weight,
- adding a new node along an existing connection,
- disabling or enabling a connection,
- multiplying a weight by a random factor,
- and randomizing a completely new connection weight.

When performing crossover, genes are lined up between parent networks according to each gene's connection innovation number. Crossover is then applied over the matching genes by randomly choosing the child gene's connection weight from one of the parents. If the gene in at least one of the parents is disabled, there is an increased chance that the inherited gene will also be disabled. Due to networks evolving differently, the length of the chromosomes between two parents may not match. If there are genes from parent ***A*** that are not in parent ***B,*** but parent ***B*** holds genes with a higher innovation number, then these genes are called *disjoint* genes and are simply carried over to the children produced. If there are genes in parent ***A*** that are not in parent ***B*** and parent ***B*** does *not* contain genes with a higher innovation number, then these genes are called *excess* genes. Excess genes are carried over to any children produced if they are held by the fitter parent and dropped if they are not. A diagram depicting how crossover works in NEAT can be seen in *Figure 19* (Stanley and Miikkulainen, 2002; Hadjiivanov and Blair, 2016).

Newly augmented topologies have a higher chance of being culled by the evolutionary process because the weights associated with their new connections have not had time to evolve, however, the fact that the connection is there may be valuable in the long run. To combat this, NEAT makes use of speciation to niche individuals with newly expanded topologies. Individuals that have similar topologies then have a better chance when competing against each other. Individuals within a species also make use of fitness sharing in order to ensure

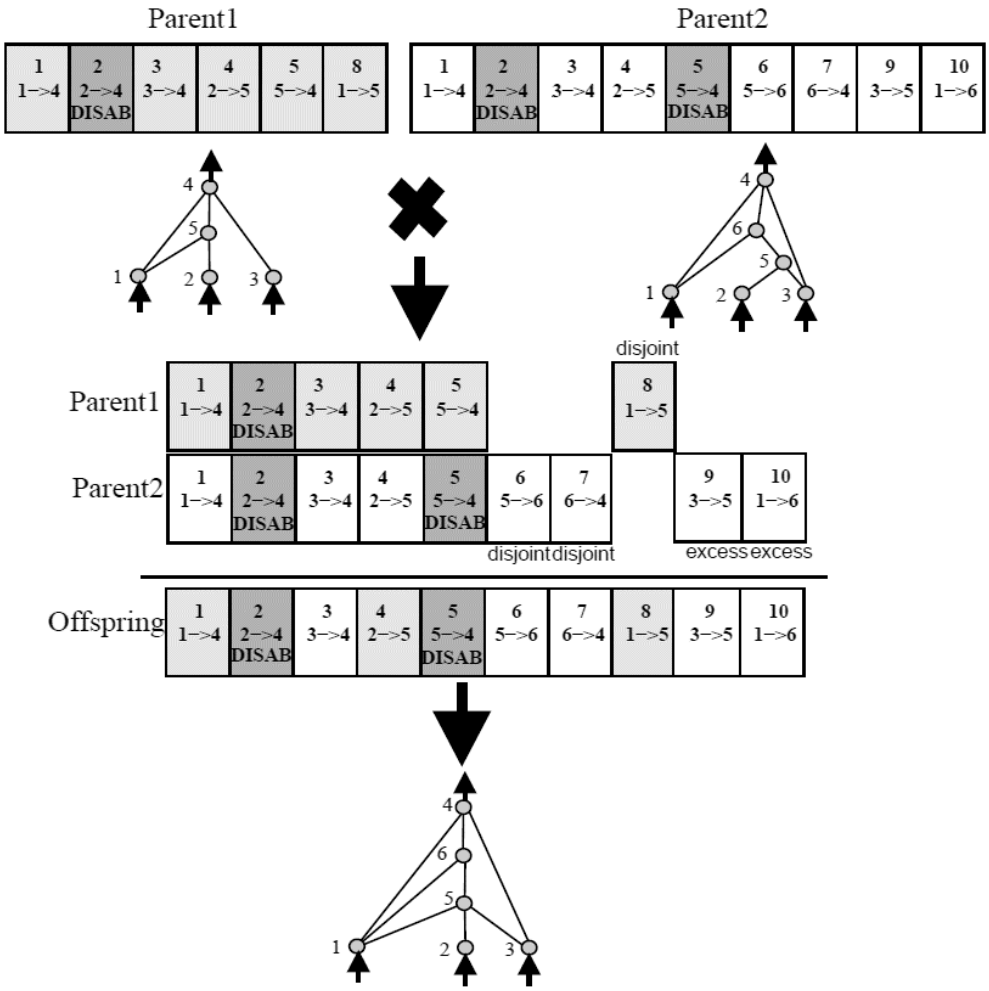


Figure 19

Diagram depicting crossover in NEAT (Stanley and Miikkulainen, 2002).

that it is unlikely a single species dominates the rest of the population (Stanley and Miikkulainen, 2002; Hadjiivanov and Blair, 2016; Acton *et al.*, 2020).

Speciation in NEAT is defined by a distance algorithm that accumulates the differences in the parameters of two networks. The distance function is calculated as shown in *Equation (3)*. *E* is the number of excess genes, *D* is the number of disjoint genes, *N* is the number of connections, and *W* is the average weight difference between genes with the same innovation number. Additionally, c_1, c_2, and c_3 are simply tuneable parameters (Stanley and Miikkulainen, 2002).

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W \quad (3)$$

2.3.5 *HyperNEAT*

Another popular approach to neuroevolution is called *HyperNEAT*. HyperNEAT extends from NEAT to be well adapted to evolving large scale neural networks and no longer uses direct encoding like NEAT does. Instead, HyperNEAT makes use of an indirect encoding method called *Compositional Pattern Producing Networks* (CPPNs) which exploits geometric patterns in the topology of an ANN's network. This is compared to the often sporadic and disorganised networks that are usually produced by NEAT (Stanley, D'Ambrosio and Gauci, 2009). HyperNEAT becomes useful when dealing with larger networks as its use of traditional gradient descent methods makes it much faster than NEAT while still utilizing the benefits of EAs to explore a problem space through the adaption of its topology (Stanley, D'Ambrosio and Gauci, 2009; Hadjiivanov and Blair, 2016).

2.3.6 *DeepNEAT*

An overarching issue with NEAT is the fact that its effectiveness quickly diminishes as network sizes increase. This is because the dimensionality of the network's own optimization space drastically increases as it mutates to become bigger. Secondly, NEAT is designed to aim for minimal complexity. This may unfortunately result in it settling on a suboptimal architecture that is smaller

than a more optimal one. NEAT may get stuck in this local optimum or take an unreasonable number of generations to begin to properly explore larger networks. Furthermore, with advancements in modern-day computing power, the development of larger networks is becoming less of an issue from a computational standpoint so the need to create minimally complex networks is no longer critical (Stanley and Miikkulainen, 2002; Acton *et al.*, 2020).

DeepNEAT strives to solve these issues by more extensively applying DNE with NEAT to DNNs. The idea centres around using a higher-level evolutionary process to optimize the parameters and structure of multiple low-level evolutionary processes. The first main difference between DeepNEAT and NEAT is that it encodes layers of nodes in its chromosomes instead of single neurons, thus moving away from the more seemingly random structures produced in NEAT to a multi-layered approach. This helps increase the efficiency of the evolutionary process as allowing arbitrary connectivity between any neurons adds additional complexity. It also makes it more likely for larger networks to develop as multiple neurons are added in a single mutation instead of just one (Miikkulainen *et al.*, 2018).

DeepNEAT further extends this idea by allowing each layer to be expressed by a different kind of network. Layers can either be convolutional, fully connected, or recurrent. Each layer also has numerous evolvable properties that include (among others) the size of the layer and activation functions used. Evolution is then applied at a higher level to the structure of the layers and simultaneously at a lower level to the weights within the layers (Miikkulainen *et al.*, 2018; Acton *et al.*, 2020).

In the same way that NEAT struggles to deal with the growing complexity of its neurons, the networks of DeepNEAT struggle to deal with the growing complexity of linked layers. To address this, *Coevolution DeepNEAT* (CoDeepNEAT) was developed. CoDeepNEAT splits the evolutionary process by

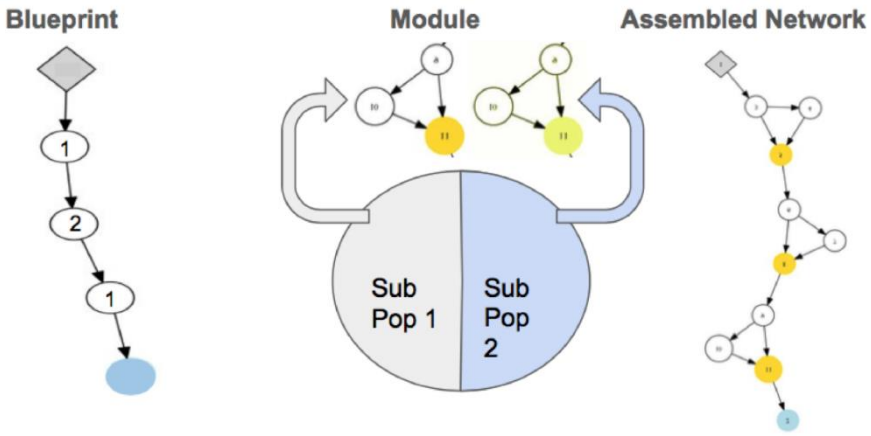


Figure 20

Diagram depicting crossover in NEAT (Stanley and Miikkulainen, 2002).

evolving populations of *modules* and *blueprints* separately. Each module is a small DNN consisting of a few layers developed in the same way as DeepNEAT while blueprints are chromosomes that map the entire network's inputs to a tree structure of different modules. This also allows certain modules to be evolved once and then used multiple times by a blueprint and hence enables the easy development of useful repeating structures (Miikkulainen *et al.*, 2018; Acton *et al.*, 2020). An illustration of how CoDeepNEAT works can be seen in *Figure 20*.

2.3.7 Spiking Neural Networks

As effective as ANNs might be, they are subject to a few limitations and are ultimately abstract representations of more complex systems that take place in biological brains. A significant issue with ANNs is that they are often rigid with regards to subtle dynamic environmental changes. This is usually a significant issue when dealing with real-world agents or when trying to port a model that has been trained in a simulation to a physical environment (Jakobi, Husbands

and Harvey, 1995; Qiu *et al.*, 2020). To address these issues, *Spiking Neural Networks* (SNNs) try to replicate the dynamic way biological entities can respond to change by mimicking biological brains in a more direct way (Tavanaei *et al.*, 2019).

Unlike traditional ANNs, biological brains send signals between neurons using trains of discrete *action potentials* (spiking voltage signals) where the frequency and timing of the spikes is significant to the neuron on the receiving end. Due to the time-sensitivity of incoming signals, SNNs have the ability to perform better than normal ANNs in time dependant environments. Furthermore, SNNs are very resilient to noise as a few erroneous spikes of information are unlikely to significantly change the average spike rate across a connection path (Tavanaei *et al.*, 2019).

The nature of SNNs makes them very well suited to real-world environments as they are time sensitive and possess the ability to deal with the dynamic nature of physical environments by being able to adapt to changes. SNNs also exhibit natural *Transfer Learning (see chapter 2.4.1)* capabilities as an agent trained in a simulated environment with an SNN is more likely to be able to adapt when moved to the real world. This ability to adjust to environmental differences or changes is known as *online adaption* as the agent is adjusting its behaviour during run-time (Qiu *et al.*, 2020).

Another benefit of SNNs is that, due to the fact that not all neurons always fire, network pathways are sparsely activated. This results in computational processing being more time and energy efficient as portions of the network that are not activated need not be evaluated (Wu *et al.*, 2019). However, this requires that the hardware or software running a SNN be implemented in such a way that it can utilize these optimizations.

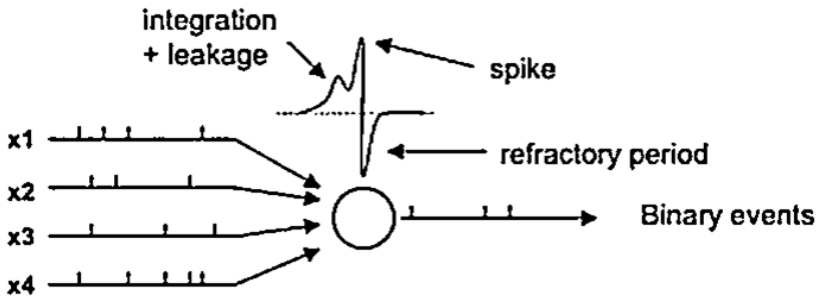


Figure 21

Diagram showing how signals are sent in SNNs (Clarke, 2018).

In a biological brain, a neuron emits a signal spike when the sum of charges in the action potential across its membrane is above a certain threshold. This process can be simulated in a SNN. There are multiple different models that approximate the complex biological process of real neurons. Some of these models include (among others) the *Spike Response* model, the *Izhikevich Neuron* model, and the *Leaky Integrated-and-Fire* (LIF) model (Tavanaei *et al.*, 2019).

One of the most popular approaches is the LIF model. In the LIF model, the current neuron's charge value is accumulated by receiving incoming signal spikes. The neuron then gradually leaks its charge after each time step that passes with higher charges leaking faster than smaller charges. This makes it more likely for a neuron to cross its threshold and fire when signals are sent to it more frequently, even if they are only small signals (Tavanaei *et al.*, 2019; Wu *et al.*, 2019).

After a neuron fires, its charge value is reset to a minimal value before it enters a *refractory period* in which the neuron will not fire no matter what happens. During this period the neuron may still accumulate charge, however, incoming signals accumulate significantly less charge as opposed to when the neuron is not in a refractory period (Fil and Chu, 2020).

SNNs are able to apply online learning rules to adjust the weights of synaptic connections in their networks. This is known as *neuro plasticity* as subtle changes in the network's weights occur depending on inputs from the environment. SNN learning through online adaption draws some parallels with gradient descent methods to some extent but also easily allows for unsupervised learning. This is because it does not require a loss function to regress against and simply works by increasing or decreasing a connection's weight over time depending on how much it is used (Qiu *et al.*, 2020).

SNN connections can be *excitatory* (which increase the strength of a signal) or *inhibitory* (which decrease the strength of a signal). SNN learning rules are predominantly categorized under a general learning rule called *Spiking Timing Dependent Plasticity* (STDP). STDP works by adjusting weights based on the relative times between spiking signals sent down a connection within a given learning window period. The idea is that if a presynaptic neuron (the neuron before a connection) fires shortly before a postsynaptic neuron (the neuron after a connection), then the connection weight will be increased (strengthened). If the postsynaptic neuron takes a long time to fire (but still fires within the learning window period), then the connection weight will be decreased (weakened) (Tavanaei *et al.*, 2019; Qiu *et al.*, 2020).

The process of strengthening a connection is called *Long-Term Potentiation* (LTP) while the process of weakening a connection is called *Long-Term Depression* (LTD). One of the most commonly used STDP rules is outlined in *Equation (4)* where w is the weight of a connection, $\boldsymbol{A}$ and $\boldsymbol{B}$ are tuneable constant parameters and T is the time of the learning window period. The variable $\boldsymbol{A}$ must be less than zero while $\boldsymbol{B}$ must be greater than zero. It is worth noting that most SNN implementations use variations of this rule so as to achieve a more efficient result depending on the task at hand (Tavanaei *et al.*, 2019).

$$\Delta w = \begin{cases} Ae^{\frac{-(|T_{pre} - T_{post}|)}{T}} & T_{pre} - T_{post} \leq 0, \quad A > 0 \\ Be^{\frac{-(|T_{pre} - T_{post}|)}{T}} & T_{pre} - T_{post} \leq 0, \quad B < 0 \end{cases} \tag{4}$$

2.3.8 Summary and Recap: Artificial Neural Networks

Many modern AI and ML solutions in place today involve some use of ANNs, often within a DNN structure. ANNs are a connectionist method to handling information as knowledge is represented in a network by the unique arrangement and configuration of the weights of the connections themselves. This allows ANNs to possess a strong ability to deal with noise and exhibit generality in a problem space (Agatonovic-Kustrin and Beresford, 2000; Poggio *et al.*, 2017; Gupta, 2020).

As the complexity of a problem increases, the necessity of "deepening" the structure of an ANN becomes more significant. Not only are the weights of a network important, but the unique structure of a network's topology can also have a drastic effect on its performance (Stanley, D'Ambrosio and Gauci, 2009). Building more advanced topologies with hierarchical levels of structures (as in the cases of DNNs) can also further assist in improving performance in complex tasks (Acton *et al.*, 2020; Gupta, 2020).

The choice of hyperparameters, activation functions and optimization algorithm can also have a drastic effect on the performance of an ANN. Despite the efforts of modern optimization algorithms, most are still susceptible to the shortcomings of gradient descent and are not always suited to dealing with multi-

objective, dynamic environments (as is often the case in real-world environments) (Ruder, 2016; Belani, Vuković and Car, 2019).

A powerful approach to ANNs that is not reliant on gradient descent methods is neuroevolution (Miikkulainen and Lehman, 2013). NEAT is a particularly significant implementation of neuroevolution, but it struggles to feasibly grow large networks. Some methods such as HyperNEAT try to mitigate these limitations by using gradient descent to optimize network weights while exploring the problem space using evolution on network topologies (Stanley and Miikkulainen, 2002; Stanley, D'Ambrosio and Gauci, 2009). Other methods such as DeepNEAT and CoDeepNEAT combat these challenges by dividing the evolutionary process into hierarchical levels to encourage the expansion and growth of networks so that they can deal with more complex problem spaces (Miikkulainen *et al.*, 2018; Acton *et al.*, 2020).

One common method of reducing an ANN's input dimensionality is by extracting feature information from input data into a latent space using CNNs which can both improve the speed and accuracy of an ANN (Karpathy and Leung, 2014; Vedaldi and Lenc, 2015).

A further extension of ANNs that more directly replicates the functionality of a human brain can been in the use of SNNs. SNNs are still able to perform in the same manner as normal ANNs except that they have the added ability to exhibit online-adaption and neuroplasticity. This makes them potentially even better than traditional ANNs at dealing with noise and their ability to deal with small adjustments makes them well suited for real-world environments (Tavanaei *et al.*, 2019; Wu *et al.*, 2019; Qiu *et al.*, 2020). Furthermore, due to the fact that neurons build up charge over time before spiking, SNNs exhibit a natural ability to retain knowledge and develop short to medium term memory as an event in the past can make it more likely for some neurons to fire in the near future. If the leaking rate of a neuron is low and the charge accumulates slowly, this ability to

draw information from the past can extend over relatively far horizons. The use of a refractory period can also assist in shaping a SNN's response to being sensitive to temporal contexts. However, it is also worth noting that SNNs often have a higher parameter dimensionality than traditional ANNs which make it harder to apply evolution to them as training times may drastically increase or optimal solutions may never even be found.

2.4 Gaining Perspective

We have addressed various methods and paradigms with regards to problem-solving using AI, however, as previously outlined, there is an aspect to problem-solving that human intelligence is capable of that is often lacking in intelligent machines. This thesis refers to this abstract factor as the concept of perspective which broadly entails looking at a problem at an appropriate hierarchical level of generalization with regards to any current or historical information. Understanding something with perspective requires contextual awareness and an ability to use prior knowledge to adapt to change. Using *Osband's* analysis of intelligence, the idea of perspective in intelligent agents can be abstractly represented by proficiency in the cognitive components of *memory*, *generalization*, and *credit-assignment*. These components of intelligence (especially credit-assignment) are areas of problem-solving that are particularly difficult for intelligent agents to handle well, despite being something that humans can do naturally (Holland and Gamez, 2009; Ke *et al.*, 2018; Osband *et al.*, 2020). Many AI approaches such as ANNs have the ability to generalize and deal with noise but struggle at handling tasks that require an active memory or temporal generality (Bhandarkar *et al.*, 2019; Badue *et al.*, 2021).

Another aspect to perspective-based learning is the idea of *intelligent* exploration. The concept of exploration in an AI's learning process is not anything novel and is often a necessary process in the search for improved solutions. However, the ability to choose which new solutions to try in an intelligent way is something that is often ignored despite being a skill that humans can do by utilizing past experiences. Furthermore, an agent should be able to generalise about its exploration and infer knowledge about some un-visited states without exploring them by simply visiting adjacent states (Gupta *et al.*, 2018). Most explorations strategies use an adaptive ϵ-*greedy* algorithm where

the chance of randomly exploring decreases as time continues, such as in the case of *ε-first* or *decreasing-ε*. Tokic introduces *Value-Difference Based Exploration* (VDBE) which increases and decreases an agent's chance of exploring based on its uncertainty regarding the current locality in its environmental state-space (Tokic, 2010). *Gupta* extends this idea of exploration to be a solvable problem in itself and introduces *Model Agnostic Exploration with Structured Noise* (MAESN). MAESN uses gradient-descent to learn how to explore intelligently. It utilizes prior experience to learn a *latent exploration space* which it can use to intelligently employ stochastic exploration based on an agent's current state in its environment (Gupta *et al.*, 2018).

In many model-based RL algorithms, an agent keeps track of an internal mapping of environmental states visited and their feedback (Watkins and Dayan, 1992). This can be seen as the agent's memory. However, this requires a potentially large database of past events and states that the agent must consider. In expansive environments the dimensionality of data stored in an agent's memory can become unmanageable (Ke *et al.*, 2018; Osband *et al.*, 2020). This is especially an issue in situations such as those experienced by space rovers as the computers used by space rovers are very small-scale and simplistic, limiting the amount of memory space and processing power available. This is due to weight and electric power limitations as well as the fact that the computer must be robust to harsh conditions and interfering cosmic rays (Cheng *et al.*, 2004; Bresina and Morris, 2007; Joyeux, Schwendner and Roehr, 2014). One way to address this issue is for an agent to be *temporally general*. This means that an agent is able to make decisions that are informed by historical events without having to explicitly store and access every individual event.

As explored by Soar, ACT-R and CLARION, there are multiple paradigms and levels in an AI's memory that can be used to store various elements of knowledge. These approaches can be differentiated as being long-term or short-term

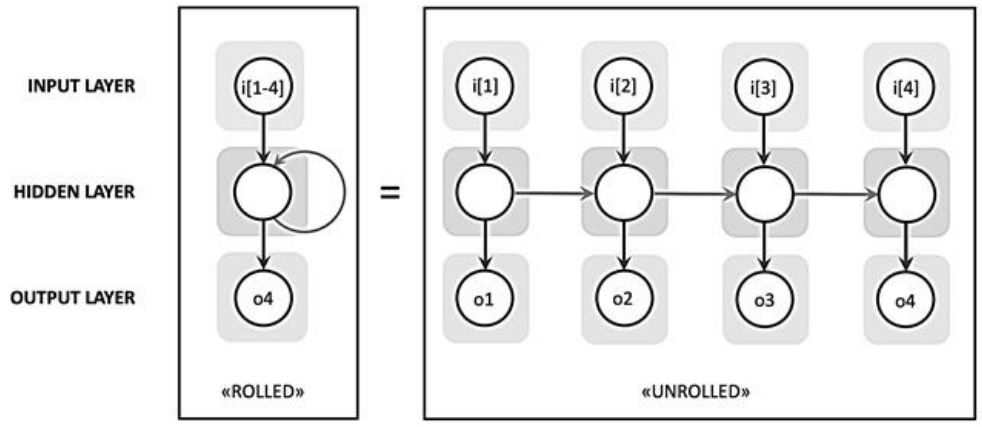


Figure 22

Diagram depicting how an RNN can be unfolded (West, 2020).

memory as well as being represented using symbolic or connectionist approaches (Scerri, 2006). As demonstrated by ANNs, connectionist methods have a unique ability to generalise and deal with noise. Furthermore, an AI that stores data in a connectionist manner can learn to compress the information into a latent space of relevant features, allowing for greater temporal generality (Agatonovic-Kustrin and Beresford, 2000; Tiu, 2020).

The simplest approach to develop short-term (working) memory using ANNs is by using RNNs. Since a RNN includes the output from its previous step in its current input, RNNs can be unfolded multiple time-steps back to access even older information. This information must somehow persist in the current state of a network along-side any current input. As a result, the depth that RNNs can be reliably unrolled quickly diminishes to only a few steps (Ke *et al.*, 2018; Tavanaei *et al.*, 2019).

A more complex model called *RNNs with Long Short-Term Memory* (LSTM) was proposed to tackle the limitations of simple RNNs. RNNs with LSTM have multiple networks dedicated to learning what information should be remembered and what information can be forgotten. This way of dealing with

memory leverages the idea of credit-assignment and only utilizes resources to remember important pieces of information so that they can exist over longer periods of time. However, LSTMs still struggle to viably deal with memory over very long horizons or in complex environments (Bhandarkar *et al.*, 2019; Osband *et al.*, 2020).

2.4.1 Hierarchical Learning

One way to approach the issues associated with long-term memory in RL while still maintaining temporal generality is to use what is called *Hierarchical Reinforcement Learning* (HRL). HRL methods approach large horizons by having different granular levels of how far an algorithm looks back. HRL also breaks problems into smaller sub-problems (sub-policies) and has different components manage and work on these sub-policies at different hierarchical levels. Some benefits of HRL implemented properly also include structured exploration of the state-space and the encapsulation of different levels of knowledge (Bosch *et al.*, 2011). A common approach to HRL is *MAXQ learning*. MAXQ learning depends on being able to break down the global goal of an environment into more achievable sub-goals and sub-tasks. However, MAXQ is not designed to be implemented in an unsupervised manner as it depends on the programmer being able to identify these sub-goals (Dietterich, 2000).

The approach of HRL need not only be applied as a means to better deal with distant temporal dependencies. HRL is also good at addressing the concept of *Transfer Learning* (TL). TL is a concept in AI that focuses on using the knowledge gained from solving one problem to solve a different but similar problem. TL can be broken down into three categories. In *inductive* TL, the target environment

(domain) and task are different from the source environment and task while in *transductive* TL the tasks are the same and just the environments (domains) differ. Finally, *unsupervised* TL is the same as inductive TL except it solves unsupervised learning tasks in the target environment (Pan, Kwok and Yang, 2008; Pan and Fellow, 2009).

A significant HRL algorithm that incorporates the concept of TL is the *Feudal Learning* (FL) algorithm which was proposed in 1993 and is inspired by Europe's medieval feudal system. The idea of FL is that there are manager components within a model that deploy tasks to worker components that in turn deploy tasks to more worker components (thus playing the role of the manager) in order to complete the task given to them by their own manager. Workers only receive reinforcement rewards based on their completion of a task given to them by their manager. This process of deploying tasks to workers can continue as many times as a particular model needs (Vezhnevets *et al.*, 2017). In 2017 FL was expanded upon to produce *Feudal Networks* (FUN) which is essentially the same idea as traditional FL except that there is only one manager and one worker. FUN represents the sub-goals given by a manager as directions in a latent space which can then be translated into actions taken as primitive sub-policies by a worker (Vezhnevets *et al.*, 2017; Nachum and Lee, 2018).

2.4.2 Meta Learning Shared Hierarchies

One method that uses HRL and draws on aspects of FUN is *Meta Learning Shared Hierarchies* (MLSH). MLSH is a model-free method that learns basic primitive policies for sub-tasks that an agent may frequently need to complete method. A higher hierarchical element of the model then learns how to use these sub-

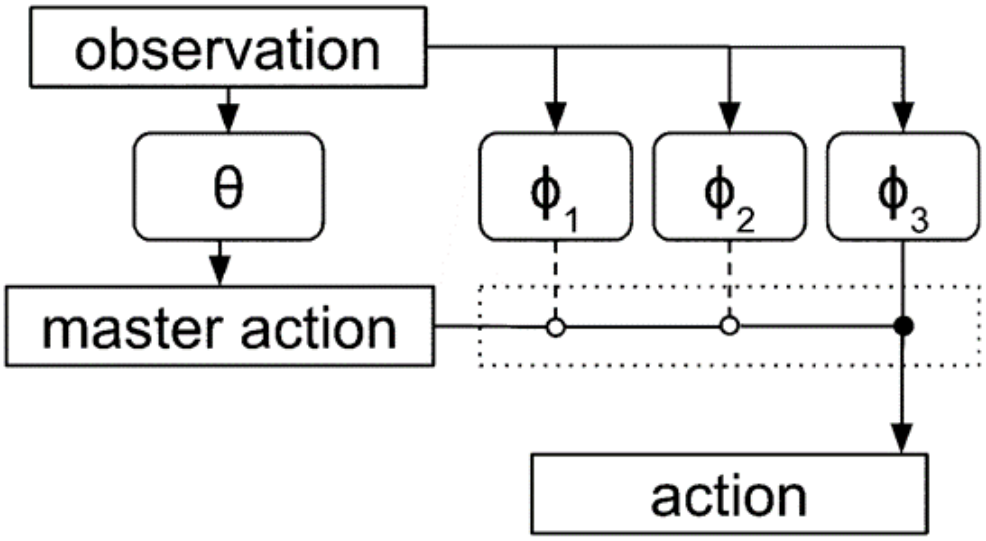


Figure 23

Diagram showing how MLSH works (Frans *et al.*, 2018).

policies to act optimally in a global policy for an agent's current environment (Frans *et al.*, 2018).

The idea of MLSH is to simply use one ANN labelled θ to select from a vector ϕ of sub-policy ANNs. The input received by the θ network and selected sub-policy network are the same. The job of the θ network is hence to learn to choose the sub-policy that is best suited to the current instance in a given environment while the job of the sub-policy ANN is to define how the agent will act (Frans *et al.*, 2018).

The sub-policy networks also learn to act optimally within their own locality. This means that an agent can be put in a new but similar MDP and simply use the already learnt sub-policies in ϕ and all it would need to do is re-learn or adjust the learnt weights of the θ network (Kocsis and Szepesvari, 2006; Frans *et al.*, 2018). This is an attempt to solve what is called the *few-shot learning* meta-learning problem which essentially entails being able to develop a deep understanding of a problem using only a limited amount of training. This

requires the leverage of previous experience and is something that humans have an innate ability to achieve (Hutter, 2014).

Traditionally, agents try to learn a single optimal policy for their current environment, however, there are often many sub-tasks that may have led an agent to its goal. An example of this kind of behaviour can be seen in nature when an animal must learn to walk before being able to learn where to go to gather food. Furthermore, the animal doesn't need to re-learn how to walk every time it looks for a different food source and learning one shared policy is often not very efficient in these kinds of cases (Turney, Whitley and Anderson, 1996; Scerri, 2006). MLSH enforces an agent's model to have a hierarchical structure that allows learning adjustments to be made to either the θ network or any of the sub-policy networks without affecting the performance of the other components. This helps an agent exhibit more general and adaptive behaviour in varying environments (Bosch *et al.*, 2011; Frans *et al.*, 2018).

MLSH also enables agents to better deal with time related tasks as it promotes an agent to focus on sub-tasks in a step-by-step manner without having to consider the entire long-term goal all at once. An agent using MLSH effectively improves its ability to tackle problems with far horizons as it can break them down into multiple problems with smaller horizons (Bosch *et al.*, 2011).

A parallel can be drawn between the different sub-policies learnt by an agent using MLSH and the non-conscious and automatic behaviours biological organisms such as humans exhibit. This kind of behaviour enables humans to focus on more complex tasks without having to learn or think about every minute action we take (Minsky, 1961; Cooper *et al.*, 1996; Turney, Whitley and Anderson, 1996; Holland and Gamez, 2009).

2.4.3 *Attention*

Further expanding the ideas surrounding perspective with regards to dealing with large amounts of current or time-based information leads to the notion of *Attention.* Attention is a concept in AI that deals with the idea that only certain inputs are significant when determining an optimal output for a given state. These relevant inputs may be dispersed temporally over potentially great horizons. The idea of Attention is often brought up in natural language processing in which select words in previous sentences give context to the current sentence (Ke *et al.*, 2018).

Attention is able to solve many issues associated with temporal generality that are difficult for other methods to effectively tackle, such as RNNs with LSTM which struggle to deal with memory over distant horizons (Ke *et al.*, 2018). HRL algorithms are able to look further back by using layers of different sized granular time-steps, but they can ignore the fact that significant events need not occur in a regular manner which may often lead to important information being missed (Bosch *et al.*, 2011). Attention focuses on learning the importance of certain events in relation to other events. This information can then be used to only select the most important pieces of historical information to be used in any further processing, thus dramatically reducing processing times.

An example of Attention can be made with the use of a *Language Model* (LM). Language is complex and multidimensional with different words and sentences relating to each other across different spaces of text and contexts. In order for a LM to predict the next word in a sentence (or in the case of an agent the next optimal action to take), certain words need to be given more Attention considering the current word and the context of the sentence (Vaswani *et al.*, 2017). A simple *Attention mechanism* essentially constructs a matrix of all words against all other words. The values inside the matrix are learnt using some

method (such as gradient descent) and define how much attention to pay to each word given another word. This can be hierarchically imposed to have Attention on groups of words, sentences, or even paragraphs (Vaswani *et al.*, 2017). While language processing is only one problem domain, the concept can be generally applied by replacing "words" with "agent inputs", and "previous sentences" with "past events".

Attention mechanisms work by encoding inputs into vectors. Three vectors are constructed. A *query* (Q) vector which represents the current input, and a set of *key-value* (K, V) pair vectors. Each query needs to be evaluated against all other inputs (keys) to determine a value vector which dictates how much attention to apply to a key given a query (Alammar, 2018). These vectors can also make use of *positional encoding* which is when the position or time of an input is relevant to its encoded vector. In the case of agents, important events that happened a long time ago may be less relevant (Vaswani *et al.*, 2017).

The extent of what Attention mechanisms are capable of is shown through the use of *Transformers* as demonstrated in the *Generative Pre-Trained Transformer* (GPT) recently created by *OpenAI.* Transformers stack Attention units on top of each other (employing the concept of HL). In GPT's second version (GPT-2), the transformer created consists of six layers of encoders and decoders. Each encoder has a *self-Attention* unit (Attention applied from the encoder to itself) and a FFNN. Furthermore, each decoder has a self-Attention unit and encoder-decoder Attention unit which is followed by a FFNN. GPT-2 and GPT-3 have been shown to be extremely good as LMs and can perform text completion with convincing context, understanding and *perspective* that may convince many people as being human (Vaswani *et al.*, 2017).

Another implementation of an Attention model combines traditional Attention mechanisms with the use of RNNs and LSTM and is called *Sparse Attentive*

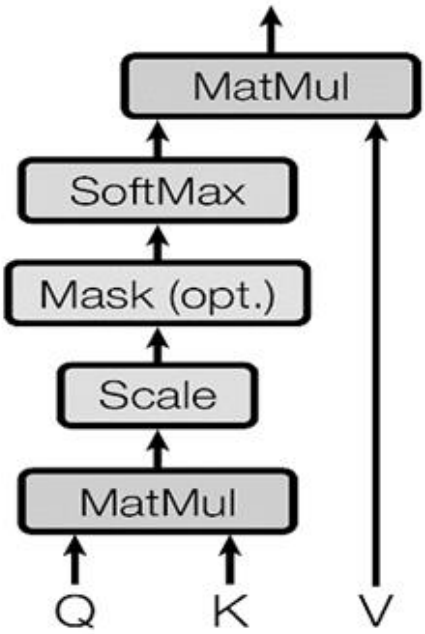


Figure 24

Diagram showing the process followed by the SDPA Attention mechanism (Vaswani *et al.*, 2017).

Backtracking (SAB). However, the experiments done in SAB's proposal show that it falls short of the Transformer in terms of performance (Ke *et al.*, 2018).

The Attention mechanism that Transformers use is called *Scaled Dot-Product Attention* (SDPA). In SDPA, the dot product of a query is computed against all keys and scaled by the square root of the number of key dimensions (d_k). A *softmax* function is then applied to find the weights of the value vector. The formula for SDPA can be seen in *Equation (5)* and is visualised in *Figure 24* (Vaswani *et al.*, 2017).

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (5)$$

2.4.4 Summary and Recap: Gaining Perspective

The ability to solve problems with a sense of perspective is necessary for any general cognitive architecture to be able to create intelligent agents that can operate in a wide variety of complex, dynamic, and time-dependant environments. This abstract concept is a significant component that differentiates strong biological intelligence as demonstrated in humans from ML and other weak AI approaches (Holland and Gamez, 2009; Osband *et al.*, 2020). Some key issues around this disparity include hierarchical memory structures as well as the ability to be general with regards to temporal information in memory (Bosch *et al.*, 2011; Squire and Dede, 2015; Frans *et al.*, 2018; Ke *et al.*, 2018). Generality in problem-solving and the ability to understand what experiences are relevant given an agent's current state (the credit-assignment problem) are also important (Vaswani *et al.*, 2017; Osband *et al.*, 2020).

A common issue that many AI solutions fall into when dealing with memory and time-related events is not addressing the fact that relevant past information can be structured hierarchically as demonstrated in HRL (Dietterich, 2000; Bosch *et al.*, 2011). Furthermore, explicitly storing data symbolically can limit generality and the ability to quickly access relevant past data (Ke *et al.*, 2018). Connectionist methods can mitigate this as approaches such as ANNs exploiting the Markov property are able to generalise by learning latent space representations of data which can be applied to an agent's understanding of its environmental state-space (Ohnishi *et al.*, 2019). The knowledge held in a network's weighted connections are representative of procedural (implicit) memory or knowledge. In the case of neuroevolution, the evolutionary process itself would be the driving factor of this kind of knowledge. Extending this, Attention mechanisms can be used to bring about a sense of semantic, declarative (explicit) and episodic memory by directly addressing the credit-assignment problem. SNNs also have

the ability to naturally implement short to medium term episodic (working) memory.

Breaking down problems into hierarchies through the use of HL not only has the ability to improve efficiency in accessing hierarchies of memory information but can also improve task completion by breaking down a global goal into more achievable sub-goals. MLSH directly addresses this by enforcing a hierarchical structure in a network and encourages the development of sub-policies that can be used when solving global policies (Frans *et al.*, 2018). It is also worth noting that Attention mechanisms can be applied in a hierarchical way to address different levels of credit-assignment (Vaswani *et al.*, 2017).

Utilizing a perspective-based approach can also be expressed in exploration. Instead of applying a static or simplistically adaptive algorithm to the process of exploration, methods such as VDBE and MAESN intelligently use exploration as needed based on the situation of an agent (Tokic, 2010; Gupta *et al.*, 2018). An extension of this idea could be to utilize the temporal and structural information provided by methods such as MLSH and Attention mechanisms to further improve intelligent exploration. Additionally, applying the approaches of MLSH, and Attention to evolutionary methods may result in better exploration of the problem space as these methods are traditionally applied using gradient descent.

2.5 General Discussion of Literature

Broadly speaking, the literature reviewed in this chapter covers a wide spectrum of general knowledge pertaining to AI. This includes unpacking what intelligence is in the context of AI and ML (as applicable to the construction of general theories of cognition) as well as various methods of implementing AI as solutions to various components that are relevant to general cognitive architectures. The cognitive architecture developed for the thesis, *Brain Evolver* (BE), is centred around creating intelligent agents using evolutionary algorithms, RL and various metaheuristics including HPO and REVAC. The different AI and ML methods reviewed generally covered ANNs, neuroevolution, SNNs, MLSH, and Attention. The literature pertaining to these approaches was explored as the outcome of the search for solutions to the various aspects of intelligence outlined in *chapter 2.1*. The purpose of reviewing ANNs and gradient descent methods along with various neuroevolutionary approaches was to survey prominent (connectionist) methods to efficient and effective general problem-solving. On the other hand, the use of literature regarding metaheuristic methods is driven by the desire to find general solutions that can explore a wide range of problem spaces in a more universally applicable way, while REVAC offers an AutoML-like solution to completing HPO.

The main analyses of intelligence explored constitute those proposed by *Sternberg*, *Thurstone*, *Minsky*, and *Osband*. The limitations of these attempts to perfectly define intelligence (and any attempt to do so) is that they are essentially abstract and ultimately just hypothetical claims (Minsky, 1961; Clarke and Sternberg, 1986; Osband *et al.*, 2020). Furthermore, the porting of a linguistic definition of some component of intelligence to an actual algorithm or piece of computer code makes it even harder to draw exact conclusions from these

analyses. Although these analyses are useful as guidelines, they should be treated as such, just guidelines.

Some well-known cognitive architectures were also briefly reviewed including Soar, ACT-R and CLARION (Scerri, 2006; Laird, 2008; Omohundro, 2008; Yliniemi, Agogino and Tumer, 2014). It is important to note that each of these projects are extremely expansive and have been in development for many years. It would hence be out of the scope of this thesis to try and replicate the depth of these cognitive architectures. The intention of this thesis is to develop a novel architecture by utilising a unique combination of AI methods. Replicating exactly what has already been done (in a general sense) is not as useful in expanding the broader field of research. The implementations of Soar, ACT-R and CLARION should hence also be treated as guidelines.

As a result, it is important for this thesis to focus on particular domains or essential features of general AI problem-solvers as well as highlight some potentially important concepts that are often over-looked, such as credit-assignment and temporal generality (Ke *et al.*, 2018; Osband *et al.*, 2020). This is where this thesis's idea of addressing these essential features by combining different prevalent AI approaches in the construction of a single architecture comes from. It also drives the purpose of conducting a broad spectrum of highly focused tests *(see chapter 3.8)* to evaluate BE, with only a few small-scale tests that represent more realistic situations (Mars rover exploration). This is because focusing on these aspects acts as a surrogate for a much wider range of complex tasks that BE (or another general cognitive architecture using similar principles) could theoretically solve (Osband *et al.*, 2020). It is important to remember that BE is just meant to be a stepping-stone in the broader field of general AI research and is intended to highlight insights that may lead to the development of more advanced, effective, and practical cognitive architectures.

In identifying some essential features of general AI, a significant aspect of intelligence highlighted by the cognitive architectures explored, as well as DL and HL, is the necessity of defining different levels or paradigms of knowledge (Scerri, 2006; Laird, 2008; Omohundro, 2008; Yliniemi, Agogino and Tumer, 2014; Vezhnevets *et al.*, 2017; Miikkulainen *et al.*, 2018). This also seems to be a key element to understanding problems with perspective alongside the credit-assignment problem and the ability to be general across time (Ke *et al.*, 2018; Osband *et al.*, 2020).

At the most fundamental level, all intelligent behaviour must learn by navigating a problem space (Newell, 1990). There are multiple ways to approach doing this, but it is important to realize that many tasks that intelligent agents encounter in real-world environments require unsupervised knowledge acquisition in POMPDPs (Belani, Vuković and Car, 2019; Jabri *et al.*, 2019; Badue *et al.*, 2021). This makes RL a natural approach to learning as it allows an agent to gather information by exploring its environment. However, applying RL as a blanket method ultimately also has its limitations as RL may not be good for certain kinds of tasks such as classification tasks or tasks that are best learnt through *regression* methods like gradient descent (Pan, Kwok and Yang, 2008; Karpathy and Leung, 2014).

Although gradient descent methods are very efficient in some cases, it is important to remember that the goal of this thesis is to find general solutions and hence, a broader approach to navigating problem spaces is needed. This leads to the use of metaheuristics as they have the ability to traverse any problem space with some guarantee of success (Lozano, Molina and Herrera, 2011). Of the metaheuristic methods reviewed (SA, ACO, PSO, and EAs), the choice of approach may depend on the task at hand. However, due to the particular wide applicability and extensive body of available research, EAs are a particularly

useful choice to use as a metaheuristic (Utzle *et al.*, 2010; Mishra, 2011; Yliniemi, Agogino and Tumer, 2014; Olague, 2016).

The limitations of metaheuristics such as EAs are that they are black-box approaches to problem-solving that do not guarantee global optimality. Due to the requirements of an appropriately large population (depending on the dimensionality of evolvable parameters in the problem space) and need to for multiple generations, they are also often more computationally expensive than other approaches (Gobeyn *et al.*, 2019). This may indeed end up being a limiting factor of BE since BE is implemented extensively with evolutionary methods. However, this is an intentional choice to directly follow the cognitive modelling approach (since biological intelligence is a product of evolution) in order to explore the potential outcomes of applying a cognitive architecture in such a manner. Since real-world tasks often entail multiple objectives, multiple niching techniques were also reviewed including fitness sharing, speciation, crowding, VEGA, and SPEA2. Ultimately, all these methods aim to uphold some level of Pareto optimality and attempt to prevent EA populations from converging on a solution to only one objective, with speciation being a particularly popular approach (Goldberg and Holland, 1988; Zitzler, Laumanns and Thiele, 2001; Stanley and Miikkulainen, 2002; Eiben and Smith, 2015).

It is necessary to identify the role of a cognitive architecture as a kind of implementation of AutoML and hence recognise the importance of HPO in the stack of problems to address when creating an intelligent entity (Cooper *et al.*, 1996; Hutter, 2014). Without, going out of scope, REVAC was explored as a viable solution for BE to tackle this issue. Like other metaheuristics (particularly when dealing with HPO), REVAC suffers from being computationally demanding and does not guarantee that it will find the most optimal hyperparameter configuration (Smit and Eiben, 2010).

In addressing another essential feature of intelligence, methods of basic problem-solving and function-mapping were looked at. This naturally led to the analysis of various connectionist NN methods due to their resilience to noise and generalisation capabilities. (Agatonovic-Kustrin and Beresford, 2000; Poggio *et al.*, 2017; Bre, Gimenez and Fachinotti, 2018). Both ANNs and CNNs are able to make use of latent spaces to learn the features of an input. However, some limitations of ANNs are that they (like metaheuristics) are also black-box approaches and can produce unexpected and undesirable behaviours if careful attention is not given to how they are trained. CNNs are also limited as they are essentially a rigidly heuristic approach to doing something that standard ANNs can theoretically do by themselves (Minsky, 1961; Karpathy and Leung, 2014; Vedaldi and Lenc, 2015; Tiu, 2020). A second limitation of simple ANNs as represented by FFNN is that they are a basic abstraction of more complex systems and processes that take place in biological brains and ignore some important aspects to problem-solving such as plasticity and time-sensitivity (Fil and Chu, 2020). However, this does not diminish the usefulness of connectionist-based approaches as a potentially fundamental component of a cognitive architecture.

In addressing some of the limitations that ANNs pose, ANNs can be expanded to evolutionary processes (neuroevolution). As a result, CNE, NEAT, HyperNEAT, DeepNEAT, and CoDeepNEAT were explored. The use of EAs on ANNs has been shown to improve a network's ability to traverse a problem space while dealing with multiple potential objectives. A common limitation of neuroevolutionary methods (as with other EAs) are their extended processing times which is especially prevalent in NEAT. DeepNEAT and CoDeepNEAT attempt to mitigate these issues without resorting back to gradient descent but are still unable to exhibit neuroplasticity or the ability to adequately assign credit to generalized time-based information (Stanley and Miikkulainen, 2002; Stanley, D'Ambrosio and Gauci, 2009; Miikkulainen *et al.*, 2018; Acton *et al.*, 2020). However, it is useful to recognise the strengths and weaknesses of each of the above mentioned

neuroevolutionary approaches when constructing the DNE method used for BE *(see chapter 3.1)*.

In an effort to further cover all aspects of intelligence, some methods to addressing time-sensitivity were explored. Expanding on the connectionist approach introduced by ANNs, these methods included RNNs, RNNs with LSTM and SNNs. It is evident from the literature that the level of time-awareness that these approaches offer is only feasible over short to medium horizons (Ke *et al.*, 2018; Bhandarkar *et al.*, 2019; Tavanaei *et al.*, 2019). However, SNNs also have the additional ability to be even more resilient to noise than traditional ANNs and employ neuroplastic behaviours. This makes SNNs a valid pathway to explore as a component of BE since there is also little research pertaining to evolutionary SNNs. A downside to SNNs is that they introduce extra parameter complexity as opposed to traditional ANN approaches (Wu *et al.*, 2019).

At this point, a few uncovered issues remain. Drawing from Soar and ACT-R's approaches to memory and sub-goal creation, these issues pertain to the lack of broad-scaled time sensitive problem-solving, credit-assignment and a method of directly addressing the hierarchical nature of many complex problems (Laird, 2008; Vaswani *et al.*, 2017; Frans *et al.*, 2018; Osband *et al.*, 2020). These challenges are the inspiration of this thesis's categorization of the perspective problem. As a result, literature pertaining to HL and Attention methods were explored. Within HL the methods FL, MAXQ, FUN and MLSH were covered which all broadly cover the idea of solving problems at different levels by creating and tackling sub-goals in order to solve a global goal, much like Soar (Dietterich, 2000; Bosch *et al.*, 2011; Vezhnevets *et al.*, 2017; Nachum and Lee, 2018). FL, MAXQ and FUN constitute prior research to MLSH which is an approach to HL that has been shown to be successful in leveraging sub-policies to solve complex problems as well as being able to demonstrate TL capabilities. A limitation of MLSH is that it

relies largely on a supervised approach to learning and is unable to organise its own sub-goals (Pan and Fellow, 2009; Frans *et al.*, 2018).

As far as addressing long-term memory and the credit-assignment problem when dealing with time sensitive tasks, Attention offers a direct solution. Not only is Attention computationally efficient, but it has been shown to achieve truly state-of-the-art results when applied in layers (HL) as well as being combined with DNNs (as demonstrated by GPT) (Vaswani *et al.*, 2017).

In summary, it was identified through the literature reviewed that in order to create a general AI problem solver or approach to AutoML, a strong theory of cognition must be developed through the creation of a general cognitive architecture. Cognitive architectures comprise of many components, however, common deficiencies in the learning methods reviewed include the ability to be general over time, assign credit to feedback, and the ability to exhibit a general sense to solving problems by hierarchically breaking tasks down. This is summed up as the notion of perspective. The literature indicates that general problem-solving boils down to function-mapping. This can be achieved through the connectionist approach provided by ANNs which can take on a more general heuristic nature by being extended to neuroevolution. However, to tackle the perspective problem, three main approaches were explored in the literature and stand out as primary methods to be tested *(see chapter 2.1.)*. These include *Attention*, *MLSH*, and *SNNs*. Using the information reviewed, these approaches can be tactically implemented by referring back to RL.

Some common principles behind RL explored include the Markov principle, Monte Carlo methods, BPO, QL, and DQL. However, QL is limited in POMDPs and large problem spaces with distant horizons as it must discretize and remember all states (Helman, 1986; Kocsis and Szepesvari, 2006; Hunt, 2010; Ohnishi *et al.*, 2019). DQL tries to address this by using ANNs to map to latent QL states while intelligent exploration methods such as VDBE and MAESN are

better able to traverse problem spaces (Watkins and Dayan, 1992; Tokic, 2010; Mnih *et al.*, 2013; Gupta *et al.*, 2018; Ohnishi *et al.*, 2019).

As proposed solutions, the use of EAs and the methods of Attention, MLSH and SNN reviewed in literature can be applied to the QL or DQL framework. In order to efficiently traverse the problem space to map potentially multi-objective tasks to latent QL states, DQL can make use of DQN through DeepNEAT networks instead of traditional ANNs. This can be extended further to include evolutionary SNNs. Another limitation of QL is its use of a reward discount factor (Dietterich, 2000). Instead of trying to remember diminishing past and potential future rewards over distant horizons, an Attention matrix can substitute for a table of learnt Q-values. This process can then also be applied hierarchically using MLSH. In this way the principles of RL can be applied with the hierarchical and time sensitive attributes of Attention, MLSH and SNNs in order to effectively tackle the perspective and credit-assignment problem in a paradigm that allows for the development of intelligent agents. Furthermore, a custom method of exploration that utilizes the above-mentioned approaches can be used to potentially improve performance over other exploration methods. The details behind the construction of these components and the custom exploration method used in the development of BE as well as the tests conducted are outlined in *chapter 3*.

Chapter 3

Methods

The purpose of this thesis is to develop a general cognitive architecture that has the ability to create adequately robust intelligent agents regardless of their target environment. This cognitive architecture is expressed in the creation of the software *Brain Evolver* (BE) which uses a model-free approach to learning. BE is designed to be completely customizable and modular so that the user can create their own unique models and test them on a suite of in-built or custom-added environments. This makes BE a tool that can be used beyond the scope of this thesis by other people wishing conduct similar research. BE is also written with an open copyright which allows others to modify or add to its source-code.

The programming language that BE is written in was chosen for performance reasons. The project uses C++ and was compiled with the MinGW-w64 compiler. All project files can be found at the link in the *appendix*. As suggested by the compiler, BE is a 64-bit program which is useful if a user intends to run large-scale tests that require a lot of RAM on a more powerful system.

Like Soar and other software implementations of cognitive architectures, BE's architecture is ultimately just a hypothetical claim to the effectiveness of its particular design as a general problem solver. In other words, there is no mathematical standard or proof that allows us to say that a piece of software has definitively achieved the goal of being a truly general cognitive architecture.

Based off the cognitive modelling approach and literature reviewed, this thesis outlined certain aspects and components of what makes something potentially intelligent. As with other cognitive architectures, a wholistic approach to

theorising about general intelligence requires multiple different architectural parts to work together. The way these elements work together may result in emergent useful behaviours and characteristics. There is little research on the merging of different AI approaches to form cognitive architectures (as opposed to research on the approaches in isolation) which is likely due to the fact that cognitive architectures often take long times to develop, and it can be difficult to do research on such a general topic. BE's theory about what components are necessary for general intelligence is hence unique and its particular combination of AI methods offer a novel avenue to explore with regards to general problem solvers.

Before defining BE's architecture (as alluded to in *chapter 2.5*), it is worth recapping the directives behind its design as the separation of intelligence into two paradigms. The first paradigm pertains to the method (or methods) by which an intelligent entity comes to learn what it knows. This thesis identifies two main ways that this occurs in nature. The first method involves learning through experience by exploring what is not known and exploiting what is known. This can be described by RL. The second method involves an adaption of generational knowledge through the use of evolution which can be represented by the physical expression an entity's phenotype as well as the development of neural structures through neuroevolution (representing instinct). Extending this idea practically to machines also highlights using evolutionary Hyperparameter Optimization (HPO) as part of an AutoML process that further places the development of intelligent behaviours on BE instead of a human designer. Implementing HPO with an EA would simply use an evolutionary method to optimize the hyper parameters of a single Deep Neural Evolutionary (DNE) run within BE. In this context, REVAC (Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters) would fulfil this role (see *chapter 3.2*).

The second paradigm involves the resultant intelligent abilities of an entity once it has learnt to act in an environment. These abilities can be broken down into the seven core capabilities of strong intelligent agents as outlined by *Osband* which include *basic task completion*, *exploration*, *scalability*, *noise reduction*, *generalization*, *memory*, and *credit-assignment* (Osband *et al.*, 2020). An extension of these capabilities identifies the necessity of combining memory and generalization and opens up a new categorization around the ability to be *temporally general*. Furthermore, the idea of exploration can better be defined as *intelligent* exploration as it is insufficient to occasionally act randomly but is rather more efficient to direct exploration based on experience. Lastly, it must be recognized that most modern implementations of AI involve some level of Deep Learning or at least a recognition that knowledge and information are often hierarchical.

Based on this summation of the literature reviewed as a guideline to BE's construction, a hypothesis was settled upon regarding a particular architecture for BE based on the scope, limitations, and goals of this thesis. These goals focus on the use of *evolutionary RL* to construct a cognitive architecture and its ability to be *general* across different types of tasks by dividing intelligence into definable components. This goal is further focused on the relevance of each of BE's subsequent parts (modules) in the context of its entire architecture, as well as highlighting the concept of temporal generality while avoiding explicitly storing all historical information. More details are given regarding these goals in *chapter 1.2*.

BE's architecture consists of six core modules. For the basis of the architecture, all agents are fundamentally built on a custom implementation of *DNE*. All of the other modules are optional and can be turned on or off in different combinations based on the user's choice. These other modules include *REVAC*, an *Attention* unit, *SNNs* (Spiking Neural Networks), *MLSH* (Meta Learning Shared

Hierarchies) and an intelligent *exploration* unit *(see Figure 25)*. The way that the components connect or join together can also be adjusted to some extent by the user. Furthermore, the structure of BE's software itself can also be broken down into five modules. These modules include a testing *environment interface*, a *user interaction* framework (GUI), a *graphics engine*, a *saving and loading* system, and the cognitive architecture itself. A detailed outline of BE's structure and implementation as a piece of software can be seen in the *appendix*.

The overall process that BE uses in order to learn, along with a general overview of its architecture can be seen in *Figure 25*. Hierarchically, REVAC is at the highest level. REVAC generates a population of potential hyperparameter solutions which are each evaluated under a general evolutionary process centred around DNE. Each individual in a DNE population represents a potential agent. For each step within a test environment's simulation, an agent must first poll input from its environment. This data is then passed through the agent's Attention unit (if enabled) before the output of the Attention unit is fed into the agent's main θ network *(see chapter 3.3)*. If MLSH is disabled, the θ network's output determines the agent's next action in its environment. If the exploration unit (which is attached to the θ network) is enabled *(see chapter 3.6)* and the agent decides to explore (rather than exploit what it already knows), it will instead perform a random action within the environment. If MLSH is enabled and the agent is not exploring, then the next action of the agent is decided by the MLSH module. Lastly, the actions performed by an agent result in feedback from its environment before the cycles starts again.

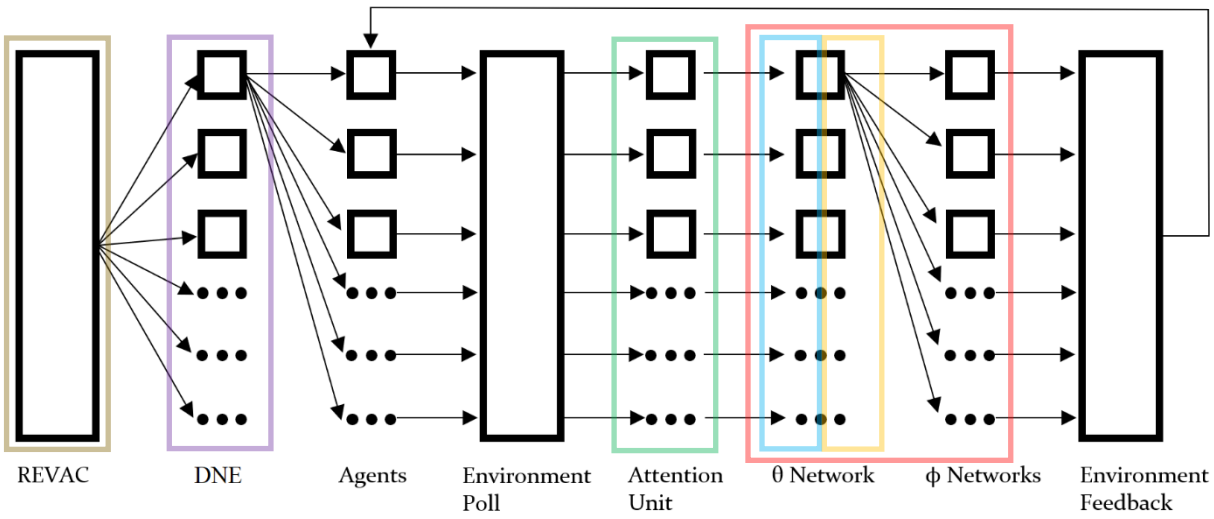


Figure 25

Diagram showing the whole process of how Brain Evolver (BE) works as well as its general architectural components. Each component is colour coded. Brown represents REVAC, DNE is purple, green is the Attention unit, blue represents an agent's main DNE network or SNN, yellow is the exploration unit, and MLSH is red.

3.1 Module One: DNE

The evolutionary process of BE's DNE implementation is represented by the *purple* highlighted column in *Figure 25* while the standard DNE networks produced for each agent by the evolutionary process are represented by the *blue* area in *Figure 25*. The choice of DNE as the core module for BE is founded on the fact that high-level biological intelligence is largely neurally based and the formation of these neural structures as well as their base-level of understanding (instinct) comes about through evolution (Turney, Whitley and Anderson, 1996). DNE is based on DNNs (Deep Neural Networks) which are good at being able to generalise, deal with noise and perform basic function-mapping (task completion). The addition of evolution to ANNs improves the ability of the

network to effectively explore the problem space while "deepening" networks improves the ability of the network to understand different hierarchical levels in data, thus enabling more complex tasks to be solved. For these reasons, DNE should be able to address (to varying degrees) the requirements of *basic task completion*, *scalability*, *noise reduction* and *generalization*, as outlined in *Osband's* analysis of intelligence (Osband *et al.*, 2020). Basic task completion, noise reduction and generalization are directly addressed by the ANN of an agent while adding additional deeper layers improves task completion and scalability. Furthermore, the use of evolutionary methods also assists in an agent's ability to potentially deal with multiple objective environments (Stanley and Miikkulainen, 2002; Miikkulainen *et al.*, 2018; Acton *et al.*, 2020).

The implementation of DNE in BE is, like DeepNEAT, based off the original implementation of NEAT (Neuro Evolution of Augmenting Topologies*)* as proposed in 2002 (Stanley and Miikkulainen, 2002). Everything is implemented in the same manner as the traditional NEAT algorithm except that there are a few alterations and additions.

The speciation process is done dynamically at each generation. The user has the option to set a cap for the number of different species that can exist at one time. This is to prevent the number of species becoming too diverse and unnecessarily slowing conversion times. Users can additionally choose a portion of the population to be dedicated as elite individuals. Furthermore, the parent selection method used for each generation can also be selected from a choice of linear, exponential, proportional, or tournament selection.

The main difference between NEAT and BE's DNE is that it no longer works with individual neurons, but layers of neurons. This is similar to the way DeepNEAT functions (Miikkulainen *et al.*, 2018; Acton *et al.*, 2020). However, all nodes and their parameters within a layer can be mutated individually. Each network node must reside in a layer whilst new layers come about through mutation. When a

new layer is mutated, it is always added as the final layer before the output layer. A set of random neurons are initiated for the new layer with the number of neurons being equal to the floor of the number half-way between the size of the previous second-last layer and the size of the output layer. Connections are then formed as to maintain the network in a fully connected state. All input connections to the new layer will hence include all connections that previously entered the final layer. All output connections and any additionally required input connections are initiated with their weights set to one so as to keep the performance of the network relatively consistent given mutations.

Once a new layer has been mutated, further structural mutation within the layer itself can occur by adding a new node to the layer or removing a node (if there are at least two nodes in the layer). The enforcement of fully connected networks and topological mutation of layers encourages deeper networks to evolve faster. Furthermore, if connections are not needed, they can always evolve to be turned off (in the same way as traditional NEAT) or have a weight of zero. This is effectively the same as a connection not being there, but the fact that it *is* there presents more avenues in the problem space for the evolutionary process to operate on.

Each node also has the ability to evolve its own activation function as well as something called an *activation offset*. The activation offset simply takes the sum of all input signals, offsets the value by some relatively small amount (between negative one and one) and then feeds the result into the node's activation function. This horizontal transformation of the activation function can become useful as it increases the inclination of some functions such as the binary function to produce drastically different outputs. The functions that a node can evolve to include the *identity*, *binary*, *ReLU*, *Leaky ReLU*, *sigmoid*, and *TanH* activation functions as outlined in *chapter 2.3*. All node output signals and connection weights are constrained to be between negative one and one. It is also worth

noting that the sigmoid function is parametrized in the same way as the original NEAT paper, as seen in *Equation (6)*.

$$sigmoid(x) = \frac{1}{1 + e^{-4.9x}} \tag{6}$$

Unlike DeepNEAT and CoDeepNEAT, BE does not make use of different types of layers. The lack of convolutional layers is due to scope restraints for the project. This opens potential for growth in BE's software as the literature reviewed highlights the benefits of CNNs as outlined in *chapter 2.3.1.* However, it is still possible for convolutional-like layers to evolve naturally due to the fact that nodes can evolve different activation functions. Secondly, recurrent networks are not utilized in BE's implementation of DNE. This is because recurrent networks cannot be reliably unfolded many time-steps into the past and add a level of unnecessary complexity to a network (Bhandarkar *et al.*, 2019; West, 2020). Instead, contextual historical data is directly addressed using BE's Attention unit.

Due to these deviations from the original NEAT implementation, there is no longer a need to keep track of innovation numbers. Since new layers are always added at the end of the network and nodes are added or removed based on their position in the layer, the structure of a network indicates its maturity while innovation data about the nodes themselves are implicitly held in their position in the network. This approach is partially inspired by the importance HyperNEAT places on positional, structural, and topological data.

Speciation also works differently to NEAT. Individuals are bucketed into species based on a hash-key that is generated by the particular structure of their DNE networks. Every unique network structure gets its own unique hash-key, so individuals that have the same topology will be put in the same species. However, since the user can place a cap on the number of species that may exist at once, a distance function is used to place individuals with uniquely structured networks

into species that are the closest to them when the species cap has already been reached. The distance function between two individuals is calculated by totalling the difference in the number of nodes across the entirety of both networks. However, nodes are only totalled in one individual if the layer or ϕ sub-network (if MLSH is enabled) exists and so differences on a hierarchically higher level are more significant. Furthermore, in order to prevent the topology of some individuals from growing too rapidly and the variation between species from becoming too sparse, individuals are only allowed to make topological mutations if they are part of a species larger than a certain adjustable size. Networks can also be limited by the user to not have more than a particular number of hidden neurons if processing times become too much of an issue.

Due to the fact that innovation numbers no longer exist, BE's DNE recombination method uses crossover that simply works by lining up nodes and connections by their position. Any excess networks (if MLSH is enabled), layers, or nodes within a layer are copied over to a child individual if they come from the fitter parent.

The evolutionary process itself can be terminated by a user defined generation cap or be set to automatically stop if no new best solution has been found after a certain number of generations. When testing the individuals of a generation in an environment, the user can choose how many times to repeat the testing process before averaging the fitness across all repetitions. More details on the adjustable parameters and settings of BE can be seen in the *appendix*. If the fittest individual of a particular generation is equal to or fitter than the best seen individual so far, it is repeatedly tested again by a different amount as chosen by the user. This allows the population to be quickly evaluated by a smaller number of repetitions while any candidate best individuals are still thoroughly tested before they may be accepted as being one of the best solutions seen.

3.2 Module Two: REVAC

REVAC is used to perform HPO and is represented by the *brown* highlighted section in *Figure 25*. Although an optional component, it is still an important part of BE's full architecture as it enables the parameters of a run to be adjusted so as to suit any environment or set of environments being tested. This facilitates the process of AutoML, alleviates the burden of configuration from the user, and assists BE's independence as a cognitive architecture.

Due to the intended flexibility of the program, the number of hyperparameters governing a particular configuration of BE can consist of up to 50 different tuneable values (depending on the user's architectural configuration) that must each take on a value between zero and one. This is quite a large number of parameters, and it can further increase to 200 independent values if online self-adaption is enabled.

Online self-adaption is done by using four values for each parameter, which can themselves be optimized through REVAC. The first value is the parameter's starting value which is also its constant value if self-adaption is not enabled. The second, third and fourth values define a function (as shown in *Equation (7)*) that adjusts a configuration's i^{th} parameter value (p_i) based on the change in MBF (as a percentage between zero and one) every n number of DNE generations, where n is specified by the user and is not included in the optimization process of REVAC. In the equation, the value x represents the percentage change in MBF. The range for a_i and b_i is between negative one and one and the range for c_i is between zero and one.

$$p_i = a_i|x - b_i| + c_i \tag{7}$$

Due to time constraints and the increased hyperparameter dimensionality threatening out-of-scope processing times, self-adaption was not enabled in the tests conducted for this thesis. However, the feature remains present in the program for others who may wish to experiment with it.

The implementation of REVAC itself works in the same way as how it is outlined in *chapter 2.2.5*. However, an additional element was added to the algorithm at the mutation phase where there is a user defined chance of an entirely new solution being created with completely randomized parameters. This is to mitigate REVAC from potentially getting stuck in a local optimum if its initial population has an "unlucky" randomization of parameters. This is a possibility due to the fact that REVAC's population cannot be too large without running into unreasonable processing times.

It is worth noting that REVAC does not operate on program settings that change the overarching architecture of BE or the fundamental way that the software runs. Those settings must be adjusted by the user. For a list of hyperparameters as displayed in BE's settings GUI and further descriptions on what each of the hyperparameters are responsible for changing, please refer to the *appendix*.

3.3 Module Three: MLSH

MLSH is represented by the *red* block in *Figure 25* and enables BE to implicitly generate sub-policies and sub-goals for itself in the context of its global goal. The enforcement of hierarchical structure has the potential to assist and improve performances in most of the aspects of intelligence as defined by *Osband* but is essentially focused on *basic task completion, scalability, generalization,* and (to a

smaller extend), *noise reduction* and *temporal generality* (Osband *et al.*, 2020). By leveraging principals of HL (Hierarchical Learning), MLSH should hypothetically be particularly good at assisting in BE's ability to be *scalable* when solving bigger, more challenging problems as opposed to just smaller tasks. This is also the reason why MLSH should (indirectly) assist in the other aspects of intelligence as it enables problems to be broken down into more manageable sub-tasks.

MLSH in BE is implemented dynamically by using an evolutionary approach. The hierarchical structures formed are generated in an unsupervised manner and are figured out by the evolutionary process itself. This is different to the original MLSH implementation which largely requires the overall structure of an implementation to be defined by the programmer. Furthermore, all networks within the MLSH module are evolved simultaneously in contrast to traditional MLSH which trains all ϕ networks first before training the θ network.

The main DNE network of an individual is referred to as the individual's θ network. In BE's dynamic implementation of MLSH, an individual is first initiated with a θ network and one ϕ network. Mutation and crossover are then applied to each individual at both a network and meta-network level. An individual can mutate to include another new ϕ network (if it is part of a species that is sufficiently large enough to allow topological mutation). The new ϕ network is given an ID based on its order of appearance within the individual (similarly to NEAT's innovation number system) and a new output node is added to the θ network with randomly weighted connections generated so as to maintain full connectedness. Networks that share IDs across individuals perform crossover within their isolated networks while excess networks from one parent are inherited if they come from the fitter parent. This allows the structure of networks to dynamically grow without adding additional hierarchies that do not benefit the population.

The output from an agent's main θ network decides which sub-policy ϕ network to use before the input received by the θ network is duplicated for the chosen ϕ network. The output from the ϕ network then constitutes any actions made by the agent for its current environment simulation step. Like the original implementation of MLSH, BE also uses *one-hot-encoding* to poll ϕ networks (Frans *et al.*, 2018). This means that only the ϕ network corresponding to the strongest output of the θ network is used to dictate an agent's action. Further additional research could hence explore the possibility of using weighted polling of ϕ networks which was also alluded to in the original MLSH paper (Frans *et al.*, 2018).

3.4 Module Four: Attention Unit

Attention is used as BE's main method of introducing temporal contextuality and is highlighted as the *green* column in *Figure 25*. Despite the Attention unit also being an optional module, its purpose is to tackle the challenges surrounding *memory*, *temporal generality*, and *credit-assignment* as outlined in *Osband's* analysis (Osband *et al.*, 2020). Furthermore, the Attention unit's output has a lower dimensionality than the agent's cumulative input across time, allowing it to generate a time-based latent space in a similar way to how convolutional layers perform feature extraction.

BE's Attention unit includes just one layer of self-Attention that applies Attention from the input of an agent's current step to the input received across all previous steps. Although stacking multiple Attention layers hierarchically (as in the case of Transformers) may have the potential to produce better results, the choice to only use one layer was purely based on scope limitations. Adding more layers not

only requires more software development, but the additional dimensionality would further increase processing times. However, should the addition of the single Attention unit prove effective, then its use warrants further development and acts as a proof-of-concept for its inclusion in BE's architecture.

The Attention mechanism used is SPDA and despite the fact that the unit only consists of a single encoding layer, it should enable an agent to remember important inputs that happened on previous time-steps. All values for each individual's Q, K, and V vectors are optimized through evolution which is an approach to Attention mechanisms that is sparsely researched. During mutation, each value across all three vectors can change to a brand-new value or shift by some offset. Furthermore, crossover is applied by lining all vector values up linearly between two individuals.

There are two options for how Attention is utilized in BE. The first option passes environmental input into the Attention unit and then passes the output of the Attention unit into the θ network. The second option includes both the original sensor input from the environment and the output from the Attention unit in the θ network's input. Lastly, the number of steps back in time that the Attention unit considers is also an adjustable parameter which can be modified by the REVAC algorithm. The inclusion of this setting was added because extending the look-back window too far for certain problems may unnecessarily increase the dimensionality of the evolutionary landscape.

3.5 Module Five: SNNs

SNNs allow BE to directly apply the cognitive modelling approach by mimicking the way human brains work in a more accurate way than traditional ANNs and is also represented by the *blue* column in *Figure 25* as it integrates itself into the way the standard DNE θ network module works. SNNs enable the creation of agents that are able to exhibit online adaptation and neuroplasticity which is useful when dealing with real-world environments. SNNs add to the capabilities of BE's custom DNE implementation by improving a network's ability to deal with *noise* as well as assist in *memory* and *temporal generality*. Due to the reduced complexity (compared to other implementations) and generally wide applicability, the approach that is used for BE's SNN implementation is the LIF model in conjunction with the STDP rule as defined by *Equation (4)* in *chapter 2.3.7*.

When SNNs are enabled, the overall structure of BE remains the same while the manner in which individuals' networks are evaluated changes. Each node includes additional parameters (as required by the LIF model) for its *resting value*, *spiking threshold*, *refractory period*, *inhibitory factor* for incoming signals during the refractory period, and *leaking rate* per time step. The activation functions used in the normal implementation of NEAT are no longer present as neurons simply fire a signal when their charge threshold is crossed. Each connection also includes additional parameters that assist in online adaption. These include a *learning window* for plastic adaptions, *LTP* rate, and *LTD* rate. All these additional parameters are then mutated and recombined in the same manner as the other parameters used in DNE.

Structurally, the user has two further configuration options. If both MLSH and SNNs are enabled, the user can choose whether SNNs are only implemented in an agent's θ network or if they are used in all networks of the individual. Using

SNNs in only the θ network allows an agent to exhibit plasticity in hierarchically higher-level decision making while maintaining rigidity in its learnt sub-policies.

3.6 Module Six: Exploration

Exploration of any environment's problem space takes place at multiple hierarchical levels within BE. REVAC is responsible for this at the highest level while the evolutionary processes that takes place within each component of an individual's architecture employs exploration at a lower level. These evolutionary processes allow the exploration and development of implicit (procedural) knowledge whilst leveraging niching techniques including speciation and fitness sharing to improve the effectiveness of the algorithm's exploration. Furthermore, the use of evolutionary methods allows BE to tackle multiple potential objectives depending on the tests run by the user. Fine adjustments that affect the way exploration takes place can be set by the user or adapted by REVAC and include (among others) the selection method (not REVAC adjustable), elitism factor, mutation chance, and crossover chance. A full list of all adjustable parameters can be seen in the *appendix*.

At the lowest hierarchical level, BE's agents are also able to intelligently employ exploration strategies within their environments. The exploration strategy devised for BE is called *Guided-ϵ* exploration (GEE) and is a component in BE's learning process that can be enabled or disabled by the user. The GEE module is indicated in *Figure 25* as the yellow column as it is attached to the main DNE θ network. GEE is inspired by MAESN as defined by *Gupta* (Gupta *et al.*, 2018). It works by incorporating the decision to perform exploratory actions as part of the functioning of an agent's θ network by utilizing an additional output node that

is simply added so that it is fully connected to the rest of the network. A further parameter is then assigned to an agent which acts as an exploration trigger threshold. If the value of the additional output node is greater than the value of the exploration trigger threshold, then any ϕ networks that may exist are not evaluated and all the agent's actuator outputs are given random values. This places the decision to explore on the agent itself and bases it on contextual information from the agent's environment. This contextual information can be supported by data passed on from the Attention unit. The latent exploration space is then explored by the evolutionary processes acting on the agent's architecture.

3.7 Summary and Recap: Methods

To summarise the methodology behind this thesis, a unique and novel general cognitive architecture was constructed based off the necessary components and processes of natural intelligence as guided by the literature reviewed. Part of the intent of identifying intelligent processes into different components also directs the construction of the general cognitive architecture (BE) as a modular piece of software so that it can be flexible for other users. This is due to the fact that the methods behind constructing BE are important as the program itself is a prime contribution of this thesis.

The six core modules for BE including DNE, REVAC, MLSH, Attention, SNNs, and the custom exploration unit (Guided Epsilon Exploration) GEE were developed and built into BE's general software structure. The uniqueness in these implementations is further demonstrated in the flexibility by which they can be

Table 1

Table showing the expected task performance focus of each module.

Focus Task / Module	DNE	MLSH	Attention	SNNs	GEE
Basic Task Completion	✓	✓	-	-	-
Exploration	-	-	-	-	✓
Scalability	✓	✓	-	-	-
Noise Reduction	✓	✓	-	✓	-
Generalization	✓	✓	-	-	-
Memory / Temporal Generality	-	✓	✓	✓	-
Credit-Assignment	-	-	✓	-	-

linked, parameterised and controlled by BE's software, as well as the fact that everything makes use of evolutionary methods as opposed to gradient descent.

Based on the hypothesis defined in *chapter 1.2* and literature reviewed, some modules are also expected to have certain performance benefits with different types of tasks, such as the Attention and SNN modules' potential ability to add contextual temporal generality and MLSH's ability to target hierarchically more complex tasks. However, these predictions needed to be examined in the testing process. The unique aspects of intelligence that each module is *intended* to target can be summarised in *table 1*. The REVAC module is not included in the table as it simply facilitates the process of AutoML by targeting HPO.

Tests were formulated as part of the methodology process in order to examine how well BE can perform at being able to generally solve a wide range of different kinds of tasks by itself (see *chapter 4*). The tests were also conducted with different architectural configurations to gain insight into the impact that each of the modules have on different kinds of tests on their own or in combination with other modules. The intended outcome of the methodology used is to ultimately make a contribution to the broader field of general problem solvers and AutoML with the analysis of BE's varying evolutionary methods and their potentially cohesive union.

Chapter 4

Experiments and Results

4.1 Experiments

In order to evaluate BE's effectiveness as a general cognitive architecture, a wide range of testing environments that target varying fundamental aspects of intelligence were created. These tests are split into three categories. The first category consists of a set of standard test functions (since problem solving can essentially be broken down to function mapping) while the second category includes a collection of common RL benchmark tests that have been directly referenced from *B-Suite*. B-Suite (short for *Behaviour Suite for Reinforcement Learning*) is the title of the original paper outlining the seven core capabilities of intelligence defined by *Osband* and is an assortment of testing environments designed to specifically target these aspects of intelligence (Osband *et al.*, 2020). Each environment was recreated for BE's custom environment interface and adjusted to fit the needs of the program whilst also incorporating user-friendly visualisations. Lastly, the third category includes test environments that abstractly represent some basic tasks that intelligent agents acting as Mars rovers might encounter.

For a complete outline of the parameters and configurations used for all tests, see *table 7*, *table 8*, *table 9*, and *table 10* at the end of *chapter 4.1.4*.

4.1.1 Test Functions

The test functions used for BE include 20 different functions that were largely sourced from *Mishra's* collection of important test functions (Mishra, 2011). The test functions can either be tested in 1D or in 2D for a total of 40 test functions. Since the rewards of a test function at all points are a dimension itself, the visualizations of the 1D test functions are in 2D and the visualisations of the 2D test functions are in 3D.

All tests with 1D and 2D test functions work in the same way. When a test function environment is initiated, all individuals are given random locations across the surface of the function. Each individual then gets ten steps to figure out how to get to the lowest point in the function. At each step, an individual can jump to any other location in the function within the specified bounds of that function (which have been manually chosen). The inputs received by the agent include their current and previous coordinates as well as their current and previous rewards from the function. Also included in an agent's input is how many steps are remaining for the current test cycle. The fitness of the agent is then only evaluated on the final step and is judged to be the agent's current position in the function.

There are also two additional function-based environments that can be selected by the user. These include *"Random 1D Functions"* and *"Random 2D Functions"*. As their names suggest, these test environments simply initiate a new random function for the start of each simulation cycle. This tests an agent's ability to learn a general rule for solving functions instead of a rule that solves a specific function.

A list of all the test functions used can be seen in *table 2* along with a 1D and 2D depiction of each function (generated by BE's graphics engine) in *table 3*. Note that for the 1D implementations, *y* is defaulted to zero.

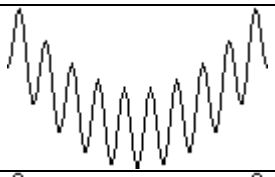
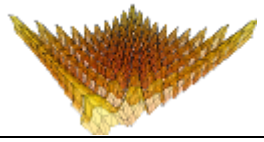
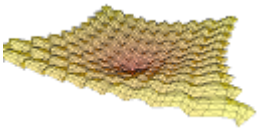
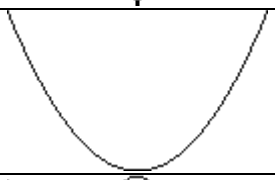
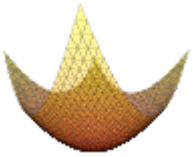
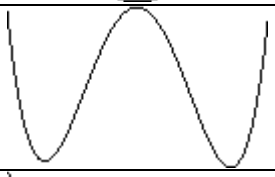
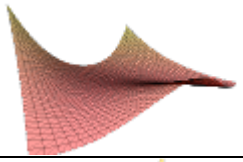
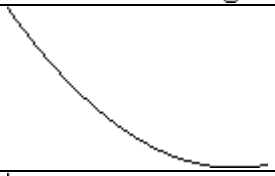
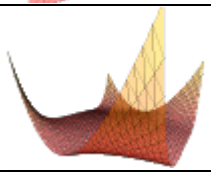
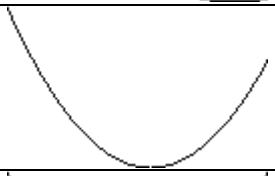
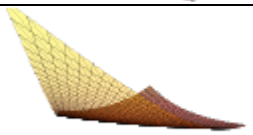
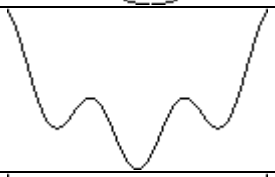
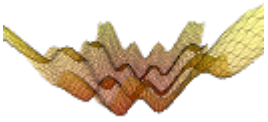
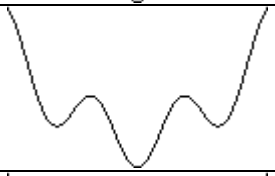
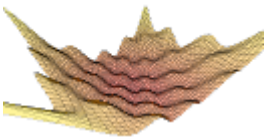
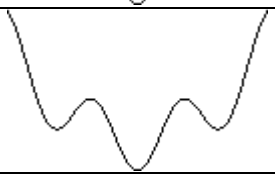
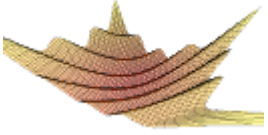
Table 2

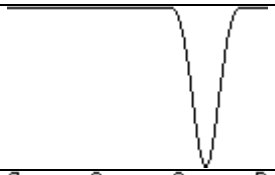
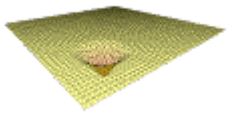
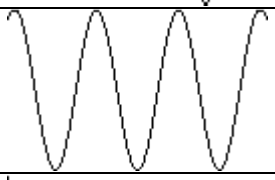
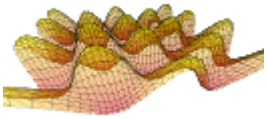
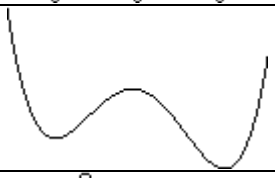
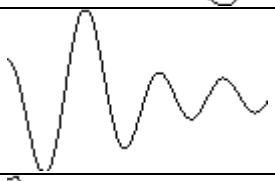
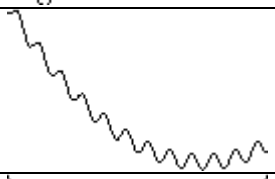
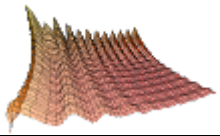
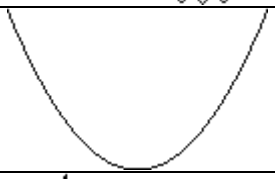
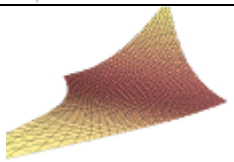
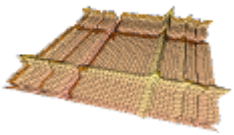
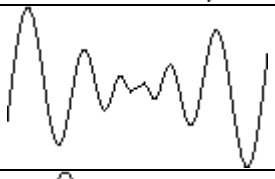
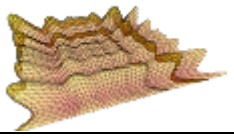
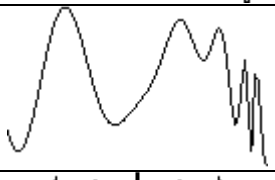
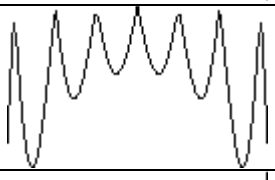
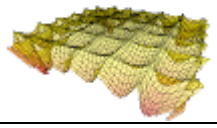
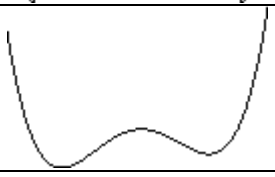
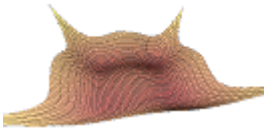
Common test functions used and their formulae.

Name	Formula
Rastrigin	$f(x,y) = 20 + x^2 - 10\cos(2\pi x) + y^2 - 10\cos(2\pi y)$
Ackley	$f(x,y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5\cos(2\pi x)+\cos(2\pi y)} + e + 20$
Sphere	$f(x,y) = x^2 + y^2$
Rosenbrock	$f(x,y) = 100(y - x^2)^2 + (1 - x)^2$
Beale	$f(x,y) = (1.5 - x + xy^2)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 - 0.281274$
Booth	$f(x,y) = (x + 2y - 7)^2 + (2x + y - 5)^2$
Bohachevsky 1	$f(x,y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7$
Bohachevsky 2	$f(x,y) = x^2 + 2y^2 - 0.3\cos(3\pi x)0.4\cos(4\pi y) + 0.3$
Bohachevsky 3	$f(x,y) = x^2 + 2y^2 - 0.3\cos(3\pi x + 4\pi y) + 0.3$
Easom	$f(x,y) = \cos(x)\cos(y)e^{-(x-\pi)^2-(y-\pi)^2} + 1$
Griewank	$f(x,y) = 0.00025(x^2 + y^2) - \cos(x)\cos\left(\frac{y}{\sqrt{2}}\right) + 1$
Himmelblau	$f(x,y) = (x + y^2 - 6)^2 + (x^2 + y - 11)^2$
Levi 3	$f(x,y) = \left(\sum_{i=1}^{5}((\mathrm{i}-1)\cos((i+1)x + i))\right)\left(\sum_{i=1}^{5}((\mathrm{i}-1)\cos((i+1)y + i))\right) + 176.542$
Levi 13	$f(x,y) = \sin^2(3\pi x) + (x-1)^2(1 + \sin^2(3\pi y)) + (y-1)^2(1 + \sin^2(2\pi y))$
Matyas	$f(x,y) = 0.26(x^2 + y^2) - 0.48xy$
Michalewicz	$f(x,y) = -\left(\sin(x)\sin^{20}\left(\frac{x^2}{\pi}\right) + \sin(y)\sin^{20}\left(\frac{2y^2}{\pi}\right)\right) + 1.8013$
Schwefel	$f(x,y) = 837.9658 - x\sin\left(\sqrt{\lvert x\rvert}\right) + y\sin\left(\sqrt{\lvert y\rvert}\right)$
Eggholder	$f(x,y) = -(y + 47)\sin\left(\sqrt{\lvert 0.5x + y + 47\rvert}\right) - x\sin\left(\sqrt{\lvert x - y + 47\rvert}\right) + 959.6407$
Holder Table	$f(x,y) = \left\lvert\sin(x)\cos(y)e^{\left\lvert 1 - \frac{\sqrt{x^2+y^2}}{\pi}\right\rvert}\right\rvert + 19.2085$
Styblinski-Tang	$f(x,y) = 0.5(x^4 - 16x^2 + 5x + y^4 - 16y^2 + 5y) + 78.33234$

Table 3

Common test functions used and their assosiated 1D and 2D images in 2D and 3D space.

Name	1D Function – 2D Image	2D Function – 3D Image
Rastrigin		
Ackley		
Sphere		
Rosenbrock		
Beale		
Booth		
Bohachevsky 1		
Bohachevsky 2		
Bohachevsky 3		

Easom		
Griewank		
Himmelblau		
Levi 3		
Levi 13		
Matyas		
Michalewicz		
Schwefel		
Eggholder		
Holder Table		
Styblinski-Tang		

4.1.2 B-Suite Environments

The environments that have been derived from the original B-Suite tests (outlined by *Osband*) have been modified to fit the needs of BE's testing interface. The original set of B-Suite tests includes a wide range of parameters that define how B-Suite runs each test which can be used to adjust the scale or difficulty. Since so many tests were run for this thesis, the recreated versions of the B-Suite tests run for BE used parameters that were narrowed down to the scope of the project. By increasing the range of the parameters or scale of the tests further, preliminary tests showed that run times began to take too long to get useful results. Furthermore, variations of the tests that included too much stochasticity were excluded so as to better control the proceedings (and hence understand the outcomes) of the tests.

Additionally, there were also three environments that were added to this section of the tests. These included two environments that directly test the ability of an agent to perform 1D and 2D gradient descent as well as a very simple environment that tests an agent's ability to move to a target. The gradient descent tests were put in place over-and-above the test functions outlined in *Table 3* as they are far simpler and test gradient descent at a more fundamental level. The *Move to Target* test was added because it too is a very straight-forward, fundamental task and tests BE's ability to remain robust to overfitting without obfuscating the simplicity of the underlying problem.

A list of all testing environments in this category and their details can be seen in *table 4*. Additionally, *table 5* shows the visualizations created for each environment (generated by BE's graphics engine) along with a general outline of what aspects of intelligence each environment tests. The target issues shown in *table 5* are partly taken from *Osband's* analyses but adjusted slightly for the unique implementations used in BE (Osband *et al.*, 2020).

Table 4

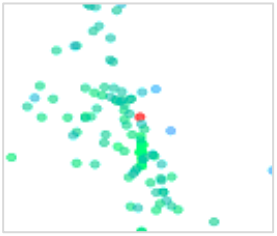
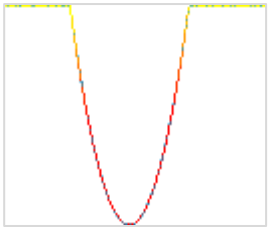
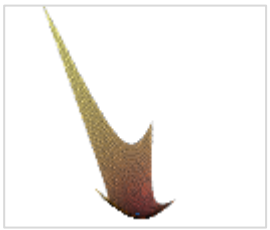
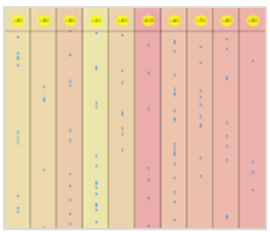
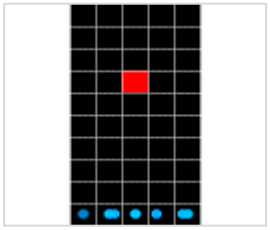
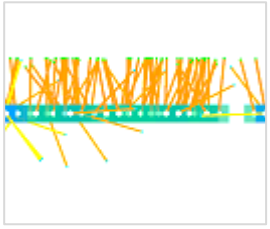
Table detailing each B-Suite test environments and how it is implemented

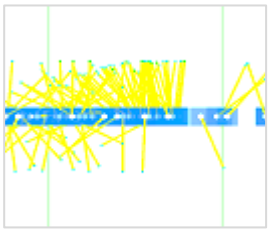
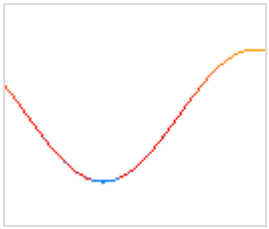
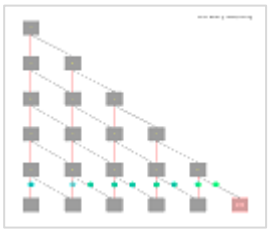
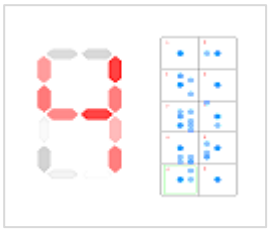
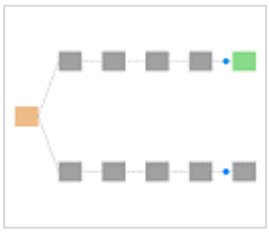
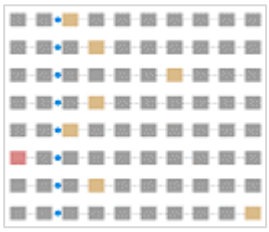
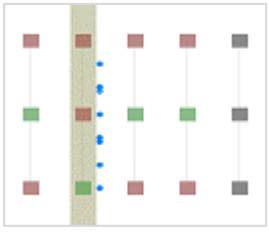
Name	Description
Move to Target	*Move to Target* is a very simple test where agents must move to a red target whose location is randomized at each step. The only information given to an agent includes its current position and target position. The closer an agent is to the target, the higher its reward. Rewards are only given if the agent moved closer to the target compared to its previous position.
1D Gradient Descent	Agents must descend a 1D slope to find its lowest point within ten steps. The gradient and direction of the slope as well as the position of the global minimum are randomized for each test cycle.
2D Gradient Descent	Agents must descend a 2D slope to find its lowest point after ten steps. The gradient and direction of the slope in both dimensions as well as the position of the global minimum are randomized for each test cycle.
One-Armed Bandit	The *One-Armed Bandit* implemented for BE is deterministic. An agent is given an array of ten possible choices to select as input with each option being worth a reward from ten to a hundred in increments of ten. The positions of these rewards are randomized at the start of each cycle. There are 20 steps in each test cycle, but rewards are only received after ten steps. An agent must therefore keep trying different options until it finds one that gives the highest reward before staying with it for the remaining number of steps.
Catch	*Catch* is a game that an agent must play in a grid world that is ten blocks high and five blocks wide. The agent occupies the bottom row and can only move a maximum of one block left or right. Targets then fall one block at a time from the highest block and start at a random location in the x-axis. Agents must move so as to "catch" the target falling towards them.
Cartpole	The *Cartpole* test requires agents to figure out the physics behind the problem. A cart is positioned within a 1D bounded area with a weighted pole attached to it standing upright (with a small randomized initial deviation). The agent must then apply forces to the cart at each time-step in either the left or right direction in order to maintain the balance of the pole so it does not swing down. It must maintain this for 50 steps whilst constantly remaining in the bounds of the environment. The input received at each step consists of the cart's position in the area, change in position, sine and cosine of the pole's angle, and change in the pole's angle. A reward is received at each step if the pole is upright such that the cosine of the angle of the pole is greater than 0.8.
Cartpole Swing-Up	The *Cartpole Swing-Up* test is exactly the same as the *Cartpole* test except all poles begin hanging down (with a small randomized initial deviation). The agent must then learn to swing the pole up before maintaining its balance in an upright position. This is significantly harder than the standard *Cartpole* test. Each agent gets 50 steps to do this but because the pole starts already hanging, it is technically impossible to get 100% accuracy (which would require the pole to always be upright) as there will always be a period in which the agent must swing the pole up.
Mountain Car	For the *Mountain Car* test, an agent must apply a force to a car to the left or to the right in order to move the car to the top of a target hill. The car starts on a smaller hill (with some random deviations to its initial starting conditions) and must use the dip between the two hills by going back and forth in a swinging motion to gather enough momentum to reach its target. An agent receives its current position, velocity, and current step as its input. Each test cycle is 50 steps, but an agent gets 20 free steps to make it to the top of the other hill. Its fitness is then calculated as 30 minus every step past 20 that the agent takes to reach its target.

Deep Sea	The *Deep Sea* test requires agents to traverse down a tree of nodes with six nodes (including the starting node) and five binary decision branches for each traversal. An agent can choose to either go left or right at each node. However, the node can either take the agent's action at face value (indicated by a plus sign) or reverse the agent's decision (indicated by a minus sign). This is randomized for each test cycle. Travelling down the left branch of the tree always results in a reward being received, however, the farthest right-hand node at the bottom of the tree issues the highest reward of 100. Each test cycle allows for enough traversals so that an agent may potentially explore the entire tree before only evaluating the final tree traversal to find the agent's fitness. The input received by an agent includes its cumulative rewards and an array representing the chain of movements the agent made in its current traversal of the tree as well as a flag indicating if it is in its final traversal or not.
Custom Simplified MNIST	The *Custom Simplified MNIST* environment has been modified from the original MNIST test. The original MNIST test simply requires the classification of 28x28 pixel hand-written numbers from zero to nine. In order to scale down the problem to fit within the scope of this thesis, this custom version only requires an agent to classify a number represented using digital segments with randomized noise. This changes the input to 14 different values. The same number is given multiple times with differently randomized noise in order to test an agent's ability to be invariant under changing noise. This also allows agents with memory to correct an initial misclassification. Rewards are only given to correct classifications.
Custom Umbrella Problem	The *Custom Umbrella* environment has been slightly modified from the original *Umbrella Problem* as outlined by *Osband* (Osband *et al.*, 2020). An agent begins at the start of a five-step sequenced event where the agent has to make a binary decision that determines where it ends up at the end of the five steps. A reward is only given at the end of the five steps and all actions taken by the agent on other steps do not actually impact anything. At each step, an agent receives an array (also of size five) of random noise. However, one of the array parameters (randomly chosen) indicates which initial decision will result in a reward. This information is only given on the first step. The five-step cycle is repeated twice, allowing the agent to learn which piece of information is valuable before having its fitness evaluated the second time. The problem is analogous to deciding to take an umbrella outside based on a weather prediction. One might engage in a set of unrelated tasks before the initial decision only becomes important later when it starts to rain.
Discounting Chain	The *Discounting Chain* environment consists of eight chains of linked nodes with each chain being made up of ten nodes. One randomized node on each chain contains a reward while only one reward across all chains contains the maximum reward available. At each step, an agent progresses along the chain before having to decide which chain to explore next when it reaches the end of its current chain. Each test cycle allows for an agent to potentially explore all chains before the final chain decision is the one that is evaluated as the agent's fitness. The input received by an agent includes data about its current position within its current chain as well as a flag indicating if it is on its last decision or not.
Memory Chain	In the *Memory Chain* test, an agent is shown a succession of five different pieces of information. Each piece of information contains a number from one to three. With each new piece of information, the environment also asks the agents what piece of information was previously given at a randomly chosen past step. Successfully choosing the right value results in a reward while the amount of content to remember increases with each step.

Table 5

Table detailing all B-Suite test environments, what each test intends to target, and example visualisations.

Name	Target Issues	Visualisation
Move to Target	• Basic Task Completion	
1D Gradient Descent	• Basic Task Completion	
2D Gradient Descent	• Basic Task Completion	
One-Armed Bandit	• Basic Task Completion • Exploration	
Catch	• Basic Task Completion	
Cartpole	• Basic Task Completion • Simple Credit-Assignment • Generalization	

Cartpole Swing-Up	• Basic Task Completion • Simple Credit-Assignment • Generalization • Exploration	
Mountain Car	• Basic Task Completion • Simple Credit-Assignment • Generalization	
Deep Sea	• Exploration • Credit-Assignment • Memory / Temporal Generality	
Custom Simplified MNIST	• Basic Task Completion • Generalization • Noise Reduction	
Custom Umbrella Problem	• Credit-Assignment • Memory / Temporal Generality • Noise Reduction	
Discounting Chain	• Exploration • Credit-Assignment • Memory / Temporal Generality	
Memory Chain	• Credit-Assignment • Memory / Temporal Generality	

4.1.3 Mars Rover Test Environments

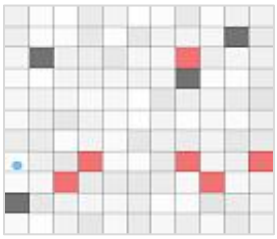
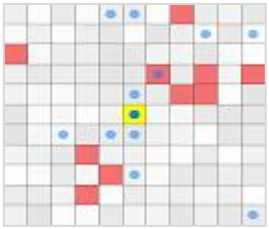
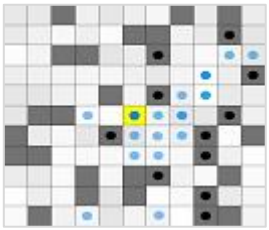
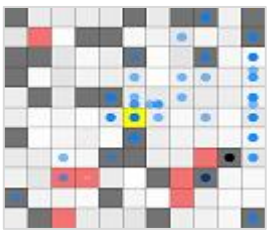
The aforementioned test environments are each highly targeted and cover a broad range of general tasks. However, in order to replicate a more realistic use-case of BE's software, four more test environments were built that abstractly represent the kinds of tasks that may be encountered by a space rover operating on Mars. These environments are still scaled down to fit within the scope of this thesis and are merely representative of larger scale problems that BE might be used to help tackle.

Drawing from the implementation GESTALT in NASA's MER mission, the environments created utilize a 11 x 11 virtual grid world which is very similar to the 10 x 10 meters^2 grid world originally used by GESTALT. As a result, this more directly replicates the type of higher-level decision making that a more advanced space rover AI may need to execute if it were in a similar situation to the original MER mission. The dimensions of the grid world were chosen to be eleven instead of ten in order to have a starting cell that is perfectly in the centre of the grid world.

The four space-rover environments created target all of the aspects of intelligence outlined by this thesis and hence represent a more general benchmark of BE as a whole. The difficulty of performing well in these environments is also considerably higher than that of the other tests so the results produced by agents operating in these environments are not necessarily expected to be very good. The details pertaining to the space rover tests created as well as an example of their visualisations can be seen in *table 6*.

Table 6

Table detailing all space rover test environments and how they are implemented along with example visualisations.

Name	Description	Visualisation
Collect Mission	This mission represents a basic space rover task that entails navigating to certain locations in order to gather sample data from that location. The locations are randomly set around the grid world. Gathering samples results in a reward being received. If two sample locations overlap a cell, they are indicated by a "x2" symbol and are worth an appropriately higher reward. Cells also go grey if they have been collected by all agents. Each agent only gets 121 steps (the number of cells in the grid world) to try and find all sample gathering locations. Agents can only move one block at a time and cannot collect the same source more than once. The input information received by an agent at each step includes information about all the blocks immediately surrounding it, the last movement it made, and how many steps are remaining. Blocks that do not contain anything also include a random noisy signal that agents must learn to ignore.	
Gather Mission	The *Gather Mission* is set up in exactly the same manner as the *Collect Mission* except once a sample has been gathered, a reward is only received for it once the agent returns to its starting position at the centre of the grid world. An additional input is given to the agents beyond those outlined in the *Collect Mission* that indicates whether an agent is currently holding any samples that needs to be deposited or not. This environment is therefore significantly harder than the *Collect Mission*.	
Navigation Mission	This mission is also set up in the exact same way as the *Collect Mission* except instead of gathering locations that are randomized around the world, hazardous obstacles are randomly placed while still ensuring there is always at least one path leading from the starting point (in the centre of the grid world) to the edge of the grid world. Agents must then try and navigate as far away from the centre as possible. If they move into a hazard, their progress is halted and they can no longer move. An additional input is given to the agents beyond those outlined in the *Collect Mission* that indicates the furthest distance that a particular agent has achieved from the starting point for its current test cycle. This test is similar to the *Collect Mission* in difficulty.	
Full Rover Mission	The *Full Rover* mission is the hardest environment and combines the *Gather Mission* with the *Navigation Mission*. Agents are rewarded in the same way as the *Gather Mission* except there are additional hazardous cells that halt an agent's progress should they move into them. Unlike the *Navigation Mission*, agents are not rewarded for how much distance they cover from the starting point as the hazardous cells merely act as obstacles to the target gathering locations. On a high level, agents must be able to explore the world in an efficient manner while avoiding dangerous cells. Only exploring areas once and remembering paths they took to deposit gathered samples is also necessary for an agent to perform well.	

4.1.4 Experimentation Process and Data Gathering

The experimentation process conducted for this thesis tests BE's ability as a general cognitive architecture to exhibit its potential for AutoML. This is to say that the tests focus on BE's architecture as a whole, whether simpler approaches for a particular problem would be more efficient or not. An example of this would be in the solution to the *Move to Target* test environment. The solution to the environment is extremely simple, and BE's architecture (due its complexity) only serves to obfuscate the optimal function. However, the intent is to test BE's process as a whole to target any problem regardless of the complexity and its ability to solve the *Move to Target* test environment without guidance is still valuable. Despite the goal of BE being its capability to eventually assist in solving complex tasks, most tests were highly simplified in order to focus on small proofs of concepts by testing BE's generality over a wide range of different kinds of tests.

All tests were run on a local machine with 16Gb of DDR4 RAM and an i7-8700K hex-core running at 3.7 GHz. However, since the machine was not dedicated solely to running tests, only four of its six cores were allocated to BE. Due to long processing times from the limited hardware available and large number of tests to complete, the scalability of the tests run was also kept very small. However, scalability was identified as being one of the core components of intelligence. Despite this, in order to keep everything in scope, the tests conducted do not evaluate the *scalability* component of intelligence. While some tests may inadvertently test this (such as in the case of the space rover tests), this aspect is not explored much further by this thesis. However, it is still important to note that MLSH's focus on scalability still assists its effectiveness in improving other aspects of intelligence as its purpose is to break problems down into simpler parts.

There are multiple functionalities and potential configurations that are also possible with BE that are not explored in the testing process. The reason for this is that these functionalities and configurations are simply out of scope and are not essential for the focus of this thesis. However, they still remain part of BE's software for others to potentially use since BE is in itself a contribution of this thesis. One of these functionalities not used includes tests that involve an agent's ability to solve multiple test environments at once. Test environments are therefore not combined and are instead run individually. This shifts the focus from testing BE's ability to create generally intelligent agents to testing BE's ability to be general itself. Another feature that was not used is online self-adaption which simply fell out of scope but may still be a potentially useful component of BE as a general cognitive architecture. However, this would have to be explored by future research.

Continuing with the effort to keep everything in scope, tests were not run on all 1D and 2D test functions individually. Instead, only two test environments were run using the *Random 1D Function* and *Random 2D Function* environments as they encompass all of the test functions. Despite all these changes to minimise testing run times as well as the use of parallelisation, completing a single run of all the tests ultimately took more than an entire month. Due to this time constraint and the fact that many preliminary tests were first attempted, only a single batch of all tests was successfully completed. This means that the results were not averaged across different runs with different initial seeds. However, this does not necessarily mean that the results obtained are unreliable as all potential best individuals in the optimized DNE tests were evaluated 100 times before averaging the obtained fitnesses. Furthermore, each REVAC generation was also repeated five times before averaging the results (MBF) of each of the individual DNE configurations.

The architectural configurations tested include all possible combinations of enabling or disabling the SNN, MLSH, Attention, and GEE components with DNE always enabled as a core component *(see table 8)*. Some of the configuration options that remained static (in order to reduce the number of tests) include *Add Raw Input to Attention Output* which was always set to off (to reduce network complexity) and *Use SNNs On Theta Network Only* which was always set to being on (to minimise SNN's additional complexity being introduced into ϕ sub-policy networks). The evolutionary parent selection method used was also always set to *exponential selection* due to its flexibility as a selection method as outlined in *chapter 2.2.6*.

The testing process used to evaluate BE involves running BE with 16 different architectural configurations repeated for each testing environment. Each test configuration-environment pair is run twice; first using REVAC with lots of small-scale DNE runs of the target test environment (representing the population of REVAC) and secondly using the optimally found parameters by REVAC on a large-scale DNE run. A complete list of all the relevant non-architectural parameters (that were not optimized by REVAC) used for all tests can be seen in *table 7*. For a full comprehensive overview of all the tests run, *table 7* can be used in conjunction with *table 8* and *table 9* (which have been split up due to the limited page space available) as well as *table 10*. *Table 8* and *table 9* show the exact architectural configurations for all tests in all test environments. Each configuration is made up by combining different modules. The modules *DNE*, *SNN*, *MLSH*, *Attention* and *GEE* are represented by the symbols *D, S, M, A,* and *E* respectively. The "*Use REVAC"* option is marked as "*N/A"* since all tests are run once with REVAC enabled and then again with it disabled. The "*REVAC Tests"* column in *table 7* corresponds to the parameters used for all tests that have REVAC enabled while the "*Optimized DNE Tests"* column corresponds to all tests where REVAC is disabled.

Table 7

Table showing all non-architecural and non-REVAC optimized parameters used for all tests.

Parameter	REVAC Tests	Optimized DNE Tests
Seed	1	1
Max Threads	4	4
Max Nodes Per Network	100	1000
Max Networks	5	10
DNE Population	50	1000
DNE Generations Max	50	1000
DNE Generations to End If No New Best	50	1000
DNE Simulation Repetitions	1	10
DNE Evaluation Repetitions	10	100
REVAC Population	80	N/A
REVAC Generation Max	100	N/A
REVAC Generations to End if No New Best	4	N/A
REVAC Evaluation Repetitions	5	N/A
REVAC Parent Pool Size	40	N/A
REVAC Mutation Range	10	N/A

Table 8

Table showing all the different configuration settings (with GEE disabled) used in the tests conducted for all environments. REVAC is marked "N/A" as it is enabled first and then disabled.

Configuration	D	DS	DM	DA	DSM	DSA	DMA	DSMA
Use REVAC	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Use SNNs	-	✓	-	-	✓	✓	-	✓
Use SNNs On Theta Network Only	-	-	-	-	✓	-	-	✓
Use MLSH	-	-	✓	-	✓	-	✓	✓
Use Attention	-	-	-	✓	-	✓	✓	✓
Add Raw Input to Attention Output	-	-	-	-	-	-	-	-
Enable Guided Epsilon Exploration	-	-	-	-	-	-	-	-
Enable Activation Offset	✓	✓	✓	✓	✓	✓	✓	✓
Enable Identity Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable ReLU Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable Leaky ReLU Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable Sigmoid Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable TanH Function	✓	✓	✓	✓	✓	✓	✓	✓
Use Linear Wheel	-	-	-	-	-	-	-	-
Use Exponential Wheel	✓	✓	✓	✓	✓	✓	✓	✓
Use Proportional Selection	-	-	-	-	-	-	-	-
Use Tournament Selection	-	-	-	-	-	-	-	-

Table 9

Table showing all the different configuration settings (with GEE enabled) used in the tests conducted for all environments. REVAC is marked "N/A" as it is enabled first and then disabled.

Configuration	DE	DSE	DME	DAE	DSME	DSAE	DMAE	DSMAE
Use REVAC	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
Use SNNs	-	✓	-	-	✓	✓	-	✓
Use SNNs On Theta Network Only	-	-	-	-	✓	-	-	✓
Use MLSH	-	-	✓	-	✓	-	✓	✓
Use Attention	-	-	-	✓	-	✓	✓	✓
Add Raw Input to Attention Output	-	-	-	-	-	-	-	-
Enable Guided Epsilon Exploration	✓	✓	✓	✓	✓	✓	✓	✓
Enable Activation Offset	✓	✓	✓	✓	✓	✓	✓	✓
Enable Identity Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable ReLU Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable Leaky ReLU Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable Sigmoid Function	✓	✓	✓	✓	✓	✓	✓	✓
Enable TanH Function	✓	✓	✓	✓	✓	✓	✓	✓
Use Linear Wheel	-	-	-	-	-	-	-	-
Use Exponential Wheel	✓	✓	✓	✓	✓	✓	✓	✓
Use Proportional Selection	-	-	-	-	-	-	-	-
Use Tournament Selection	-	-	-	-	-	-	-	-

Table 10

Table summarizing all experiments run. The target issues have been abbreviated as B, E, N, G, M, and C which corresponds to the first letter of each target issue's description.

Environment	Steps	B	E	N	G	M	C
Move to Target	5	✓	-	-	-	-	-
1D gradient descent	10	✓	-	-	-	-	-
2D gradient descent	10	✓	-	-	-	-	-
One-Armed Bandit	20	✓	✓	-	-	-	-
Catch	10	✓	-	-	-	-	-
Cartpole	50	✓	-	-	✓	-	✓
Cartpole Swing-Up	50	✓	✓	-	✓	-	✓
Mountain Car	50	✓		-	✓	-	✓
Deep Sea	60		✓	-	-	✓	✓
Custom Simplified MNIST	10	✓		✓	✓		
Custom Umbrella Problem	10	-	-	✓	-	✓	✓
Discounting Chain	90	-	✓	-	-	✓	✓
Memory Chain	5	-	-	-	-	✓	✓
Collect Mission	121	✓	✓	✓	✓	✓	✓
Gather Mission	121	✓	✓	✓	✓	✓	✓
Navigation Mission	121	✓	✓	✓	✓	✓	✓
Full Rover Mission	121	✓	✓	✓	✓	✓	✓
1D Test Functions	10	✓	-	-	-	-	-
2D Test Functions	10	✓	-	-	-	-	-
All environment tests above are repeated for each of the following configurations:							
Excluding GEE:	D	DS	DM	DA	DSM	DSA	DMA	DSMA
Including GEE:	DE	DSE	DME	DAE	DSME	DSAE	DMAE	DSMAE

To summarise the experiments outlined throughout this chapter, an overview of all tests conducted can be seen in *table 10*. The table simply defines all test environments and the number of simulation steps used. The target issue of each environment is also shown in order to highlight the purpose behind each test. All target issues *(Basic Task Completion*, *Exploration*, *Noise Reduction*, *Generalization*, *Memory/Temporal Generality*, and *Credit Assignment)* have been abbreviated as *B, E, N, G, M,* and *C* respectively. The configurations depicted have also been abbreviated in the same manner as *table 8* and *table 9*. Ultimately, *table 10* encompasses every test run using BE for this thesis.

4.2 Results

The results obtained from the tests outlined in *chapter 4.1.4* were all stored in separate save files by BE. An additional *Python* program was then written called *data_reader.py* to read the data from the collection of save files before summarising all the information into four *MS Excel* spreadsheets named *Average Results*, *Average REVAC Results*, *Highest Results*, and *Highest REVAC Results*. The data pertaining to each Excel spreadsheet corresponds to the values obtained at the end of a particular run.

The results that hold the most interest are those found in the *Highest Results* spreadsheet as shown in *table 15* and *table 16* as they represent the performance of the "end products" or resultant intelligent agent solutions produced by BE's AutoML process for all the environments and architectural configurations. The results in *Average Results* indicate the solution population's stability or reliability in relation to the highest results while the REVAC counterparts highlight the benefit of running larger scale DNE runs with REVAC tuned parameters as opposed to the results produced using REVAC by itself.

The data collected in the four files can be seen in the tables to follow. The columns represent the results for different architectural configurations while the rows represent the different test environments used. Each table has been split up (due to space constraints) in the same manner as *table 8* and *table 9* so that the results obtained with GEE are represented on a separate table. Symbols have also been used as a shorthand in the same way as *table 8* and *table 9* to indicate an architectural module except that the *default* architectural configuration simply indicates that only the core DNE module was used in combination with BE's general learning and evaluation process. The symbol *"D"* (DNE) has been omitted for convenience.

Table 11

Average Optimized DNE Results with GEE Disabled

Environment/Config:	Default	S	M	A	S+M	S+A	M+A	S+M+A
Move to Target	75.97	23.41	78.30	70.77	24.47	25.92	67.12	20.69
1D gradient descent	85.90	92.27	86.01	91.90	92.68	92.51	86.71	91.99
2D gradient descent	82.99	89.91	88.32	86.86	90.70	90.57	86.48	90.85
One-Armed Bandit	68.13	70.07	69.05	69.66	69.85	73.63	68.17	71.07
Catch	38.65	72.34	36.87	28.77	72.47	83.49	51.64	74.66
Cartpole	17.25	46.57	67.55	47.87	46.51	45.25	52.47	44.42
Cartpole Swing-Up	29.19	10.87	32.39	13.39	10.84	10.14	12.75	10.84
Mountain Car	33.51	44.42	31.47	0.00	0.00	0.00	0.00	0.00
Deep Sea	74.86	83.33	75.19	79.28	83.33	83.33	72.77	83.33
Custom Simplified MNIST	35.54	22.64	20.80	58.22	23.15	25.67	35.77	17.99
Custom Umbrella Problem	68.30	80.25	63.56	63.90	72.50	77.11	78.96	74.58
Discounting Chain	66.64	75.12	65.36	70.37	69.77	70.15	65.81	70.15
Memory Chain	54.63	63.21	46.69	51.97	60.49	57.51	49.58	53.79
Collect Mission	47.24	14.38	30.25	4.52	10.88	2.45	6.33	2.20
Gather Mission	6.16	0.00	20.57	0.88	0.14	0.00	1.66	0.00
Navigation Mission	61.49	55.59	78.18	28.72	55.35	17.51	30.39	12.84
Full Rover Mission	0.61	0.00	1.70	0.92	0.00	0.00	0.48	0.00
1D Test Functions	63.42	78.12	63.91	68.49	76.43	76.78	68.25	77.21
2D Test Functions	74.09	79.36	74.24	76.84	79.36	78.59	73.41	79.59

Table 12

Average Optimized DNE Results with GEE Enabled

Environment/Config:	Default+E	S+E	M+E	A+E	S+M+E	S+A+E	M+A+E	S+M+A+E
Move to Target	69.55	26.10	79.83	72.57	26.31	26.83	70.94	25.01
1D gradient descent	80.08	86.46	83.88	83.48	86.03	86.32	83.12	85.11
2D gradient descent	82.94	87.50	82.97	80.82	87.38	86.26	80.76	88.03
One-Armed Bandit	71.39	66.98	70.70	70.50	73.14	71.76	67.67	69.44
Catch	68.54	56.22	48.82	53.36	37.84	44.38	22.86	38.49
Cartpole	39.81	56.58	20.19	29.06	49.70	43.34	44.03	39.55
Cartpole Swing-Up	41.01	14.83	49.66	18.39	9.33	8.72	18.44	9.12
Mountain Car	37.12	4.48	60.38	38.78	4.26	4.41	28.65	3.92
Deep Sea	78.06	78.46	76.51	74.58	81.32	77.12	75.89	80.65
Custom Simplified MNIST	18.28	20.84	15.05	27.26	21.32	21.99	19.37	18.01
Custom Umbrella Problem	59.05	56.71	62.81	60.10	65.33	63.07	55.72	59.97
Discounting Chain	61.20	61.02	61.13	63.34	62.25	60.69	61.14	62.71
Memory Chain	46.89	37.87	47.79	46.58	37.57	46.00	43.65	42.15
Collect Mission	33.60	26.69	30.38	19.06	28.12	27.94	19.73	19.36
Gather Mission	7.83	7.80	7.45	4.21	7.32	7.12	4.84	4.97
Navigation Mission	64.26	57.93	71.34	26.04	61.30	15.31	21.80	15.34
Full Rover Mission	0.21	0.21	0.22	0.21	0.22	0.24	0.22	0.21
1D Test Functions	60.19	69.17	57.09	67.15	64.04	62.88	62.30	66.00
2D Test Functions	68.01	72.96	72.51	67.79	70.04	74.99	69.39	70.76

Table 13

Average REVAC Results with GEE Disabled

Environment/Config:	Default	S	M	A	S+M	S+A	M+A	S+M+A
Move to Target	90.73	31.68	90.56	82.71	29.83	25.58	80.56	26.80
1D gradient descent	94.12	94.04	94.15	94.38	94.28	94.31	94.38	94.34
2D gradient descent	91.06	91.89	91.15	91.19	91.87	91.86	91.20	91.86
One-Armed Bandit	88.23	80.56	88.27	88.11	80.05	80.74	88.22	79.84
Catch	78.00	64.55	84.00	78.97	70.40	68.68	83.85	70.82
Cartpole	88.49	49.01	91.35	86.60	48.73	46.13	53.06	45.98
Cartpole Swing-Up	37.49	16.20	35.81	30.50	17.00	14.99	29.21	14.73
Mountain Car	45.47	0.74	51.25	1.54	0.66	0.00	2.41	0.00
Deep Sea	83.17	83.33	83.13	82.37	83.33	83.33	83.23	83.33
Custom Simplified MNIST	46.39	28.47	46.13	46.30	29.74	33.36	46.38	35.02
Custom Umbrella Problem	88.32	89.38	88.53	87.85	87.60	89.32	88.70	90.55
Discounting Chain	82.59	81.16	82.44	82.87	80.40	76.53	82.83	76.27
Memory Chain	71.27	54.40	71.40	71.53	54.67	67.07	71.65	66.61
Collect Mission	26.01	11.13	27.88	12.81	8.22	3.71	12.78	3.55
Gather Mission	4.33	0.00	5.92	2.56	0.00	0.00	1.89	0.00
Navigation Mission	79.42	54.22	77.35	50.62	54.94	19.88	51.70	19.57
Full Rover Mission	1.42	0.00	1.33	0.26	0.00	0.00	0.32	0.00
1D Test Functions	82.31	80.99	82.59	83.11	80.85	80.74	83.16	80.72
2D Test Functions	84.17	82.78	83.86	84.20	82.75	82.78	84.23	82.70

Table 14

Average REVAC Results with GEE Enabled

Environment/Config:	Default+E	S+E	M+E	A+E	S+M+E	S+A+E	M+A+E	S+M+A+E
Move to Target	86.55	37.96	84.17	74.45	38.24	38.13	74.84	38.25
1D gradient descent	93.03	91.97	92.77	93.07	92.05	92.17	92.94	92.06
2D gradient descent	90.20	89.60	90.27	90.04	89.50	89.43	90.12	90.14
One-Armed Bandit	87.79	77.45	88.08	87.61	77.80	77.30	87.84	77.32
Catch	62.22	58.55	61.17	58.42	58.78	59.08	63.62	60.00
Cartpole	77.68	48.41	83.26	67.83	49.79	46.09	79.07	45.89
Cartpole Swing-Up	31.68	14.91	29.83	21.47	14.89	14.90	21.73	14.87
Mountain Car	59.58	21.91	63.99	23.99	21.67	21.80	24.26	21.72
Deep Sea	83.00	83.33	83.27	82.99	83.33	83.33	82.84	83.35
Custom Simplified MNIST	44.41	22.61	45.75	43.90	25.89	29.75	43.89	29.96
Custom Umbrella Problem	87.30	86.62	87.57	87.10	86.82	86.68	87.10	86.72
Discounting Chain	80.67	78.88	80.71	80.73	78.81	78.79	80.83	78.76
Memory Chain	70.04	47.06	70.11	69.90	47.06	46.63	70.10	50.01
Collect Mission	43.84	43.17	43.84	40.94	43.22	43.22	41.19	41.12
Gather Mission	18.90	18.55	18.62	15.91	18.47	18.56	16.04	16.48
Navigation Mission	80.35	51.33	76.35	49.12	39.28	19.82	49.90	21.67
Full Rover Mission	1.77	1.76	1.73	0.99	1.72	1.75	0.96	1.03
1D Test Functions	79.78	74.25	79.76	80.67	74.59	74.02	80.49	74.20
2D Test Functions	82.76	77.17	82.60	81.77	77.16	77.37	82.41	77.59

Table 15

Highest Optimized DNE Results with GEE Disabled

Environment/Config:	Default	S	M	A	S+M	S+A	M+A	S+M+A
Move to Target	96.43	34.53	94.11	83.99	38.67	37.34	90.33	31.64
1D gradient descent	92.71	92.27	99.35	99.44	92.68	92.95	99.22	92.18
2D gradient descent	88.73	89.91	89.00	92.17	90.75	90.57	91.37	90.98
One-Armed Bandit	77.33	70.07	81.07	82.34	69.85	82.92	76.40	77.62
Catch	100.00	77.00	100.00	100.00	96.00	100.00	100.00	76.00
Cartpole	100.00	49.92	100.00	99.96	51.10	50.26	90.68	48.86
Cartpole Swing-Up	63.96	31.20	63.96	51.58	32.20	25.76	42.00	38.08
Mountain Car	100.00	100.00	100.00	0.00	0.00	0.00	0.00	0.00
Deep Sea	83.33	83.33	83.33	83.33	83.33	83.33	83.33	83.33
Custom Simplified MNIST	50.60	30.00	37.80	71.30	26.00	30.80	78.00	35.30
Custom Umbrella Problem	78.00	80.25	77.00	77.00	84.00	100.00	100.00	100.00
Discounting Chain	68.74	77.96	68.74	78.41	80.20	76.33	69.67	74.27
Memory Chain	63.00	69.20	62.40	74.60	65.60	73.20	77.60	72.00
Collect Mission	62.00	19.10	63.50	8.10	17.70	2.45	8.10	2.40
Gather Mission	13.40	0.00	51.60	0.88	0.40	0.00	1.66	0.00
Navigation Mission	87.00	78.40	86.00	36.00	88.80	28.60	38.00	19.60
Full Rover Mission	1.70	0.00	3.50	0.92	0.00	0.00	0.48	0.00
1D Test Functions	71.11	78.51	71.32	81.14	76.96	81.56	86.10	80.66
2D Test Functions	77.33	79.36	77.14	77.17	79.36	78.59	76.19	79.59

Table 16

Highest Optimized DNE Results with GEE Enabled

Environment/Config:	Default+E	S+E	M+E	A+E	S+M+E	S+A+E	M+A+E	S+M+A+E
Move to Target	94.33	32.56	95.49	87.28	32.56	32.56	87.36	32.56
1D gradient descent	94.92	90.44	91.76	98.84	90.23	90.66	97.82	90.76
2D gradient descent	89.22	89.18	88.89	91.53	89.83	89.18	92.29	89.02
One-Armed Bandit	84.81	66.98	78.43	75.56	80.40	71.76	74.47	77.18
Catch	100.00	81.00	100.00	100.00	77.00	73.00	100.00	93.00
Cartpole	99.94	65.22	100.00	100.00	60.54	48.50	100.00	43.12
Cartpole Swing-Up	65.96	29.94	67.50	28.46	12.16	12.04	36.02	12.04
Mountain Car	100.00	9.93	100.00	100.00	9.93	9.93	76.73	9.93
Deep Sea	83.33	83.33	83.33	83.33	83.33	83.33	83.33	83.33
Custom Simplified MNIST	35.40	26.30	34.20	44.50	30.60	34.00	31.90	26.10
Custom Umbrella Problem	69.00	65.00	69.00	65.00	65.33	65.00	72.00	65.00
Discounting Chain	65.82	65.82	65.82	69.45	65.82	65.82	72.37	65.82
Memory Chain	62.00	39.40	60.60	71.60	52.80	71.40	66.60	72.00
Collect Mission	44.30	36.40	36.40	34.40	36.40	36.40	34.40	34.60
Gather Mission	12.80	12.80	12.80	11.00	12.80	12.80	9.40	11.00
Navigation Mission	86.00	77.80	87.00	36.00	87.40	19.60	36.80	18.80
Full Rover Mission	0.80	0.80	0.80	0.70	0.80	0.80	0.70	0.70
1D Test Functions	72.10	74.02	72.05	82.54	73.13	67.64	81.66	70.76
2D Test Functions	76.90	72.96	76.40	76.30	72.51	74.99	76.79	72.44

Table 17

Highest REVAC Results with GEE Disabled

Environment/Config:	Default	S	M	A	S+M	S+A	M+A	S+M+A
Move to Target	95.05	40.33	93.79	88.66	38.35	34.86	87.25	34.73
1D gradient descent	95.62	95.07	95.93	97.72	95.40	95.39	98.21	95.67
2D gradient descent	92.75	92.92	93.19	93.19	92.86	92.98	92.86	92.93
One-Armed Bandit	93.26	89.54	94.34	93.26	85.88	89.06	94.16	86.30
Catch	98.00	82.00	98.00	96.00	86.00	84.00	98.00	92.00
Cartpole	100.00	54.88	100.00	100.00	53.84	48.40	100.00	47.72
Cartpole Swing-Up	48.48	25.32	45.20	35.44	23.32	20.24	37.88	20.04
Mountain Car	100.00	19.47	100.00	99.60	19.93	0.00	100.00	0.27
Deep Sea	83.33	83.33	83.33	83.33	83.33	83.33	83.33	83.33
Custom Simplified MNIST	58.40	37.60	58.80	61.20	40.00	44.00	58.80	45.40
Custom Umbrella Problem	94.00	98.00	96.00	94.00	100.00	96.00	96.00	98.00
Discounting Chain	87.55	86.21	87.43	87.06	86.50	81.79	87.76	83.21
Memory Chain	76.00	61.20	76.00	78.40	61.20	75.20	77.20	74.80
Collect Mission	42.40	20.60	44.20	16.80	13.60	8.20	16.60	8.00
Gather Mission	15.40	0.00	15.20	7.00	0.00	0.00	7.00	0.00
Navigation Mission	96.00	70.40	100.00	60.80	73.20	22.80	63.20	22.00
Full Rover Mission	2.80	0.00	2.80	1.60	0.00	0.00	2.40	0.00
1D Test Functions	87.73	85.24	87.37	88.66	84.88	85.82	87.67	86.27
2D Test Functions	88.25	86.76	91.38	88.15	87.01	87.16	89.74	86.84

Table 18

Highest REVAC Results with GEE Enabled

Environment/Config:	Default+E	S+E	M+E	A+E	S+M+E	S+A+E	M+A+E	S+M+A+E
Move to Target	93.39	40.05	91.51	85.36	40.95	40.03	86.11	40.58
1D gradient descent	95.45	94.63	94.95	95.15	94.52	94.63	95.14	94.57
2D gradient descent	92.48	92.08	92.65	92.18	91.81	91.49	93.23	91.82
One-Armed Bandit	93.28	86.10	94.96	93.26	85.80	86.88	93.26	85.80
Catch	80.00	72.00	84.00	80.00	70.00	72.00	84.00	78.00
Cartpole	100.00	56.36	100.00	100.00	55.32	48.24	100.00	47.48
Cartpole Swing-Up	43.56	17.16	39.76	27.96	16.68	16.12	31.68	16.52
Mountain Car	100.00	30.47	100.00	93.73	26.53	25.40	92.73	25.40
Deep Sea	83.33	83.33	83.33	83.33	83.67	83.33	83.33	83.67
Custom Simplified MNIST	57.80	30.60	54.60	54.00	36.20	40.40	55.40	38.60
Custom Umbrella Problem	96.00	94.00	94.00	94.00	94.00	94.00	94.00	94.00
Discounting Chain	85.37	83.70	87.41	85.73	83.90	83.70	87.34	83.70
Memory Chain	76.00	52.80	76.00	76.00	53.60	57.20	76.40	59.20
Collect Mission	49.20	46.00	50.00	44.40	45.80	46.00	44.80	45.80
Gather Mission	22.40	21.20	24.40	20.80	21.00	21.00	19.00	19.40
Navigation Mission	98.00	67.20	94.00	62.80	55.60	22.80	61.60	25.60
Full Rover Mission	3.20	3.20	3.00	2.60	3.20	3.20	2.40	2.60
1D Test Functions	85.83	80.36	84.26	85.89	81.40	80.80	84.93	81.14
2D Test Functions	87.74	85.03	87.50	87.41	86.76	84.27	86.70	85.92

4.3 Summary and Recap: Experiments and Results

Before any experiments could take place, BE had to be successfully developed and checked for the absence of any programmatical and logical bugs. Furthermore, the system created for BE that allows experiments to be run was built in such a way that all subsequent tests executed for the purpose of this thesis could be easily setup through the program's GUI.

The goal behind the experiments conducted is to test BE's ability to find general AI solutions (by itself) for a wide range of possible task scenarios. The test environments coded into BE were hence chosen to stress this. As BE is just an attempt at a general cognitive architecture, the effectiveness of its architecture and all its components is hypothetical. Consequently, experiments were run on all test environments *for all* configurations (in a general subset of the possible configurations BE has to offer) so as to test the relevance of BE's claim to its specific architecture. There is also no AutoML process in BE that automatically configures the architectural design other than using all components at once and having less useful components evolve to not interfere with other internal processes.

Based on the literature reviewed, hypothetical predictions were also made outlining what each module within BE would be capable of achieving (as summarised in *table 1*). The experiments used were set up so as to test the validity of these predictions and hence provide insight into the benefits of using each module in an evolutionary context, thus contributing to the broader field of research into general AI.

The use of REVAC within the testing process was also separated out. In other words, each test involved a HPO process (done by REVAC), and an evaluation

process done by running the optimized parameters for the current configuration and environment on an extended DNE test. This was done so that two separate results could be obtained for each test-pair (instead of just running everything with REVAC and using those as the final results). Having two separate sets of results means that the effectiveness of REVAC's contribution to the AutoML process can also be evaluated, since REVAC implements HPO and doesn't form part of the core modules involved in the operation of a resultant agent.

As already stated, processing time was a significant issue for the set of experiments that needed to be run which was mostly due to the sheer number of them. Since each test-pair consists of two test-runs repeated across 19 environments for all 16 configurations, there were a total of 608 separate tests. However, it is important to remember that these long processing times are not necessarily indicative of a potential inefficiency in BE's implementation as the experiments evaluate BE as a whole and as a possible general cognitive architecture. Practically using BE to develop (or assist in developing) a general agent would not involve running so many tests as the objective purpose would likely be more specific and could hence be done in a much more manageable timeframe.

Chapter 5

Discussion

5.1 Overview

After gathering and processing all data obtained through the experimentation process conducted for this thesis, the information collected was then analysed. The following chapter outlines and discusses the various analyses made whilst also further processing the data and representing it in subsequent tables.

The first objective is to discuss the validity of the predictions made regarding how each BE module will perform in the context of each of the different target aspects of intelligence as outlined in *table 1*. The data is also looked at and analysed by focusing on BE's overall performance in solving all test environments, subtasks, and aspects of intelligence. Based on BE's general ability to solve problems, further discussion is given to the effectiveness of BE's architecture by comparing different configurations to the default core configuration that uses DNE in isolation.

Furthermore, the usefulness of REVAC in the role of BE's implementation of AutoML is also evaluated along with identifying which modules were most valuable given certain conditions. Special attention is also subtly focused on the performance of BE in the space rover tests as well as the aspects of intelligence pertaining to *perspective*, such as memory/temporal generality and credit assignment.

5.2 Discussion and Analysis of Results

The analyses of the results collected draws back to the information presented in *chapter 4.2*. The tables in *chapter 4.2* contain a lot of data to comprehend despite the fact that it has already been largely abstracted and summarised from the 608 files of raw data that were produced from all the tests run. However, as previously mentioned, the data that holds the most initial interest consists of the highest results obtained by BE using an optimized DNE test for any given environment. One might notice that not all of the highest results were obtained from the optimized DNE tests and that many came from the REVAC tests. However, the highest results from the REVAC tests should not be counted above the optimized DNE tests as the optimal agents run by REVAC were configured to not be evaluated 100 times like the optimized DNE tests, but 10 times instead *(see table 7).* This was done to avoid REVAC taking too long to run and was assumed to be a good-enough approximation to facilitate the REVAC process. However, the outcome is that there is an increased chance that an agent will get an inflated optimal result by chance.

Based on the analyses of the different test environments as outlined in *chapter 4.1,* certain environments target different aspects of the core components of intelligence. This is directly shown in the *"Target Issues"* column in *table 5* in *chapter 4.1.2* which also highlights the frequency by which credit-assignment is an element of focus, thus mirroring the importance of the idea of *perspective*. Furthermore, there is also an expectation as to which BE modules will help solve which kinds of problems as specified in *chapter 3*. Keeping this in mind, *table 19* and *table 20* highlight which architectural configurations were expected to result in better performance for each environment while comparing them to the descending order of how the configurations actually performed. The configurations that were expected to perform better in a particular environment

are highlighted in green while those expected to perform worse (due to possible added architectural complexity) are blue. Ambiguous or neutral configurations are orange. The *1st* rank represents the highest performance while rank *16* is the lowest. Contiguous results with the same values are marked as *"N/A"*.

The expected colour coded performances for the B-Suite tests were devised by cross-referencing the "*Target Issues"* column in *table 5* as well as the information depicted in *table 10* with the aspects of intelligence that each module is *intended* to target as summarised in *table 1* in *chapter 3.7*. Modules that target aspects of intelligence present within an environment (green) are hence expected to have better performance as opposed to ones that may hinder performance (blue). The ambiguous modules (orange) vary between including MLSH, DNE, GEE, and sometimes Attention. DNE is the backbone of all configurations so it does not ever get highlighted blue, and while exploration may not always be a target aspect of a particular environment, it is mostly unclear what prediction to make with regards to GEE's effect on performance. Furthermore, since MLSH hypothetically has the ability to improve many aspects of intelligence, it was only marked as green where the test environment was complex enough to have multiple sub-goals, such as first swinging up the pole before balancing it in the *Cartpole Swing-Up* test. In some cases (such as the rover mission tests) it is expected that all the modules will assist in performance, so configurations that combine multiple modules were predicted to perform better.

Table 19

Configuration performances from 1st to 8th rank. Green indicates an expected improvement, blue an expected decrease, and orange indicates a neutral prediction. Yellow fields achieved 100% while red fields achieved 0%. "N/A" fields indicate multiple identical results.

Environment/Rank:	1st	2nd	3rd	4th	5th	6th	7th	8th
Move to Target	D	ME	DE	M	MA	MAE	AE	A
1D gradient descent	A	M	MA	AE	MAE	DE	SA	D
2D gradient descent	MAE	A	AE	MA	SMA	SM	SA	S
One-Armed Bandit	DE	SA	A	M	SME	ME	SMA	D
Catch	SM	SMAE	SE	SME	S	SMA	SAE	SA
Cartpole	A	DE	MA	SE	SME	SM	SA	S
Cartpole Swing-Up	ME	DE	M	D	A	MA	SMA	MAE
Mountain Car	SME	SMAE	SE	SAE	MAE	S	ME	M
Deep Sea	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Custom Simplified MNIST	MA	A	D	AE	M	DE	SMA	ME
Custom Umbrella Problem	SM	S	D	M	A	MAE	ME	DE
Discounting Chain	SM	A	S	SA	SMA	MAE	MA	AE
Memory Chain	MA	A	SA	SMAE	SMA	AE	SAE	S
Collect Mission	MA	A	M	D	DE	SME	SE	SAE
Gather Mission	MAE	M	D	SME	SE	SAE	ME	DE
Navigation Mission	SM	SME	ME	D	M	DE	S	SE
Full Rover Mission	M	D	A	SME	SE	SAE	ME	DE
1D Test Functions	MA	AE	MAE	SA	A	SMA	S	SM
2D Test Functions	SMA	SM	S	SA	D	A	M	DE

Table 20

Configuration performances from 9th to 16th rank. Green indicates an expected improvement, blue an expected decrease, and orange indicates a neutral prediction. Yellow fields achieved 100% while red fields achieved 0%. "N/A" fields indicate multiple identical results.

Environment/Rank:	9th	10th	11th	12th	13th	14th	15th	16th
Move to Target	SM	SA	S	SME	SMAE	SE	SAE	SMA
1D gradient descent	SM	S	SMA	ME	SMAE	SAE	SE	SME
2D gradient descent	SME	DE	SE	SAE	SMAE	M	ME	D
One-Armed Bandit	SMAE	MA	AE	MAE	SAE	S	SM	SE
Catch	ME	MAE	MA	M	DE	D	AE	A
Cartpole	SMA	SAE	SMAE	ME	MAE	M	D	AE
Cartpole Swing-Up	SM	S	SE	AE	SA	SME	SMAE	SAE
Mountain Car	DE	D	AE	N/A	N/A	N/A	N/A	N/A
Deep Sea	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Custom Simplified MNIST	SAE	MAE	SA	SME	S	SE	SMAE	SM
Custom Umbrella Problem	SME	SMAE	SE	SAE	AE	SMA	SA	MA
Discounting Chain	M	D	SME	SMAE	SE	SAE	ME	DE
Memory Chain	MAE	SM	D	M	DE	ME	SME	SE
Collect Mission	ME	SMAE	MAE	AE	SA	SMA	S	SM
Gather Mission	SMAE	AE	MA	A	SM	N/A	N/A	N/A
Navigation Mission	MA	MAE	AE	A	SA	SMA	SAE	SMAE
Full Rover Mission	SMAE	MAE	AE	MA	N/A	N/A	N/A	N/A
1D Test Functions	SE	SME	DE	ME	M	D	SMAE	SAE
2D Test Functions	MAE	ME	AE	MA	SAE	SE	SME	SMAE

At a high-level overview, the predictions outlined do *not* hold. There also seems to be little pattern at all as to which configurations will result with better performances given a particular environment. Almost no trend is observed in relation to whether the number of modules enabled in a configuration affects performance. However, by applying a very broad generalisation, a slight bias towards better performance with configurations that only have a few modules enabled emerges. This can be seen by looking at the top three columns and is in contradiction to part of the hypothesis for this thesis which predicts that the cumulation of modules would result in greater performance than the modules on their own. In short, at a practical level, these predictions are unreliable.

The difference in the results compared to what was expected could be due to any number of reasons. However, a reasonable hypothesis may deduce that the way the different solution-space complexities of each configuration behave in an evolutionary sense depends on some potentially intricate underlying function of the environment that is not easily understood. In other words, systems often behave unexpectedly or drastically different when put to practical use, as demonstrated by the results. Furthermore, it may be that adding complexity seems to not always favour an evolutionary approach in this context. This is not to say that the possibility that the evolutionary methods used in BE will not obtain very good results given enough processing time, as they are far less likely to get stuck in local optima as opposed to gradient descent methods. It may just be infeasible to expect such complex systems to optimize themselves on any usable timescale. This aligns with some of the motivations behind methods such as HyperNEAT that identified both the benefits and efficiency limitations of evolutionary methods (Stanley, D'Ambrosio and Gauci, 2009).

Some of the values shown in *table 19* and *table 20* are greyed-out as *"N/A"*. In these cases, the results obtained were exactly the same for all the *N/A* fields and it is therefore meaningless to rank them. In some other fields, there are a few

configurations that also have the same performances without being marked as *N/A*, however, they are limited to two or three in a row and are hence not a major issue. Long sequences of *N/A* fields can be seen in the *Deep Sea* environment tests which is most likely because all configurations have ended up optimising to the locally optimal solution of always figuring out the left path movement (which results in the immediate reward). The maximum reward for only moving right is hence always missed, even with GEE enabled. This could also be the fault of REVAC since the smaller DNE runs used by REVAC may have forced a configuration to be evolved that favours more immediate rewards as opposed to exploration (which could take more generations to optimize).

Some other tests that had sequences of the same result include the 100% accuracy rating held by the *Catch*, *Cartpole*, *Mountain Car*, and *Custom Umbrella* environment tests. These tests, especially *Catch*, were optimally solved by BE. Since multiple configurations achieved 100%, their ranking is no longer relevant. Furthermore, the *Mountain Car*, *Gather Mission*, and *Full Rover Mission* environment tests all hold a sequence of configurations that were unable to find any solution and achieved fitnesses of 0%. *Mountain Car* is particularly interesting as it holds both a sequence of 100% and 0%. This indicates that the environment is easy to solve given that some slightly inconspicuous avenue in the problem space is explored (such as swinging in the dip to gather momentum), but impossible if it is not. All the results that did not achieve 0% were obtained with GEE enabled which indicates that the module assists in the exploration process to some degree. Another example of GEE's use is in the *One-Armed Bandit* environment in which DNE with GEE performed the best, which is as expected since the environment is a basic task that targets exploration.

Another broad analyses of *table 19* and *table 20* identifies a subtle trend in which MLSH appears relatively often in the higher ranked configurations as opposed to DNE by itself. Attention also sees multiple instances in the top rankings while

SNNs and GEE see mixed results between high and low ranking configurations (possibly due to the unnecessary increase in problem space dimensionality they bring). This indicates that the use of the additional modules beyond DNE may still be useful depending on the task at hand, despite what the unpredictability of the configuration rankings may initially seem to indicate. A potential solution may be to incorporate different configurations into the REVAC process or add a meta-meta optimization process as it is evident that the choice of architectural configuration is critical in the AutoML process and cannot be easily predicted by a human. However, this runs into even greater issues regarding processing times and further brings to light the efficiency issue of solely using evolutionary methods. This would of course require more research to explore.

Since the idea of perspective is also a relevant idea for this thesis, looking at the SNN and (particularly) Attention modules in this context can allude to their contribution to temporal generality and (in the case of Attention) credit assignment. The B-Suite environments *Cartpole*, *Cartpole Swing Up*, *Mountain Car*, *Deep Sea*, *Custom Umbrella Problem*, *Discounting Chain* and *Memory Chain* all target credit assignment. Using the Attention unit received mixed results in this regard with only better results than the default configuration in four of the seven environments. However, the strength of the Attention unit's contribution to perspective is directly shown in the results of the *Discounting Chain* and particularly *Memory Chain* which are the most rigorous tasks when it comes to credit assignment and temporal generality. Attention improved the results over the default configuration from 68.74% to 78.41% and 63.00% to 74.60% respectively. The SNN module also saw similar success by beating the default configuration in *Custom Umbrella Problem*, *Discounting Chain*, and *Memory Chain* which all focus on temporal generality. Furthermore, SNNs also saw some success in the *Custom Simplified MNIST* as demonstrated by its frequent appearance in the higher ranks of *table 19*.

Despite the fact that the predictions made do not seem to hold up to the results, comparing configurations that made improvements over the standard DNE configuration is also a focus goal of this thesis. Hence, *Table 21* and *table 22* show the highest and average results (respectively) and their corresponding configuration for each environment. The average results are the final average values for the population of an entire optimized DNE run and show the stability of a configuration across the whole population within the evolutionary process of DNE. The percentage increase that a configuration offers over the default configuration is also shown while the fields marked *"N/A"* indicate that the result value is the same as the default configuration.

The results from *table 21* and *table 22* align with the general expectation that certain module configurations should offer performance improvements above the default evolutionary process of DNE, despite any additional architectural complexity. Besides the *Move to Target* test (for which the default environment was expected to have the best result), the *Catch*, *Cartpole*, *Mountain Car*, *Deep Sea*, and *Collection Mission* tests were the only environments that didn't adhere to this pattern. However, most of these cases can be explained as *Catch*, *Cartpole*, and *Mountain Car* could not have achieved any better results as the default configuration already achieved 100% and *Deep Sea* led all configurations to get stuck in the same local optima. The *Collect Mission* test was the only test that truly defied expectation in both tables as the default configuration produced the best result.

Table 21 and *table 22* also reconfirms the trend favouring simpler configurations over those that have many modules enabled at once, further countering the prediction made that the addition of all modules would have the greatest performances. Of the four additional modules that can be enabled, five of the top configurations in *table 21* only used one additional module, eight used two additional modules, three configurations used two, and none used all four.

Table 21

Table showing the best configurations per environment and their improvements over the default configuration.

Environment	Best Result	Configuration	Increase % From Default Configuration
Move to Target	96.43	D	N/A
1D gradient descent	99.44	A	7.26
2D gradient descent	92.29	MAE	4.01
One-Armed Bandit	84.81	DE	9.67
Catch	100.00	N/A	N/A
Cartpole	100.00	N/A	N/A
Cartpole Swing-Up	67.50	ME	5.53
Mountain Car	100.00	N/A	N/A
Deep Sea	83.33	N/A	N/A
Custom Simplified MNIST	78.00	MA	54.15
Custom Umbrella Problem	100.00	MA/SA/SMA	28.21
Discounting Chain	80.20	SM	16.67
Memory Chain	77.60	MA	23.17
Collect Mission	62.00	D	N/A
Gather Mission	51.60	M	285.07
Navigation Mission	88.80	SM	2.07
Full Rover Mission	3.50	M	105.88
1D Test Functions	86.10	MA	21.08
2D Test Functions	79.59	SMA	2.92

Table 22

Table showing the best configurations using the average DNE population data per environment and their improvements over the default configuration.

Environment	Best Average	Configuration	Increase % From Default Configuration
Move to Target	79.83	ME	5.08
1D gradient descent	92.68	SM	7.89
2D gradient descent	90.85	SMA	9.47
One-Armed Bandit	73.14	SME	7.35
Catch	83.49	SA	116.02
Cartpole	67.55	M	291.59
Cartpole Swing-Up	49.66	ME	70.13
Mountain Car	60.38	ME	80.19
Deep Sea	83.33	S/SM/SA/SMA	11.31
Custom Simplified MNIST	58.22	A	63.82
Custom Umbrella Problem	80.25	S	17.50
Discounting Chain	75.12	S	12.73
Memory Chain	63.21	S	15.71
Collect Mission	47.24	D	N/A
Gather Mission	20.57	M	233.93
Navigation Mission	78.18	M	27.14
Full Rover Mission	1.70	M	178.69
1D Test Functions	78.12	S	23.18
2D Test Functions	79.59	SMA	7.42

It is evident from both *table 21* and *table 22* that BE was successfully able to generate good results across a wide range of task environments with at least one of its configurations. This is in line with the main goal of this thesis of attempting to create a general cognitive architecture that is able to find solutions to a wide set of disparate environments using AutoML. Even the averages of the DNE populations have relatively good results compared to the highest results, which indicates that the performance of the population as a whole increased over time and adhered to the expectation of how an effective evolutionary process should behave.

It is worth noting the 67.50% accuracy for the best *Cartpole Swing Up* test is better than it initially seems since there are multiple steps that the agent must take to swing the pole up before it can receive a reward (as previously stated in *table 4*). Furthermore, BE achieved very good results in the standard 1D and 2D test functions with high scores of 86.10 % and 79.59% respectively. Since these environments test the ability of BE to create agents that are general themselves (due to the fact that any of the test functions could be given in succession to an agent), they are particularly impressive and are a positive outcome for SNNs. However, BE struggled with the *Gather Mission* test and failed the *Full Rover Mission* test, indicating that at larger, more practical scales, the architecture still needs improvement and is either unable to solve such tasks or may require infeasible processing times to do so. This is in contradiction to one of the potential goals of this thesis as BE was partly created to possibly solve tasks like the *Full Rover Mission* test. This means that BE was not able to obtain a sense of *perspective* at a higher level. However, it has also been stated that these kinds of tasks are very challenging to tackle, and it was partly expected that poor results in the space rover missions would be a possible outcome.

In a further effort to analyse the general performance of each of the different modules and their appropriate configurations, *table 23* shows all possible

configurations and their average performance across all environments. This analysis is performed to see if any modules offer a general performance increase across all tasks as the results shown in *table 19* and *table 20* indicate that it is not always obvious which modules will assist in which kinds of tasks. This means that there may be some trends in which certain configurations are more likely than others to contribute positively to an agent's performance. The MBFs for each configuration are compared to the default configuration by giving the percentage increase from the default configuration as well as the *p-value* of a *t-test* done on the comparison of the two pieces of data. The formula for a t-test is given in *Equation (8)* where $\bar{x}$ is the mean of a set, s is the standard deviation and n is the size of the set (which in this case is 19 as there are 19 environments). The p-value is obtained from the t-value by using an appropriate lookup table for a *two-tailed* t-test using 18 degrees of freedom since the null hypothesis proposes that the configuration in question has a distribution that is the same as the default configuration, which can be rejected if it is either higher *or* lower than the default configuration.

Table 15

Table showing the general performances of each configuration across all environments compared to the default configuration. Statistically significant results are blue.

Configuration	MBF Across All Tests	Delta % From Default Configuration	P-Value
Default (DNE)	72.39	N/A	N/A
S	60.05	-12.33	0.199
M	74.20	+1.81	0.826
A	63.07	-9.32	0.364
S+M	56.51	-15.88	0.121
S+A	54.46	-17.93	0.097
M+A	63.64	-8.75	0.399
S+M+A	52.76	-19.62	0.065
D+E	70.40	-1.99	0.825
S+E	53.68	-18.71	0.048
M+E	69.50	-2.89	0.749
A+E+E	66.13	-6.26	0.506
S+M+E	54.40	-17.99	0.062
S+A+E	50.50	-21.89	0.025
M+A+E	64.77	-7.62	0.414
S+M+A+E	50.96	-21.43	0.033

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{8}$$

The way the data is presented in *table 23* in combination with the unpredictability depicted by *table 19* and *table 20* highlights the fact that each environment has a certain configuration bias that isn't necessarily humanly predictable. None of the configurations had a positive delta percent over the default configuration except for MLSH in isolation. However, *table 21* and *table 22* both show that certain configurations are indeed beneficial. This ties in with the *No-Free-Lunch* theorem which states that no black-box algorithm can outperform a task-specific solution when averaged across all theoretical tasks, and the more dependency BE places on AutoML, the more it relies on being a black-box approach (Eiben and Smith,

2015). Although the fact that MLSH was the only configuration to achieve a positive delta (which is testament to its strength as a general AI approach), the result was not statistically significant. Results are statistically significant if their p-value is less than 0.05. Furthermore, the only statistically significant results had negative deltas and were obtained by configurations with multiple modules enabled. This is once again further evidence to suggest that additional complexity (at least in the context of BE's evolutionary system) is not necessarily good.

Turning attention to REVAC, one of the potential issues surrounding REVAC is that since DNE runs within REVAC have to be scaled down in order to maintain reasonable processing times, REVAC may end up optimizing a set of configurations to work for the scaled down DNE runs and not the extended DNE runs. This can result in undesirable performance as the scaled down DNE runs are encouraged to grab immediate rewards as opposed to exploring more distant and potentially greater rewards due to the limit on the number of generations available. This can be seen in the fact that all results for the *Deep Sea* test got stuck in the same locally optimal results. Keeping this in mind, it may (or may not) be that a human-configured test could achieve greater results in these cases as the user can configure a DNE run with these expectations in mind. However, human-configured tests were not explored for this thesis as they were out of scope and the focus is on whether BE can figure out everything itself, regardless of any potentially better results that may be possible with more human intervention. *Table 24* shows the difference between the best results achieved for REVAC and the best results achieved by the extended optimized DNE tests across all environments for each configuration and their associated *p-values*.

It has already been identified earlier in this chapter that some REVAC results are higher than the optimized DNE results, however *table 24* shows that each configuration always does better with REVAC than the optimized DNE tests when averaged across all environments, except in the case of SNNs in isolation.

Table 16

Table showing the comparison of REVAC's MBF results vs the extended optimized DNE's MBF results.

Configuration	REVAC MBF	Optimized DNE MBF	Delta	P-Value
Default (DNE)	76.58	72.39	-4.19	0.643
S	59.41	60.05	0.64	0.951
M	77.00	74.20	-2.8	0.748
A	72.15	63.07	-9.08	0.411
S+M	59.22	56.51	-2.71	0.805
S+A	55.22	54.46	-0.76	0.949
M+A	72.53	63.64	-8.89	0.426
S+M+A	55.66	52.76	-2.9	0.805
D+E	75.95	70.40	-5.55	0.544
S+E	59.80	53.68	-6.12	0.512
M+E	75.60	69.50	-6.1	0.507
A+E+E	71.82	66.13	-5.69	0.555
S+M+E	59.30	54.40	-4.9	0.607
S+A+E	57.45	50.50	-6.95	0.472
M+A+E	72.21	64.77	-7.44	0.435
S+M+A+E	57.88	50.96	-6.92	0.490

On the other hand, none of these differences are statistically significant as all the p-values are greater than 0.05. However, the results do show that REVAC is optimizing its parameters to the scaled down DNE configurations that it uses, and that running extended optimized DNE tests (although not *statistically* worse) is at the very least, not worth it.

It has also already been pointed out that REVAC may obtain higher results by chance and that the results obtained by the optimized DNE tests were more robustly evaluated. However, the consistent rate that DNE is not statistically different indicates that REVAC should either not be used (in which case a human would attempt to configure everything, thus diminishing the extent of the AutoML process), or a new approach to HPO needs to be considered that allows optimal configurations to be found for extended tests without drastically increasing processing times. Further research would need to be done on this avenue as REVAC indeed seems to be too inefficient when combined with all the other evolutionary processes present in BE.

Chapter 6

Conclusions

The primary goal of this thesis was to attempt to create a general cognitive architecture (*Brain Evolver*) using evolutionary *Reinforcement Learning* (RL) with a subtle focus on the concept of *perspective*. The architecture (BE) is intended to fully implement the process of *AutoML* which entails automatically creating and configuring intelligent agents that can perform sufficiently well in any target environment. In short, this goal was achieved as the software BE was successfully developed such that it is able to solve a wide variety of tests on its own. Despite BE's limitations and potential need for further development, the task of creating a general problem solver in the pursuit of true, general AI is an extremely challenging one, and the results obtained (as shown in *table 21*) indicate that BE was able to adequately solve almost all of the test environments except for the *Full Rover Mission* test.

A further success of BE is that it has been created to be customizable, modular, and flexible enough so that others may possibly use it as a tool for further research. This is an important aspect of the original intent behind the goal of creating a general architecture, and much effort was put into creating a piece of software that not only has a wide range of functionalities but is also user-friendly with a clear GUI and insightful visualisations.

Since the creation of a cognitive architecture is such a broad and open-ended task, one of the first goals of the thesis was to try find a definable analysis of intelligence to use as a guideline in the construction of BE. As expected, this required navigating through an extensive pool of related research by trying to

pick what is relevant and useful while still focusing on the generality of the research topic. The literature reviewed therefore successfully covers a relatively wide base of approaches and methods in AI, however, it is all directed by the initial analysis of what general intelligence really is, as predominantly outlined in *chapter 2.1*. This analysis is also abstracted from a large body of research and hypothetical opinions. Ultimately, the use of the cognitive modelling approach in taking inspiration from nature and moving away from gradient descent methods largely scoped the construction of BE (governing the initial choice to focus on evolutionary RL) along with significant inspiration from *Osband's* analysis of intelligent agents (Osband *et al.*, 2020). This resulted in an effective categorisation regarding certain elements of intelligence to target, along with a direction outlining how BE may implement such solutions. Furthermore, the idea of perspective is illuminated in the attempt to obtain temporal contextuality and high-level problem comprehension without resorting to rigidly or explicitly storing information gathered from an environment. Perspective is hence abstractly represented by credit-assignment and temporal generality.

Osband's analysis of intelligence also predominantly contributed to the tests constructed for BE (*B-Suite*) as they directly relate to the elements of intelligence outlined. However, additional environments were built that included common test functions and a set of space rover simulation tests. The addition of the space rover simulation tests is based on the fact that one of the drives behind creating more robust, general AI with the capability of understanding tasks with perspective is to perform well in similar complex real-world environments.

The analysis of intelligence led to the review of multiple different approaches to AI including RL, various metaheuristic and evolutionary approaches, Hyper Parameter Optimization, various gradient descent and evolutionary network-based approaches, Hierarchical Learning, and Attention. As outlined in the goals of this thesis, a set of questions were defined in the context of general cognitive

architectures based on the implementation of solutions to these approaches using core components of intelligence. These questions involved the construction of BE's modules and the subsequent analysis of their effectiveness and contribution to the general problem-solving skills and AutoML process of BE. This thesis hence focused on BE's ability to be general as opposed to being overly concerned with what the best performances achieved in isolation were. Reasonable results for most environments were thus valued higher than greater results for only a few environments.

The modules successfully developed include *Deep Neural Evolution* (DNE), *Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters* (REVAC), *Meta Learning Shared Hierarchies* (MLSH), *Attention*, *Spiking Neural Networks* (SNNs), and a custom exploration unit *Guided Epsilon Exploration* (GEE). The hypothesis thus proposed for this thesis is that these methods implemented together in BE will sufficiently solve the standard test functions selected, the tasks outlined by *B-Suite*, and the space rover tests. As already stated, most environments were adequately solved which hence supports the hypothesis.

An extension of the hypothesis and goal of analysing intelligence sees the prediction that all modules combined will perform better than those on their own, and that certain modules will be better at solving certain tasks. There is also the counter argument of complexity (relating to the *curse of dimensionality*) and the often-slow conversion rates of evolutionary methods. However, the results shown in *table 19* and *table 20* do *not* support this part of the hypothesis as there seems to be little correlation between which module configurations have better performance, with a slight trend indicating that configurations using multiple modules actually perform slightly worse on average. This highlights the unreliability of using human predictions to guide the construction of an AI solution. It means that different modules may or may not help given a particular

environment and supports the need to further extend the AutoML process. However, this refocuses the issue of complexity and the practical useability of such an implementation given limited hardware and time, and encapsulates one reason why true, general intelligence is so difficult to research. It may indeed be, to some degree, that such intelligent systems *require* unreasonably extensive processing times. However, it may also be that by continually attempting to find a research pathway to general AI, regardless of the immediate results, these kinds of efforts will eventually lead to the elusive goal of true AI.

On the other hand, the three additional environments appended to the *B-Suite* tests targeted whether BE could still solve simple tasks given its complex architecture. These three environments included the *Move to Target*, *2D Gradient Descent,* and *1D Gradient Descent* tests. While all environments were adequately solved, *Move to Target* achieved an accuracy of 96.43%. This highlights the heuristic nature of evolutionary approaches as a drawback since the *Move to Target* test simply required a direct relationship between all inputs and outputs as a solution, making it trivial to achieve 100% using a gradient descent-based method. *1D Gradient Descent* achieved a strong result of 99.44% but *2D Gradient Descent* achieved a slightly lower 92.29% which again highlights the issue, especially if agents need to be very precise in the tasks they have been given. It can be concluded by this that in order to solve more complex tasks, greater complexity and perspective may come at a detrimental cost to fine-grained preciseness and accuracy.

Despite the unpredictability of a configuration's performance given a particular environment, *table 21* and *table 22* show that certain configurations and modules can have very significant performance increases beyond the default configuration (which consists of DNE by itself). This is particularly prevalent in the space rover missions, especially in the *Gather Mission* test which achieved a 285.07% increase using MLSH over the default DNE configuration. However, it must also not be

understated that the use of *Deep Learning* present in BE's custom implementation of DNE was able to obtain significant results across the general collection of tests conducted, thus supporting its design choice as the fundamental component in BE.

It is very important to note that, beyond the failure of BE to solve the *Full Rover Mission* test, the highest result obtained across all four space rover tests was 88.80% accuracy, which is still not sufficient to be a practical solution for any real-world space rover mission. This is because these kinds of tasks required by space rover missions are often very high-risk and one small, incorrect action can sometimes have a disastrous impact. With this in mind, 88.80% percent accuracy is still simply too unreliable for such high-stake tasks and much more development would be needed before such systems would be trusted. However, for simpler, less important tasks, a rover may be more likely to rely on AI without human intervention. Solutions provided by BE may also still be useful for high-risk tasks if they are combined with other, more predictable and reliable methods and used as a recommendation of sorts. This can also be related back to the *problem statement* of *chapter 1.1* which largely outlined the use of intelligent agents in our society's growing dependence on technology and AI alike. The conclusion drawn is that practically, more predictable and simpler solutions are most likely going to be the predominantly favourable choice in the context of our current understanding of AI and that as more complex or experimental solutions (such as those offered by BE) develop, they would best be suited to first become generally used in less conspicuous or inconsequential tasks.

Another conclusion made by further exploring the data found that different configurations do *not* have a general improvement across all environments beyond finding a specific configuration for that environment. Additionally, REVAC was both problematic and useful. Its usefulness comes in its ability perform HPO which, as indicated by the unpredictability of the results, is a

valuable component of AutoML. However, it is problematic in the sense that its optimizations do not apply well to the extended optimized DNE tests. This means that if it must optimize large scale DNE tests, its processing times would start to increase unreasonably.

Although the evolutionary methods used to implement BE's modules are shown to be imprecise or heuristic-like in nature (as is a common drawback of evolution), when coupled with the concept of RL, evolution can be concluded to be very useful. Many of the environments were partially observable and required agents to discover things by themselves. This means that, due to the unsupervised nature of many of the problems, it would have been very challenging to solve them using gradient descent as there would be no source of knowledge to regress against. Evolutionary methods hence work well in these cases as the reward and punishment feedback system of RL aligns neatly with the fitness function of an evolutionary method.

In analysing the individual components, it was concluded that MLSH performed significantly well and was on average the most valuable additional module beyond DNE by itself. This highlights the importance of addressing *hierarchies* within problems as the basis of any AI that can tackle complex, high-level problems.

Both the SNN module and GEE had mixed results when enabled, but both often seemed to unnecessarily increase complexity without getting drastic performance increases. This is not to say that the SNN module and GEE were not beneficial for some environments. GEE seemed to assist in environments such as the *Cart Pole Swing Up* test (which targets exploration) but could not assist the *Deep Sea* test environment from getting out of the common local optimum. Ultimately, GEE's performance (as a novel method) proved to be not very relevant and could have potentially been replaced by a simpler exploration method. However, the results indicate that under the right circumstances it can

still be useful, and it may also be possible that GEE could be more valuable if it is given enough time to optimize itself.

SNNs had successful results in the *Catch*, *Mountain Car*, *Custom Umbrella Problem*, and *2D Test Functions* test environments but had negative results for multiple other environments. Some good results were also seen with SNNs in the *Custom Simplified MNIST* environment, indicating that it may indeed be assisting with noise reduction (since both SNNs and *Custom Simplified MNIST* target this issue). Additionality, SNNs formed part of the top performing configurations in the standard 1D and 2D test functions (which test generality of the agents produced themselves) and thus supports SNNs use in this case. However, SNNs add a significant increase in evolvable parameters and due to the degree to which changing them can affect performance, a longer evolutionary period or larger population may be needed to optimize the SNN module further. An example of this would be the *learning window* parameter of an SNN node which initially starts out as being only one time-step and would first need to evolve to become larger before it can evolve to be optimized.

The Attention module also saw some strong results in assisting with environments that required temporal context and credit assignment (perspective) as outlined in *chapter 5.1*. It is hence also evident from the results that it can be concluded that additional temporal context *can* be obtained by BE despite the fact that agents never explicitly store historical information. This is a positive result that aligns with one of the goals of this thesis. However, to have significant practical results, this *implicit* approach to memory would most likely need to be combined with a more *explicit* approach to historical knowledge, like other cognitive architectures such as *Soar* (Cooper *et al.*, 1996; Laird, 2008).

Since the notion of perspective also relates to solving problems with an appropriate level of overview, Attention and MLSH have been concluded to be the most effective at improving an agent's ability to achieve this abstract quality.

However, some environments see SNNs and DNE also achieve similar results. This leads to another conclusion which alludes to the idea that perspective may not even be a relevant concept at the level of understanding that the agents produced by BE acquired for each test environment. In other words, despite the intentions behind the purpose of each module and AI method, the results indicate that practically, some methods are simply more likely to optimize to a better function for an environment than others. Furthermore, it is possible that more meaningful results (for all experiments) may only be seen at larger test scales and that by keeping the scale of individual tests lower in order to have more of them ultimately proved to be less helpful. Additionally, although *scalability* was put aside as a focus, it seems that BE does indeed struggle with scalability due to the fact that the largest scaled test (the *Full Rover Mission*) was the only environment that it could not solve in the allotted scope.

Finally, it is worth being reminded that the research done for this thesis as well as its analyses and conclusions are quite broad and general. This is as intended since the insights provided will hopefully guide further research in the topic of general AI and advise on what to avoid and what to explore. There is currently a certain informality about approaching general AI research (and many other black-box approaches in general), however, the broad analyses presented by this thesis through BE will hopefully assist in slowly formalizing and bringing about further understanding in the subject matter. Additionally, while BE may be far from the single source of truth that it aims to target, it is, and can most certainly be, a useful *tool* in assisting the development of other systems that use similar methods.

5.1 Potential Future Research

The two biggest challenges surrounding this thesis involved the scope of developing the program *Brain Evolve* itself due its complexity as a piece of software and the time taken to run all the experiments. The issue of inefficiency opens an avenue for further research that aligns more with the approach that *HyperNEAT* takes in which evolution takes place at a more granular level while gradient descent is incorporated back in at a fine-grained level in order to mitigate slow processing times (Stanley, D'Ambrosio and Gauci, 2009). As already alluded to, combining elements of explicit memory with BE's implicit approach to temporal information could also be an area of additional research.

Other potential routes that could be explored involve adding multiple Attention layers (like a Transformer) and including other types of networks such as CNNs and RNNs as modules into the system. Additionally, everything could be incorporated into the MLSH process whereby each ϕ sub-network could evolve to be a different module, thus resulting in a system that works similarly to the *blueprint* model present in *DeepNEAT*. The addition of all these components will most likely not solve the issues experienced around performance and hence begs for potential experiments to be done at very large scales with much faster computers. As already stated, it may be that significant results only begin to emerge at larger scales when using more complex architectures like BE's (Vaswani *et al.*, 2017; Acton *et al.*, 2020).

There were also multiple configuration combinations that were not tested for BE as well as the online-adaption system that was neglected. Some components such as the *activation offset* feature and the use of different possible activation functions for each node were enabled and left for the evolutionary process to optimize. Only one selection method was also used *(exponential selection)*. It was hence difficult to test the significance (in the given scope) of these features and

additional research could see more variations being explored. The capabilities of BE's ability to implement Transfer Learning and solve multiple objectives was also not targeted by the tests conducted for this thesis but remains an open avenue as BE allows for agents to be simultaneously or consecutively tested on multiple environments. Furthermore, there is also room to find a more efficient method of HPO. It is also worth exploring whether the focus of general, true AI is slightly misplaced since increasing a system's capabilities through complex AI architectures often decreases its predictability, making it unsuited for tasks such as those encountered by space rovers.

References

Acton, S. *et al.* (2020) 'Efficiently Coevolving Deep Neural Networks and Data Augmentations', *2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020*, pp. 2543–2550. doi: 10.1109/SSCI47803.2020.9308151.

Adyatama, A. (2019) *Particle Swarm Optimization*, *RPubs*. Available at: https://rpubs.com/argaadya/intro-pso (Accessed: 24 November 2021).

Agatonovic-Kustrin, S. and Beresford, R. (2000) 'Basic Concepts of Artificial Neural Network (ANN) Modeling and Its Application in Pharmaceutical Research', *Journal of Pharmaceutical and Biomedical Analysis*, 22(5), pp. 717–727. doi: 10.1016/S0731-7085(99)00272-1.

Alammar, J. (2018) *The Illustrated Transformer*. Available at: http://jalammar.github.io/illustrated-transformer/ (Accessed: 13 December 2020).

Armstrong, J. M. (2004) 'After the Ascent: Plato on Becoming Like God', *Oxford Studies in Ancient Philosophy*, 26(December 2003), pp. 171–183.

Aspinall, A. and Gras, R. (2010) *K-Means Clustering as a Speciation Mechanism Within an Individual-Based Evolving Predator-Prey Ecosystem Simulation*. University of Windsor. doi: 10.1007/978-3-642-15470-6_33.

Badue, C. *et al.* (2021) 'Self-Driving Cars: A Survey', *Expert Systems with Applications*, 165. doi: 10.1016/j.eswa.2020.113816.

Bansall, S. (2019) *Agents in Artificial Intelligence*, *Geeks for Geeks*. Available at: https://www.geeksforgeeks.org/agents-artificial-intelligence/ (Accessed: 10 December 2020).

Barto, A. G. and Sutton, R. S. (1999) 'Reinforcement Learning', *MIT Press, Cambridge*. Massachusetts Institute of Technology, pp. 126–134. doi: 10.1017/s0269888999003082.

Belani, H., Vuković, M. and Car, Ž. (2019) 'Requirements Engineering Challenges in Building AI-Based Complex Systems', *IEEE*.

Bhandarkar, T. *et al.* (2019) 'Earthquake Trend Prediction Using Long Short-Term Memory RNN', *International Journal of Electrical and Computer Engineering*

(IJECE), 9(2), p. 1304. doi: 10.11591/ijece.v9i2.pp1304-1312.

Blickle, T. and Thiele, L. (1996) 'A Comparison of Selection Schemes Used in Evolutionary Algorithms', *Evolutionary Computation*, 4(4), pp. 361–394. doi: 10.1162/evco.1996.4.4.361.

Bosch, A. van den *et al.* (2011) 'Hierarchical Reinforcement Learning', in *Encyclopedia of Machine Learning*. Boston, MA: Springer US, pp. 495–502. doi: 10.1007/978-0-387-30164-8_363.

Boukas, E. *et al.* (2017) 'Global Localization for Future Space Exploration Rovers', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10528 LNCS, pp. 86–98. doi: 10.1007/978-3-319-68345-4_8.

Branke, J., Kaußler, T. and Schmeck, H. (2001) 'Guidance in Evolutionary Multi-Objective Optimization', *Advances in Engineering Software*, 32(6), pp. 499–507. doi: 10.1016/S0965-9978(00)00110-1.

Bre, F., Gimenez, J. M. and Fachinotti, V. D. (2018) 'Prediction of Wind Pressure Coefficients on Building Surfaces Using Artificial Neural Networks', *Energy and Buildings*, 158(November), pp. 1429–1441. doi: 10.1016/j.enbuild.2017.11.045.

Bresina, J. L. *et al.* (2003) 'Mapgen: Mixed Initiative Planning and Scheduling for the Mars '03 MER Mission', *Proceedings of iSAIRAS*. Available at: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:MAPGEN:+Mixed+Initiative+Planning+and+Scheduling+for+the+Mars+?03+MER+Mission+?#0.

Bresina, J. L. and Morris, P. H. (2007) 'Mixed-Initiative Planning in Space Mission Operations', *AI Magazine*, 28(2), pp. 75–88.

Brinton, C. and Atm, M. (2017) 'A Framework for Explanation of Machine Learning Decisions Analysis', *International Joint Conference on Artificial Intelligence - Explainable AI (XAI) Workshop*. Available at: http://home.earthlink.net/~dwaha/research/meetings/ijcai17-xai/2. (Brinton XAI-17) A Framework for Explanation of Machine Learning Decisions.pdf.

Bryndin, E. (2020) 'Technology Self-Organizing Ensembles of Intelligent Agents with Collective Synergetic Interaction', *Automation, Control and Intelligent Systems*, 8(4), p. 29. doi: 10.11648/j.acis.20200804.11.

Champrasert, P., Suzuki, J. and Otani, T. (2009) 'Constraint-Based Evolutionary QoS Adaptation for Power Utility Communication Networks', *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, (Figure 1), pp. 395–403. doi: 10.1109/ICTAI.2009.113.

Cheng, Y. *et al.* (2004) 'The Mars Exploration Rovers Descent Image Motion Estimation System', *IEEE Intelligent Systems*, 19(3), pp. 13–21. doi: https://doi.org/10.1109/MIS.2004.18.

Clarke, A. M. and Sternberg, R. J. (1986) 'Beyond IQ: A Triarchic Theory of Human Intelligence', *British Journal of Educational Studies*, 34(2), p. 205. doi: 10.2307/3121332.

Clarke, P. (2018) *ETA Adds Spiking Neural Network Support to MCU*, *eeNews Europe*. Available at: https://www.eenewseurope.com/news/eta-adds-spiking-neural-network-support-mcu-0 (Accessed: 14 December 2020).

Cooper, R. *et al.* (1996) 'A Systematic Methodology for Cognitive Modelling', *Artificial Intelligence*, 85(1-2 SPEC. ISS.), pp. 3–44. doi: 10.1016/0004-3702(95)00112-3.

Daniel Buchmueller (2017) 'Tether Compensated Airborne Delivery'. United States.

Dietterich, T. G. (2000) 'Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition', *Journal of Artifcial Intelligence Research*, 13, pp. 227–303.

Donald Michie (1968) '"MEMO" Functions And Machine Learning', *Nature*, 218(5136), pp. 19–22. Available at: https://stacks.stanford.edu/file/druid:jt687kv7146/jt687kv7146.pdf.

Dréo, J. (2006) *Find the Shortest Path with ACO*. Available at: https://commons.wikimedia.org/wiki/File:Aco_shortpath.svg (Accessed: 24 November 2021).

Eiben, A. E. and Smith, J. E. (2015) 'Problems to Be Solved', in *Evolutionary Computation*, pp. 1–12. doi: 10.1007/978-3-662-44874-8_1.

Ferguson, D. and Stentz, A. (2007) 'Field D*: An Interpolation-Based Path Planner and Replanner', *Springer Tracts in Advanced Robotics*, 28. doi: 10.1007/978-3-540-48113-3_22.

Fil, J. and Chu, D. (2020) 'Minimal Spiking Neuron for Solving Multilabel Classification Tasks', *Neural computation*, 32(7), pp. 1408–1429. doi: 10.1162/neco_a_01290.

Frans, K. *et al.* (2018) 'Meta Learning Shared Hierarchies', *ICLR*, pp. 1–11.

Fujimoto, S., Meger, D. and Precup, D. (2018) 'Off-Policy Deep Reinforcement Learning Without Exploration', in *Proceedings of the 36th International*

Conference on Machine Learning.

Glover, F. (1986) 'Metaheuristics', *Encyclopedia of Operations Research and Management Science*. Springer, New York.

Gobeyn, S. *et al.* (2019) 'Evolutionary Algorithms for Species Distribution Modelling: A Review in the Context of Machine Learning', *Ecological Modelling*, 392(June 2018), pp. 179–195. doi: 10.1016/j.ecolmodel.2018.11.013.

Goldberg, D. E. and Holland, J. H. (1988) 'Genetic Algorithms and Machine Learning', *Machine Learning*, 3, pp. 95–99. doi: https://doi.org/10.1023/A:1022602019183.

Gomez, F. and Miikkulainen, R. (2006) 'Efficient Non-linear Control Through', *Control*, pp. 654–662.

Gupta, A. *et al.* (2018) *Meta-Reinforcement Learning of Structured Exploration Strategies*. University of California, Berkeley.

Gupta, D. S. (2020) *Fundamentals of Deep Learning – Activation Functions and When to Use Them?*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/ (Accessed: 12 December 2020).

Hadjiivanov, A. and Blair, A. (2016) 'Complexity-Based Speciation and Genotype Representation for Neuroevolution', *2016 IEEE Congress on Evolutionary Computation, CEC 2016*, pp. 3092–3101. doi: 10.1109/CEC.2016.7744180.

Hassabis, D. (2017) 'Chess Match of the Century', *Nature*, 544, pp. 413–414.

Helman, P. (1986) 'The principle of optimality in the design of efficient algorithms', *Journal of Mathematical Analysis and Applications*, 119(1–2), pp. 97–127. doi: 10.1016/0022-247X(86)90147-2.

Holland, O. and Gamez, D. (2009) 'Artificial Intelligence and Consciousness', *Encyclopedia of Consciousness*, pp. 37–45. doi: 10.1016/B978-012373873-8.00004-9.

Hunt, J. (2010) 'A Short Note on Continuous-Time Markov and Semi-Markov Processes'.

Hutter, F. (2014) *Meta-learning*, *Studies in Computational Intelligence*. doi: 10.1007/978-3-319-00960-5_6.

Jabri, A. *et al.* (2019) 'Unsupervised Curricula for Visual Meta-Reinforcement Learning', *Advances in Neural Information Processing Systems*, 32(NeurIPS).

Jakobi, N., Husbands, P. and Harvey, I. (1995) *Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics*. University of Sussex. doi: 10.1007/3-540-59496-5_337.

Joyeux, S., Schwendner, J. and Roehr, T. M. (2014) 'Modular Software for an Autonomous Space Rover', *International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS)*.

Karpathy, A. and Leung, T. (2014) 'Karpathy Large-Scale Video Classification with Convolutional Neural Networks', *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 10–20. Available at: http://cs.stanford.edu/people/karpathy/deepvideo.

Ke, N. R. *et al.* (2018) 'Sparse Attentive Backtracking: Temporal Credit Assignment Through Reminding', in *32nd Conference on Neural Information Processing Systems*, pp. 1–12.

Kocsis, L. and Szepesvari, C. (2006) 'Bandit Based Monte-Carlo Planning', *European conference on machine learning*, pp. 282–293. Available at: https://link.springer.com/content/pdf/10.1007/11871842_29.pdf.

Laird, J. E. (2008) 'Extending the Soar Cognitive Architecture', *Frontiers in Artificial Intelligence and Applications*, 171(1), pp. 224–235.

Langley, P., Choi, D. and Shapiro, D. (2004) 'A Cognitive Architecture for Physical Agents', *Aaai-2004*, pp. 1469–1474.

Long, L. N. *et al.* (2007) 'A Review of Intelligent Systems Software for Autonomous Vehicles', *Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Security and Defense Applications, CISDA 2007*, (Cisda), pp. 69–76. doi: 10.1109/CISDA.2007.368137.

Lozano, M., Molina, D. and Herrera, F. (2011) 'Editorial Scalability of Evolutionary Algorithms and Other Metaheuristics for Large-Scale Continuous Optimization Problems', *Soft Computing*, 15(11), pp. 2085–2087. doi: 10.1007/s00500-010-0639-2.

M, R. (2019) *The Ascent of Gradient Descent*, *Clairvoyant*. Available at: https://blog.clairvoyantsoft.com/the-ascent-of-gradient-descent-23356390836f (Accessed: 28 November 2021).

Mahesh, K., Nallagownden, P. and Elamvazuthi, I. (2016) 'Advanced Pareto Front Non-Dominated Sorting Multi-Objective Particle Swarm Optimization for Optimal Placement and Sizing of Distributed Generation', *Energies*, 9(12), p. 982. doi: 10.3390/en9120982.

Mahoney, M. S. (1988) 'The History of Computing in the History of Technology', *Annals of the History of Computing 10*, 10, pp. 113–125. Available at: https://www.princeton.edu/~hos/mike/articles/hcht.pdf.

Maimone, M. W., Leger, P. C. and Biesiadecki, J. J. (2007) 'Overview of the Mars Exploration Rovers' Autonomous Mobility and Vision Capabilities', *IEEE International Conference on Robotics and Automation, Space Robotics Workshop*, pp. 1–8.

Matrossov, S. *et al.* (1992) 'RCL'S Advanced High Mobility Locomotion Systems'.

McCarthy, J. *et al.* (2006) 'A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence', *AI Magazine*, 27(4), pp. 12–14.

McLeod, S. A. (2007) 'Skinner - Operant Conditioning', pp. 1–4. Available at: http://www.simplypsychology.org/operant- conditioning.html.

Miikkulainen, R. *et al.* (2018) 'Evolving Deep Neural Networks', *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pp. 293–312. doi: 10.1016/B978-0-12-815480-9.00015-3.

Miikkulainen, R. and Lehman, J. (2013) *Neuroevolution*, *Scholarpedia*. Available at: http://scholarpedia.org/article/Neuroevolution (Accessed: 13 December 2020).

Minsky, M. (1961) 'Steps Toward Artificial Intelligence', *Proceedings of the IRE*, 49(1), pp. 8–30. doi: 10.1109/JRPROC.1961.287775.

Mirfenderesgi, G. and Mousavi, J. (2015) 'Adaptive Meta-Modeling-Based Simulation Optimization in Basin-Scale Optimum Water Allocation: A Comparative Analysis of Meta-Models', *Journal of Hydroinformatics*, 18. doi: 10.2166/hydro.2015.157.

Mishra, S. K. (2011) 'Performance of Repulsive Particle Swarm Method in Global Optimization of Some Important Test Functions: A Fortran Program', *SSRN Electronic Journal*. doi: 10.2139/ssrn.924339.

Mnih, V. *et al.* (2013) 'Playing Atari with Deep Reinforcement Learning', pp. 1–9. Available at: http://arxiv.org/abs/1312.5602.

Nachum, O. and Lee, H. (2018) 'Data-Efficient Hierarchical Reinforcement Learning', in *32nd Conference on Neural Information Processing Systems*.

Nadikattu, R. R. (2016) 'The Emerging Role of A.D.M.E.', *International Journal of Creative Research Thoughts*, 4(4), pp. 906–911. doi: ISSN: 2320-2882.

Newell, A. (1990) *Unified Theories of Cognition*. Cambridge, Massachusetts:

Harvard University Press.

Odziemczyk, W. (2020) 'Application of Simulated Annealing Algorithm for 3D Coordinate Transformation Problem Solution', *Open Geosciences*, 12(1), pp. 491–502. doi: 10.1515/geo-2020-0038.

Ohnishi, S. *et al.* (2019) 'Constrained Deep Q-Learning Gradually Approaching Ordinary Q-Learning', *Frontiers in Neurorobotics*, 13(December), pp. 1–19. doi: 10.3389/fnbot.2019.00103.

Oke, S. A. (2008) 'A literature Review on Artificial Intelligence', *International Journal of Information and Management Sciences*, 19(4), pp. 535–570.

Olague, G. (2016) 'Evolutionary Computing', *Natural Computing Series*, (9783662436929), pp. 69–140. doi: 10.1007/978-3-662-43693-6_3.

Omohundro, S. M. (2008) 'The Basic AI Drives', *Frontiers in Artificial Intelligence and Applications*, 171(1), pp. 483–492. doi: 10.18254/s2077518000009748-1.

Osband, I. *et al.* (2020) 'Behaviour Suite for Reinforcement Learning', in *ICLR*, pp. 1–19.

Pan, S. J. and Fellow, Q. Y. (2009) 'A Survey on Transfer Learning', *IEEE*, pp. 1–15. doi: 10.1109/TKDE.2009.191.

Pan, S. J., Kwok, J. T. and Yang, Q. (2008) 'Transfer Learning via Dimensionality Reduction', *Proceedings of the National Conference on Artificial Intelligence*, 2, pp. 677–682.

Perspective (2020) *Oxford Learner's Dictionaries*. Available at: https://www.oxfordlearnersdictionaries.com/definition/english/perspective (Accessed: 9 December 2020).

Poggio, T. *et al.* (2017) 'Why and When can Deep-but Not Shallow-Networks Avoid the Curse of Dimensionality: A Review', *International Journal of Automation and Computing*, 14(5), pp. 503–519. doi: 10.1007/s11633-017-1054-2.

Qiu, H. *et al.* (2020) 'Towards Crossing the Reality Gap with Evolved Plastic Neurocontrollers', *GECCO 2020 - Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pp. 130–138. doi: 10.1145/3377930.3389843.

Rubenstein, D. I. (1982) 'Risk, Uncertainty and Evolutionary Strategies', pp. 91–111.

Ruder, S. (2016) *An Overview of Gradient Descent Optimization Algorithms*. Available at: https://ruder.io/optimizing-gradient-descent/ (Accessed: 12 December 2020).

Saha, S. (2018) *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way, Towards Data Science*. Available at: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (Accessed: 28 November 2021).

Sandra May (2021) *NASA's Picture Dictionary - Rover, NASA*. Available at: https://www.nasa.gov/audience/forstudents/k-4/dictionary/Rover.html (Accessed: 1 August 2021).

Scerri, P. (2006) *Cognition and Multi-Agent Interaction - From Cognitive Modeling to Social Simulation*. Edited by R. Sun. Cambridge University Press.

Schatten, A. (1995) 'The Application of Software Agent Technology to Health Care'.

Shen, Y. *et al.* (2019) 'Interpreting the Latent Space of GANs for Semantic Face Editing', *IEEE*, pp. 9243–9252.

Shyalika, C. (2019) *A Beginners Guide to Q-Learning, Towards Data Science*. Available at: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c (Accessed: 23 November 2021).

Smit, S. K. and Eiben, A. E. (2010) 'Beating the "World Champion" Evolutionary Algorithm via REVAC Tuning', *2010 IEEE World Congress on Computational Intelligence, WCCI 2010 - 2010 IEEE Congress on Evolutionary Computation, CEC 2010*. doi: 10.1109/CEC.2010.5586026.

Squire, L. R. and Dede, A. J. O. (2015) 'Conscious and Unconscious Memory', *Cold Spring Harbor Laboratory Press*. doi: 10.1101/cshperspect.a021667.

Stanley, K. O., D'Ambrosio, D. B. and Gauci, J. (2009) 'A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks', *Artificial Life*, 15(2), pp. 185–212. doi: 10.1162/artl.2009.15.2.15202.

Stanley, K. O. and Miikkulainen, R. (2002) 'Evolving Neural Networks Through Augmenting Topologies', *Evolutionary Computation*, 10(2), pp. 99–127. doi: 10.1162/106365602320169811.

Stevens, R. and Soller, A. (2005) 'Machine Learning Models of Problem Space Navigation: The Influence of Gender', *Computer Science and Information Systems*, 2(2), pp. 83–98. doi: 10.2298/csis0502083s.

Tambe, M. *et al.* (1995) 'Intelligent Agents for Interactive Simulation Environments', *AI Magazine*, 16(1), pp. 15–39.

Tavanaei, A. *et al.* (2019) 'Deep Learning in Spiking Neural Networks', *Neural*

Networks, 111, pp. 47–63. doi: 10.1016/j.neunet.2018.12.002.

Taylor, A. (2017) *When the Brain's Wiring Breaks*, *UNC Health Talk*. Available at: https://healthtalk.unchealthcare.org/when-the-brains-wiring-breaks/.

Thurstone, T. (1962) 'PMA (Primary Mental Abilities)'. Available at: http://www.worldcat.org/title/pma-primary-mental-abilities/oclc/8067751&referer=brief_results.

Tiu, E. (2020) *Understanding Latent Space in Machine Learning*, *Towards Data Science*. Available at: https://towardsdatascience.com/understanding-latent-space-in-machine-learning-de5a7c687d8d (Accessed: 10 December 2020).

Tokic, M. (2010) 'Adaptive ε-Greedy Exploration in Reinforcement Learning Based on Value Differences', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6359 LNAI, pp. 203–210. doi: 10.1007/978-3-642-16111-7_23.

Turney, P., Whitley, D. and Anderson, R. W. (1996) 'Evolution, Learning, and Instinct: 100 Years of the Baldwin Effect', *Evolutionary Computation*, 4(3), pp. iv–viii. doi: 10.1162/evco.1996.4.3.iv.

Utzle, T. S. *et al.* (2010) 'Parameter Adaptation in Ant Colony Optimization IRIDIA – Technical Report Series Parameter Adaptation in Ant Colony Optimization', *IRIDIA*, (January).

Vaswani, A. *et al.* (2017) 'Attention is All You Need', *Advances in Neural Information Pro-cessing Systems*, pp. 6000–6010. Available at: https://papers.nips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Vedaldi, A. and Lenc, K. (2015) 'MatConvNet: Convolutional Neural Networks for MATLAB', *MM 2015 - Proceedings of the 2015 ACM Multimedia Conference*, pp. 689–692. doi: 10.1145/2733373.2807412.

Vezhnevets, A. S. *et al.* (2017) 'FeUdal Networks for Hierarchical Reinforcement Learning', in *Proceedings of the 34th International Conference on Machine Learning*.

Voß, S. (2001) 'Meta-Heuristics: The State of the Art', *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, 2148, pp. 1–23. doi: 10.1007/3-540-45612-0_1.

Watkins, C. J. C. . and Dayan, P. (1992) 'Q-Learning', *Machine Learning*, 8, pp. 279–292. doi: 10.4018/978-1-59140-993-9.ch026.

West, M. (2020) *Explaining Recurrent Neural Networks*. Available at: https://www.bouvet.no/bouvet-deler/explaining-recurrent-neural-networks (Accessed: 13 December 2020).

Wilde, H., Knight, V. and Gillard, J. (2020) 'Evolutionary Dataset Optimisation: Learning Algorithm Quality Through Evolution', *Applied Intelligence*, 50(4), pp. 1172–1191. doi: 10.1007/s10489-019-01592-4.

Wu, Y. *et al.* (2019) 'Direct Training for Spiking Neural Networks: Faster, Larger, Better', *The Thirty-Third AAAI Conference on Artificial Intelligence*, pp. 1311–1318. doi: 10.1609/aaai.v33i01.33011311.

Yliniemi, L., Agogino, A. K. and Tumer, K. (2014) 'Multirobot Coordination for Space Exploration', *AI Magazine*, 35(4), pp. 61–74. doi: 10.1609/aimag.v35i4.2556.

Yudowsky, E. (2008) 'Artificial Intelligence as a Positive and Negative Factor in Global Risk', *Artificial Intelligence*, pp. 1–45.

Zelinka, I., Senkerik, R. and Pluhacek, M. (2013) 'Do Evolutionary Algorithms Indeed Require Randomness?', *2013 IEEE Congress on Evolutionary Computation, CEC 2013*, pp. 2283–2289. doi: 10.1109/CEC.2013.6557841.

Zheng, Y. C. (2020) 'Mars Exploration in 2020', *Innovation(China)*, 1(2), p. 100036. doi: 10.1016/j.xinn.2020.100036.

Zitzler, E., Laumanns, M. and Thiele, L. (2001) 'SPEA2: Improving the Strength Pareto Evolutionary Algorithm', *Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems*, pp. 95–100. doi: 10.1.1.28.7571.

Zitzler, E. and Thiele, L. (1999) 'Multiobjective Evolutionary Algorithms : A Comparative Case Study and the Strength Pareto Approach', *IEEE Transactions on Evolutionary Computation*, 3(4), pp. 257–271.

Appendix

All Brain Evolver project files:

https://github.com/davegj101/BrainEvolver

A: Brain Evolver Software Structure

A.1 Environment Interface

BE includes an interface class that serves as a communication standard between all agents and all environments. Since the code for BE is open source, this allows users to code and add their own new custom testing environments to BE. The interface ensures that newly coded environments fit in with how the program's system works and that agents' behaviours comply with what the interface expects. The methods and variables included in the interface class outline the paradigm for which agents created by BE must interact with an environment. It is also important to note that the RL rewards received by an agent as it learns also constitute the resultant fitness of the agent for a particular test simulation.

In order to add a new environment, a user must create new *environment_x.h* and *environment_x.cpp* files in *Brain Evolver Project Files -> src -> environments -> simulations*. These files must execute the logic of the newly added environment by implementing all the expected methods and variables defined in *Brain Evolver Project Files -> src -> environments -> management -> environments.h*.

Additionally, the new environment must be imported into *Brain Evolver Project Files -> src -> environments -> management -> environment.h* before adding the option to create the new environment in the constructor method of *Brain Evolver Project Files -> src -> environments -> management -> environment.cpp* by following the same procedure used to add the other environments. Lastly, the name of the new environment must be added to at the bottom of the *Brain Evolver Project Files -> src -> general -> globals.cpp* file so that it can be accessed through the GUI.

All environments must assign valid values to a set of variables. These variables include:

- the environment *name*,
- the number of *steps* per test,
- the number of expected *input* and *output* nodes,
- the *maximum fitness* obtainable for the environment,
- the *highest reward* an agent can receive on a given step,
- and the *update rate* of the environment (which is used to set the simulation speed of visualization animations).

There are seven methods that must also be overridden. These methods include functions to:

- *setup* the environment,
- perform any *reset* actions between steps,
- poll the dimensions of the internal state *parameters* required by an agent,
- poll the environmental *input* for the current step,
- *update* the environment and agent feedback based on an *action* taken by an agent,
- *draw* a visualisation of the *environment*,
- and to *draw* a visualisation of the *individuals* in the environment.

A.2 Graphics Engine

BE includes its own custom graphics engine that makes use of *OpenGL*. The engine was designed in a multithreaded way so that a user can interact with the program's GUI while any visualizations being displayed by test simulations are simultaneously displayed without affecting the responsiveness of the GUI. It enables multiple windows to be stacked and displayed at once while the communication and management of each window is handled in separate but concurrently safe threads. The thread-safe communication is implemented in such a way as to only minimally impact performance and is a very important aspect to the graphics engine since different draw instructions may be given to different windows across different threads. All visualizations are also separate to any AI processing. This means that if BE is running a computationally intense simulation, the program will remain responsive without the frame-rate being likely to drop.

The graphics engine includes some high-level commands that can be executed. These include:

- the initialization of a *new window*,
- *closing* of an open window,
- *updating* the responsiveness of all windows,
- *polling* if a window is ready to be used,
- *posting* a *frame* to a window,
- and a number of *draw* commands. These commands include:
    - drawing a *rectangle*,
    - a *tetragon*,
    - a *line*,
    - a *circle*,
    - and displaying *text*.

The graphics engine also includes a custom 3D engine that allows tetragon shapes to be projected and ordered into a 3D space. This 3D component is used when visualizing two-dimensional test functions *(see chapter 3.8.1)*. These 3D visualisations can also be interacted with by controlling the user's mouse. A user can click and drag the visualisation to rotate it or use the scroll wheel to zoom in or out. Any 3D function is represented using a mesh of partially transparent polygons. The user can also adjust the resolution of the number of polygons that are used when rendering 3D functions. This can be increased for better visual fidelity and decreased for better performance.

Users can also cap all visualizations to only display a maximum number of individuals, even if there are more individuals present in the population. This further assists any performance issues. Additionally, users can toggle whether to see the fitness of each individual by pressing the *spacebar* and can speed up or slow down the simulation rate by using the arrow keys. These interactive functions are enforced by the environment interface class. Users can also toggle whether environment simulation visualisations are even displayed at all. If visualizations are disabled, the training process will run at the maximum speed capable by the hardware being used. Visualizations can be toggled back on at any point to see how a test is progressing. Further details regarding the implementation of the test environments and their visualizations can be seen in *chapter 3.8*.

A.3 File System

BE includes a saving and loading system to ensure that progress is not lost should something go wrong during a long testing session. When REVAC is enabled, BE only saves progress between REVAC generations. When REVAC is not being used, it saves between DNE generations. The saving process can be set so that it occurs automatically. If saving is enabled, a user can either choose to have the program automatically save once everything is complete or every certain number of generations.

All sessions are saved and loaded from a single file named *save.cfg* which is located in the *save* folder in the root directory of the program. If a previous session has been saved, a user can choose to either load the whole session as it was last saved so that it can continue running from where it left off, or they can choose to just load the settings and parameters used for that session. The latter option is useful if tests are run so has to perform hyperparameter configuration (using REVAC) before running longer DNE level tests on the optimally found parameters. A user can then just load the parameters discovered by the REVAC tests without loading the REVAC session itself before changing some parameters to define a new test.

A.4 Performance

BE's performance is based on a few factors. Firstly, the fact that BE was coded in C++ was a performance-based decision. Furthermore, the multithreaded approach to BE's graphics engine and the way it interacts with simulation visualizations and the program's GUI was also designed with performance in

mind. In this way, the operation of the program's GUI does not affect the running of any program tests.

A significant addition to BE that enables scalable performance comes in the ability of the program to parallelise its tests. Tests can run in parallel at two different granular levels. The first level is course-grained and is implemented by REVAC while the second is more fine-grained and is implemented by DNE. These different levels cannot operate at the same time. If REVAC is enabled, then parallelization takes place at the course-grained level. If REVAC is not enabled, then it takes place at the fine-grained level. Users also have some control over the extent of the parallelization by setting the maximum number of threads the program can use for running tests (which is by default set to be linear).

If a system has 128 available cores but a user only wants to dedicate half of the system's resources to a particular set of BE tests, then they can set the maximum number threads to be 64. This does not mean that the program will always use the maximum number of threads, but it will try grab as many threads (up to the maximum) when it can. At the course-grained level, new threads are created by dividing up the evaluations of the population of potentially hyperparameterized DNE solutions. At the fine-grained level, new threads are created during the simulation and evaluation processes where all individuals' evaluations for a generation are divided up between the available threads.

Certain architectural and coding practice decisions also went into ensuring that the program runs as efficiently as possible. Besides more routine requirements such as the choice of data structures, algorithms, and variable datatypes, one of the main architectural choices that had a large impact on performance was BE's choice to use a structure based DNE (as outlined in *chapter 3.1*). This allowed node and connection values to be quickly accessed in arrays while the lack of a need for innovation numbers and the use of structure makes speciation, mutation, and crossover significantly more efficient. It is worth nothing that two

previous versions of BE were coded during the development of this thesis. The first was coded in Python which proved to be too slow while the second was coded in C++ but made use of a more traditional approach to NEAT than the current custom implementation of DNE. This implementation's performance was far worse than the current model and its increased run times due to its lack of efficient processing was one of the motivational factors to re-coding the whole program and redesigning the algorithm.

Due to the dynamic way that an agent's components can grow and evolve in BE, a highly parallel implementation of the code that makes use of graphics cards ended up falling out of the scope of the project and was hence not implemented. This is a drawback of the program's performance and future work on further optimizing BE could see the inclusion of either CUDA or OpenCL into BE.

A.5 Parameters and Settings

BE's software is governed by a wide range of parameters and settings. Some of these settings are tuneable by REVAC and some are not. All adjustable values in BE are divided into four categories including *runtime variables*, *configuration*, *settings*, and *parameters*. Runtime variables pertain to how BE's software runs and does not interfere with how the logic of the AI operates (and are hence not influenced by REVAC). A list of all the runtime variables (as seen in the program) is given below. Some of these settings have already been outlined. It is also important to note that binary variables are set by a user by either entering a one or zero for *true* or *false*.

Runtime Variables:

- **Seed:** Allows randomizations to be predictably repeated. If set to zero, then there is no seed and all random processes are completely unpredictable.
- **3D Graph Polygons:** The number of polygons to render when displaying an environment that has 3D visualizations.
- **Render Population Max:** The maximum number of individuals to render in an environment's visualization. The other individuals are still processed in the background.
- **Simulation Speed:** How fast the visualization of an environment simulation animates.
- **Show On-Screen Fitness:** Display all individual's fitnesses in any environment's visualisation.
- **Max Threads:** Limit the number of CPU threads that BE can use.
- **Auto-Save Interval:** BE will automatically save a run every *n* number of generations. If set to zero, then the run will only be saved once it is complete.
- **Stats Graph Resolution:** The resolution of the graph in the GUI depicting a run's average and best fitnesses found across all generations (*see chapter 3.7.6*).

The *configuration* category pertains to how the overarching structure of BE's cognitive architecture works. These settings allow certain components to be enabled or disabled and also allow for some adjustability in how any active components interact with each other. All configuration values are also not adjustable by REVAC and must be set by the user. A list of all the configuration settings (as seen in the program) is given below. The names of the settings are self-explanatory in their function.

Configuration:

- **General:**
  - Use REVAC
  - Use SNNs
  - Use SNNs On Theta Network Only
  - Use MLSH
  - Use Attention
  - Add Raw Input To Attention Output
  - Enable Guided Epsilon Exploration
- **Activation Functions:**
  - Enable Activation Offset

    - Enable Identity Function
    - Enable Binary Function
    - Enable ReLU Function
    - Enable Leaky ReLU Function
    - Enable Sigmoid Function
    - Enable TanH Function
- **Selection Methods** (only one can be selected at a time):
    - Use Linear Wheel
    - Use Exponential Wheel
    - Use Proportional Selection
    - Use Tournament Selection

The *settings* category pertains to how DNE and REVAC operate. Again, these settings are not tuneable by REVAC and must be set by the user. A list of all the adjustable values in the settings category (as seen in the program) is given below. Most of the names of the settings are self-explanatory in their function but additional clarification has been given for a few of them.

Settings:

- **DNE:**
    - Population
    - Generation Max
    - Generations To End If No New Best - *(If an improvement is not made within this number of generations, then the DNE process terminates.)*
    - Simulation Repetitions – *(This is the number of repetitions that all agents must go through when being tested for each generation.)*
    - Evaluation Repetitions – *(This is the number of repetitions that only the fittest agents with the potentiality of being the "best seen" solutions must go through)*
    - Max Hidden Nodes per Network
    - Max Networks
- **REVAC:**
    - Population
    - Generation Max
    - Generations To End If No New Best - *(If an improvement is not made within this number of generations, then the REVAC process terminates.)*

- Evaluation Repetitions
- Parent Pool Size
- Mutation Range

Lastly, the *parameters* category pertains to all hyperparameters that can be adjusted by REVAC. These values can also be manually set by the user and be constrained to only be REVAC adjusted within a certain tuneable range. Furthermore, these parameters can also be subject to online self-adaption as each parameter has four values that can be set as demonstrated by *Equation (7)* in *chapter 3.2*. A list of all the parameters (as seen in the program) is given below. The names of the parameters should also be self-explanatory in their function. However, clarification is also given to parameters with more obscure names.

Parameters:

- **General:**
  - Crossover Chance
  - Mutation Chance
  - Attention Memory Length Factor – *(This is the look-back depth that is included into the Attention unit's temporal window as a fraction of the maximum number of simulation steps across all testing environments.)*
- **Selection:**
  - Max Number of Species Factor – *(The factor is a portion of the total population size.)*
  - Species Discard Factor – *(This is the weakest portion of the total number of species that is discarded.)*
  - Min Champion Species Size Factor – *(This is the minimum number of individuals that a species must have as a portion of the total population before the species can have champions. Champion individuals employ elitism and pass on to the next generation without change.)*
  - Exponential Selection Factor
  - Tournament Pool Size Factor
- **Connection Mutation:**
  - Mutate Connection Enabled Chance
  - Mutate Connection Weight Chance
  - Shift Connection Weight Chance

    - Shift Connection Weight Scale
    - Mutate LTP Rate Chance
    - Shift LTP Rate Chance
    - Shift LTP Rate Scale
    - Mutate LTD Rate Chance
    - Shift LTD Rate Chance
    - Shift LTD Rate Scale
    - Mutate Learning Window Chance
    - Shift Learning Window Chance
    - Shift Learning Window Scale
- **Node Mutation:**
    - Mutate Activation Function Chance
    - Use Identity Function Chance
    - Use Binary Function Chance
    - Use ReLU Function Chance
    - Use Leaky ReLU Function Chance
    - Use Sigmoid Function Chance
    - Use TanH Function Chance
    - Mutate Activation Offset Chance
    - Shift Activation Offset Chance
    - Shift Activation Offset Scale
    - Mutate Activation Threshold Chance
    - Shift Activation Threshold Chance
    - Shift Activation Threshold Scale
    - Mutate Leaking Factor Chance
    - Shift Leaking Factor Chance
    - Shift Leaking Factor Scale
    - Mutate Refractory Period Chance
    - Shift Refractory Period Chance
    - Shift Refractory Period Scale
- **Topology Mutation:**
    - Min Topology Mutation Species Size Factor - *(This is the minimum number of individuals that a species must have as a portion of the total population before any individuals in that species can make a topological mutation.)*
    - Mutate New Node Chance
    - Mutate Remove Node Chance
    - Mutate New Layer Chance
    - Mutate New Network Chance

- **Attention Mutation:**
    - Mutate Attention Weight Chance
    - Shift Attention Weight Chance
    - Shift Attention Weight Scale
- **Exploration Mutation:**
    - Mutate Epsilon Exploration Chance
    - Shift Epsilon Exploration Chance
    - Shift Epsilon Exploration Scale

A.6 Graphical User Interface

The *Graphical User Interface* (GUI) of BE enables the interaction of the user with the program. When starting the program, the user is presented with the main window as shown in *Figure 26*. The buttons down the top left-hand side of the window each open up new windows corresponding to the categories of parameters and settings that the user can adjust as outlined in *chapter* 3.7.5 alongside another button that selects what testing environments will be used. The windows dedicated to changing parameters and settings are all structured in a similar manner and are all scrollable. An example of the window dedicated to changing parameters (with online self-adaption disabled) can be seen in *Figure 27*. To enter a new parameter or setting value, the user must click on an empty field to highlight it, type in the new desired value, and then press the *enter* key. Invalid entries are ignored and values that are out of bounds default to the closest valid boundary value.

In the top-middle section of the main window is another list of buttons pertaining to actions that can be performed by the program during runtime. The function of the buttons is listed below.

- The **START/PAUSE** button does as its named and starts and pauses a run of a set of tests.

- The **SHOW/HIDE SIMULATION** button enables or disables any test environment visualisations. When set to disabled, the program will run at maximum speed.

- The **VIEW BEST NETWORK** button brings up an interactive window where users can inspect details about the currently best found individual. More details are given on this functionality shortly.

- The **LOAD PROGRESS** button loads an entire previously saved session as is so that it can continue from where it left off.

- The **LOAD SETTINGS** button only loads the values for the runtime variables, configuration, settings, parameters, and environments selected for a previously saved session. These values can then be further adjusted before starting a new run.

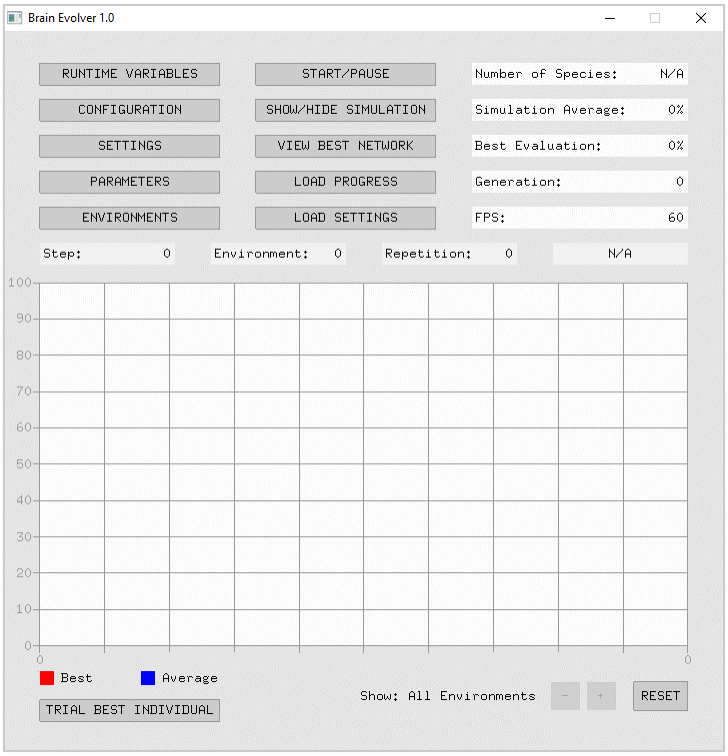


Figure 26

Main GUI window displayed when program starts.

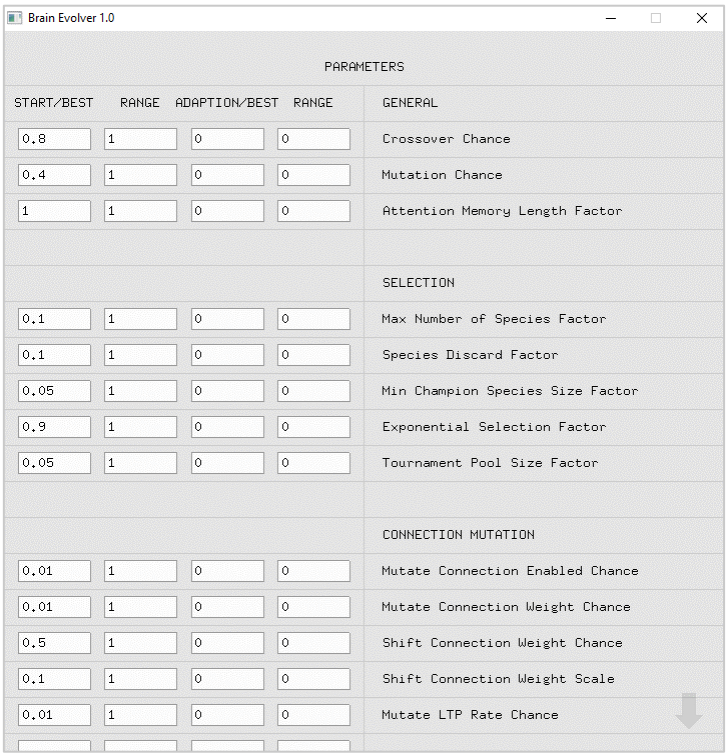


Figure 27

GUI Window dedicated to changing and viewing parameters.

The window produced by clicking on the *view best network* button can only be accessed once a run has been started. It shows the structure of the current best network. A user can click on one of the nodes to see the parameters for that particular node. Furthermore, if a user clicks on a second node that is connected to the first one, the program will highlight the connection between the two nodes and show the parameters pertaining to that connection. Examples of these interactions can be seen in *Figure 28* and *Figure 29* respectively. Initially, the network shown represents the agent's θ network (referenced as network zero). However, users can switch to view different ϕ networks (if MLSH is enabled) by clicking the plus or minus buttons at the top right of the window. If Attention is enabled, users can also view the Q, K, and V vectors of the agent's Attention matrix by clicking on the buttons with the corresponding letters at the top right of the window. A depiction of what this may look like can be seen in *Figure 30*.

On the top right-hand section of the main window is a set of fields on top of each other depicting the current information regarding a particular run. This information includes the current *number of species* (if REVAC is not enabled), the most recent *average population fitness*, the *best fitness found* so far, the current *generation*, and the program's *frames-per-second* (FPS) which can be used to monitor if the program is under excessive processing strain.

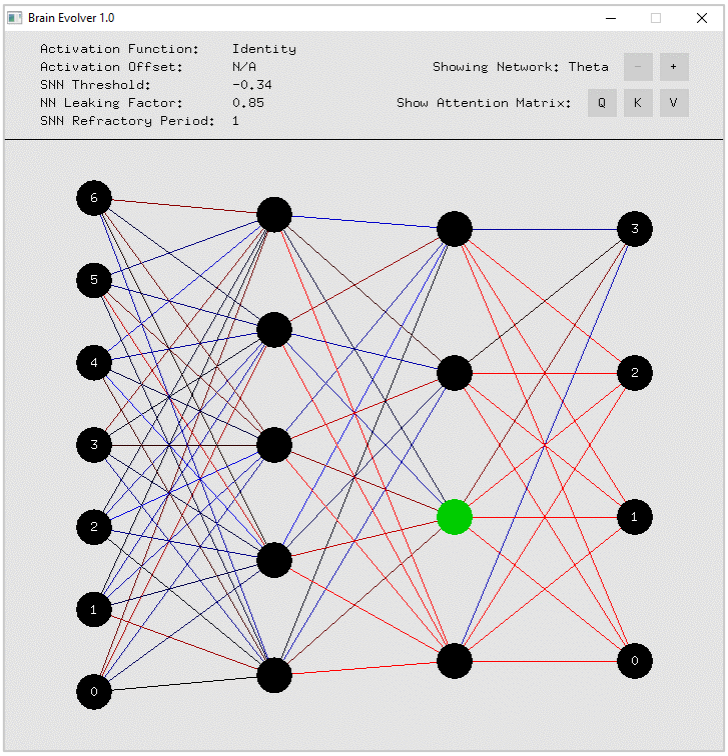


Figure 28

View best network GUI window with a single node highlighted.

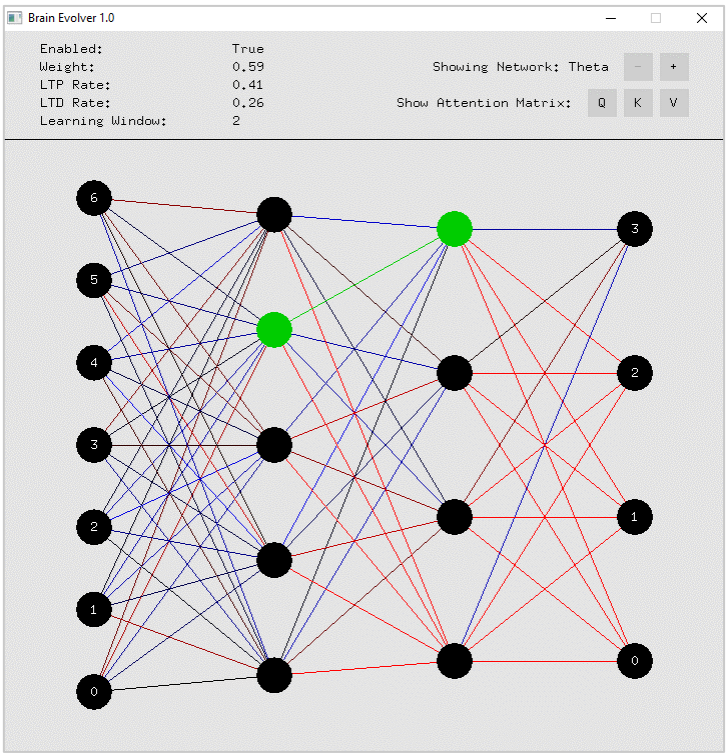


Figure 29

View best network GUI window with a connection highlighted between two selected nodes.

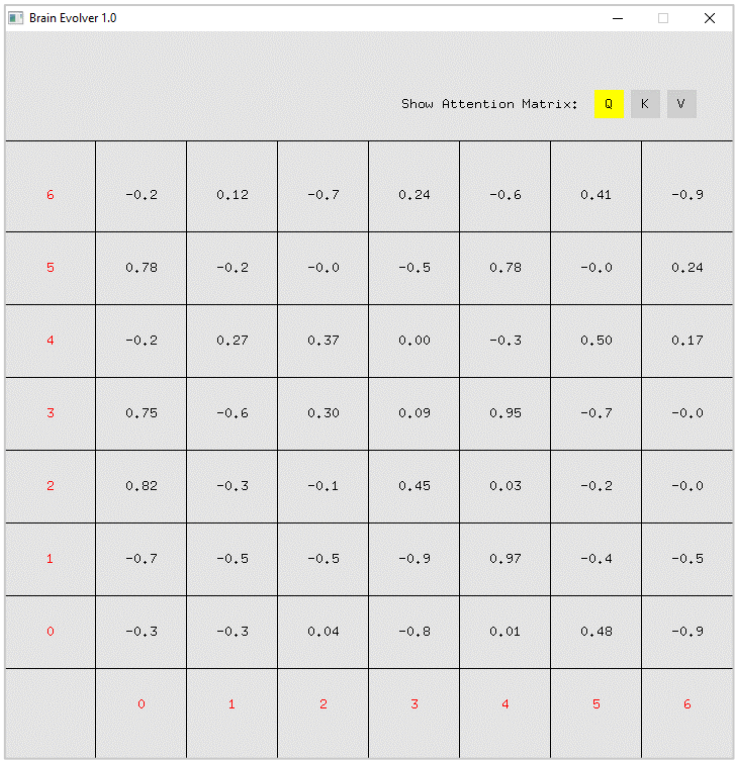


Figure 30

View best network GUI window showing the values of the Attention matrix's Q-vector.

In the lower-middle half of the main window exists the *progress graph*. The progress graph draws two lines as the program runs, one in red and one in blue. These lines plot the average population fitness (in blue) and the best fitness found so far (in red) against the progression of generations. This graph is useful as it keeps track of the historical information about the evolutionary DNE process (if REVAC is disabled) or the evolutionary REVAC process (if REVAC is enabled). This can be used to gain insights into how the population converges (or doesn't) to potential optimal solutions. By default, the progress graph shows the cumulative fitness of individuals across all test environments selected. However, if more than one test environment is selected, the user can see the progress of the population for just one of the test environments by clicking on the plus or minus buttons below the graph. An example of how a run might be represented by the progress graph and the main window can be seen in *Figure 31*.

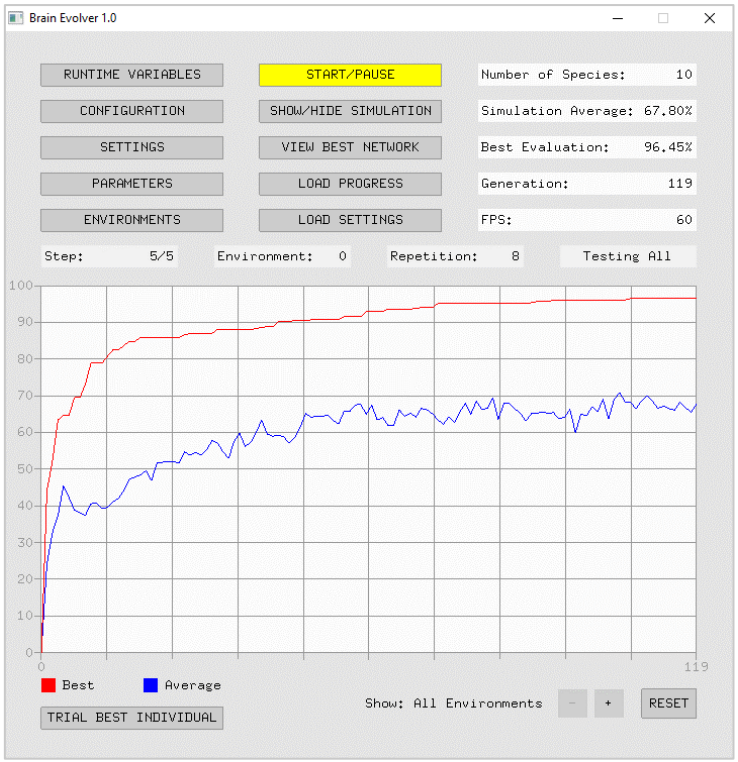


Figure 31

The main window GUI showing statistics pertaining to a test during a run.

Just above the top of the progress graph is a row of more information fields running laterally across the screen. This information pertains to the progression of the current evaluation process and is also different depending on whether REVAC is enabled or not. If REVAC is disabled, the information displays the current simulation *step*, *environment* being evaluated (since users can set multiple test environments to be evaluated one after the other), evaluation *repetition*, and current algorithm *status*. The status can indicate a few algorithm states. These include *testing all* individuals, *testing a potential best* individual, *evolving* the population between generational evaluation tests, *saving* the current progression of a session, and indicating that the algorithm is *complete*. If REVAC is enabled, the information displayed on this row constitutes the current evaluation *repetition*, the number of evaluated DNE solutions within a repetition that are *complete*, and the current *generation* of the *slowest* DNE solution.

At the bottom left of the window is a *trial best individual* button that pauses the progress of the current session (if it has not finished yet) and runs a test on the selected environments of just the best solution found so far by itself. This is useful to see the behaviour of what the program considers to be the optimal found solution. Lastly, there is a button at the bottom right of the main window which allows the program to be reset. Any current session is discarded, and all settings and parameters are set to their default values.