Damage recovery for robot controllers and simulators evolved using Bootstrapped Neuro-Simulation

Brydon Andrew Leonard

214 036 685

April 2019

Submitted in fulfilment of the requirements for the Master of Science degree in the Faculty of Science at the Nelson Mandela University

Supervisor: Dr. Mathys C. du Plessis

Declaration

In accordance with Rule G5.6.3, I, Brydon Andrew Leonard (214 036 685), hereby declare that this dissertation for the degree of Master of Science is my own work and that it has not previously been submitted for assessment or completion of any postgraduate qualification to another university or for another qualification.

Brydon Andrew Leonard

April 2019

Acknowledgements

I would like to take this opportunity to thank those who have played a role in supporting me through my studies. Firstly, I'd like to thank my supervisor, MC du Plessis. Without his constant feedback and constructive criticism, what I've achieved this year would not have been possible.

I would also like to thank Grant Woodford. Not only did he lay the groundwork upon which my research builds, but his willingness to take the time to talk through problems and help me understand more complex concepts has played a very significant role in my success this year.

I would like to make special mention of my parents. They have never held back when investing in my education. Their support, both financially and emotionally, is what has allowed me to reach where I am today. I am incredibly grateful for everything they have done.

Finally, I'd like to thank my family and friends. Some showed their support from the sidelines, and others struggled through their studies with me. People are what make life enjoyable, so I'm grateful for all those that play a role in mine.

Abstract

Robots are becoming increasingly complex. This has made manually designing the software responsible for controlling these robots (controllers) challenging, leading to the creation of the field of evolutionary robotics (ER). The ER approach aims to automatically evolve robot controllers and morphologies by utilising concepts from biological evolution.

ER techniques use evolutionary algorithms (EA) to evolve populations of controllers - a process that requires the evaluation of a large number of controllers. Performing these evaluations on a real-world robot is both infeasibly time-consuming and poses the risk of damage to the robot. Simulators present a solution to the issue by allowing the evaluation of controllers to take place on a virtual robot.

Traditional methods of controller evolution in simulation encounter two major issues. Firstly, *physics simulators* are complex to create and are often very computationally expensive. Secondly, the *reality gap* is encountered when controllers are evolved in simulators that are unable to simulate the real world well enough due to simplifications or small inaccuracies in the simulation, which together cause controllers in the simulation to be unable to transfer effectively to reality.

Bootstrapped Neuro-Simulation (BNS) is an ER algorithm that aims to address the issues inherent with the use of simulators. The algorithm concurrently creates a simulator and evolves a population of controllers. The process starts with an initially random population of controllers and an untrained simulator neural network (SNN), a type of robot simulator which utilises artificial neural networks (ANNs) to simulate a robot's behaviour. Controllers are then continually selected for evaluation in the real world, and the data from these real-world evaluations is used to train the controller-evaluation SNN.

BNS is a relatively new algorithm that has not yet been explored in depth. An in-

vestigation was, therefore, conducted into BNS's ability to evolve closed-loop controllers. BNS was successful in evolving such controllers, and various adaptations to the algorithm were investigated for their ability to improve the evolution of closed-loop controllers. In addition, the factors which had the greatest impact on BNS's effectiveness were reported upon.

Damage recovery is an area that has been the focus of a great deal of research. This is because the progression of the field of robotics means that robots no longer operate only in the safe environments that they once did. Robots are now put to use in areas as inaccessible as the surface of Mars, where repairs by a human are impossible.

Various methods of damage recovery have previously been proposed and evaluated, but none focused on BNS as a method of damage recovery. In this research, it was hypothesised that BNS's *constantly learning* nature would allow it to recover from damage, as it would continue to use new information about the state of the real robot to evolve new controllers capable of functioning in the damaged robot.

BNS was found to possess the hypothesised damage recovery ability. The algorithm's evaluation was carried out through the evolution of controllers for simple navigation and light-following tasks for a wheeled robot, as well as a locomotion task for a complex legged robot.

Various adaptations to the algorithm were then evaluated through extensive parameter investigations in simulation, showing varying levels of effectiveness. These results were further confirmed through evaluation of the adaptations and effective parameter values in real-world evaluations on a real robot. Both a simple and more complex robot morphology were investigated.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Term
ANN	Artificial Neural Network
BNS	Bootstrapped Neuro-Simulation
BO	Bayesian Optimisation
EA	Evolutionary Algorithm
EC	Evolutionary Computation
ER	Evolutionary Robotics
FFNN	Feedforward Neural Network
FLNN	Functional Link Neural Network
FRW	Fake Real World
GA	Genetic Algorithm
GP	Gaussian Process
IT&E	Intelligent Trial-and-Error
LSTM	Long Short-Term Memory
MCTS	Monte Carlo Tree Search
NEAT	Neuroevolution of augmenting topologies
RE	Reset-Elitist
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RO	Research Objective
RTE	Reset-Free Trial-and-Error
RW	Real World
SBX	Simulated Binary Crossover
sIT&E	Safety-Aware Intelligent Trial-and-Error
SNN	Simulator Neural Network
SRNN	Simple Recurrent Neural Network
SSE	Sum of Square Errors

Chapter 1

Introduction

1.1 Introduction

Evolutionary robotics (ER) is a field that deals with the automatic evolution of the software that controls robots (controllers) and their physical forms (morphologies) (Zagal and Ruiz-Del-Solar, 2007; Bongard, 2013; Hiller and Lipson, 2012). ER applies concepts from evolutionary computation (EC) to robotics. To evolve these controllers, a population of candidate individuals is created, each representing a possible controller. The controllers are then evaluated and those that perform best are given the opportunity to reproduce with each other to produce a new population of individuals. Over time, the population improves and converges to fewer, more promising, controllers until a satisfactory controller is created. These evolutionary methods require the evaluation of many controllers, and it is infeasible from a time perspective to perform these evaluations on real-world robots (Zagal and Ruiz-Del-Solar, 2007). Robot controllers are therefore evolved in simulation before the best evolved controller is transferred to the real robot. This approach has a number of issues. The *reality gap* causes controllers to be unable to transfer from simulation to reality, and the simulators often require a large amount of data gathering or domain knowledge to construct.

Bootstrapped neuro-simulation (BNS) (Woodford, Pretorius, and du Plessis, 2016) is a new method of controller and simulator creation, which aims to solve these issues. The algorithm starts by creating an initially random population and simulator neural network

(SNN). Controllers are then chosen from the population and evaluated in the real world. The data from these real-world evaluations is then used to improve the simulator so that the controller population can improve in simulation. This process repeats until a controller has been created that is able to successfully solve the task at hand. This method has shown great promise and has been successfully used for the evolution of controllers for several robot morphologies of varying complexity (Woodford et al., 2016; Woodford, du Plessis, and Pretorius, 2017).

An ER method such as BNS evolves controllers for a specific real-world robot, but changes in morphology, motor performance and sensor performance caused by damage can hinder the performance of the evolved controllers. As the field of robotics progresses, robots are used for more complex and interesting applications, which often carry a greater risk of damage. ER-based methods of robot damage recovery are therefore of great interest to the ER community (Papaspyros, Chatzilygeroudis, Vassiliades, and Mouret, 2016; Cully, Clune, Tarapore, and Mouret, 2015; Chatzilygeroudis, Vassiliades, and Mouret, 2016; Bongard, Zykov, and Lipson, 2006). While many methods of damage recovery have been developed, the topic of damage recovery for BNS-evolved controllers is completely unexplored and may offer benefits over existing methods of damage recovery.

Section 1.2 presents the hypothesis guiding this research, followed by a discussion of the research objectives in Section 1.3. The research methodology is then discussed in Section 1.4, before the topics of each chapter are briefly explained in Section 1.6.

1.2 Hypothesis

This research hypothesises that:

BNS can be shown to recover from physical damage to a robot and be augmented with new adaptations to improve this damage recovery.

1.3 Research Objectives

Before any research can be conducted investigating BNS's damage recovery abilities, it is important to investigate not only the current state of research on the BNS algorithm, but

also that of ER and ER damage recovery. The first research objective is therefore:

RO1: Investigate existing ER damage recovery methods.

The simulator built into the BNS algorithm is always learning about its environment. It is, therefore, very likely that the algorithm has the inherent ability to recover from damage by simply observing the way the robot functions after damage has occurred. This possibility should be investigated. This leads to the second research objective:

RO2: Determine BNS's damage recovery capabilities.

While BNS is likely to possess the ability to recover from damage even before any changes are made to the algorithm, it is unlikely that its performance represents the best possible damage recovery. A number of changes to the algorithm could be proposed to further improve its performance. These changes need to be implemented and their effects evaluated, leading to the third research objective:

RO3: Propose, implement, and evaluate adaptations to improve BNS's damage recovery capabilities.

BNS has never been used to evolve controllers more complex than simple sequences of commands. While these controllers served as an excellent place to start evaluating the algorithm, they are not easy to apply in the real world: they are unable to adapt to changes and simply execute their commands one after another. Such controllers are called open-loop controllers. Closed-loop controllers, on the other hand, are able to make observations about the world around them and use those observations to drive their actions. Since closed-loop controllers are so much more applicable to the real world, they are more likely to be used and damaged in the real world, which makes their ability to recover from damage that much more vital. The importance of both the evolution of closed-loop controllers and their damage recovery leads to research objectives four and five:

RO4: Propose, implement, and evaluate a method of evolving complex controllers using BNS.

RO5: Propose, implement, and evaluate adaptations for BNS to allow for damage recovery for complex controllers.

Finally, once BNS has been shown to recover from damage to both simple and more complex controllers for a simple robot, the algorithm must be shown to recover from damage to robots with more complex morphologies. It would not be possible to evaluate the algorithm on every possible morphology, so only one will be chosen, but the ability of the algorithm to transfer from a simple to a complex one would indicate that the algorithm is not limited to only simple robots. The sixth and final research objective is therefore:

RO6: Demonstrate and investigate the transferral of BNS's damage recovery to a more complex robot.

1.4 Methodology

Instead of intuition or introspection, the positivistic approach to research aims to gain knowledge through empirical evidence (Easterbrook, Singer, Storey, and Damian, 2008). These objective observations ensure that bias on the part of the researcher is kept to a minimum. This research used a positivistic approach since the research questions were such that they could be answered through empirical study.

The deductive approach focuses on testing a particular hypothesis or theory. This approach was thus appropriate for this research as it is focused on testing hypotheses in order to challenge the theory that BNS can be augmented to enable it to recover from damage to a robot. An experimental and grounded theory method will be used in the answering of the research questions. The experimental process followed will be discussed in Chapter 3, after the theoretical background of the topic is covered in Chapter 2.

1.5 Envisioned Contribution

This research will make both theoretical and practical contributions to the field of ER. Various adaptations to the BNS algorithm will be proposed. These adaptations will be based on enhancing aspects of the algorithm in ways that are likely to improve its ability to recover from damage.

Once proposed, it is important that the adaptations are evaluated so that recommendations can be made for their use. This would not be possible without the implementation

of a damage system to function in conjunction with existing BNS implementations. The system will allow for the evaluation of damage recovery without the risks involved with inflicting real damage on the robot.

In order to obtain statistically significant results, a large number of parameter combinations will need to be assessed. Therefore, the parameter combinations will be evaluated in simulation before promising configurations are transferred to the real world. Where simulators exist already, they will be used, but where they do not, new simulators will be implemented.

1.6 Dissertation Layout

Here, a brief description of each chapter's contents is given.

Chapter 2 - Literature Review: Topics relevant to the research are investigated, including machine learning topics such as evolutionary computation and neural networks. Evolutionary robotics is then discussed, as is the use of simulation in the field. Finally, existing work investigating damage recovery in evolutionary robotics is discussed.

Chapter 3 - Experimental Methodology: The chapter discusses the logic behind each step taken in this research, as well as the methodology followed when implementing and performing the evaluation for each investigation. The chapter also presents the novel adaptations to be evaluated and describes the content of each of the investigations that comprise this research.

Chapter 4 - Investigation A: The chapter presents the results of the evaluation of BNS's damage recovery. It also presents the first results of the performance of BNS with damage adaptations.

Chapter 5 - Investigation B: The results of investigation B are presented and discussed. The investigation focuses on the evolution of closed-loop controllers using BNS, but not damage, thereby laying the groundwork for investigation C.

Chapter 6 - Investigation C: The results of closed-loop controller damage recovery are presented in this chapter.

Chapter 7 - Investigation D: The chapter presents the results of damage recovery evaluation using simple controllers, but for a complex hexapod robot.

Chapter 8 - Conclusions and Future Work: The final chapter draws conclusions from the investigations and discusses possible future avenues of research.

Appendix A The Mann-Whitney U Test.

Appendix B The conference paper presented at the 2018 11th International Conference on Computer Science and Information Technology (ICCSIT 2018) and published in the Journal of Computers (JCP).

Appendix C The paper submitted to the Journal of Robotics and Autonomous Systems.

Chapter 2

Literature Review

2.1 Introduction

Before any investigations can be discussed in detail, it is crucial that an in-depth literature review be conducted in order to establish the groundwork for this research. This chapter investigates existing work related to the fields of evolutionary robotics, specifically regarding simulator development and damage recovery.

Section 2.2 discusses artificial neural networks, including their biological origins, the construction of the neurons and layers from which the networks are constructed, and techniques for their training. Section 2.3 presents an in-depth discussion of evolutionary algorithms (EAs), a sub-field of evolutionary computation (EC). This discussion covers the various components of an EA, as well as optional additions that can be made to the algorithm, based on the specific use-case. Section 2.4 demonstrates how these EAs can be used in the field of robotics to evolve robot controllers.

Sections 2.5 and 2.6 discuss types of simulators used in evolutionary robotics (ER), their associated drawbacks, and prior research, which has attempted to address these issues. Simulator neural networks (SNNs), a type of ANN-based robot simulator able to effectively evolve controllers which bridge the reality gap, are discussed in Section 2.7.

Bootstrapped neuro-simulation (BNS) is a method of concurrent controller and simulator development utilising SNNs, discussed in Section 2.8. BNS is the algorithm which is to be augmented with damage recovery functionality. Finally, Section 2.9 presents an

in-depth investigation of existing methods of damage recovery, and Section 2.10 concludes the chapter.

2.2 Artificial Neural Networks

2.2.1 Biological Origins

The brain is a powerful parallel computer able to perform tasks such as pattern recognition, perception, and motor control much faster than any electronic computer (Engelbrecht, 2007). These abilities are appealing for computer science applications, as are the brain's ability to learn, remember and generalise. This has led to research into the algorithmic modelling of the brain's systems. These models are known as artificial neural networks (ANNs).

2.2.2 Artificial Neurons

Biological neural networks are composed of interconnected *neurons*. Similarly, ANNs are composed of multiple interconnected *artificial neurons* (Engelbrecht, 2007). Each of these artificial neurons has sets of input and output connections, a set of weights associated with these connections, a constant bias value, and an activation function (Figure 2.1).

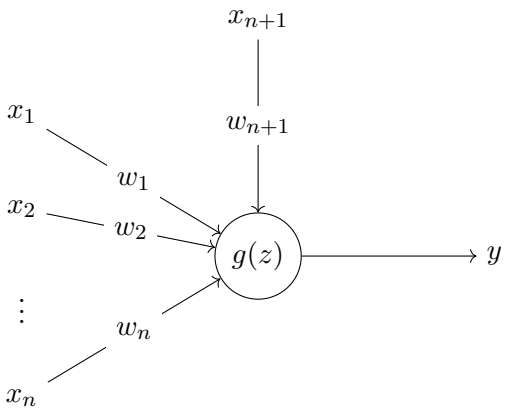


Figure 2.1: Artificial Neuron

To obtain the output of a neuron, the weighted inputs to the neuron must first be calculated. These weighted inputs are the product of each of the neuron's input connections $x_1..x_n$, and each connection's associated weight $w_1..w_n$. The product of an additional

input called the bias x_{n+1}, and its weight w_{n+1}, is also computed. The bias is used as a threshold value. These input values are then aggregated through either summation (equation (2.1)) or, in some cases, multiplication (equation (2.2)) to obtain the net input (z) to the neuron (Engelbrecht, 2007).

$$z = \sum_{i=1}^{n+1} x_i w_i \tag{2.1}$$

$$z = \prod_{i=1}^{n+1} x_i^{w_i} \tag{2.2}$$

The net input is passed into the neuron's activation function, which determines whether the neuron should *fire* by converting the input to the neuron into an output signal (Engelbrecht, 2007). There are many potential activation functions from which to choose. Four are presented here:

1. **Linear function** (Figure 2.2a):

$$g(z) = \lambda z \tag{2.3}$$

   where λ is a parameter determining the slope of the function.

2. **Step function** (Figure 2.2b):

$$g(z) = \begin{cases} \gamma_1 & \text{if} \quad z < \theta \\ \gamma_2 & \text{if} \quad z \geq \theta \end{cases} \tag{2.4}$$

   The step function produces one of two values, depending on the value of z relative to the threshold value, θ. A common use of the step function is to create binary output, where $\gamma_1 = 0$ and $\gamma_2 = 1$.

3. **Sigmoid function** (Figure 2.2c):

$$g(z) = \frac{1}{1 + e^{-\delta z}} \tag{2.5}$$

   δ is a parameter controlling the steepness of the function. Usually $\delta = 1$ (Engelbrecht, 2007).

4. **ReLU function** (Figure 2.2d):

$$g(z) = \begin{cases} \lambda z & \text{if} \quad z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

As with the linear function, λ is a constant value determining the slope of the function. Unlike the linear function, the ReLU function produces a value of 0 for any input values $z \leq 0$.

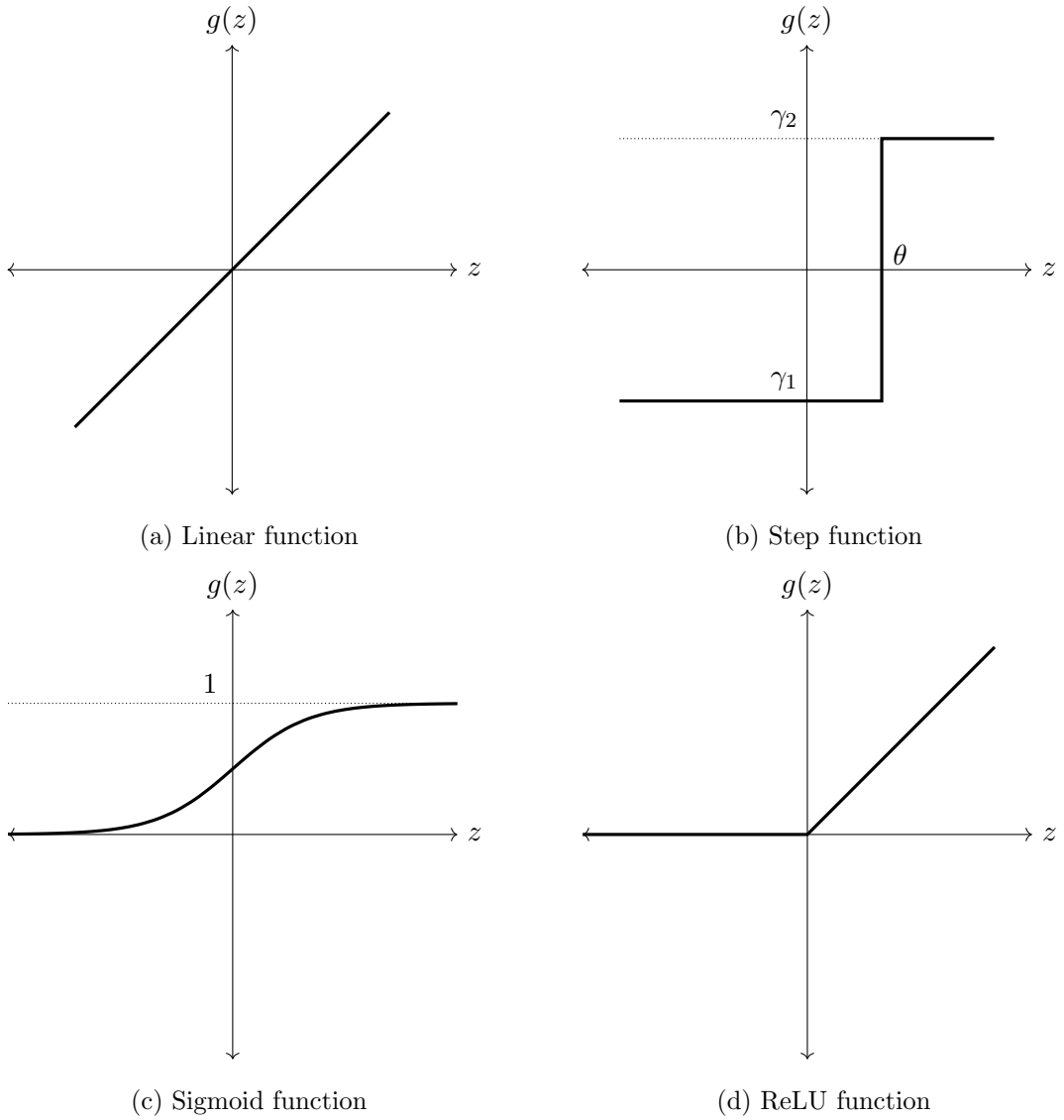
The sigmoid function is a commonly utilised activation function for ANNs. Recently, the use of rectified linear units has grown in popularity (Maas, Hannun, and Ng, 2013), due to issues inherent in the use of sigmoidal functions, discussed further in Section 2.2.5.

A single neuron can be used to realise a linearly separable function with no error (Engelbrecht, 2007). This means that the neuron is able to separate input vectors with above-and below-threshold responses in an I-dimensional space with an I-dimensional hyperplane, which forms the boundary between the two vector classes. This is easy to visualise in two dimensions where this means that the vectors must be able to be separated by a line, as in Figure 2.3.

2.2.3 Networks of Neurons

The ability to learn complex functions is desirable, but since single neurons are limited to learning only linearly separable functions, a layered network of neurons must be used (Engelbrecht, 2007). These layered networks are known as artificial neural networks (ANNs). There are various neural network structures (topologies) that have been shown to be useful in solving different problem types. Arguably, the simplest is the feedforward neural network (FFNN) (Figure 2.4).

Each neuron in layer n_l of an FFNN connects to each neuron in the layer $n_l + 1$, with the outputs of the neurons in layer n_l acting as inputs for those in layer $n_l + 1$ (Engelbrecht, 2007). Each neuron has a bias value, as when operating as a single neuron; these are not shown in Figure 2.4, for the sake of readability. While the figure shows a network with three input neurons, three hidden neurons in one hidden layer, and two output neurons, an FFNN can have any number of hidden layers, and each layer in the network can have



(a) Linear function

(b) Step function

(c) Sigmoid function

(d) ReLU function

Figure 2.2: Activation functions

any number of neurons. The individual neurons in a network are not restricted to a single activation function: a different function can be used for each.

Generally, the input layer of an FFNN uses only linear activation functions (Engelbrecht, 2007). This is not true for a functional link neural network (FLNN), which expands the input layer to include a set of functions used to transform the input. This allows for higher-order combinations of input values.

Simple recurrent neural networks (SRNNs) use the outputs of later layers as inputs for

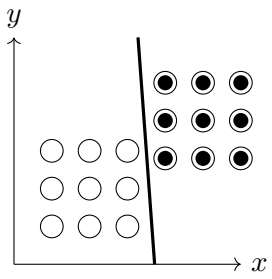


Figure 2.3: Linearly separable points

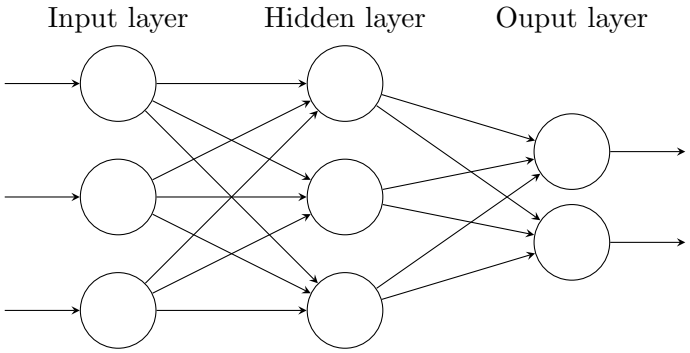


Figure 2.4: An example Feedforward Neural Network

previous layers in the network (Figure 2.5) (Engelbrecht, 2007). There are many ways for an SRNN to be implemented, with a recurrent connection able to be created between any neuron and any previous neuron in the network. These networks have been shown to be *Turing complete* (Siegelmann and Sontag, 1995).

SRNNs are able to use their recurrent connections to learn temporal properties of data. A recurrent network architecture, used specifically for this property, is the time-delay neural network, which accepts input from multiple time periods. A long short-term memory (LSTM) network (Hochreiter, 1997) is a type of recurrent network that has been shown to be effective in solving problems where a longer-term memory is required than that offered by an SRNN.

2.2.4 Data Normalisation

It is important that data be scaled to the active range and domain of the activation functions in use in an ANN (Engelbrecht, 2007). This improves the performance of the

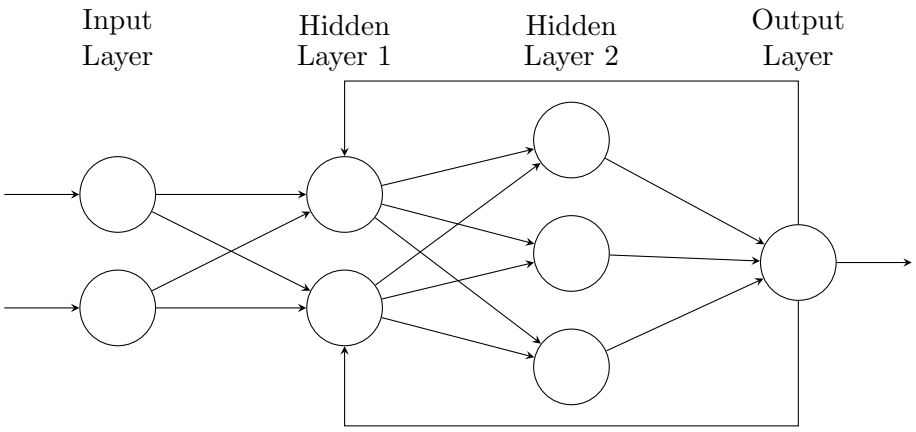


Figure 2.5: An example Simple Recurrent Neural Network

ANN by ensuring that changes in input values lead to large changes in output values. The active range for the sigmoid activation function is $[-\sqrt{3}, \sqrt{3}]$.

One method of data normalisation is *amplitude scaling*. It is important to note that the *input* and *output* values for the normalisation function do not correspond with the input and output values of the ANNs. When normalising data to be used as input for an ANN, the output of the normalisation function is the input of the ANN. When *de-normalising* output data from the neural network, the output data acts as input to the normalisation function, which is able to perform the de-normalisation. The input high ($b_{1,2}$) and low ($b_{1,1}$) bounds specify the range of values of the input to the normalisation function, while the output high ($b_{2,2}$) and low ($b_{2,1}$) values specify the range of possible output values from the normalisation function. The normalisation function is:

$$b_2 = \frac{b_1 - b_{1,1}}{b_{1,2} - b_{1,1}}(b_{2,2} - b_{2,1}) + b_{2,1} \tag{2.7}$$

where b_1 is the unscaled input, and b_2 is the scaled output.

2.2.5 Training

A network's optimal weights for complex problems and network topologies are not simple to determine. Therefore, methods of determining these values have been developed, allowing the network to *learn* them automatically. A selection of these methods is discussed in this section.

Gradient descent

Gradient descent is a method of training a single artificial neuron (Engelbrecht, 2007). In order to use the method, an error function must be defined for measuring the accuracy of the neuron's output. A commonly used function is the sum of squared errors (SSE), shown in equation 2.8. n_s is the number of *input-target vector pairs* (patterns) in the training set and, for a pattern $q \in (1, n_s)$, t_q and o_q are the target value and actual output value of the neuron, respectively.

$$\varepsilon = \sum_{q=1}^{n_s} (t_q - o_q)^2 \tag{2.8}$$

Gradient descent aims to find the set of weight values for the neuron that minimises the error, ε. The algorithm achieves this by calculating the gradient of ε in weight space. The neuron's weights are then slowly adjusted, moving them in the direction of ε's negative gradient. These adjustments will move the weights towards local optima, where the error is smaller than the error for nearby values for the weight (Figure 2.6). Each of the neuron's weights is adjusted in this way, and the error of the neuron is reduced.

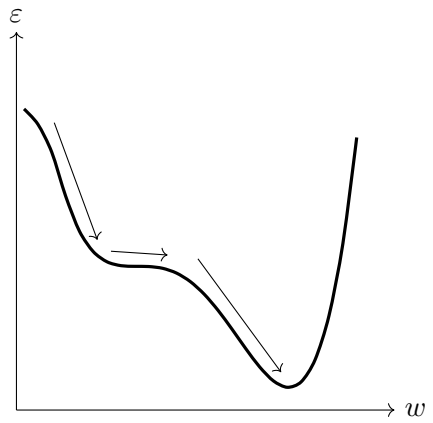


Figure 2.6: Gradient descent weight updates

The weights are updated using:

$$w_i(t) = w_i(t-1) + \Delta w_i(t) \tag{2.9}$$

$$\Delta w_i(t) = \eta(-\frac{\partial \varepsilon}{\partial w_i}) \tag{2.10}$$

$$\frac{\partial \varepsilon}{\partial w_i} = -2(t_q - o_q)\frac{\partial f}{\partial z_q}x_{q,i}w_i \tag{2.11}$$

where η controls the learning rate, f is the activation function, z_q is the net input for pattern q, and $x_{q,i}$ is the input i corresponding to pattern q ($z_q = \sum_{i=1}^{n_v+1} x_{q,i}w_i$, n_v is the number of inputs to a neuron). The calculation of $\frac{\partial f}{\partial z_q}$ requires f to be differentiable and, therefore, presents an issue for non-differentiable activation functions.

As an example, given the differentiable sigmoid activation function (equation 2.5),

$$\frac{\partial f}{\partial z_q} = o_q(1 - o_q) \tag{2.12}$$

and, therefore

$$\frac{\partial \varepsilon}{\partial w_i} = -2(t_q - o_q)o_q(1 - o_q)x_{q,i}w_i \tag{2.13}$$

Backpropagation

Using gradient descent optimisation over an entire network is known as *backpropagation* (Engelbrecht, 2007). Backpropagation consists of two phases. In the feedforward pass, the output of the network is calculated. The backward propagation phase then uses the error of the output of the network to propagate changes from the output layer of the network to the input layer. Weights are adjusted proportionally to the derivative of ε with respect to w.

The backpropagation training process stops once a stopping criterion is reached (Engelbrecht, 2007). Common stopping criteria are:

- A maximum number of training cycles (epochs)
- When the network's error on a validation set is low enough
- When overfitting is observed. Overfitting is when the network begins to memorise the training set and lose its generalisation ability.

Training networks that utilise gradient-based training methods, of which backpropagation is one, have two major pitfalls. The *vanishing gradient* problem is encountered in ANNs with many hidden layers. Gradient-based training methods update a parameter's value by understanding how a small change in that value would affect the network's output. If a change in the parameter causes too small a change in the output, the network

cannot update the parameter effectively. This is due to the way that specific activation functions, such as the sigmoid function, compress the input into a small range of output values; these functions are asymptotic, meaning that there is very little difference between output values once the input becomes very large or very small. This problem is compounded as the number of layers in the network increases, since each subsequent layer maps the input region to an even smaller output region. The use of activation functions such as the rectified linear unit (ReLU) has been shown to solve the vanishing gradient problem (Maas et al., 2013).

Another problem solved through the use of ReLU over a sigmoid activation function is that a sigmoid-based network cannot produce an absolute 0 (Maas et al., 2013). It is possible for the output to be very near to 0, but this is potentially less powerful than an activation function able to produce a true 0, such as the ReLU function. In classification applications, the ReLU function may, therefore, improve performance.

Optimal network topology

The topology of a neural network has a significant impact on the network's ability to solve a problem (Floreano and Mattisussi, 2008) and its selection is a challenge in the implementation of neural networks (Hunter, Yu, Pukish, Kolbusz, and Wilamowski, 2012). The topology for a given problem can be based on any number of factors, including empirical observations or previous experiments with similar problems. Approaches such as Neuro-Evolution of Augmenting Topologies (NEAT) aim to remove some of this guesswork by evolving not only the weights but also the topology of the network (Stanley and Miikkulainen, 2002).

Transfer learning

Transfer Learning is a concept based, fundamentally, on the idea of *learning to learn* (Sinno Jialin Pan and Yang, 2010). An example is how the training of an object classification system intended to identify pears may be expedited by beginning with a system already trained to identify apples. Transfer learning also refers to the ability of a network trained on one set of data to generalise to an unseen data set.

Dropout

During training, it is possible to encounter an issue known as *over-fitting*. The ANN begins to memorise its training patterns, instead of generalising and finding the underlying link between inputs and outputs (Engelbrecht, 2007). This can happen when the network is too complex but given too few training patterns. The high number of adjustable weights allows the network to learn the individual patterns.

One way of alleviating this issue is known as *dropout* (Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov, 2014). During training, neurons are randomly chosen to be *dropped*. When this happens, the neuron's output is ignored for the feedforward phase, and its weights are not updated during the backpropagation phase. This causes the co-dependency of the neurons to decrease since they cannot rely on each other to be present and must be accurate in the absence of those other neurons.

2.2.6 Applications

ANNs have many real-world applications, from robot control (Pretorius, du Plessis, and Cilliers, 2010) to data mining (Engelbrecht, 2007). *Deep learning* is the use of many-layered neural networks, which has recently become prominent. These large ANNs are able to perform even more complex tasks, such as speech recognition, visual object recognition, object detection, and natural language processing (Lecun, Bengio, and Hinton, 2015). These ANNs have been pivotal in solving many seemingly insurmountable problems encountered in artificial intelligence research, and investigation is ongoing to test their application in increasingly diverse fields.

2.3 Evolutionary Algorithms

2.3.1 Evolutionary Computation

Evolutionary computation (EC) is a sub-field of computational intelligence, well-suited to finding adequate solutions to both dis- and continuous optimisation problems (Engelbrecht, 2007). EC makes use of evolutionary algorithms (EAs), inspired by evolution in nature, which promotes the survival of *good* genes. This occurs in nature because the indi-

viduals with good genes express those genes as traits or characteristics that increase their probability of survival, such as the ability to run faster or camouflage themselves better. Since these individuals are more likely to survive, they are more likely to reproduce. The genes that make the individuals fitter are, therefore, likely to be passed on to the following generation.

2.3.2 Basic Process

EAs are a class of population-based metaheuristics, able to perform a stochastic search through a solution space to find optimal solutions to a given problem (Engelbrecht, 2007; Bongard, 2013). The pseudocode for an EA is given in Algorithm 1.

Algorithm 1: The Evolutionary Algorithm

```
Let generation t = 0
Initialise a population, C(0), of n individuals
while stopping condition(s) not true do
    Evaluate the fitness of each individual in C(t)
    Create new population C(t + 1) using selection and reproduction operators
    Apply mutation operators to individuals in C(t + 1)
    Advance to the next generation: t = t + 1
end
```

The EA begins with the initialisation of a population of randomly generated individuals. These individuals are represented by *chromosomes*, comprised of *genes*, encoding various traits affecting the behaviour of the individual (Section 2.3.3). In order to evaluate the fitness of the individuals in the population, a *fitness function* (Section 2.3.4) must be defined.

Once the fitness values have been calculated, a reproduction step occurs, where parents are chosen using a *selection* operator (Section 2.3.6) and their genetic material is combined to create children, in a process called *crossover* (Section 2.3.7).

Once the reproduction is complete, the new population of individuals undergoes *mutation* (Section 2.3.8). In this process, the values of an individual's genes are mutated, introducing small random changes.

This process repeats until some *stopping condition* is satisfied (Section 2.3.9).

2.3.3 Chromosomal Representations

When implementing an EA, potential solutions must be represented as a set of variables (genes) (Beasley, Bull, and Martin, 1993; Engelbrecht, 2007). These sets of genes are known as *chromosomes*. As an example, for a bridge design problem, the genes could represent the lengths of individual beams supporting the bridge that the algorithm is required to optimise.

The design of the form of the chromosomes to be used in the EA is critical, as the efficiency of the algorithm depends greatly on these representations. Generally, EAs represent the individuals' chromosomes as vectors of values. Classical genetic algorithms require that the chromosome representation be *Binary-Coded*. That is, consisting of only binary (1 and 0) values. There are, however, a number of advantages to the use of real-valued chromosomal representations (Herrera, Lozano, and Verdegay, 1998). EAs using these representations are known as *Real-Coded* EAs.

The use of real-valued parameters makes it possible to use variables with large or even unknown domains (Herrera et al., 1998). This is difficult to achieve with binary representations since an increase in the size of the domain of the variables leads to decreased precision.

Real values are also able to exploit the *graduality* of functions (Herrera et al., 1998). Graduality refers to the fact that a small change of a variable's value leads to a small change in the output. Lastly, the coding and decoding steps that are required with binary coded EAs are avoided since the form of the chromosome matches that of the search space.

A negative consequence of the use of real-valued representations is that the magnitude of mutation may need to be specified by an additional *mutation magnitude* parameter.

2.3.4 Fitness Functions

In natural evolution, individuals with the best characteristics have the highest probability of surviving and reproducing. EAs require a method of expressing this *fitness* of the individuals in the population mathematically (Engelbrecht, 2007; Herrera et al., 1998;

Beasley et al., 1993); the fitness function maps chromosomal representations to scalar values.

The fitness function usually provides an absolute measure of an individual's fitness, but this is not always necessary. What is important is the ability to compare two individuals based on their fitness. In situations such as an EA learning to play a game, each individual's fitness can be based on their performance when playing against other individuals in the population. Fitness functions are integral to the function of EAs since they determine what it means for an individual to be *good* in the eyes of the EA.

When considering fitness functions, it is important to note that they are quantitative measures which aim to capture the critical features of what it means for an individual to be successful. It is easy to overestimate the accuracy with which this measure represents the underlying qualitative success that is intended (Lehman, Clune, Misevic, Adami, Beaulieu, Bentley, Bernard, Beslon, Bryson, Chrabaszcz, Cheney, Cully, Doncieux, Dyer, Ellefsen, Feldt, Fischer, Forrest, Frénoy, Gagné, Goff, Grabowski, Hodjat, Hutter, Keller, Knibbe, Krcah, Lenski, Lipson, MacCurdy, Maestre, Miikkulainen, Mitri, Moriarty, Mouret, Nguyen, Ofria, Parizeau, Parsons, Pennock, Punch, Ray, Schoenauer, Shulte, Sims, Stanley, Taddei, Tarapore, Thibault, Weimer, Watson, and Yosinksi, 2018). Evolution often exploits these differences to find solutions which are simpler than *real* solutions but satisfy the experimenter's fitness function well.

An example of such exploitation is shown in Figure 2.7. The EA was tasked with evolving a robot morphology and controller capable of jumping as high as possible off of the ground. The experimenters judged the height jumped by the robot as the change in the height of the initially lowest point of the creature. Instead of jumping, the robot in the figure learned that it could simply somersault, thrusting its very light lower limb into the air.

2.3.5 Exploration and Exploitation

An important consideration to be made when using EAs is the balance between exploration and exploitation (Engelbrecht, 2007; Črepinšek, Liu, and Mernik, 2013). Exploration is the process of visiting new and unexplored areas of the search space, while exploitation is the concentration of the search in a specific promising area in an attempt to refine the

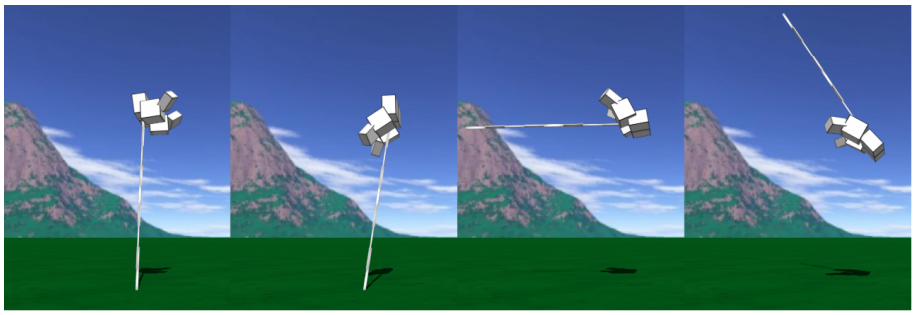


Figure 2.7: Evolution exploiting a flaw in a fitness function (Lehman et al., 2018)

candidate solutions.

A population of individuals that are too similar to each other, and thus are not adequately dispersed through the search space, is said to have low *diversity* (Engelbrecht, 2007). Such a population is likely to converge prematurely, causing it to become trapped in local optima. Alternatively, without sufficient exploitation, solutions cannot be adequately refined.

2.3.6 Selection Operators

Selection operators select the individuals from the population that are to reproduce and create offspring (Engelbrecht, 2007). This step is responsible for the *survival of the fittest* concept that is central to the evolution process.

Each selection operator has a level of *selection pressure* (Engelbrecht, 2007; Harvey, Husbands, Cli, Harvey, Husbands, and Cli, 1993). Those with higher pressure are more likely to select fitter individuals over less fit alternatives. This leads to a more rapid decline in diversity than an operator with lower selection pressure.

Four examples of selection operators are (Engelbrecht, 2007; Beasley et al., 1993):

- **Random selection:** This operator selects a random individual from the population. It has the lowest selection pressure of the operators discussed here since the least and the most fit individuals have equal likelihood of selection.

- **Best selection:** Select the best individual from the population. This operator has the highest selection pressure of the operators discussed here. Convergence with this operator will be extremely fast.

- **Proportional selection:** With this operator, the probability of an individual being selected is the proportion of that individual's fitness in the total fitness of the population:

$$p(c_k) = \frac{f(c_k)}{\sum_{i=1}^{n_p} f(c_i)} \tag{2.14}$$

  where $p(c_k)$ is the probability of individual c_k being selected, $f(c)$ is the fitness of c, and n_p is the number of individuals in the population. This operator is also known as *roulette wheel* selection.

  Since selection is directly proportional to fitness, it is possible that fitter individuals may dominate too strongly, leading to a rapid decrease in the diversity of the population.

- **Tournament selection:** This selection operator selects a random group of n_t individuals, where n_t is a parameter to the EA, and $n_t \leq n_p$. The best individual from this random group is then selected. Tournament selection could be said to be a combination of random and best selection. The larger the tournament, the higher the selection pressure; with a tournament the size of the entire population, tournament selection is the same as selecting the best individual.

It is not guaranteed that the fittest individual from each population will survive to the next. Elitism is a process which preserves the fittest individuals by copying them directly to the new population without mutation. The higher the number of individuals transferred (elitists), the faster the population will converge (Engelbrecht, 2007).

2.3.7 Crossover

After individuals have been selected for reproduction, their genetic material is combined to create offspring (Engelbrecht, 2007). Three methods of implementing this crossover, initially intended for use with EAs using binary-coded chromosomes, however also able to be used with real-coded chromosomes, are:

- **Single-point crossover:** A point, corresponding with a specific position in the chromosomes of both parents, is chosen. Two children can then be created, with the

children's chromosomes matching one parent up until the crossover point, and the other parent afterwards.

- n**-point crossover:** Instead of a single point being chosen, where the chromosome of the offspring switches between parents, n points are chosen. At each of these points, the parent from which the genetic material is taken is swapped.
- **Uniform crossover:** Uniform crossover gives each gene in the offspring's chromosome an equal probability of being taken from each parent.

There are crossover methods explicitly designed for use with real-coded chromosomes, such as simulated binary crossover (SBX), which aims to simulate the behaviour of single-point crossover for real-coded chromosomes (Deb and Agrawal, 1995). Two parents c_1 and c_2 are used to produce two offspring $\tilde{c_1}$ and $\tilde{c_2}$:

$$\tilde{c}_{1j} = 0.5((1+\gamma_j)c_{1j} + (1-\gamma_j)c_{2j}) \tag{2.15}$$

$$\tilde{c}_{2j} = 0.5((1-\gamma_j)c_{1j} + (1+\gamma_j)c_{2j}) \tag{2.16}$$

$$\gamma_j = \begin{cases} (2v_j)^{\frac{1}{\eta+1}} & \text{if} \quad v_j \leq 0.5 \\ (\frac{1}{2(1-v_j)})^{\frac{1}{\eta+1}} & \text{otherwise} \end{cases} \tag{2.17}$$

where $j \in [0, n_g]$, n_g is the number of genes in the individuals' chromosomes, $v_j \sim U(0,1)$ and $\eta > 0$ is the distribution index. Deb, Pratab, Agarwal, and Meyarivan (2002) suggested that $\eta = 1$.

The chosen crossover operator is repeatedly used, with the selection operator selecting parents each time, until a new population of individuals has been created.

2.3.8 Mutation

The aim of mutation is to introduce diversity into the population (Engelbrecht, 2007). This is achieved by probabilistically changing the values of the individuals' genes by small amounts. Mutation is applied to each gene of the individual with probability m_r. The probability is usually a small value to ensure that good solutions are not distorted excessively.

Binary-coded chromosomes are mutated by flipping their genes' values from 0 to 1 or 1 to 0. Uniform mutation for real-coded chromosomes allows for mutation without a specific mutation magnitude parameter (Engelbrecht, 2007):

$$c_i' = \begin{cases} c_i + \Omega(0, x_{max} - x_i) & \text{if v} < 0.5 \\ c_i + \Omega(0, x_i - x_{min}) & \text{otherwise} \end{cases} \tag{2.18}$$

where c_i is the i^{th} gene of individual c, c_{min} and c_{max} are the minimum and maximum values of the genes, respectively, $v \sim U(0,1)$, and $\Omega(x_1, x_2) \sim U(x_1, x_2)$.

Another method of mutation allows the magnitude of the mutation to be specified as a parameter m_g to the algorithm. Each mutation operation then mutates the targeted gene using

$$c_i' = \Omega(-m_g, m_g) + c_i \tag{2.19}$$

or

$$c_i' = \Psi(m_g) + c_i \tag{2.20}$$

where $\Psi(x) \sim N(0, x^2)$.

EAs with initially high mutation rates, which become smaller over time, are shown to have improved convergence, speed, and accuracy over those with constant mutation rates (Engelbrecht, 2007).

2.3.9 Stopping Conditions

An EA continues the evolutionary cycle until some stopping condition is reached (Engelbrecht, 2007). Possibly the simplest condition is a limit on the number of evolution cycles that can occur. Convergence is also a useful criterion in deciding when evolution should cease (Beasley et al., 1993). A performance measure can be chosen to represent the population's aggregate fitness. The measure could be one of any number of measures, such as the fitness of the best, or average, individual. The rate of change of this fitness measure can then be used to monitor convergence of the population; as convergence increases, the rate of improvement of the population's fitness will decrease (Engelbrecht, 2007). Evolution can thus be stopped when this rate of improvement has slowed to a point where the best fitness has not improved more than a threshold value e over a number of generations n_g, where e and n_g are parameters to the algorithm.

2.3.10 Multiple Populations

EAs lend themselves to parallelisation, allowing the workload of computing the EA to be split among multiple processing units (Engelbrecht, 2007; Alba and Tomassini, 2002). There are a number of methods of implementing this parallelisation, one of which is an *island* EA. Island EAs consist of multiple populations, each of which can be computed on a separate processor. Selection, crossover, and mutation happen within each population, entirely independently of the others. Periodically, individuals are chosen to migrate among populations. Island EAs are useful not only for parallelisation; they also increase the diversity of the EA (Alba and Tomassini, 2002).

A necessary step in the construction of an Island EA is the specification of a migration policy, which defines (Engelbrecht, 2007; Alba and Tomassini, 2002):

- **A communications topology**, specifying the connections between islands. These are the paths along which information can be shared and along which individuals can migrate. This topology will determine how rapidly good solutions are communicated to each population. Sparse topologies communicate these solutions more slowly and thus facilitate the emergence of a more varied set of solutions.

- **A migration rate**, which determines the frequency of migration among populations. Ideally migration should occur once each population has converged in order to prompt a resurgence of exploratory evolution.

- **A selection mechanism**, which is responsible for deciding which individuals should migrate. This mechanism can utilise any fitness selection method.

- **A replacement strategy**, to select the individuals of the destination island to be replaced by the arriving migrant individuals.

An alternative to a predefined policy is a probabilistic one, such as is used by a dynamic model, where migration decisions are made probabilistically (Engelbrecht, 2007). Migration occurs with a fixed probability; the migrating individual is chosen using tournament selection, migrates to a random island, and is accepted probabilistically based on its fitness relative to the average fitness of the destination island.

2.3.11 Neuroevolution

EAs provide an appealing alternative to gradient-based learning methods for the training of neural networks (Floreano and Mattisussi, 2008; Stanley and Miikkulainen, 2002). The properties of the network can be encoded in genes and evolved by the EA. This approach holds advantages over other learning methods since it can be used to evolve many defining features of a network simultaneously, such as the network's structure and weights. It also gives more flexibility with regards to the definition of performance criteria than an error function.

A simple form of the network encoding takes the form of a weight vector representing the weights of each connection in the network (Figure 2.8). This specific representation cannot be used for the evolution of the network's topology alongside the weights, such as is done by NEAT; it does not encode for the topology directly.

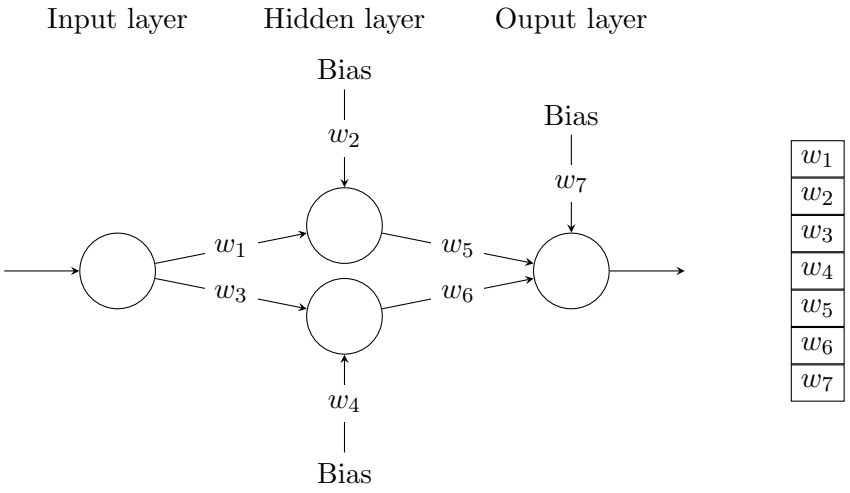


Figure 2.8: A FFNN and a feasible corresponding weight vector representation

Neuroevolution utilises crossover to create child networks. Each ANN can be said to consist of a number of smaller sub-structures (Stanley and Miikkulainen, 2002). Neuroevolution assumes that fit individuals are fit because each of the sub-structures making up their full structure has moderately good fitness. It is also assumed that combining good sub-structures can create good structures. When these assumptions are not true, the environment is said to be *deceptive*.

The *competing conventions problem*, also known as the *permutation problem* (Han-

cock, 1992; Whitley, 1995), is one that can arise in a deceptive environment (Angeline, Angeline, Saunders, Saunders, Pollack, and Pollack, 1994; Yao and Liu, 1997; Stanley and Miikkulainen, 2002). In Figure 2.9, the activation functions throughout the hidden layer are constant, and bias weights are excluded in the interest of readability. Networks 2.9a and 2.9b are reflections of each other and, given the same input, both networks would produce the same result. If these networks represent fit individuals in the population and are chosen to reproduce, a uniform crossover operation could produce network 2.9c. Instead of achieving the goal of creating a fitter individual through the crossover of genetic material, a child has been created that contains repeated components and has lost a portion of the parent's computational ability.

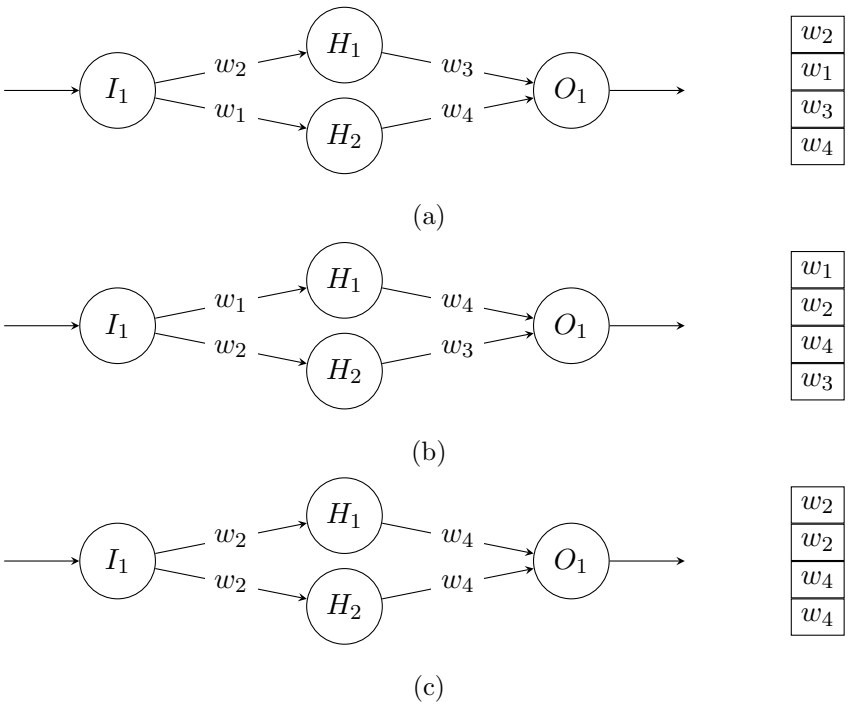


Figure 2.9: Reflected neural networks

The NEAT algorithm aims, among other things, to solve the competing conventions problem by evolving both the topology and weights of the network simultaneously (Stanley and Miikkulainen, 2002). While NEAT can offer advantages over other methods of neuroevolution, previous research has shown that this is not true in all cases (Pretorius, du Plessis, and Gonsalves, 2017). A number of other approaches have been developed

with the aim of addressing the competing conventions problem (Whitley, 1995), but research has found that it does not cause as negative an impact as has often been reported (Hancock, 1992).

2.4 Evolutionary Robotics

The manual design of mobile, autonomous, and adaptive robots is difficult. Industrial robots are able to operate autonomously, but they are not adaptive and they cannot respond to external stimuli; rather, they execute the same sequence of actions repeatedly. Remote controlled drones are adaptive but not autonomous, as they rely on their human operator for control (Bongard, 2013).

It is possible to encode the robots' controllers and morphologies as individuals of an EA and use concepts from EC to evolve them automatically (Zagal and Ruiz-Del-Solar, 2007; Bongard, 2013; Hiller and Lipson, 2012). This method also means that few assumptions need to be made in the robots' implementation, allowing for the discovery of robot behaviours and morphologies that may not have been envisioned by the human designers.

2.4.1 A Brief Background of Evolutionary Robotics

The aim of artificial intelligence is to produce human-like intelligence, where the definition of intelligence is based on something similar to the Turing test. Evolutionary robotics (ER), on the other hand, focuses on producing *behaviours* which appear intelligent to an observer (Bongard, 2013). In ER, the belief is that *intelligence* is an emergent property of the increasingly complex interactions between the robot and its environment, instead of being something contained in just the robot's *brain*. This concept is known as *embodied cognition*.

Early ER experiments made interesting discoveries with regard to embodied cognition. An ER system, when tasked with moving a robot towards specific shapes and away from others, learned to use both motion and vision to identify shapes, instead of attempting to classify the shapes before making a decision (Floreano and Mattisussi, 2008). The system used only a small number of pixels, but also used information about how the values of

those pixels changed over time. The method of object identification was unintuitive, and it is unlikely that a human researcher would have chosen to solve the problem in the same way.

Interestingly, not only does biological evolution inspire ER, but discoveries in ER can be used to assist research into biological evolution (Bongard, 2013). The outcomes of the ER process often provide researchers with new ways of thinking about biologically evolved traits. If an evolved robot expresses similar traits or behaves in a similar manner to a biological animal, investigation of the process that led to the emergence of that behaviour in the robot can provide insight into the same behaviours or traits in nature.

This automatic generation of unexpected solutions extends beyond the creation of controllers. Unexpected robot morphologies can be generated too (Hiller and Lipson, 2012; Lipson, Bongard, Zykov, and Malone, 2006). Research has shown that allowing the morphology of the robot to evolve alongside the controllers produces significantly better results, though much of this research has taken place in simulation because of difficulties involved with the bridging of the reality gap (Section 2.5.2), and the complexity of the real-world fabrication of robot morphologies for evaluation in the ER process (Bongard, 2013; Cheney, Maccurdy, Clune, and Lipson, 2013).

2.4.2 The Evolutionary Robotics Process

The ER process is related closely to that of EAs. The process is illustrated in Algorithm 2. A population of controllers is created with genes encoding aspects of the individuals in the population. In the example of neural network-based controllers, the genes can code for the weights of connections between neurons. These individuals are decoded, and controllers created. The controllers are then transferred to a robot and evaluated, based on their ability to complete a given task. Once this evaluation is completed, a fitness function returns a fitness value for each individual. A selection operator is then applied to select pairs of parents between which a crossover operation is performed to create the new population. Finally, the individuals in this new population have small random changes introduced through the application of a mutation operator.

Algorithm 2: The basic Evolutionary Robotics Algorithm

Let generation $t = 0$

Initialise a population, $C(0)$, of n individuals

while stopping condition(s) not true **do**

Decode each individual in $C(t)$ into a controller in D

Evaluate the fitness of each controller in D by determining how well it performs the required task

Create new population $C(t + 1)$ using selection and reproduction operators

Apply mutation operators to individuals in $C(t + 1)$

Advance to the next generation: $t = t + 1$

end

2.5 Simulators in Evolutionary Robotics

In ER, the evolutionary process requires the evaluation of each individual in every generation. If these evaluations are performed on a real robot, the process becomes unrealistically time-consuming (Zagal and Ruiz-Del-Solar, 2007). Real-world evaluations also introduce the risk of damage to physical robots (Papaspyros et al., 2016). This is especially true in the case of controllers that perform poorly or cause the robot to perform erratic and hazardous movements. These may occur early in the evolution process while individuals are largely random, or later, due to random mutation.

Simulators allow for much faster exploration of the search space (Lund and Miglino, 1996) with no risk of damage to a physical robot (Sofge, Potter, Bugajska, and Schultz, 2003).

2.5.1 Types of Simulators in Evolutionary Robotics

Simulators in ER can be grouped into three broad classes: physics-based, empirical, and hybrid approaches. Physics-based simulators aim to use mathematical models to describe the interaction between a robot and its environment. These simulators can be challenging to develop due to the complex models upon which they are based (Koenig and Howard, 2004).

Empirical approaches use experimentally collected data to construct simulators (Lipson

et al., 2006). The resulting simulators are able to represent the *fuzzy* characteristics of the real world.

Hybrid simulators incorporate aspects of both empirical and physics-based approaches (Jakobi et al., 1995). An example is the *anytime learning* approach which uses a physics-based model, with empirically determined parameter values (Parker, 2000).

2.5.2 The Reality Gap

While simulation in ER yields many benefits, it is not without issues of its own. Due to the complexity of representing the real world, simplifying assumptions are made in simulators' development (Brooks, 1992), causing the accuracy of the simulation to decrease. These inaccuracies mean that a controller that performs well in the simulator may not do so in the real world.

Controllers evolved in simulation may also learn to exploit the inaccuracies in the simulation. In this case, the inaccuracies prevent the controller from transferring to reality at all, since the controller relies on the inaccuracy in order to be effective in the completion of its task. Lehman et al. (2018) went so far as to describe the use of EC techniques in simulators as *automated bug discovery* since the solutions evolved in simulation very often learn to exploit bugs in the simulation itself. These issues of transference and exploitation of simulation inaccuracies together form the *reality gap*.

An example, presented by Koos, Mouret, and Doncieux (2013a) (Figure 2.10) clearly shows the reality gap. The vertical axis represents the distance travelled by a quadruped robot, while the horizontal axis represents the number of generations of the EA. As the population of controllers improves, it increasingly exploits aspects of the simulation which may not accurately reflect reality. The improvements in simulation actually lead to the controller performing worse in the real world.

Figure 2.11 shows another example of a robot failing to cross the reality gap. Jakobi et al. (1995) evolved controllers to move a wheeled robot forwards as fast as possible while avoiding obstacles. 2.11a shows the path followed by the robot in simulation, while 2.11b shows its path in the real world.

Figure 2.12 shows the simulated execution of a controller on a *creature* which has learned to exploit a bug in the simulation. Each image shows one timestep of the robot's

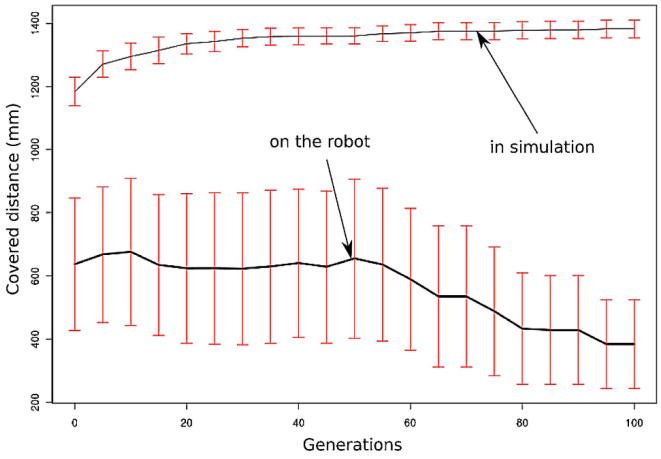
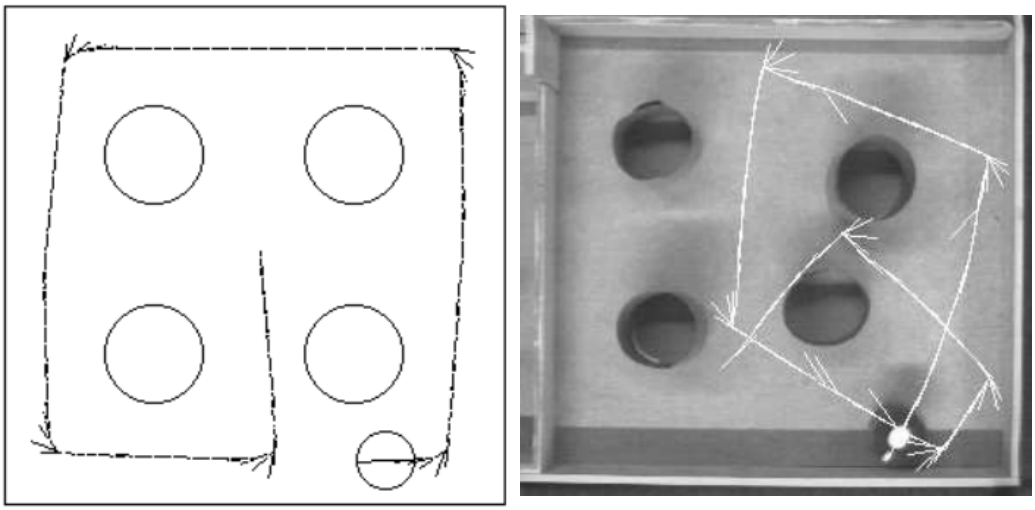


Figure 2.10: The Reality Gap (Koos et al., 2013a)



(a) Robot's path in simulation (b) Robot's path in reality

Figure 2.11: A robot's path in reality and simulation (Jakobi et al., 1995)

execution as it moves to the right. The simulator's time steps were large; this allowed the creature to penetrate the ground between steps, which would not be possible in the real world. This collision causes the simulator to repel the creature strongly, providing it with *free* energy and allowing it to vibrate forwards along the ground (Lehman et al., 2018).

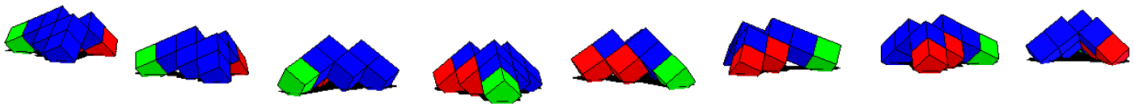


Figure 2.12: Evolution exploiting a bug in the simulator (Lehman et al., 2018)

2.6 Solving The Reality Gap

As an open issue in ER (Sofge et al., 2003), there have been numerous attempts to address the reality gap. Three of these are discussed in this section.

2.6.1 The Transferability Approach

The transferability approach is focused on discovering controllers that transfer well between simulation and reality (Koos et al., 2013a). The algorithm solves for two objectives: fitness and transferability. The fitness calculation is done as in other ER approaches, while the transferability is estimated by a transferability function, which represents the level of accuracy with which the simulator mimics reality, for a given action. This function is usually approximated with a regression algorithm (Koos et al., 2013a).

The fitness and transferability of a controller often seem to be conflicting objectives, with an increase in one having a negative effect on the other (Koos et al., 2013a). Thus, a Pareto-based multiobjective optimisation algorithm is used (Deb et al., 2002). These algorithms are based on the concept of Pareto dominance:

Definition 2.6.1. *Pareto dominance*: A solution x^* is said to dominate another solution x if (1) and (2) hold:

1. The solution x^* is not worse than x with respect to all objectives.

2. The solution x^* is strictly better than x with respect to at least one objective.

All non-dominated solutions in a search space form the Pareto front, which represents the set of optimal *trade-offs* for the given problem. In the case of the transferability approach, they are the solutions which have the best balance between transferability and fitness.

2.6.2 Real-World Evolution

Floreano and Mondada (1994) evolved a robot controller without a simulator by performing the evolution process on the physical robot. This approach yields excellent results, but comes with the issues inherent in real-world testing: it is time-consuming and risks damage to the robot.

2.6.3 Concurrent Controller and Simulator Development

Instead of treating the training and deployment of a controller as a one-way process, concurrent methods of controller and simulator development use information gathered from the deployment of the controller to inform the simulator and the training of new controllers.

Zagal and Ruiz-Del-Solar (2007) proposed the Back To Reality Algorithm. This algorithm co-evolves both the robot and simulator by first evolving a controller in simulation and then transferring it to a real robot (Zagal and Ruiz-del Solar, 2005). The controller is then evolved further on the real robot, and the difference in fitness between the simulator and real world calculated and used to train the simulator. The optimisation of the original researchers' UCHILSIM simulator was done through parameter modification. This method, while effective, still requires a time-consuming evolution process in the real world.

Bongard et al. (2006) used a self-modelling system to allow a controller to build up an internal model of its morphology, discussed further in Section 2.9.2. Self-modelling is effective in the evolution of transferable robot controllers, but, like many other methods of ER, requires the very time-consuming construction of a physics simulator before evolution can take place.

The transferability approach, discussed in Section 2.6.1, is a concurrent method of sim-

ulator and controller development. Once again, this method requires the time-consuming construction of a physics simulator. Koos et al. (2013a) suggest the use of both an accurate and simplified simulator in order to reduce computational complexity further, but this also implies the implementation of a second physics simulator in addition to a transferability estimation function.

The anytime learning approach uses a simulator, initially constructed to be as accurate as possible, to evolve a population of controllers (Parker, 2000). Periodically the best, worst, and a random individual from this population, are chosen and tested on a real-world robot. The collected data is then used to evolve a population of model parameters, with the aim of using them to improve the simulator. An illustration of the process is shown in Figure 2.13. While this method is effective at updating the simulator to be as close to the real world as possible, the only parameters able to be modified by the algorithm are those chosen by the researcher. If there are unanticipated sources of simulator inaccuracy, they cannot be addressed automatically by this method.

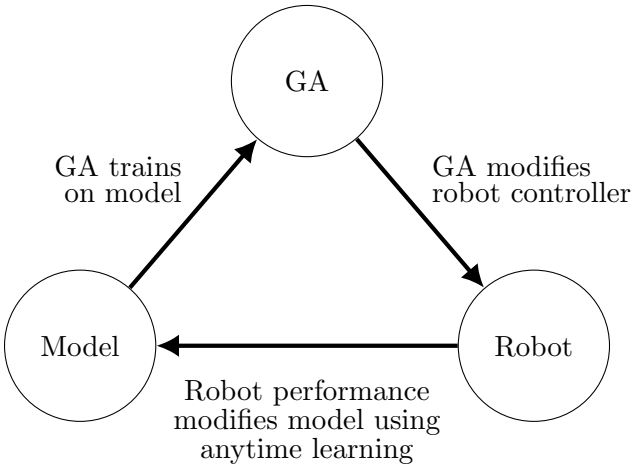


Figure 2.13: Anytime Learning process (adapted from Parker (2000))

2.7 Simulator Neural Networks

Simulator neural networks (SNNs) are an empirical approach to robot simulation (Pretorius, du Plessis, and Cilliers, 2013). SNNs make use of artificial neural networks which are trained to take sensor information and commands and produce a prediction of the

change in the robot's state. An example SNN is shown in Figure 2.14. In this case, the simulator is designed for a differentially steered Khepera III robot, as used by Woodford et al. (2016), and consists of three neural networks, each predicting one of the changes in x, y, and rotation of the robot, respectively.

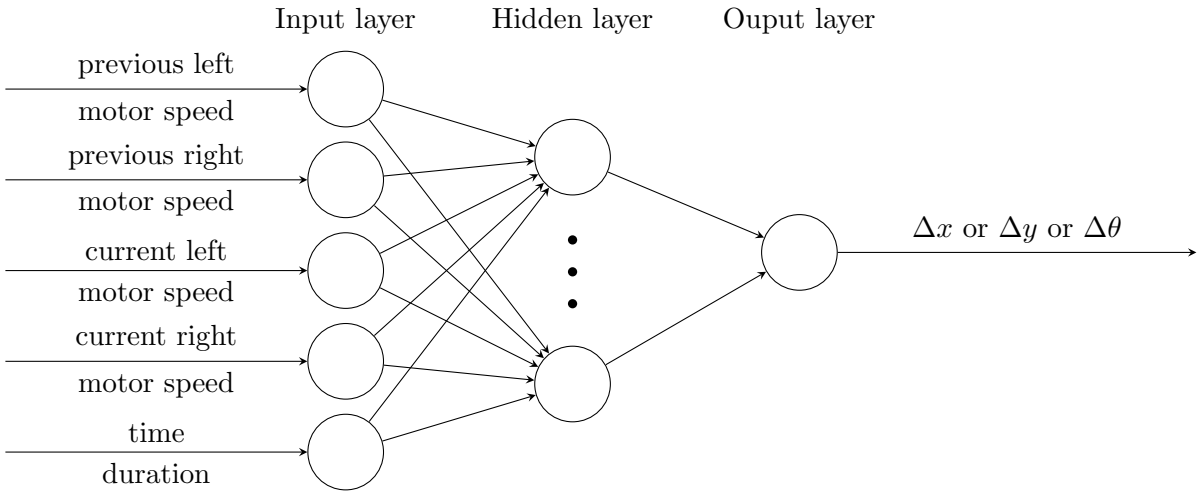


Figure 2.14: Simulator Neural Networks for the Khepera III (Woodford et al., 2016)

NNs are typically trained using collected training data. Since the simulator is intended to simulate a real-world robot, this data comes from the real world, which inevitably contains large quantities of noise (Jakobi et al., 1995). It is also impossible to sample every possible combination of parameter values from the real world, such as every possible motor speed. ANNs address both of these issues.

- ANNs' high noise tolerance allows them to filter out noise and find the underlying relationships in real-world data (Basheer and Hajmeer, 2000).
- ANNs' generalisation ability allows them to approximate the outputs for inputs that were not presented to the network during training, by finding the relationships between input and output values (Pretorius et al., 2013).

SNNs can be constructed without prior knowledge of the underlying mathematical model of the system being simulated (Pretorius et al., 2013), making them an appealing alternative to physics-based approaches. The process of training is as follows:

1. Training data is obtained by:

(a) executing random commands on a real-world robot and

(b) recording the change in the robot's physical state and relevant sensor values.

2. A SNN is trained using the collected data.

3. The SNN is used as a simulator to evolve robot controllers.

4. The controllers are transferred to the real robot for evaluation in the real world.

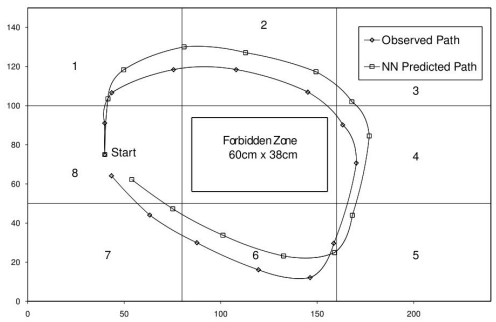
SNNs have also been shown to perform comparably to physics simulators while offering computational efficiency that is orders of magnitude better than the physics simulators of comparable accuracy (Pretorius, du Plessis, and Gonsalves, 2014).

Pretorius, du Plessis, and Cilliers (2009) used SNNs to evolve a robot controller for a differentially steered robot. The robot was required to navigate from one region of the testing area through the others, in order, while avoiding the centre *forbidden zone*. The process was successful and controllers were evolved that were able to bridge the reality gap and perform their task in the real world (Figure 2.15a). Individual ANNs simulating each aspect of the robot that is to be simulated were also shown to be more effective than single ANNs simulating all aspects of the robot.

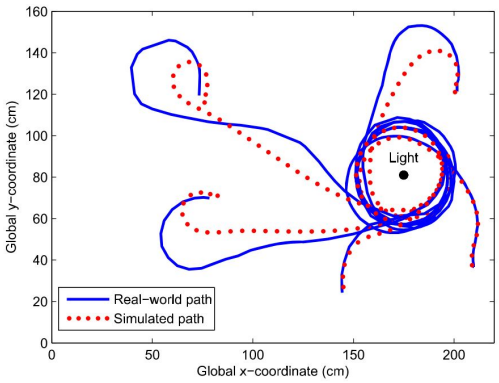
Pretorius et al. (2010) showed that multiple SNNs can be used together, specifically to simulate both the motion of a differentially steered robot and its light sensor readings, based on its position relative to a light source. These simulators were used to evolve closed-loop controllers capable of navigating towards and remaining near a light source (Figure 2.15b). The problem simulated was simple and could arguably have a physics simulator constructed for it fairly easily, but the use of SNNs reduced the complexity of simulator construction and was able to automatically take into account intricacies which would otherwise have required explicit implementation in a physics simulator.

It has been demonstrated that the simulation of more complex systems, such as an *inverted pendulum* balancing robot, is possible (Pretorius et al., 2013). SNNs were able to simulate both the on-board gyroscopic and tilt sensors and were used to evolve controllers that were able to bridge the reality gap. This finding showed that SNNs are not limited to the simulation of simple systems, but could potentially be used for the evolution of robot controllers able to perform complex tasks in the real world.

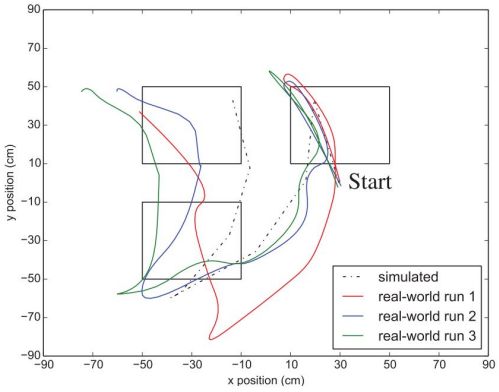
SNNs have also been used to simulate the behaviour of much more complex robotic systems, such as a snake-like robot (Woodford, du Plessis, and Pretorius, 2015). Data was gathered and an SNN trained, which was used to evolve a controller for the robot capable of completing a navigation task. The controllers transferred reasonably to the real world (Figure 2.15c) and proved that the use of SNNs is viable for the evolution of robot controllers for robot morphologies of much greater complexity. Closed-loop controllers were also successfully evolved using SNNs, allowing the robot to move towards, and remain near, a light source, using light sensors on the robot (Figure 2.15b) (Pretorius et al., 2013).



(a) Path followed by an open-loop controller evolved using SNNs (Pretorius et al., 2009)



(b) Path followed by a closed-loop controller evolved using SNNs (Pretorius et al., 2010)



(c) Path followed by an open-loop controller for a snake robot evolved using SNNs (Woodford et al., 2015)

Figure 2.15: Paths followed by controllers evolved using SNNs

2.8 Bootstrapped Neuro-Simulation

There are disadvantages to the traditional approach used for SNN development (Section 2.7). Since it is a one-directional process, the simulator must be developed before the training of controllers can begin (Woodford et al., 2015). Time may also be wasted while the simulator learns to simulate behaviours that are not required for the successful evolution of controllers for a given task. Finally, once the simulator is developed, it is unable to adapt to changes in the robot's performance that may occur due to mechanical wear, damage, or environmental factors.

Woodford et al. (2016), therefore, proposed using the concepts of bidirectional training and SNNs in conjunction. The resulting approach, called *Bootstrapped Neuro-Simulation* (BNS), trains a population of controllers using an initially random SNN. Controllers are continually chosen from this population to be evaluated in the real world. Data from these real-world tests are then used to train the SNN, which continues to train the controller population. A diagram of the training process is shown in Figure 2.16 and shows the following steps (Woodford et al., 2015):

1. Controllers are selected from the evolving population for real-world evaluation. The selection of these controllers can be done using any evolutionary algorithm selection method.

2. The selected controller is evaluated in the real world, independently of the controller evolution process, which continues in the background.

3. The data from this real-world evaluation is collected and placed in a training buffer for the SNN.

4. The data in the training buffer is integrated into the training of the SNN.

5. The simulator used for training controllers is periodically updated with a new, more highly trained simulator and is used to evolve controllers.

6. The process repeats until the stopping conditions are met.

Experiments have confirmed that the BNS is able to generate transferable controllers for navigation tasks on a differentially steered robot in as few as fifteen real-world eval-

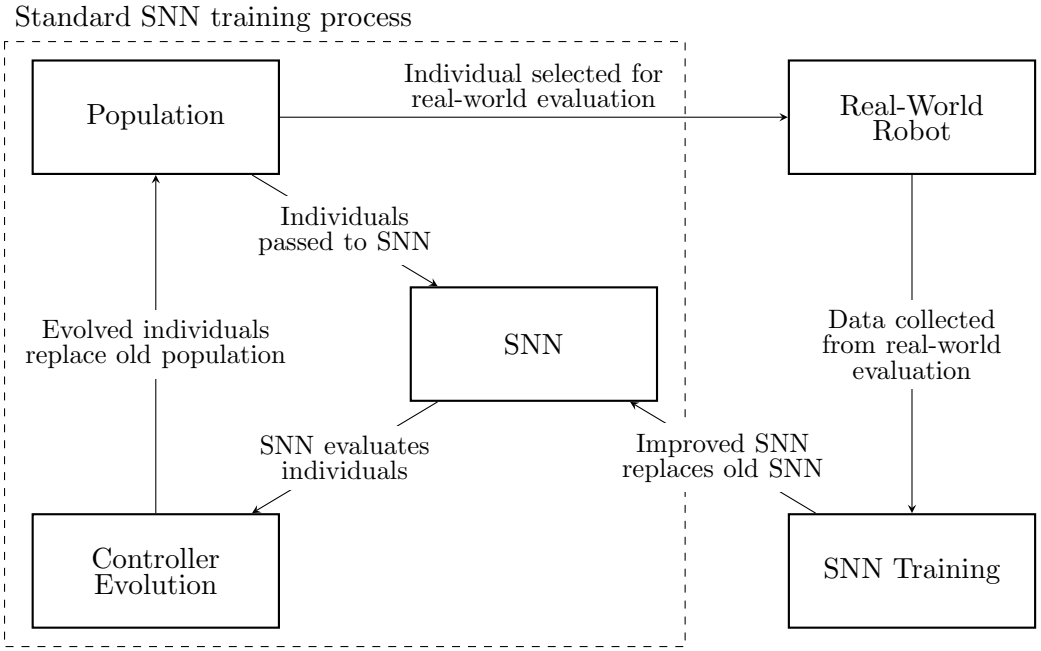


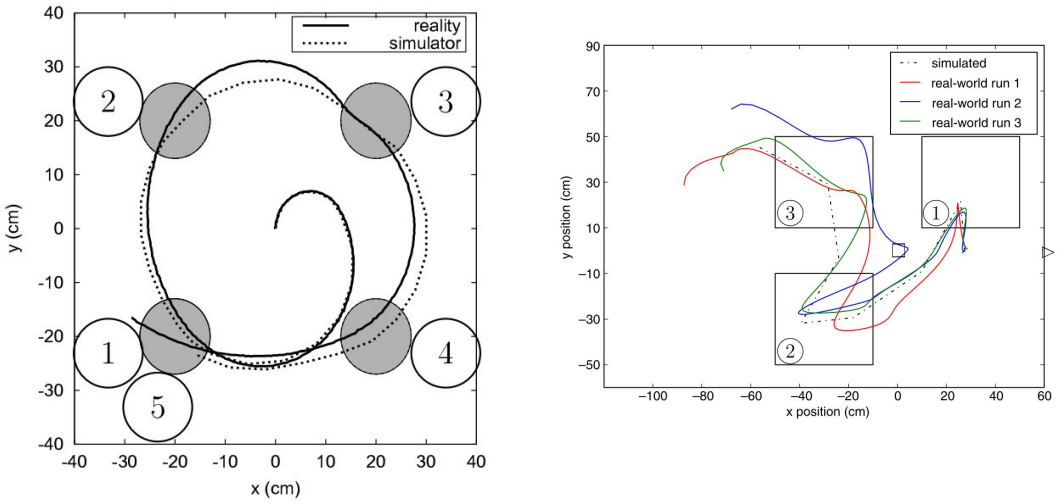
Figure 2.16: The BNS process

uations (Figure 2.17a) (Woodford et al., 2016) and locomotion of a snake-like robot in nineteen real-world evaluations (Figure 2.17b) (Woodford et al., 2017). Simulators trained using the BNS approach are less generalised than those created using traditional SNN creation methods, but are able to be trained more rapidly; their use may be situational and based on the specificity of the controller's intended task (Woodford et al., 2016).

The controllers evolved using BNS have, thus far, all been open-loop controllers. These controllers do not take in information about their environment, but rather execute a fixed set of commands. Previous SNN research has shown that SNNs can be used to evolve closed-loop controllers able to react to external stimuli (Pretorius et al., 2013), but the same has not been shown for BNS.

2.9 Damage

Many early robotic systems operated only in safe environments under close human supervision (James G. Bellingham and Kanna Rajan, 2007). Now, as the field of robotics progresses, robots are able to work in more hostile environments in which it is unsafe or impossible for humans to work (Sanderson, 2010). Given this fact, it is inevitable that



(a) Path followed by a differentially steered robot controller evolved using BNS (Woodford et al., 2016)

(b) Path followed by a snake robot controller evolved using BNS (Woodford et al., 2017)

Figure 2.17: BNS Results

these robots will become damaged (Verma, Gordon, Simmons, and Thrun, 2004; Sanderson, 2010). It is vital that robots are able recover from damage, since a non-functioning or badly functioning robot may have significant adverse financial or scientific effects, or may negatively affect public opinion of the technology (Verma et al., 2004).

Section 2.9.1 discusses how damage can be detected while Sections 2.9.2 and 2.9.3 discuss algorithms which can be implemented to recover from damage. Finally, Sections 2.9.4 to 2.9.6 discuss three methods of damage recovery based on the intelligent selection evaluation of candidate controllers in the real world.

2.9.1 Detection and Recovery

Many, but not all, methods of damage recovery require damage to be *detected* before recovery can take place. Traditional approaches to damage detection involve the use of sensors to detect and understand damage (Bongard and Lipson, 2005) to allow the robot to determine an appropriate response (Papaspyros et al., 2016; Chatzilygeroudis et al., 2016). This method of fault identification is complex because the search space of possible faults is large. Identification of faults also necessitates the addition of more fault detection sensors,

which further increase the complexity of the robot (Sofge et al., 2003) and, therefore, the probability of damage occurring (Chatzilygeroudis et al., 2016).

Recently, a large amount of research has been conducted investigating methods of automatic damage recovery using reinforcement learning through trial-and-error approaches. This approach avoids the diagnostic step of damage recovery entirely (Chatzilygeroudis et al., 2016; Koos et al., 2013a; Cully et al., 2015; Papaspyros et al., 2016). Many of these methods are also able to automatically detect damage, since they have predetermined expectations for the outcome of given actions. When the actual outcome differs significantly from their expectations, they can assume that damage has occurred.

2.9.2 Continuous Self-Modelling

Proposed by Bongard et al. (2006), continuous self-modelling aims to improve robots' robustness under uncertainty and allow them to recover from physical damage. Using this method, the robot infers its morphology through self-directed exploration. Initially, a random motor command is executed and the data is collected to create fifteen candidate models in an internal physics simulator. Thereafter, the process is as follows:

1. Search for the motor action that, when executed, causes the most disagreement among the fifteen internal models.
2. Execute the action while gathering sensor data.
3. Use the gathered data to update the candidate models.
4. Repeat from (1) until the process has been run fifteen times.

The motor action is chosen for the amount of disagreement it causes among models since this has been found to elicit the most information about the robot (Bongard et al., 2006). The amount of disagreement between candidate models is also a good indicator of model error; experiments with the least disagreement are, therefore, determined to be the most successful. If the robot detects motor values or sensor inputs that do not align with its internal models, possibly caused by damage to the robot, the same process as was used to construct the initial model is used to restructure it. This allows the robot to discover qualitatively different behaviours to compensate for the damage.

In the initial studies, the candidate models were represented as 3D models in physics simulators. Implicit representations, such as could be achieved with neural networks, would likely align more accurately with the way animals understand their morphology, but have yet to be implemented (Bongard et al., 2006).

In summary, self-modelling is a novel idea that allows robot controllers using this technique to effectively recover from damage by maintaining an internal model of their own morphology.

2.9.3 T-Resilience Algorithm

The Transferability-Resilience (T-Resilience) algorithm builds on the concept of a self-model and adapts it, along with the transferability approach (Section 2.6.1), for damage recovery (Koos, Cully, and Mouret, 2013b). T-Resilience, like continuous self-modelling, has an internal self-model and is able to automatically detect damage when sensor readings differ drastically from its predictions.

Unlike continuous self-modelling, T-Resilience does not update its internal model when damage is detected; instead, the assumption is made that the old model, while inaccurate, must simulate the damaged robot at least somewhat (Koos et al., 2013b). A large portion of the robot's motors should still rotate in the same planes, and the robot's weight distribution should be at least similar to the undamaged robot.

The transferability approach acknowledges that simulations are never perfect. The T-Resilience algorithm applies this concept, simply treating the model of the undamaged robot as an inaccurate model of the damaged one (Koos et al., 2013b,a). It is assumed that there are behaviours which still work in the presence of the damage and will thus transfer well from the inaccurate model to the real world. For example, a six-legged hexapod could develop a gait that does not utilise one of its legs. If that leg were to become damaged, the gait would still transfer well from the internal model to the real world.

Pseudocode for the T-Resilience algorithm is shown in Figure 3, where Discovery-Loop() runs N generations of a multi-objective EA on a population P, maximising the performance of the solutions as predicted by the self-model, the approximate transferability between the self-model and real robot, and a helper-objective, which aims to maintain diversity in the population.

Algorithm 3: T-Resilience Algorithm

Initialize a random population P

Initialize an empty training data set D

for i = 0 →Desired number of real-world tests **do**

Randomly select x from P

Compute the behaviour, $b(x)$, of x

Transfer x to the real robot

Execute x and estimate its real-world fitness value, $f_r(x)$

Calculate the fitness for x from the self model, $f_s(x)$

Calculate the transferability score $T(b(x)) = \parallel f_s(x) - f_r(x) \parallel$

Record $b(x)$ and $T(b(x))$ to D

Improve the transferability function using D

DiscoveryLoop()

end

Select the new controller

T-Resilience algorithm is based on the novel idea that even after damage, there are likely still behaviours able to transfer effectively between the simulator and reality.

2.9.4 Intelligent Trial-and-Error

The Intelligent Trial-and-Error (IT&E) (Cully et al., 2015) algorithm is based on the concept that animals in nature, after experiencing an injury, attempt to find methods of locomotion that will minimise pain while allowing them to maintain their mobility. They use their previous experiences and intuition to intelligently select methods to evaluate, instead of trying them at random.

IT&E uses a discrete and finite search space called the *behaviour space*, where each dimension describes a behavioural characteristic of the robot. A mapping of behaviours in the space to a performance measure, the *behaviour-performance map*, is created to act as the robot's intuition (Cully et al., 2015). A *behaviour-confidence map* is also saved, recording the robot's confidence in its prediction of each behaviour. The behaviour-performance and-confidence maps are used in conjunction to inform the selection of behaviours for eval-

uation in the real world. When these real-world evaluations take place, data is gathered and used to update both spaces, increasing confidence in the prediction and updating the predicted behaviour. The process ends when the robot predicts that it has found the most effective behaviour.

Before constructing the behaviour space, the robot's designers must describe the dimensions of the space, as well as a performance measure (Cully et al., 2015). In the example of a Hexapod robot, behaviours could be placed into a six-dimensional space, with each dimension representing the amount of time that one of the legs is making contact with the ground. A performance measure could be defined as the distance that the robot travels in a given time. Once these have been determined, the behaviour space can be constructed offline, using the MAP-Elites algorithm (Algorithm 4).

The MAP-Elites algorithm computes a total of n_t controllers to be added to the behaviour-performance map. The designers of the algorithm used $n_t = 40$ million. The algorithm begins by evaluating R random controllers in the simulator and placing them into the behaviour space. R was chosen to be 400 by the designers. The remaining $n_t - R$ controllers are created through the random mutation of the existing controllers. Whenever a controller is evaluated, its position in the behaviour space is calculated. If the position is empty, the controller is placed there. Otherwise, if the position is already occupied, the old controller is replaced if the new one has higher fitness.

Once the computation of the behaviour-performance map is complete, it is loaded onto the robot. When damage occurs, the robot uses Bayesian optimisation to select controllers to test in the real world, balancing exploration (verifying the performance of controllers with low confidence) and exploitation (verifying the performance of controllers with high expected performance). When each test concludes, the data gathered is used to update the confidence by lowering the uncertainty of the prediction and update the performance of the controller, as well as those of its neighbours in the behaviour space. Controllers with similar behaviour are assumed to perform similarly. Another controller is then chosen and evaluated. This process, illustrated in Figure 2.18, repeats until a controller evaluated on the real robot performs better than 90 per cent of behaviours in the behaviour-performance map.

IT&E has shown excellent results, allowing robots to recover from damage by rapidly

Algorithm 4: MAP-Elites Algorithm

Initialize behaviour space B

Initialize controller space C

for $i = 1 \rightarrow R$ **do**
    c = Random controller
    x = BehaviouralDescriptor(c)
    f = Fitness(c)
    if $B[x] = null \;||\; B[x] < f$ **then**
        $B[x] = f$
        $C[x] = c$
    end
end

for $i = 1 \rightarrow (n_t$ - $R)$ **do**
    c = RandomSelect(B)
    c' = RandomVariation(c)
    x = BehaviouralDescriptor(c')
    f = Fitness(c')
    if $B[x] = null \;||\; B[x] < f$ **then**
        $B[x] = f$
        $C[x] = c$
    end
end

return behaviour-performance map (B and C)

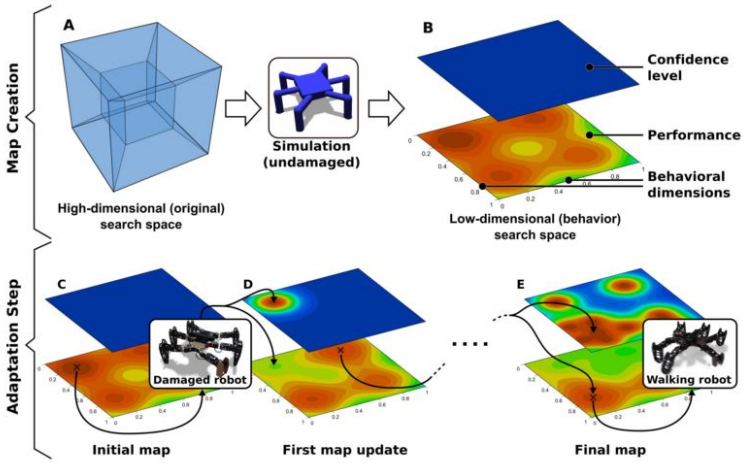


Figure 2.18: Behaviour evaluation in IT&E (Cully et al., 2015)

learning new gaits. The method also works on undamaged robots, improving their gaits to achieve higher performance.

The specific selection of behavioural characteristics to act as dimensions for the behaviour space was shown to have little effect on the algorithm's ability to recover from damage. This remained true even when the characteristics were chosen entirely randomly from the list of possible options.

In summary, IT&E established two concepts; the behaviour-performance map, positions controllers and their behaviours relative to others, and the intelligent selection of controllers for real-world testing.

2.9.5 Safety-Aware IT&E

A disadvantage of IT&E, identified by Papaspyros et al. (2016), is its lack of safety constraints. Without such constraints, the robot is not prevented from attempting potentially harmful behaviours; since IT&E focuses solely on the reward, it is likely to become damaged in the trial and error process. To this end, Papaspyros et al. proposed Safety-Aware Intelligent Trial-and-Error (sIT&E) which replaces the Bayesian Optimisation (BO) of IT&E with a constrained BO procedure.

sIT&E's behaviour space has one additional dimension for each safety constraint that should be applied to the process. sIT&E then optimises for the performance of the con-

troller, while guiding the search through safe areas of the search space.

sIT&E thus showed that additional dimensions can be added to the behaviour space to allow for guided optimisation based on a greater number of constraints.

2.9.6 Reset-Free Trial-and-Error

Reset-Free Trial-and-Error (RTE) is an RL-based method of damage recovery, inspired by IT&E (Chatzilygeroudis et al., 2016). Chatzilygeroudis et al. (2016) state that an ideal trial-and-error damage recovery algorithm should:

- not require reset after each real-world trial,
- scale well for complex robots, and
- explicitly take the environment into account, to prevent additional damage occurring as a result of the chosen real-world trials.

RTE addresses these issues using the MAP-Elites algorithm to populate a behaviour space with performance values to create a behaviour-performance map, as in IT&E (Chatzilygeroudis et al., 2016). This maps behaviours to performance, but can also be used to map actions to outcomes. Once the robot becomes damaged, this mapping is no longer correct but gives some approximation of the new, correct mapping. The mapping can then be corrected to reflect the real world. A total of n_d Gaussian processes (GPs) are used to create new predictions, where n_d is the number of dimensions in the behaviour space. The priors for these GPs are the mappings in the now incorrect behaviour-performance map.

Since GPs are probabilistic models, they produce not only predictions but also the uncertainty associated with each prediction. In RTE, this uncertainty is exploited by a Monte Carlo Tree Search (MCTS) to choose the next real-world test.

The RTE loop can be put simply as:

1. Select next best action
2. Execute action
3. Update Gaussian process

4. Repeat until the task is completed

RTE proposes the use of the behaviour-performance map with Gaussian processes that correct the incorrect predictions of the map to create new, more accurate predictions.

2.10 Conclusions

In nature, neural networks are found in the brains of animals and facilitate advanced processing tasks. Artificial neural networks are networks of artificial neurons, which imitate these biological neural networks. ANNs are able to learn complex tasks and have many real-world applications.

Another field of research inspired by nature is the field of evolutionary algorithms. These algorithms are based on the model of biological evolution and are able to evolve good solutions to problems, employing the concept of the *survival of the fittest*. An interesting application of EAs is in the evolution of ANNs, which presents an alternative to the traditional, gradient-based methods of ANN training.

Evolutionary robotics draws concepts from EAs and allows for the automatic evolution of robot controllers and morphologies. The field offers an appealing alternative to the complex task of manual controller creation, but has drawbacks of its own; the evolution process requires the real-world evaluation of huge numbers of controllers. It would be infeasible to attempt to perform these on a real-world robot. Therefore, simulators are used for the evaluation of controllers in the population. Small differences between the simulation and the real world may cause the evolution process to create controllers that perform well in simulation but are unable to transfer to the real world. This problem is known as the reality gap.

There have been many methods developed, attempting to bridge the reality gap. One such method is Bootstrapped Neuro-Simulation. BNS uses a simulator neural network, improving the efficiency and ease of construction of the simulator, but unlike traditional SNN development, trains the SNN and controller population concurrently. This means that the time-consuming data-gathering step usually involved with simulator development is no longer necessary.

As methods of robot development have advanced, the robots' proposed applications

have become increasingly complex. Robotic systems are now expected to operate in areas inaccessible and hazardous for humans, making it impossible for direct human intervention in the case of damage to the robot. Damage recovery for ER is, therefore, another area worthy of receiving a great deal of research effort.

Proposed methods of damage recovery have had excellent results, but none have been applied to BNS. It is likely, however, that the implementation of entirely new damage recovery functionality is not necessary for BNS; the algorithm is continuously gathering data from the real world and may, therefore, have the ability to recover from damage automatically. The recovery process may resemble the existing trial-and-error approaches as BNS selects promising controllers to evaluate in the real world.

Chapter 3

Experimental Methodology

3.1 Introduction

The literature review conducted in Chapter 2 established the groundwork upon which ER research, specifically into the use of BNS, can be conducted. While the use of BNS has shown great promise, the chapter showed that there are many aspects of its use that have yet to be investigated. One of these areas is damage recovery, the primary focus of this research.

This chapter discusses the investigations conducted in this research: damage recovery for simple robots, closed-loop controller evolution, damage recovery for closed-loop controllers, and damage recovery for complex robots. The guiding logic behind the choice of these investigations is discussed, as well as the approach for each investigation is discussed broadly and the contributions of this research.

Figure 3.1 shows a breakdown, putting each investigation and its processes in context within the scope of this research. Items with solid borders are original contributions, developed as a part of this research, while dashed borders represent items developed as parts of previous research by other researchers. The meaning of each area of the diagram is discussed further in its corresponding section, but it is important to understand that the nodes in Figure 3.1, item 1, define the environments wherein evaluation took place. These environments are the real world (RW), fake real world 1 (FRW 1), and fake real world 2 (FRW 2). Nodes of the corresponding shapes elsewhere in the diagram indicate the

environments used for the respective experiments. The concept of different environments is complex, but it is not necessary that it is understood at the outset; it will be discussed further in Section 3.2.3, once the necessary groundwork has been laid. The remainder of this chapter describes the individual investigations, as well as their motivation within the context of the research and ROs.

3.2 Investigation A: Damage recovery for simple robots

This investigation focused on addressing RO2 (determining BNS's damage-recovery capabilities) and RO3 (proposing and evaluating adaptations to improve the algorithm's damage recovery) by utilising two systems. The first was a system developed by Woodford et al. (2016) to execute BNS and evolve open-loop controllers for a differentially-steered Khepera III robot (Figure 3.1, item 2). The second was a newly implemented system able to simulate damage on the robot (Figure 3.1, item 3.1). These were used together to evaluate BNS's inherent damage recovery abilities as well as to propose and evaluate adaptations to improve the algorithm's ability to recover from damage (Fig 3.1, item 3).

3.2.1 Damage

Before RO2 was addressed and BNS's damage-recovery ability evaluated, a suitable problem needed to be selected. In a previous investigation, Woodford et al. (2016) implemented BNS to solve a navigation problem for a differentially-steered Khepera III robot (Figure 3.1, item 2). This system was augmented with functionality for the evaluation of BNS's damage recovery.

The system was used to execute the BNS algorithm and evolve controllers capable of completing the task. The robot was then damaged, and the algorithm's recovery observed. The algorithm's performance over time was expected to be similar to that shown in Figure 3.2, where the controllers' performance suddenly drops, but then slowly recovers. It was expected that the final performance would not always be able to reach the pre-damage levels, since the robot would no longer be functioning at its full potential.

It would have been infeasible to inflict real damage on the physical robot. Doing so would have meant replacing components on the robot after each experiment in the real

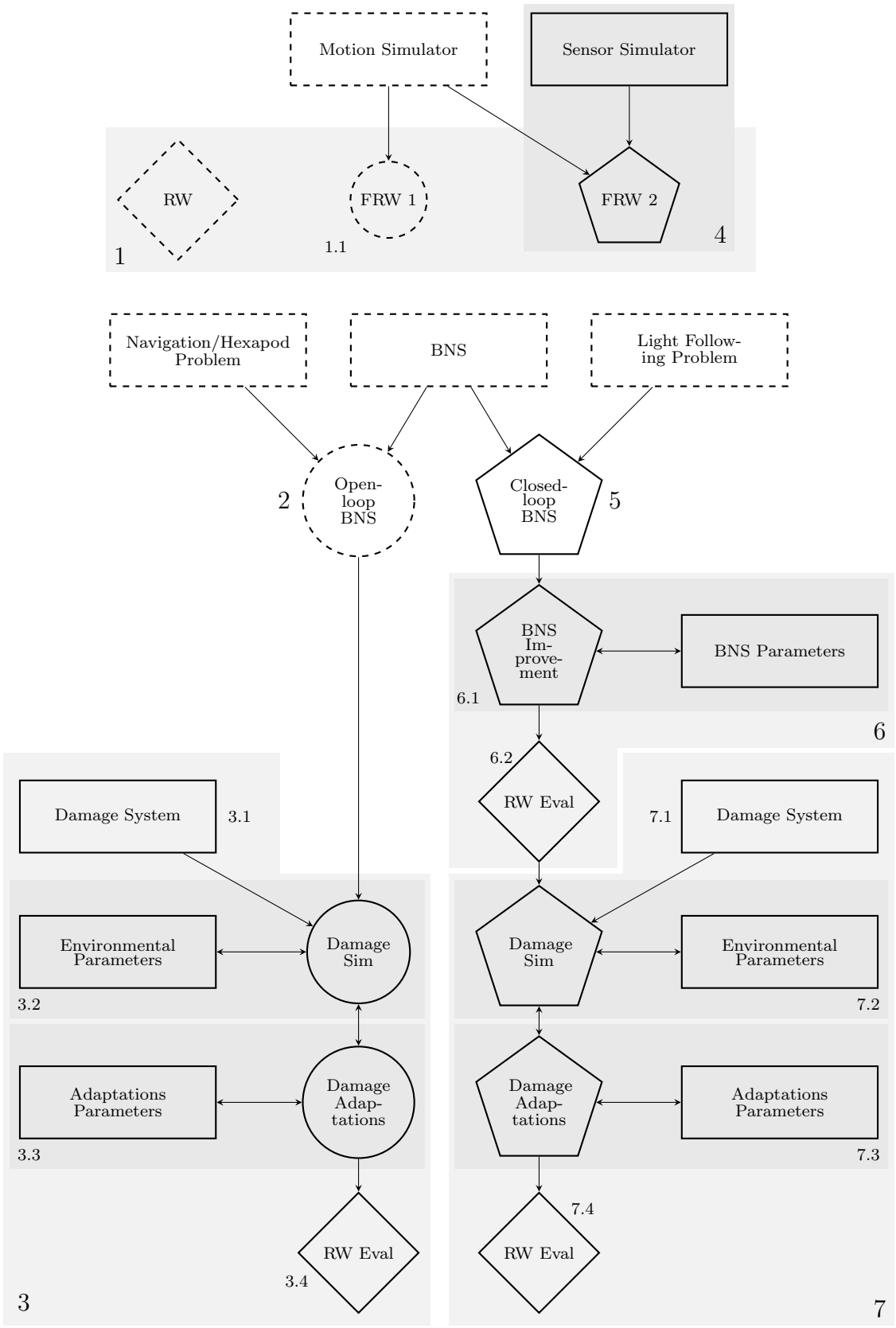


Figure 3.1: Research Structure

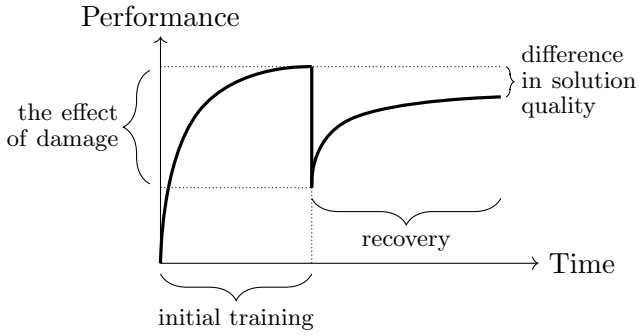


Figure 3.2: An example of the expected behaviour over time of BNS when recovering from damage

world. The damage was, therefore, achieved by implementing a damage system (Figure 3.1, item 3.1), which modified commands sent to the robot at the final step before their transmission, thereby changing the behaviour of the robot without informing the rest of the system.

The procedure for damage recovery evaluation was as follows:

1. Allow sufficient time for a controller to be evolved, capable of completing the given task (Figure 3.2, "initial training").

2. Simulate damage to the robot, affecting its ability to complete the task (Figure 3.2, "the effect of damage").

3. Observe the time taken for the system to recover and the quality of the post-recovery solution (Figure 3.2, "recovery and difference in solution quality").

When inflicting damage on the robot, four parameters controlled how the damage affected it. These included the time when damage was inflicted, the type of damage inflicted, the magnitude of the damage, and the problem being solved by the robot. The effects of these *environmental* parameters were evaluated to determine how different types of damage affect the robot (Figure 3.1, item 3.2).

3.2.2 Adaptations

It was hypothesised that the performance of BNS's damage recovery could be improved through the implementation of adaptations to the algorithm. The following three issues were anticipated to be adversely impacting performance:

- The convergence of the controller population may result in low diversity within the population. This may cause the algorithm to be unable to effectively search for new solutions once damage occurs.
- The existing, inaccurate simulator may be trapped in a local minimum and be unable to effectively adapt to the new state of the robot.
- After damage occurs, the system continues to train using the full set of gathered training data. This means that the simulator is trained on data which is incorrect and reflects the performance of the robot in its undamaged state, as opposed to its new damaged state, thereby misleading the SNN.

Four novel adaptations were proposed to address these issues. The adaptations constitute the main contribution of this research and were created as a part of these investigations for use with BNS. They were intended to be capable of being used for the evolution of controllers for different problems and robot morphologies. The evaluation of these adaptations (Figure 3.1, item 3.3) allowed RO3 to be addressed. The adaptations are discussed in the following subsections.

Adaptation 1: Reset the controller population

The *population reset* adaptation was proposed as an attempt to address the issue of low population diversity. When damage occurs, the population of controllers is randomly re-initialised. This causes an increase in diversity which may aid in the search for new controllers by enabling a higher degree of exploration.

Adaptation 2: Increase the mutation rate and magnitude

In EAs, a high mutation rate facilitates an increased rate of exploration (Engelbrecht, 2007). The same is true for a high mutation magnitude. It was, therefore, hypothesised that increasing the mutation rate or magnitude after damage has been applied may assist with the discovery of better solutions by promoting exploration of the search space. The algorithm was evaluated with mutation rates and magnitudes that increased after damage was applied, stayed constant, and combinations of the two.

Adaptation 3: Sliding window training

The sliding window adaptations aimed to address the issue of *old* training data. When using a sliding window of size x, the data used to train the SNN consists of the x newest training patterns, instead of the entire set of data. This approach is based on the concept of transfer learning since a trained SNN is continuously trained on *new* data sets. A major advantage held by this adaptation is that it requires no knowledge of the occurrence of damage. It can be used from the beginning of the execution of the BNS algorithm, whereas the others need to be used at the time that damage occurs.

Adaptation 4: Reset the simulator

The difference in a robot's behaviours before and after damage may be significant enough that re-initialising the SNN to a random state allows for a better simulator to be trained more efficiently than otherwise. Alternatively, it is possible that transfer learning may cause performance to improve if the simulator for the damaged robot is trained by starting with the existing simulator. It was unclear which of these options would lead to better results.

The set of training data remains the same before and after the simulator is reset since this adaptation is focused on the state of the simulator, not the training data. It is possible, however, for the simulator reset and training window adaptations to interact and together offer a more significant improvement than can be offered individually.

3.2.3 Parameter Evaluation

BNS allows for the rapid creation of both controllers and simulators, but the mutation and sliding window adaptations have many possible values for the mutation magnitude, mutation rate, and window size parameters. Additionally, the adaptations had the potential to interact which each other, and each needed to be evaluated with multiple combinations of the other parameters. This resulted in the number of configurations being too large for real-world evaluation to be feasible.

The solution to this problem was to create a *fake real world* in simulation (Figure 3.1, item 1.1). In a fake real world, the core of the BNS algorithm is unchanged but, instead

of gathering training data from a real-world robot, data is gathered from a fake-real-world robot. This fake-real-world robot was created using a pre-constructed robot simulator that did not form part of the BNS process. This pre-constructed simulator is called the *static* simulator, and the simulator being evolved by BNS is called the *dynamic* simulator. The static simulator for the Khepera's motion was created by Woodford et al. (2016) for previous research.

The fake real world allowed far more configurations of environment and adaptation parameter values to be evaluated than would have been possible in the real world (Figure 3.1 items 3.2 and 3.3, respectively). Once promising configurations were found, they were used for real-world experiments to confirm their ability to transfer to the real world (Figure 3.1, item 3.4).

It is important to note that the implementation of a fake real world and these parameter evaluations are not a necessary step in the implementation of the algorithm. They were conducted in these investigations in order to make recommendations with regard to the use of adaptations and their corresponding parameter values. Individuals using BNS to evolve controllers will not be required to perform this time-consuming step.

3.2.4 Results Analysis

The results from this investigation can be found in Chapter 4. The ability to perform experiments in simulation allowed each configuration to be evaluated thirty times. These results were then used to gain statistically significant insights into the algorithm's performance.

The Mann-Whitney U Test is a nonparametric test with a null hypothesis stating that two populations are identical, given samples drawn from each population (Wackerly, Mendenhall, and Scheaffer, 2008). The calculation of the Mann-Whitney U statistic can be found in Appendix A. This test allows the results of evaluations of different parameter configurations to be compared to each other. If the difference between the results is significant, then the change in configuration can be said to have significantly affected the performance of the algorithm.

Only the performance after damage has occurred is of interest since this research is focused explicitly on damage recovery. Therefore, observations are created for each

simulated execution of BNS by calculating the sum of performance values from the real-world evaluation when damage occurs until the end of the execution:

$$O_\tau = \sum_{i=n_b}^{n_r} f_\tau(i) \quad \tau = 1, 2..30 \tag{3.1}$$

where O_τ is the observation for the BNS execution τ, n_b is the fake-real-world evaluation at which damage occurs, n_r is the total number of fake-real-world evaluations carried out per execution, and $f_\tau(i)$ is the best controller's fitness at fake-real-world evaluation i in execution τ. Since each configuration is used for thirty simulated executions, 30 O_τ values will be obtained for each configuration. These can then be used to determine whether each configuration causes the results to be significantly different from another. Comparing only the final performance value of the fake-real-world evaluations would mean that information about the rate at which the system recovers would be ignored, whereas using the sum of performance values after damage occurs allows the rate of recovery as well as the final performance value to be taken into account. This can be seen in Figure 3.3 where lines a and b show performance recovering to the same final value, but at different rates.

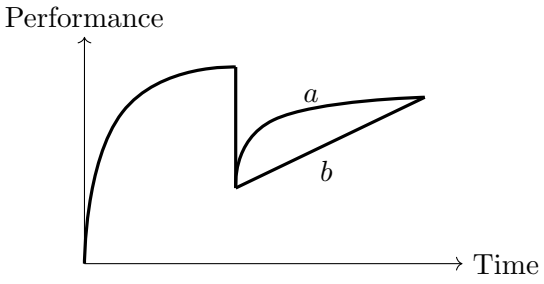


Figure 3.3: An example of differing rates of damage recovery

The diversity within the population of individuals was also calculated. Plots of the population's diversity over time are included where they are able to enrich the discussion of the results and provide insight into the cause of a configuration's effects on algorithm performance.

3.3 Investigation B: Closed-loop controller evolution

The evolution of open-loop controllers is a topic that has been researched since the creation of the BNS algorithm (Woodford et al., 2016, 2017) and continues to be the focus of a great deal of research. As a result, more is known about best practices for the evolution of open-loop controllers than for closed-loop, which have received no prior research in the context of BNS. Investigation B focuses on BNS's ability to evolve such closed-loop controllers, thereby addressing RO4: the proposal, implementation and evaluation of a method of evolving closed-loop controllers with BNS. Additionally, since this investigation was the first to do so, experiments were conducted in order to optimise BNS and make recommendations for parameter values and adaptations when evolving closed-loop controllers (Figure 3.1, item 6).

Since the objective of this investigation was not to construct a new problem for BNS to solve, but instead to evaluate its ability to evolve a closed-loop controller, a classic light-following ER problem, which has also been solved using a standard SNN (Pretorius et al., 2013), was chosen. A light source was placed in the middle of the testing area; the objective was to evolve controllers able to navigate the robot towards the light source by using sensors on the robot. The fake real world used to execute BNS (Figure 3.1, item 5) was a version of that used in investigation A, modified for the evolution of closed-loop controllers.

3.3.1 Proposed Adaptations

Given that the evolution of closed-loop controllers with BNS is a topic that has received no prior attention, a parameter study was carried out, and the effects of four adaptations to BNS were evaluated (Figure 3.1, item 6.1). The intention was to create a configuration of BNS optimised specifically for the evolution of closed-loop controllers.

A lack of diversity in the population is a concern when evolving controllers using BNS. If the population of controllers converges before the simulator is able to achieve a sufficient level of accuracy, the evolved controllers may become too specialised for the nascent simulator and be unable to adapt as the simulator is updated. Headless Chicken mutation, intermittent population reset, and Island EAs were proposed as adaptations to

address this issue. Similarly, if the simulator can be trained more rapidly, the EA will have access to a more accurate simulator earlier in the evolution process. Data augmentation aims to increase the rate at which the simulator improves. These adaptations are discussed further in this section.

Adaptation B1: Headless Chicken mutation

The Headless Chicken algorithm works during the reproduction stage of the EA (Engelbrecht, 2007). After the first individual is selected for reproduction there is a probability of p_h that, instead of the second parent also being selected from the population, it is a randomly generated chromosome. p_h is the *Headless Chicken probability*. This adaptation serves the same purpose as a mutation operator by introducing random variation into the population, though in this case the magnitude of the variation is much greater.

Adaptation B2: Intermittent population reset

Using BNS, the execution of controllers on the real-world robot is much slower than the process of evolving controllers. This means that for every real-world evaluation, several generations of controller evolution take place which may cause the population to prematurely converge to a solution which performs well only in the new simulator. It may, therefore, be beneficial to reset the population of controllers intermittently, in a process called *intermittent population reset*.

However, if the entire population is reset, there is a reduced benefit to BNS's use over a standard SNN. This is because immediately after each reset a new population is being trained on an existing simulator, which closely mirrors the standard method of non-concurrent controller evolution using SNNs. A number of individuals can be transferred to the new population, while all others are randomised in order to maintain a small amount of information from before the reset. This process resembles elitism, where the best individuals are kept between generations of an EA (Engelbrecht, 2007). These individuals are, therefore, referred to as *Reset-Elitists* (REs).

Adaptation B3: Island EA

Another method of maintaining diversity in the population of an EA is the use of an Island EA (Engelbrecht, 2007). Instead of a single population, multiple populations each evolve independently, promoting the discovery of multiple solutions. Individuals are then allowed to migrate between islands periodically, based on a migration policy.

The migration policy determines:

- how often migration occurs,
- how many individuals migrate,
- how individuals are selected for migration,
- how individuals are selected for replacement in the destination population, and
- between which islands individuals migrate.

Adaptation B4: Data augmentation

Robots such as the Khepera III appear to be symmetrical along at least one axis. It is, therefore, reasonable to assume that for each training pattern, another can be generated through the reflection of the original pattern. This process is known as data augmentation. Using this process to generate additional training patterns may allow the simulator to improve more rapidly due to the larger set of training patterns. It is also possible, however, that the assumption of symmetry is false, which may negatively affect performance.

3.3.2 Sensor Simulator

As in investigation A, the number of possible configurations was too high for a parameter study to be conducted in the real world. The solution was, once again, the use of a fake real world, allowing a large number of configurations to be evaluated and optimal configurations found (Figure 3.1, item 6.1). Unlike the previous investigation, not only the motion of the robot was required to be simulated but also the robot's sensors' readings, so that the simulated robot is able to use the readings to make decisions as it would in

the real world. This is the approach that was used in previous research by Pretorius et al. (2013).

The sensor simulator was combined with the motion simulator used in investigation A to create a new fake real world consisting of two simulators (Figure 3.1, item 4). This new fake real world allowed the general behaviour of the controllers to be determined and promising parameter configurations to be used in real-world evaluations (Figure 3.1, item 6.2).

3.3.3 Results Analysis

Once again, the Mann-Whitney U Test was used to obtain significance values for each of the evaluated parameter configurations. Unlike the other investigations presented here, where observations were calculated with the sum of performance values *after* damage had occurred, in this investigation the values were added from the first real-world evaluation since damage was not inflicted in this investigation. The simulated results informed the choice of configurations for evaluation in the real world (Figure 3.1, item 6.2). Additionally, the population diversity results are once again included where relevant.

This investigation was the only one in this research that did not focus directly on damage recovery. It was, however, a worthwhile investigation which was able to shed light on BNS's application to the evolution of more sophisticated robot controllers. The results of the investigation can be found in Chapter 5.

3.4 Investigation C: Damage recovery for closed-loop controllers

Once RO4 was addressed, the system was further augmented with a damage system to allow for the evaluation of damage recovery in order to address RO5: the proposal, implementation, and evaluation of damage recovery adaptations for closed-loop controllers (Figure 3.1, item 7.1). The same problem as investigation B was solved; therefore, this investigation's implementation consisted of only the damage system.

3.4.1 Damage

In investigation A, the types of damage inflicted on the robot were limited to simple damage to the robot's wheels. This was due to the open-loop nature of the controllers; more complex damage could not have been compensated for and damage to the robot's sensors would not have had any effect since they were not used. In this investigation, the types of damage that the system could recover from were much more varied. The sensors could be damaged by increasing or decreasing their readings, adding additional noise, or turning one of them off entirely.

3.4.2 Parameter Evaluation

As in investigation A, environmental parameters are those that control specific aspects of the damage that is inflicted on the robot, such as the time that damage is inflicted, the type of damage inflicted, and the magnitude of the damage. A parameter study was conducted to determine the effects of each of these parameters on BNS's ability to recover from damage (Figure 3.1, item 7.2). Additionally, the adaptations and parameters from investigation B were re-evaluated in order to determine their impact on damage-recovery performance.

Once the effects of the environment and algorithm parameters were determined, the standard BNS damage adaptations, as were evaluated in investigation A, were evaluated for damage to a robot using a closed-loop controller (Figure 3.1, item 7.3). High performing parameter configurations were then transferred to a real-world robot to confirm their transferability (Figure 3.1, item 7.4). Statistical observations were obtained using equation (3.1). These results can be found in Chapter 6.

3.4.3 Results Analysis

As in the previous investigations, equation (3.1) was used to obtain observations regarding the performance of the configurations. These were then compared using the Mann-Whitney U Test.

3.5 Investigation D: Damage recovery for complex robots

The next step in the evaluation of BNS's damage recovery ability was an evaluation on a robot with a more complex morphology (RO6). The process followed for this investigation was very similar to that for investigation A and, as such, it shares the same elements of the diagram as investigation A. Once again, an existing BNS system, created to evolve hexapod controllers, was used as a starting point. (Figure 3.1, item 2). The system used BNS to evolve hexapod robot controllers with the goal of moving as far as possible in any direction.

3.5.1 Damage

A new damage system was implemented and integrated with the Hexapod BNS system (Figure 3.1, item 2). This damage system was more complex than those used in previous experiments, due to the number of possible points of failure on a six-legged robot. The effects of each type of damage on the algorithm's performance were determined before any investigations into damage recovery were performed. This process formed a part of the parameter investigation which evaluated the effects of different environmental parameters, such as the type of damage (Figure 3.1, item 3.2). The results for this investigation can be found in Chapter 7.

3.5.2 Parameter Evaluation

The effects of the previously evaluated BNS damage recovery adaptations were evaluated on the hexapod robot (Figure 3.1, item 3.3). Additionally, two variations on each of these were evaluated, individually and in combination with the others, due to the increased complexity of the hexapod robot:

- The *Complete Restart* adaptation causes the BNS algorithm to restart from a randomly initialised state with a new population of controllers, no training data, and a new SNN when damage occurs.
- The *Training Data Reset* adaptation discards all training data when damage occurs. This adaptation is based on the same concepts as a sliding window, but is at a

disadvantage since it requires knowledge of the occurrence of damage.

The hexapod robot is very complex. This causes the effects of damage to the robot to be much more complex than the effects of damage on the Khepera robot, meaning that the differences in behaviour between the undamaged and damaged robot may be much bigger than in the case of the Khepera. This, in turn, implies that the pre-damage training data may be even more detrimental in the case of the hexapod than it was for the Khepera robot. These two new adaptations aim to address this issue. Ideally, continuing to evolve controllers from the point when damage occurs using the existing population would offer an increased rate of recovery. The robot's complexity means that this may not be the case. The performance of restarting the algorithm entirely was evaluated since, if it performed better, there would be no reason to recommend any adaptations; the algorithm could simply be restarted and allowed to evolve new controllers. A training data reset serves a similar purpose, and is likely to work for the same reason as the sliding window; it discards old, invalid training patterns.

Woodford (2018) found that, for complex robots, resetting the dynamic simulator intermittently throughout the BNS process brought about a significant increase in performance. Therefore, in this investigation, unlike in the previous ones, the simulator is reset once every twenty real-world evaluations. While these resets are likely to form part of all BNS implementations for complex robots, it is important to determine their impact on damage recovery specifically. Therefore, in general, experiments used configurations where the simulator was reset once every twenty real-world evaluations, but one set of experiments was run without the simulator resets to establish their effects on the algorithm's performance.

As in the previous four investigations, a fake real world was constructed by Woodford (2018) (Figure 3.1, item 1.1). The fake real world used for the hexapod is a different one to that which was used for investigation A, but they share an element in the diagram as they were constructed in the same way: from only a motion simulator.

Finally, promising parameter configurations were used in real-world evaluations (Figure 3.1, item 3.4). This once again served to demonstrate the ability of the configurations to transfer effectively to reality.

3.6 Conclusion

In this chapter, the four investigations that comprise this research were presented. Investigation A aimed to evaluate BNS's damage recovery for open-loop controllers evolved to complete a task on a Khepera III robot. The population reset, mutation rate and magnitude changes, sliding windows, and simulator reset adaptations were proposed as improvements to the algorithm. Investigation B aimed to evolve closed-loop controllers to complete a light-following task; the robot was required to use its light sensors to navigate towards a light source and remain near it for the duration of its execution. Four new adaptations to the algorithm were proposed for the evolution of closed-loop controllers (Headless Chicken mutation, intermittent population resets, Island EAs, and data augmentation). Investigation C then used the implementations created for investigation B to evaluate damage recovery for closed-loop controllers. Finally, investigation D focused on damage recovery for complex hexapod robots with open-loop controllers. Two final adaptations were proposed for the hexapod due to its increased complexity (complete restart and training data reset).

The use of a fake real world for parameter evaluations was discussed. The number of parameter configurations evaluated is too great for them to be carried out in the real world. The evaluations are therefore performed using a fake real world to gather data when the real world would usually be used. The presentation and analysis of the results were also discussed; since each configuration could be evaluated a large number of times, the Mann-Whitney U test was used with the large amount of data to obtain the significance of the differences among the performance of different configurations.

Chapter 4

Damage Recovery for Simple Robots

4.1 Introduction

This chapter presents the results of investigation A: damage recovery for simple robots. BNS was used to evolve controllers capable of completing a simple navigation task using a differentially-steered robot. The robot was then damaged, and BNS was shown to possess the ability to recover from damage automatically, without the need for specialised damage detection systems. Additionally, adaptations were proposed and evaluated for their ability to improve this damage recovery further. The evaluations were carried out both in the fake real world (FRW1), and in the real world on a physical robot. Figure 4.1, item 3 shows the steps followed in conducting the research for this investigation.

Section 4.2 discusses the implementation details for the experiments, Section 4.3 presents and discusses the results of parameter evaluations in FRW1, Section 4.4 shows the results of real-world evaluations, and Section 4.5 draws conclusions and makes proposals for potential future work.

4.2 Implementation Details

This section discusses the implementation details of the investigation. The result of this implementation is represented by item 2 in Figure 4.1. Section 4.2.1 discusses the Khep-

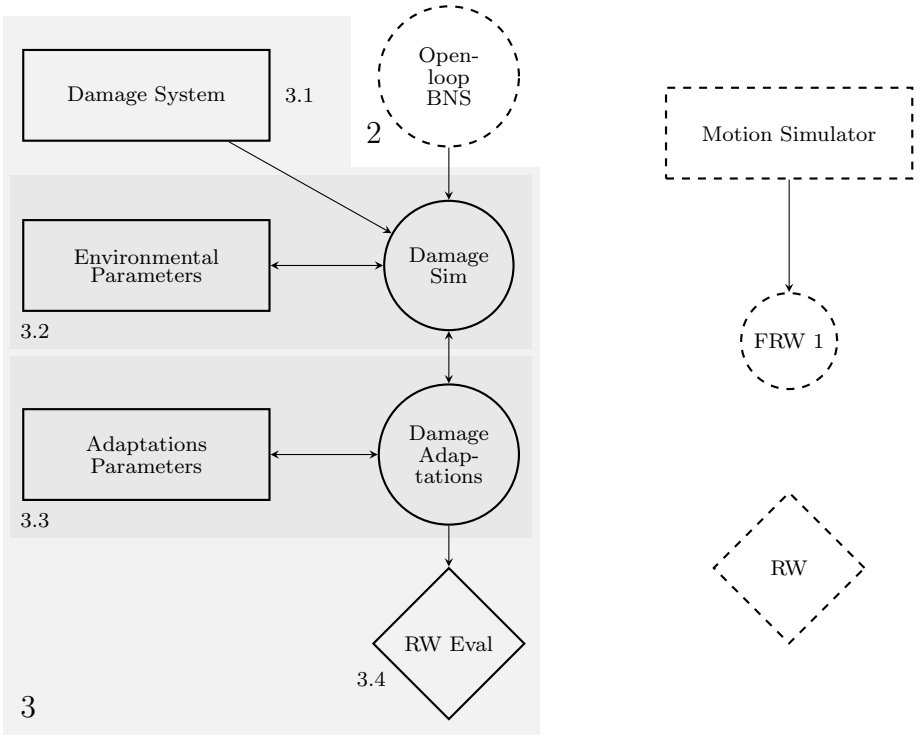


Figure 4.1: Investigation A focus

era III, the robot on which the investigation was carried out. Sections 4.2.2 and 4.2.3 discuss the problems that were solved by the system and the damage inflicted, respectively. Section 4.2.4 then discusses the controllers evolved to solve the problem. Section 4.2.5 discusses the implementation of the EA used by the BNS system. Section 4.2.6 discusses the environment in which the real-world evaluations were carried out. Section 4.2.7 discusses the implementation of a motion simulator for the Khepera III, while Section 4.2.8 provides details of the parameter evaluation.

4.2.1 Khepera III

The Khepera III robot (Figure 4.2) is used frequently in ER research (Floreano and Mondada, 1994; Koos et al., 2013a; Woodford et al., 2016). The robot is differentially-steered, meaning that it steers by varying the speeds of its two wheels. It also has an array of sensors, including ambient light sensors, ultrasonic sensors, and infrared sensors. BNS has already been used in previous research for the evolution of simulators and controllers for the robot (Woodford et al., 2016), which allowed this investigation to focus solely on the

damage recovery performance of the algorithm.

Figure 4.2: The Khepera III Robot (K-team Corporation)

4.2.2 Problems

Two problems were solved by BNS in this investigation. The first problem, the *simple* problem, required the robot to move through four *objectives*, shown in Figure 4.3. The robot started at position E and had to navigate the objectives in the order A, B, C, D, A.

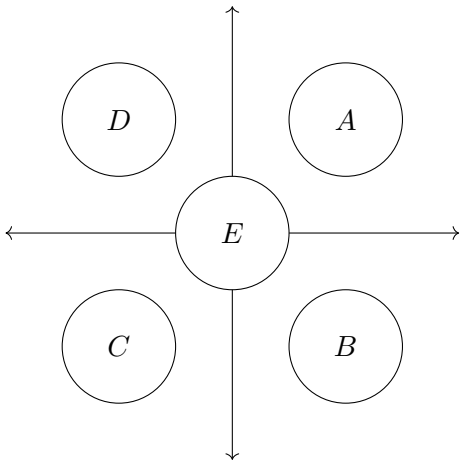


Figure 4.3: Problem Layout

The second problem was called the *infinity* problem because the shape traced by controllers successful in solving the problem resembles an ∞ symbol. In this case, the robot also started in position E (Figure 4.3) but had to navigate the objectives in the order A, B, E, D, C, E, A.

BNS was evaluated on each of these problems with varying levels of damage, different

types of damage, and different times before damage infliction.

4.2.3 Damage

The system responsible for inflicting damage on the robot is represented by item 3.1 in Figure 4.1. The system does not inflict physical damage to the robot, but rather simulates the damage in such a way that other parts of the system are unaware of its interference and experience the simulated damage as they would real damage.

The damage inflicted in this investigation was in the form of an under-performing wheel; one of the robot's wheels was slowed by a chosen percentage. BNS then needed to adapt to this by, for example, increasing the magnitude of all commands to that wheel.

4.2.4 Controllers

Each controller used in this investigation was represented by a chromosome of variable length. These chromosomes encoded the properties of the controllers, formed a part of the EA, and were able to be decoded into a controller. Each chromosome encoded the controller's multiple commands, which were made up of three values: the left motor speed, right motor speed, and duration of the command. The controllers were open-loop and executed the commands in sequence, from beginning to end.

The fitness of a controller is calculated based on the path followed by the robot during the execution of the controller. For the given problems, the goal is to minimise both the total distance travelled and the number of commands, while reaching as many objectives as possible. At the end of the execution of each command, the squared Euclidean distance from the robot to the current active objective is summed. An objective is *reached* when the robot ends a command within a minimum radius of the objective. When this happens, the next objective becomes active. After the controller's execution is complete, the final sum of these distances to the objectives is added to the product of a penalty value and the number of unreached objectives. Pseudocode for the calculation of the fitness values is shown in Algorithm 5.

Algorithm 5: Fitness calculation

```
objectives = list of objectives
path = list of positions reached by robot
sum = 0
penalty = 10000
minRadius = 9
activeObjective = 0
for position p in path do
    d = distanceToObjective(activeObjective)
    if d < minRadius then
        activeObjective++
    end
    sum += d
end
fitness = sum + (totalObjectives - activeObjective) * penalty
return fitness
```

4.2.5 Evolutionary Algorithm

The EA for this investigation was implemented using single-point crossover. Gene mutation was performed by probabilistically modifying values based on a uniformly distributed random variable. The magnitude and probability of this mutation was a parameter to the algorithm. Various values were evaluated as a part of this investigation 4.1. These choices of values were based on previous research investigating the same basic navigation problems by Woodford et al. (2016).

In order to measure the change in the diversity of the population over time, a method of measuring this diversity is required. Before calculating the diversity of the population, the population's *average individual*, $\bar{c}$, was calculated. This was done by finding the average value for each gene in the individuals' chromosomes which was, in turn, done by finding the mean of all values for the genes. An average individual was then created from this average chromosome. The diversity of the population, d_p, was subsequently calculated using equation (4.1):

$$d_p = \sum_{i=1}^{n_p} \sum_{j=1}^{n_g} |c_{ij} - \bar{c}_j| \tag{4.1}$$

where c_{ij} and $\bar{c}_j$ are the j^{th} genes of the i^{th} and average individual respectively, n_p is the number of individuals in the population, and n_g is the number of genes in each individual.

4.2.6 Testing Area

A ceiling-mounted camera faced down towards the robot which had coloured tracking markers attached to its top for positional tracking of the robot in a 2.7m×1.8m area on a skid-resistant white board. The tracking system had been implemented by Woodford et al. (2015) for previous ER research.

4.2.7 Khepera Motion Simulator

The Khepera motion simulator was implemented for use in prior research by Woodford et al. (2016). The SNN consists of three ANNs, each simulating either the robot's change in x (Δx), y (Δy), or rotation ($\Delta\theta$). Each ANN has five input neurons (Figure 4.4): the previous, and new, left and right motor speeds, as well as the duration of the robot's new command. They each have twenty hidden neurons, a number determined to be effective in previous research (Pretorius, 2010).

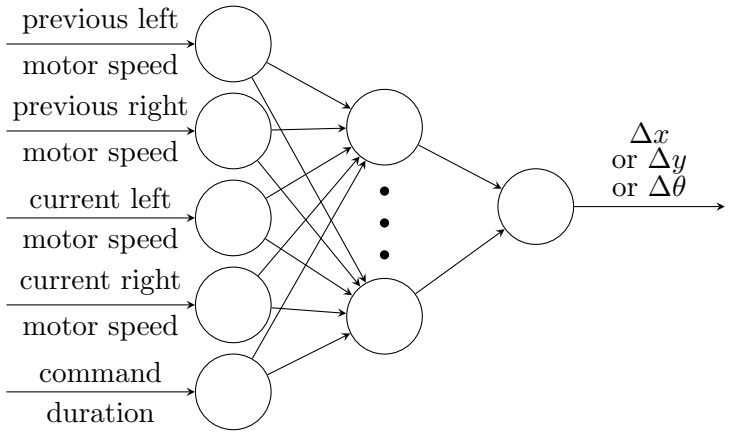


Figure 4.4: Khepera Motion SNN (Woodford et al., 2016)

4.2.8 Parameter Evaluation

The values of the adaptation parameters evaluated in simulation are shown in Table 4.1 (Figure 4.1, item 3.3). Additionally, a number of environment parameter values (Table 4.2) were used during evaluation to evaluate BNS's performance under differing conditions (Figure 4.1, item 3.2). These parameters specified the real-world generation at which the damage was applied, which of the robot's wheels was slowed, the percentage by which the robot's wheel was slowed, and the problem being solved. It was observed that, in general, the problems were able to be solved with fewer than twenty-five real-world evaluations. Thus, the real-world evaluations at which damage was applied were all chosen to be greater than this number. All combinations of these parameters were evaluated; in total, 448 discrete parameter configurations were evaluated.

Table 4.1: Adaptation parameter values

Parameter	Values
Sliding window size	300, 150, none
Reset controller population	True, False
Reset simulator	True, False
Mutation rate	0.1, 0.2
Mutation magnitude	2000, 3500

Table 4.2: Environment parameter values

Parameter	Values
RW evaluation at which damage applied	30, 40, 50
Slowed wheel	Left, Right
Damage percentage	20%, 50%
Problem	Simple, Infinity

Each parameter configuration was used for thirty simulated experiments. For each of these experiments, the fitness of the fittest individual was saved at each real-world evaluation, which resulted in 100 fitness values per experiment. The fitness values of all thirty experiments were then averaged at each real-world experiment to obtain 100 *average* fitness values for each configuration. These values represent the average fitness over time of the fittest individuals from each of the thirty experiments and are the values presented in this investigation.

4.2.9 Real World

After the completion of the parameter study, high-performing parameter configurations were identified. These configurations were used in real-world evaluations and compared to real-world evaluations without the adaptations. Previous research has shown that results obtained from BNS in simulation transfer well to the real world (Woodford et al., 2016, 2017). The goals of the real-world experiments conducted in this investigation are, therefore, to show that this transferability is maintained when BNS experiences damage and when the proposed adaptations are implemented in the real world. These evaluations were performed on the Khepera III robot, and their results are shown in Section 4.4.

4.3 Results and Discussion

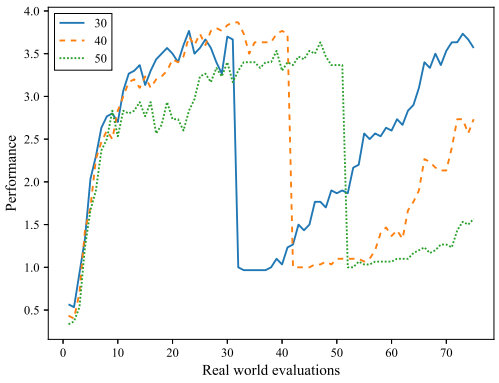
In this section, the results of the experiments are presented. First, in Section 4.3.1, the performance of BNS, in simulation, without changes is shown. Sections 4.3.2 to 4.3.5 show the simulated results of BNS with the adaptations implemented. The vertical and horizontal axes of the figures in these sections represent the average performance of the controllers based on the followed path, and the real-world evaluations, respectively. The specific environmental configurations shown were chosen, as their results were seen as representative of the performance of the algorithm in general since too many results were obtained for every configuration to be presented.

4.3.1 No Adaptations

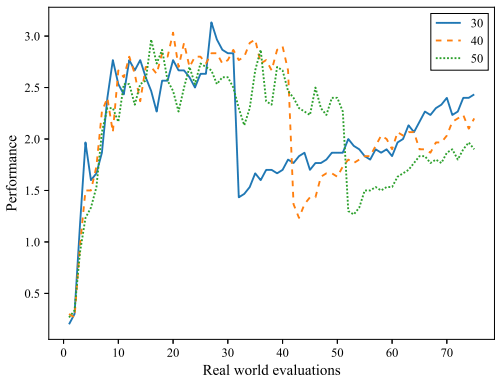
Figures 4.5a, 4.5b, 4.5c, and 4.5d each show the mean performance of thirty BNS evaluations with no adaptations.

In Figure 4.5a, the effect of inflicting damage on the robot at different real-world evaluations is shown. In this case, damage was inflicted by slowing the left wheel by 20 percent. The damage caused a very clear decrease in performance at 30, 40, and 50 real-world evaluations in Figures 4.5a and 4.5b. After a short delay, each of the lines begins to slope upwards with similar gradients, showing that each recovered at a similar rate.

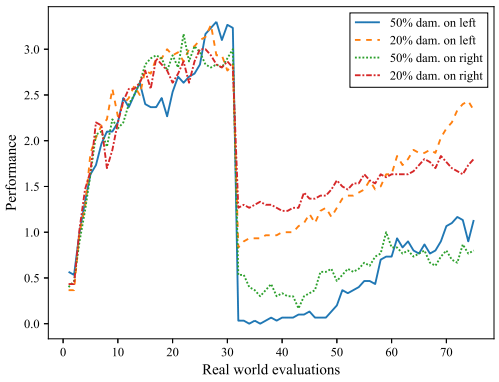
Figure 4.5c shows the effects of the application of different combinations of damage types on a system solving the basic problem. Damage was applied at thirty real-world



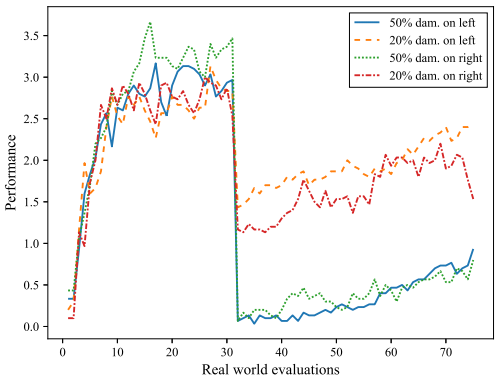
(a) Simple problem with damage inflicted at different numbers of real-world evaluations



(b) Infinity problem with damage inflicted at different numbers of real-world evaluations



(c) Simple problem with different damage types inflicted



(d) Infinity problem with different damage types inflicted

Figure 4.5: Performance over time for BNS recovering from damage while solving different problems

evaluations. It is clear that more damage causes a larger negative effect on performance; slowing the wheels by 50 percent caused the performance to drop significantly more than 20 percent.

Finally, Figure 4.5d shows the effects of an evaluation with the same parameter values as Figure 4.5c, but for a system solving the infinity problem. The results are similar to those for the simple problem, with more damage causing a larger drop in performance.

There is variation in each plot even before damage, due to the probabilistic nature of the BNS process. It is, however, clear from these results that BNS has the ability to recover from damage even before the implementation of any adaptations.

The diversity of the controller population decreases as more evaluations are run. It is, therefore, interesting that varying the time when damage was inflicted had minimal impact on BNS's rate of damage recovery. This implies that the diversity of the population and simulator do not play a prominent role in the ability of BNS to recover from damage.

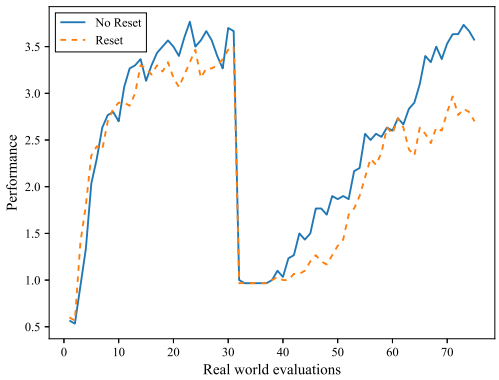
Figures 4.5c and 4.5d show that while the algorithm always recovered, or at least showed improvement, from all types of damage, it took significantly longer when faced with specific types of damage. The task of recovering from these damage types likely presented a more complex problem than others and was thus slower. Damage to the right wheel was a type of damage from which the system recovered slowly; when damaging the right wheel on a right-turning path, the robot entered a spiralling behaviour, which was likely a more complex behaviour from which to recover than the straight line followed by robots with newly damaged left wheels. This behaviour is demonstrated clearly in the real world results presented later in this chapter.

These results show that the changes in performance caused by the use of different parameter configurations are similar in magnitude regardless of the problem type. Therefore, in the interest of presenting as many useful results as possible, unless explicitly stated, all further results were obtained in the context of 20 percent damage application to the left wheel at thirty real-world evaluations.

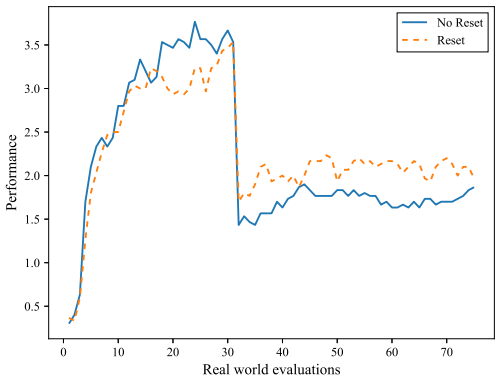
4.3.2 Controller Population Reset

Figure 4.6 shows the results of resetting the controller population after damage occurs for four different experiment configurations. This adaptation did not have a significantly

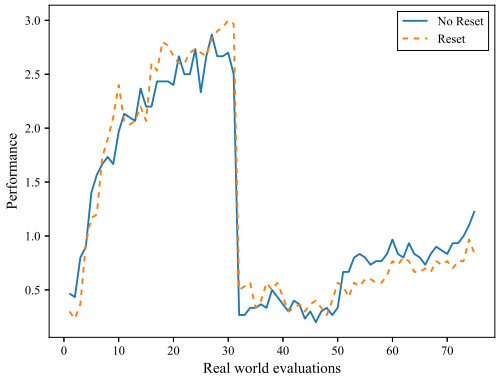
positive impact on performance. In Figure 4.6a it can be seen that in the case of 20 percent damage to the left wheel, it caused a significant ($p < 0.05$) decrease in performance.



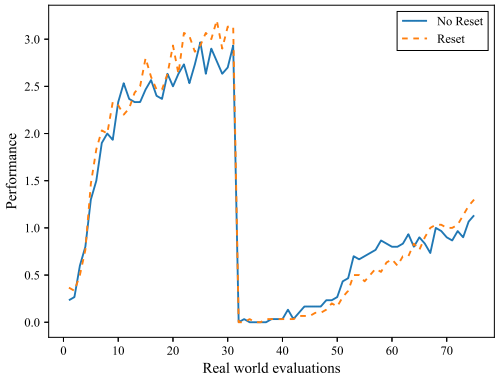
(a) 20% damage inflicted to left wheel



(b) 20% damage inflicted to right wheel



(c) 50% damage inflicted to left wheel



(d) 50% damage inflicted to the right wheel

Figure 4.6: Performance over time with the controller population reset adaptation

The results show that, at best, resetting the controller population caused no change in the performance of the algorithm and, at worst, caused the performance to worsen. The reason may be that the locations of solutions to the problem in the search space for the undamaged robot were near those for the damaged robot. The proximity meant that these solutions were able to be found effectively through mutation of the existing population, whereas resetting the population and restarting evolution caused the promising solutions to be lost.

It is possible that a problem that requires radically different solutions for an undamaged robot versus a damaged one may benefit from a population reset, but that was not the

case for the problems investigated here. However, in that case, it would also be expected that Figure 4.6d would show an improvement with a population reset since it presents the problem in which damage had the most considerable effect.

Figure 4.7 shows an example of the diversity over time of the population when it is reset after damage. The population reset does not cause a significant increase in diversity in the population at the real-world evaluation after damage. Since multiple generations of controller evolution happen for every real-world evaluation, the population was able to converge to pre-reset levels of diversity before the next real-world evaluation occurred, causing there to be no perceptible increase in diversity. This rapid convergence is likely the same reason that this adaptation was unable to offer any performance benefits.

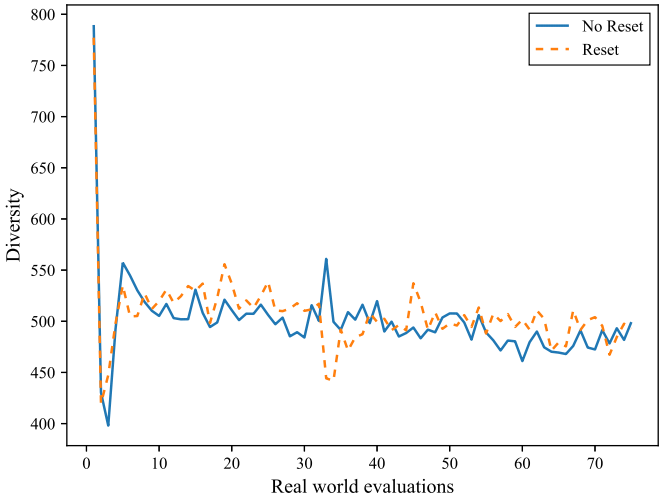
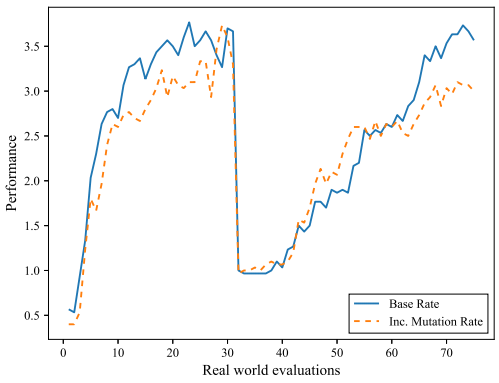


Figure 4.7: Population diversity over time with the controller population reset adaptation

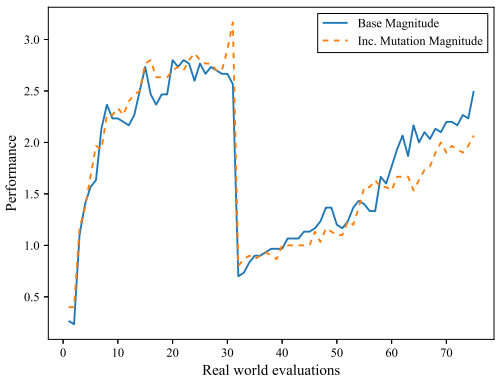
4.3.3 Mutation Rate and Magnitude Increase

Neither an increase in the mutation rate nor the magnitude had a significant positive impact on the ability of BNS to recover from damage. Figure 4.8 shows the results of a change to each parameter individually (Figures 4.8a and 4.8b), as well as the results of their interaction when increased simultaneously (Figure 4.8c).

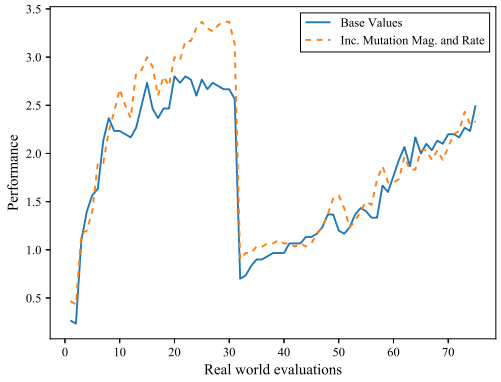
In all evaluated cases, the changes in the parameter values caused small decreases in overall performance. In the case of changes to the mutation magnitude, the likely



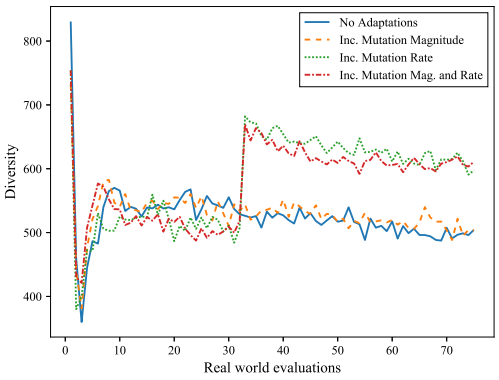
(a) Performance over time with an increased mutation rate



(b) Performance over time with an increased mutation magnitude



(c) Performance over time with an increased mutation rate and magnitude



(d) Diversity over time for different post-damage mutation configurations

Figure 4.8: The effects of mutation rate and magnitude modifications on the simple problem

reason for this is that the high mutation prevented effective convergence of the population (Figure 4.8d). Since the population was unable to converge effectively, it was unable to refine potential good solutions sufficiently.

4.3.4 Sliding Window Training

Adding a sliding window to the data used for training had the largest positive impact of all evaluated adaptations. A selection of the results of the evaluation of the adaptation on the simple problem is shown in Figure 4.9. The plots displayed each show the results of the use of no window, a window of size 150, and a window of size 300. The results shown are for 20 percent damage to the left wheel (Figure 4.9a), 20 percent damage to the right wheel (Figure 4.9b), 50 percent damage to the left wheel (Figure 4.9c), and 50 percent damage to the right wheel (Figure 4.9d). The corresponding significance values are presented in Table 4.3.

Table 4.3: p-values for Figure 4.9

(a) p-values for Figure 4.9a

	None	**150**
150	0.00000	
300	0.00001	0.05543

(b) p-values for Figure 4.9b

	None	**150**
150	0.00000	
300	0.00000	0.15146

(c) p-values for Figure 4.9c

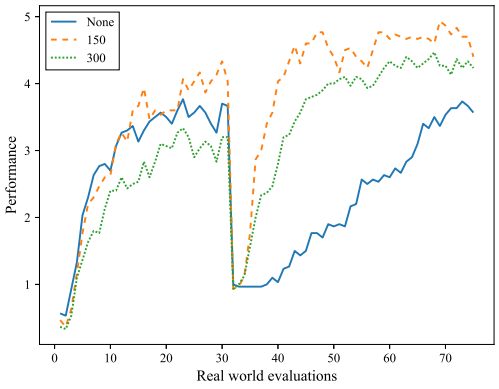
	None	**150**
150	0.00000	
300	0.00530	0.00760

(d) p-values for Figure 4.9d

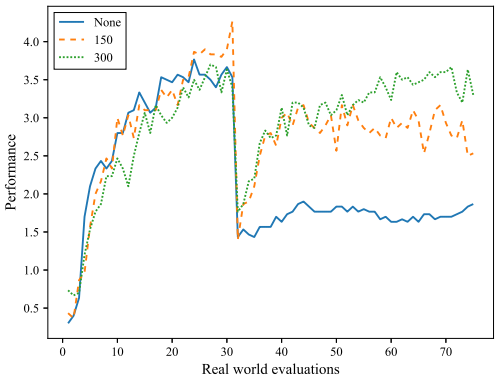
	None	**150**
150	0.00000	
300	0.00000	0.44161

The rate of damage recovery showed a significant ($p < 0.01$) improvement in all cases when a sliding window was used. The choice of window size may be problem-dependent; Figure 4.9a shows a window of size 150 to be significantly better ($p < 0.01$) than a window of size 300, while Figure 4.9b shows that there was a non-significant decrease in performance when the smaller window was used. What is clear, though, is that in all cases the use of a window caused an improvement regardless of the window's size.

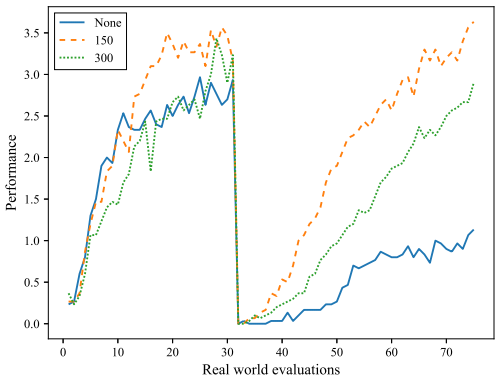
This improvement was likely because once the robot becomes damaged, many previous training patterns become incorrect and no longer accurately reflect the real-world robot. The use of a sliding window means that after a short delay the invalid training patterns



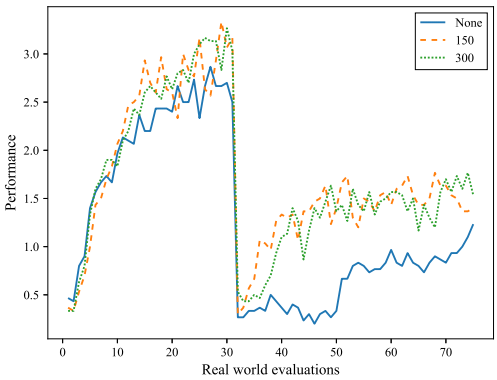
(a) Solving the simple problem with damage to the left wheel



(b) Solving the simple problem with damage to the right wheel



(c) Solving the simple problem with the left wheel slowed by 50 percent



(d) Solving the simple problem with the right wheel slowed by 50 percent

Figure 4.9: Performance over time for different sliding window sizes

are no longer used. The algorithm is then able to train the simulator exclusively on new, relevant data. Without a sliding window, the algorithm is required to gather the same number of new patterns as old patterns before even half of the training data accurately represents the real world, and the algorithm is able to improve.

Figure 4.9c shows that using a sliding window may cause the rate of training to be increased even before damage. This is an interesting discovery which means that the BNS as a whole can be improved through the implementation of this adaptation. A sliding window should, therefore, be considered for use in all BNS implementations, for both the improved rate of learning and the significant improvement in damage recovery that it facilitates. Further investigation should be conducted in this regard; it is out of the scope of this research on damage recovery.

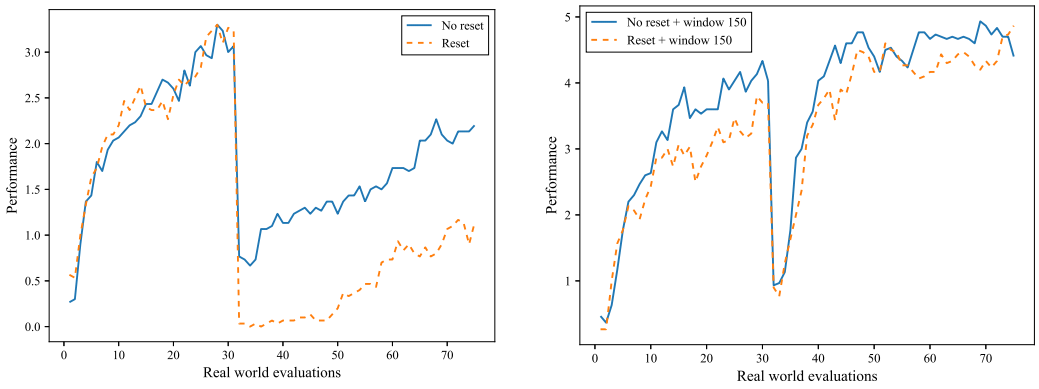
4.3.5 Simulator Reset

Figure 4.10a shows the results of resetting the simulator after damage is inflicted on the robot. This adaptation offered no benefit to the rate of improvement over unchanged BNS; both approaches adapted to the damage at the same rate. The simulator reset also caused the population of controllers to start to move away from previously discovered solutions, causing a significantly ($p < 0.00001$) larger drop in performance than when the simulator was not reset.

The reason for the drop in performance is clearly shown in Figure 4.11. Each of the plots shows the accuracy of the simulator's predictions for change in x position. Figure 4.11a shows the accuracy of the newly initialised SNN, Figure 4.11b shows the accuracy of the simulator on real-world evaluation before damage occurs, and Figure 4.11c shows the accuracy of the simulator immediately after damage has occurred. The horizontal axis represents the actual observed value, while the vertical axis represents the value predicted by the dynamic BNS simulator. The accuracy of the simulator immediately after damage occurs is worse than the trained simulator, but it is better than the accuracy achieved by a new random simulator. Resetting the simulator is, therefore, unlikely to offer improved performance.

It was hypothesised that resetting the simulator would have a benefit to a recovery process already using a sliding window (Section 4.3.4). Experimentation showed this not

to be true. Figure 4.10b shows the results of BNS with a sliding window of size 150, compared to the results of BNS with a sliding window of size 150 and simulator reset. There was no significant difference between the performance of each set of experiments; resetting the simulator had no benefit to the BNS process when a sliding window was already in use.



(a) Simulator reset after damage

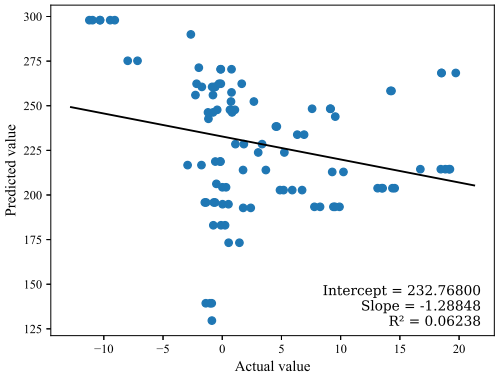
(b) Simulator reset and a sliding window

Figure 4.10: Performance over time for the simple problem

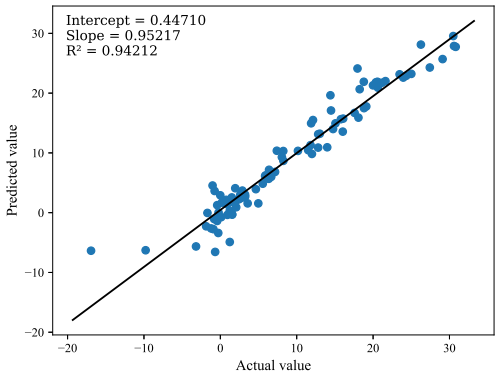
4.4 Real-World Results

The real-world results presented here represent the output from item 3.4 in Figure 4.1. The layout of the objectives for each problem are shown in Figure 4.3. Figures 4.12 and 4.13 show composite images of the robot's movement around the testing area. The first shows a robot completing the simple problem, and the second is a robot completing the infinity problem. The yellow marker is positioned at the front of the robot and is the direction in which the robot moves. Each image shows the path of the robot before damage (Pre-Damage), after damage to its left wheel (Post-Damage) and after recovery (Post-Recovery).

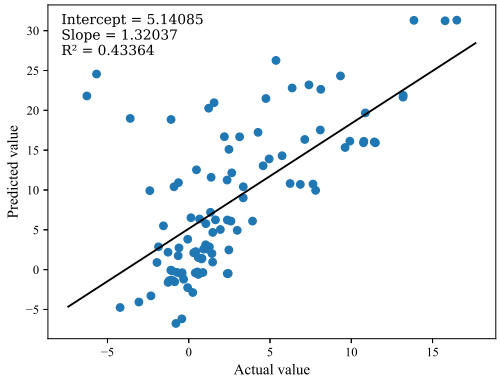
Figure 4.12 shows the robot completing the simple problem by starting in the centre and moving in a circle around that starting position, thereby moving through the four objectives, not shown in these images. The robot is then damaged, and its path veers to the side. In post recovery, the robot once again drives in a circle around its starting



(a) Initial simulator accuracy



(b) Trained simulator accuracy



(c) Simulator accuracy after damage

Figure 4.11: Simulator predicted values vs actual values

position.

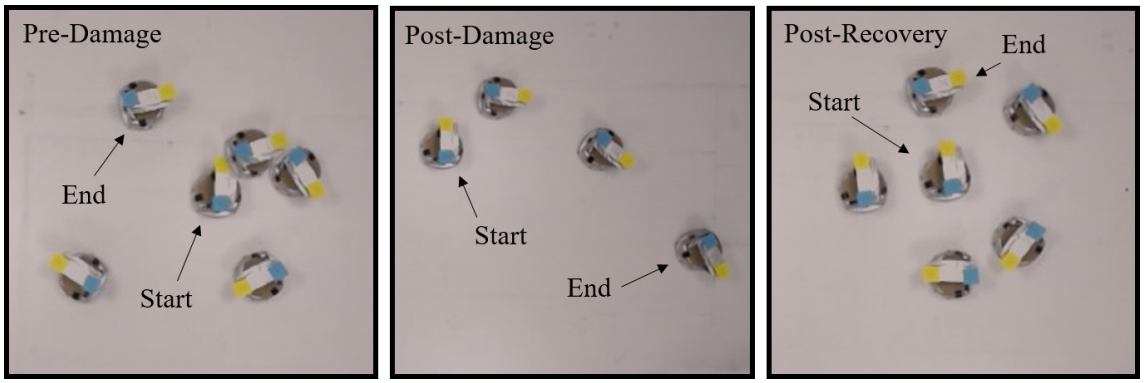


Figure 4.12: A composite image of the Khepera robot completing the simple problem

Figure 4.13 has the order of the robot's positions labelled as the path followed by the robot solving the infinity problem loops over itself multiple times. After the robot completed the task and was damaged, it could no longer turn far enough to the left after reaching position 4, resulting in the robot driving in a circle from position 4 to 6. Once the robot had recovered, instead of turning clockwise at the bottom-right of the ∞ shape, it instead turned counter-clockwise, from positions 3 to 4, before completing the remaining parts of the shape as before.

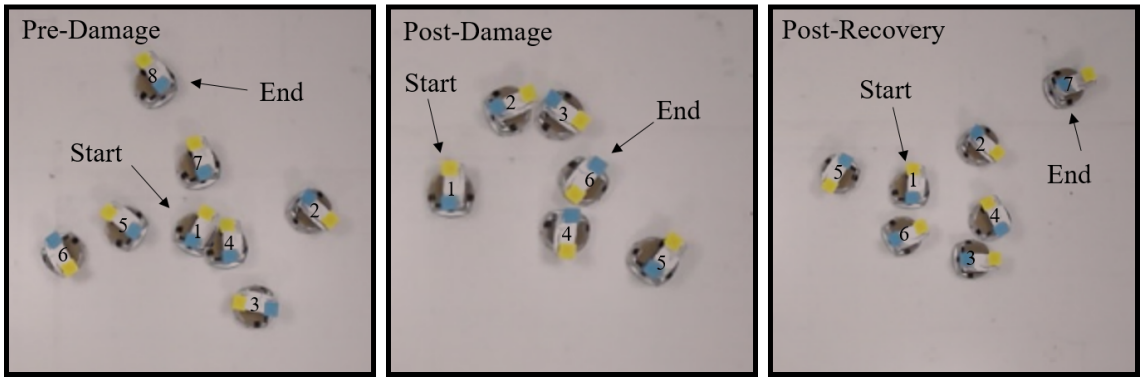


Figure 4.13: A composite image of the Khepera robot completing the infinity problem

Figure 4.14 shows the real-world paths followed by controllers evolved by BNS to solve one of the two problems while recovering from damage. Each plot shows three lines, representing the pre-damage, post-damage, and post-recovery performance of the robot.

The specific configurations used for the presented real-world results were chosen to

represent as wide a selection of configurations as possible. The performance of the algorithm without adaptations is shown for damage which was easily resolved by the algorithm (Figure 4.14a), as well as damage which could not be resolved (Figure 4.14c). Results were then also shown for an experiment using a sliding window for both the simple problem (Figure 4.14b) and the infinity problem (Figure 4.14d). The number of real-world evaluations taken for recovery is shown in parentheses along with each plot.

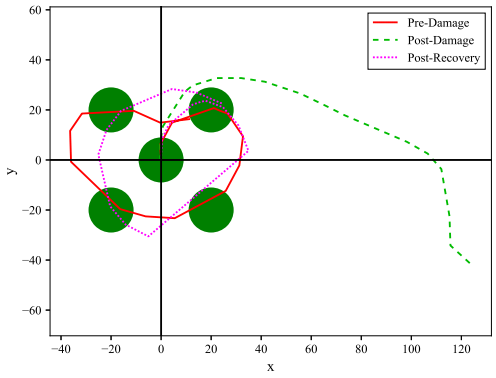
In Figure 4.14a the algorithm solved the simple problem. After damaging the left outer wheel, the robot no longer turned far enough in the clockwise direction, which caused it to follow a path to the side. Thirty-two real-world evaluations later, the robot had once again learned to complete its task by driving slower in order to compensate for the damaged wheel.

Figure 4.14b shows an experiment using a sliding window of size 150 and solving the simple problem. The damage to the robot's left wheel affected the robot in a similar manner to Figure 4.14a, but it took only ten real-world evaluations to recover from the damage, due to the window's positive effects on the rate of improvement.

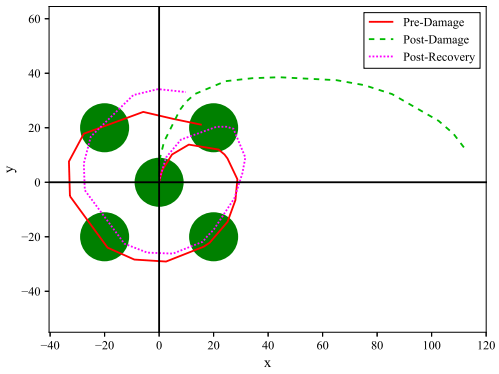
Damage to the right wheel of the robot creates a more complex problem for BNS to solve. Figure 4.14c shows the path followed by the robot, with controllers evolved using unchanged BNS, when it had learned to complete the simple problem. The figure shows the damaged robot driving in circles due to the slow inner wheel (marked as A). The robot was unable to recover by the seventy-eighth real-world evaluation and continued to drive in circles. Experiments in simulation suggest that the robot would eventually recover the ability to complete the task, but not in a time frame reasonable for real-world experimentation.

The final real-world experiment, shown in Figure 4.14d, used a sliding window of size 150 and attempted to solve the infinity problem. The robot learned to solve the problem with an infinity-shaped pre-damage path. Once the damage was inflicted to the robot's left wheel, BNS adapted to the damage by preferring counter-clockwise rotation. This is visible in the post-recovery path where, instead of turning to the right after reaching objective E (in the centre), the robot turned almost 360° in a counter-clockwise direction to reach objective D (at the top-right).

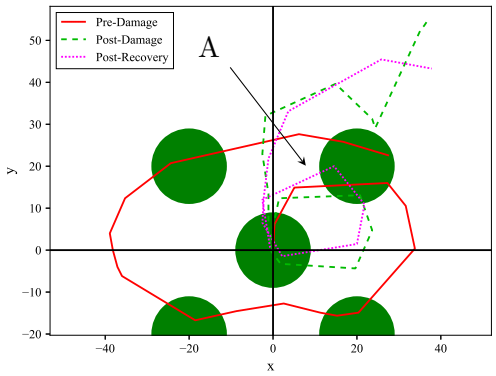
This real-world evaluation is clear evidence of BNS's ability to develop new behaviours



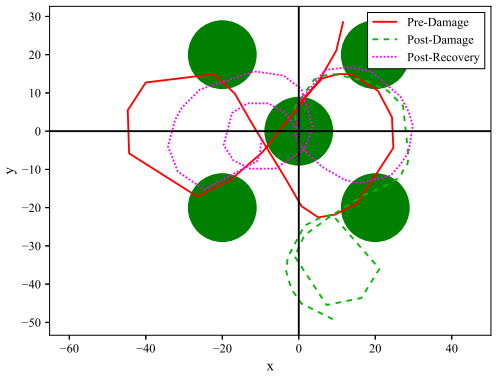
(a) Paths followed by controllers evolved using unchanged BNS (32 RW evaluations for recovery)



(b) Paths followed by controllers evolved using BNS with a sliding window size 150 (10 RW evaluations for recovery)



(c) Paths followed by controllers evolved using BNS with the right wheel damaged (algorithm did not recover)



(d) Paths followed by controllers evolved to solve the infinity problem with a sliding window size 150 (20 RW evaluations for recovery)

Figure 4.14: Real-world paths followed by evolved controllers

to solve a given problem. The result alleviates concerns that population and simulator convergence would prevent BNS from discovering new solutions, only allowing it to adapt existing solutions.

4.5 Conclusion

The results presented in this chapter have shown BNS capable of recovering from damage to a simple differentially-steered robot without any changes to the algorithm. Four adaptations to the algorithm were proposed based on their potential to improve BNS's performance by focusing on different issues that may arise upon the occurrence of damage. Evaluations were carried out both in a fake real world, which allowed for the rapid evaluation of many parameter configurations, and in the real world, which allowed the transferability of the results to be confirmed.

The use of a sliding window of training data caused the largest improvement in performance. Without the adaptation, the algorithm is unable to improve until at least as many post-damage training patterns have been collected as pre-damage, since - until that point - the majority of the training data is incorrect as it was obtained from the robot in its old, undamaged, state. The use of a window means that the old incorrect training data is discarded much sooner and the simulator is able to improve once again. In one case the sliding window was able to improve performance even before damage occurred. This improvement may indicate that the adaptation is useful not only for more than just problems that require automatic damage recovery, but for any BNS application, as a general improvement to the algorithm.

Resetting the population of controllers showed no significant improvement to the algorithm in the majority of cases. This result indicates that for the problems presented here, a lack of diversity in the population does not present an issue for the algorithm.

Neither a post-damage increase in the mutation rate nor the magnitude showed any improvement in the algorithm's damage recovery performance. The algorithm was also evaluated with both increasing simultaneously, with no improvement in performance. Since changing the mutation parameters would have increased the population diversity, this result corresponds with the population reset result since both adaptations seek to address a lack of diversity within the population, which was, therefore, likely not an issue for the given problems.

Finally, resetting the dynamic simulator showed no improvement in performance. When used without a sliding window, the reset caused a momentary drop in real-world

performance as the controller population began optimising for the re-initialised simulator. The performance decrease was not as large without a reset since the simulator, while incorrect, was at least similar to the real world. A sliding window prevented the drop in performance from being as large, by training the new simulator only on new, correct training data, but performance did not improve any more than when using only a sliding window. These results suggest that transfer learning *did* cause the previously trained simulator to be a better starting point for training the new simulator.

These results showed that BNS is able to recover well from damage. The logical next steps in evaluating BNS's damage recovery is to evaluate it on both more complex controllers and more complex robot morphologies.

Chapter 5

Closed-loop Controller Evolution

5.1 Introduction

Previous research using BNS has only aimed to evolve open-loop controllers, which are unable to react to external stimuli (Woodford et al., 2016, 2017). Closed-loop controllers are able to use information about the robot's state and environment in order to choose actions to perform. A closed-loop controller can function in many more environments than a single open-loop controller can, as it is able to react to changes in its environment.

This chapter presents the first experiments evaluating BNS's ability to evolve intelligent, closed-loop controllers. The problem solved by BNS in this investigation was the same as previous research by Pretorius et al. (2013). Controllers were evolved to use their light sensors to navigate a differentially steered robot towards a light source. They were then required to remain as close to the light source as possible for the remainder of their execution. An example of the path followed by a robot exhibiting the desired behaviour is shown in Figure 5.1. The robot started at the position marked with a red circle and moved to begin circling the light source, which was placed at the origin of the set of axes, on the large green circle.

Since the controllers needed to take readings from the robot's sensors in order to function, a simulator was developed to simulate these sensor readings. This investigation was the first instance of multiple distinct SNN simulators being trained concurrently.

Section 5.2 discusses this investigation's implementation (Figure 5.2, item 5), including

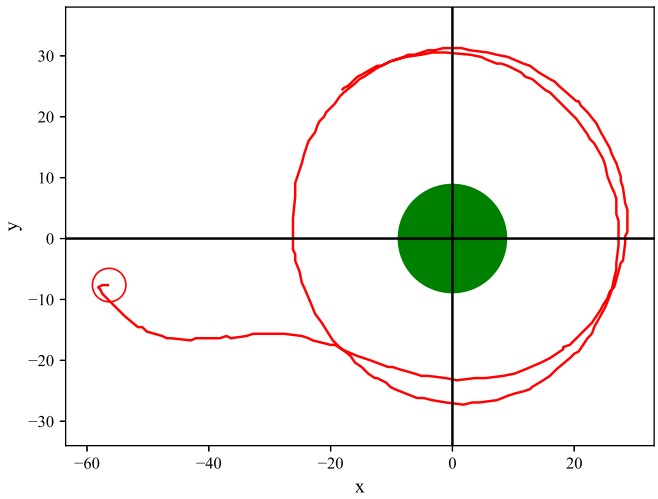


Figure 5.1: The path followed by a robot completing the light-following problem

the implementation of the adaptations to the BNS algorithm for closed-loop controllers. Section 5.3 presents the results of the experiments in simulation (Figure 6.1, item 6.1). Section 5.4 presents the results of real-world evaluations, which are represented by item 6.2 in Figure 6.1, and Section 5.5 concludes the chapter.

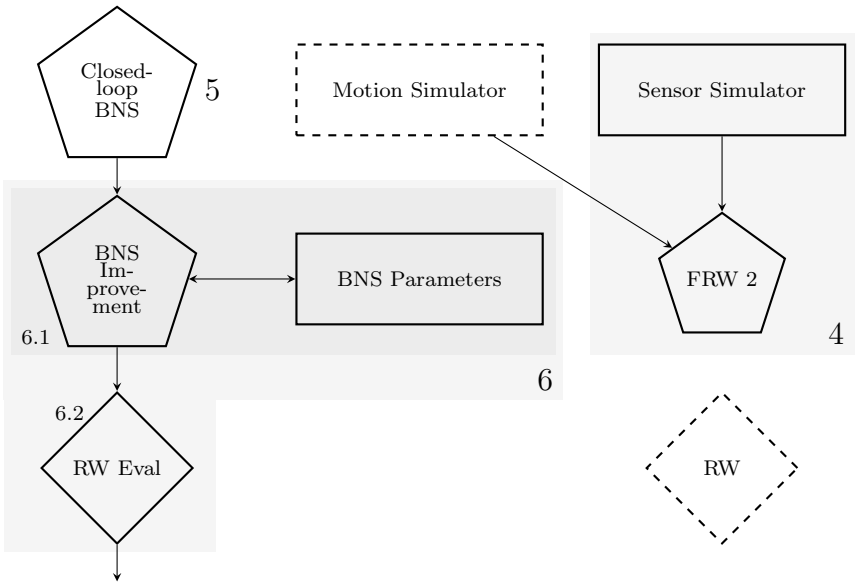


Figure 5.2: Investigation B focus

5.2 Implementation Details

The Khepera III has a set of built-in ambient light sensors, and non-concurrently-evolved SNNs have previously been used for the evolution of light-perception controllers for a similar differentially steered robot (Pretorius et al., 2013). This made the Khepera III an ideal robot for use in this investigation.

Section 5.2.1 discusses the sensors on the robot and Section 5.2.2 discusses the implementation of controllers able to use these sensor readings to react intelligently. Section 5.2.3 then discusses the implementation of an EA to evolve controllers. Section 5.2.4 describes the real-world testing environment used for this investigation. Section 5.2.5 discusses the implementation of a simulator to simulate the robot's sensor values. The output of these sections is represented by item 5 in Figure 5.2. Finally, Section 5.2.6 discusses the values of each parameter evaluated in the parameter study and is represented by item 6.1 in Figure 5.2.

5.2.1 Khepera III Sensors

The robot is equipped with an array of infrared sensors for close-range object detection and ambient light readings (K-team Corporation). The robot's nine outward facing infrared light sensors are positioned around the base of the robot. Six of these sensors are placed around the front and sides of the robot (Figure 5.3), at -85°, -45°, -10°, 10°, 45°, and 85° from the centre of the robot (Figure 5.3), and three face the rear. Two additional sensors face downwards, allowing the robot to detect markings on the ground.

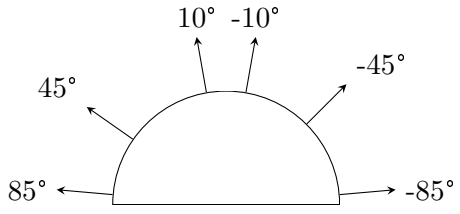


Figure 5.3: Khepera forward sensor positions

Figure 5.4 shows the variables used to describe the Khepera's position relative to a light source. ϕ is the angle between the Khepera's orientation and the vector from the Khepera to the light source. d is the distance from the Khepera to the light. The ambient

light sensors return values v, $v \in [0, 4200]$, where a smaller value represents more light. The responses of the front six sensors, based on the robot's orientation relative to the light source, are shown in Figure 5.5. The horizontal axis represents the physical robot's orientation to the light, while the vertical axis represents the reading of each of the robot's forward facing sensors when the robot is at that orientation. In order to obtain the values in the figure, 18 000 light sensor readings were taken from different angles and positions and aggregated to produce smooth plots. At each angle ϕ_a in the plot for sensor ϕ_s, the moving average of the sensors' responses is shown. The value is obtained by calculating the mean of all readings in $[\phi_a - 10, \phi_a + 10]$ for that sensor.

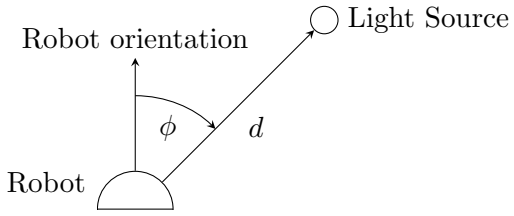


Figure 5.4: Khepera sensor parameters (Pretorius et al., 2013)

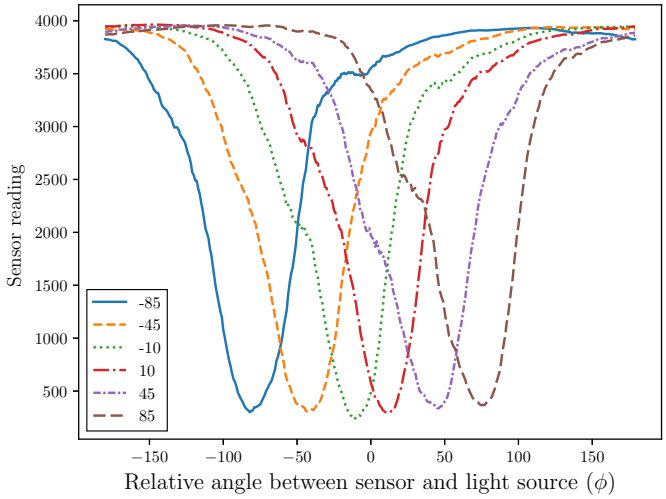


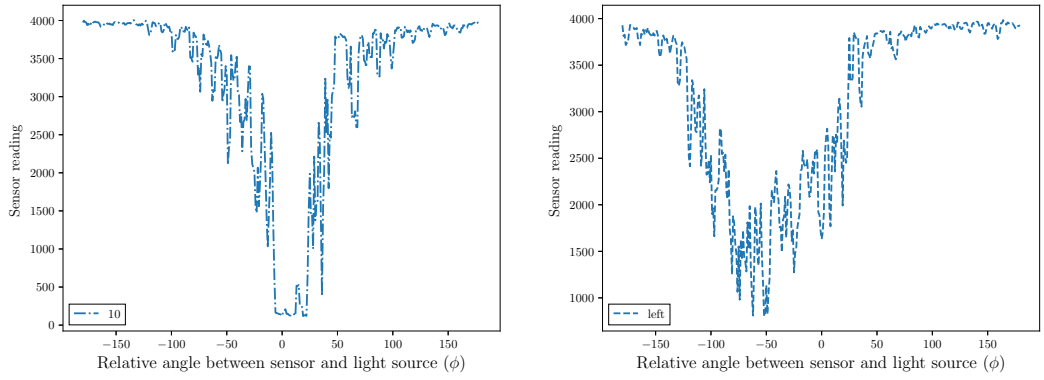
Figure 5.5: The Khepera III's light sensors' responses at different angles to the light

While Figure 5.5 is able to show the trends of the Khepera's sensors' responses, it is misleading with regard to the levels of noise in the sensors. Figure 5.6a shows a more realistic plot of a single reading taken at each angle with the 10° sensor. Around 0°, the

sensor was entirely unable to gather any useful data, since between -5° and 25° the sensor returns the same value for the majority of readings.

In previous research by Pretorius et al. (2013), robots using light-sensitive sensors have made use of only two sensors. This configuration also resembles that of a Braitenberg Vehicle (Braitenberg, 1986). The Khepera's additional sensors were, therefore, able to be used to obtain more reliable readings in a single command, while taking a similar approach to this previous research. Instead of using each sensor reading individually, the means of the readings from the forward facing sensors on each side of the robot were used as *virtual sensors*. i.e. the -85°, -45°, and -10° together formed the right virtual sensor.

The information from the virtual sensors is more useful than that from two single sensors because there is no range of angles where the sensor is unable to return useful readings, as was the case for single sensors. The range of angles where the sensors are able to detect some level of light is also larger, making each sensor useful for a wider range of angles (Figure 5.6b).



(a) Realistic single sensor readings

(b) Realistic virtual sensor readings

Figure 5.6: Khepera realistic responses

5.2.2 Controllers

Each controller in these investigations used a neural network to choose its next action based on the light sensor readings. The execution of a controller happened in a *command cycle* in which each set of motor values was maintained for 400ms, the light sensors read, and these readings used to generate a new set of motor values. This cycle was repeated

100 times for a total controller execution time of 40 seconds.

The length of time taken for the robot to respond to a request for its light sensor readings is approximately 50ms. The readings were, therefore, requested 75ms before the end of each command.

The controller network topology is based on the topology used by Pretorius et al. (2010). The controllers were implemented using a recurrent neural network with two inputs, and with two outputs used as recurrent inputs. The inputs to the network were the values of the robot's left and right sensors. The outputs are the desired motor speeds.

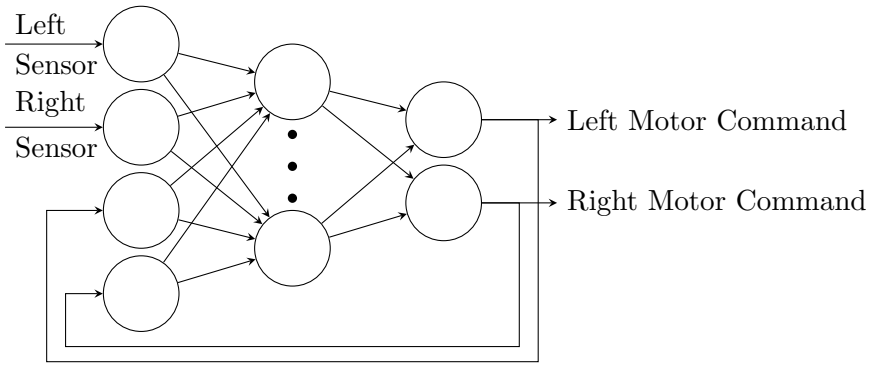


Figure 5.7: Controller ANN

5.2.3 Evolutionary Algorithm

Individual encoding

The individuals in the population were encoded as weight vectors, representing the weights of the ANN controllers. Every controller in the population had the same number of neurons in their hidden layer, which was set by a parameter passed into the BNS algorithm.

The calculation of the population's diversity, d_p, took place using equation (5.1)

$$d_p = \sum_{i=1}^{n_p} \sum_{j=1}^{n_g} (c_{ij} - \bar{c}_j)^2 \tag{5.1}$$

where n_p is the number of individuals in the population, n_g is the number of genes that comprise each individual, c_{ij} is the j^{th} gene of the i^{th} individual, and $\bar{c}_j$ is the j^{th} gene of the average individual. The average individual is calculated as in investigation A.

Fitness function

When executing a closed-loop controller, its behaviour and performance are likely to vary based on its starting position, since it may have learned how to complete the task from some starting areas, but not others. Therefore, in order to obtain an overall measure of the controller's performance, the fitness of a controller is based on its average performance over ten executions, each with a different, random starting position. The performance of each of these executions is based on the path followed by the robot during the execution, and a higher fitness is awarded to controllers that cause the robot to move towards, and remain near, the light.

Early in the process of evolution using BNS, the behaviour of controllers evaluated in the real world is erratic. This means that the execution of these controllers risks a robot navigating towards and colliding with the light fitting (Section 5.2.4). In order to reduce the risk of damage to the robot and the fitting, a system was implemented to automatically terminate the controller's execution in the event that the robot moved closer than a *safety boundary* to the light. A disadvantage of this approach is that the simulated behaviour of the robot, and its sensors, very near to the light was unpredictable due to the lack of data gathered in this area. This problem was addressed by specifying a goal radius around this safety boundary; controllers were then rewarded for remaining as close to the goal radius as possible. equation (5.2) describes the fitness function:

$$f_k = -\sum_{i=1}^{10}\sum_{j=1}^{100}\frac{1}{d_i}|\|p_{ij}\|_2 - r_g| \tag{5.2}$$

where the light source is at the origin, d_i is the robot's distance from the light at the initial position of execution i, p_{ij} is the j^{th} position in the path followed by the controller in its i^{th} execution, and r_g is the goal radius.

This function improves on that used by Pretorius et al. (2010) by differentiating more clearly between behaviours, such as circling at a constant distance and moving towards and then away from the light, both of which evaluate identically in the previous fitness function.

The theoretical optimal behaviour could be achieved by a controller which began its execution on the radius r_g and then moved in a perfect circle, remaining at a distance of

r_g from the light source. Such a controller would achieve a fitness value of 0.

Reproduction

The selection of individuals for reproduction was performed using tournament selection, as was the selection of individuals to be evaluated in the real world. The size of the tournament is a parameter to the algorithm. Once individuals are selected for reproduction, a crossover operation is performed. Three methods of crossover, Single-point, Uniform, and Simulated Binary Crossover were evaluated, with the choice of the crossover operator being a parameter for the algorithm.

Parents are repeatedly chosen from the population and combined using the crossover operator to create the new population. Each individual in the new population then undergoes mutation, where each of their genes is changed probabilistically, based on a normally distributed random variable.

5.2.4 Testing Area

A removable light fitting was installed in the centre of the tracking area for this investigation. The testing environment was otherwise identical to that used in Chapter 4.

When the light fitting was in use, its light could not be allowed to illuminate the top of the tracking markers; the tracking system was calibrated to the specific colours of the markers, and the light changed the colour of the markers sufficiently that the tracking system was unable to track them. The markers were, therefore, elevated above the light fitting by mounting them on a frame attached to the robot (Figure 5.8).

5.2.5 Khepera Sensor Simulator

The construction of SNNs for the simulation of the Khepera's sensor readings was done very similarly to that of the SNNs for motion simulation. Given the robot's orientation to and distance from the light, the two SNNs are able to predict one of the robot's left and right virtual sensor readings.

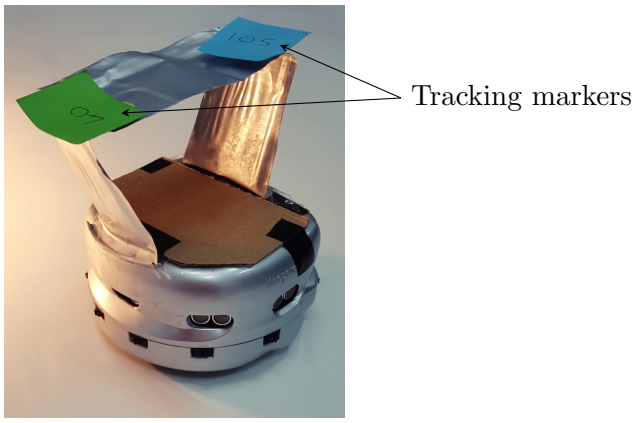


Figure 5.8: The Khepera III Robot with elevated tracking markers

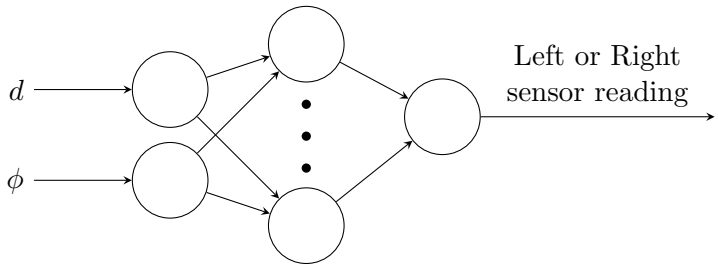


Figure 5.9: Sensor SNN

5.2.6 Parameter Evaluation

Parameter evaluation in this investigation was carried out in the fake real world. Static SNNs were created, using the same methodology as was used by Pretorius et al. (2013), to act as a fake real world, and used to evaluate parameter combinations.

Static simulator creation

In order to train a static simulator, data was gathered from the real world. Random commands were sent to the robot, causing the robot to roam the testing area. Every 400ms, the robot's position and orientation to the light source, as well as its ambient light sensor readings, were saved. Each of these readings could then be used to create one training pattern for each of the two sensor SNNs. The data gathering process was allowed to continue for several hours; a total of 16440 training patterns were created for each SNN.

Resilient backpropagation was then used to train the sensor SNNs for 40000 epochs. The final accuracy of the left sensor simulator is shown in Figure 5.10. Several network topologies were evaluated; a network with a single hidden layer containing twenty-five hidden neurons, each utilising a sigmoid activation function, was found to produce good results.

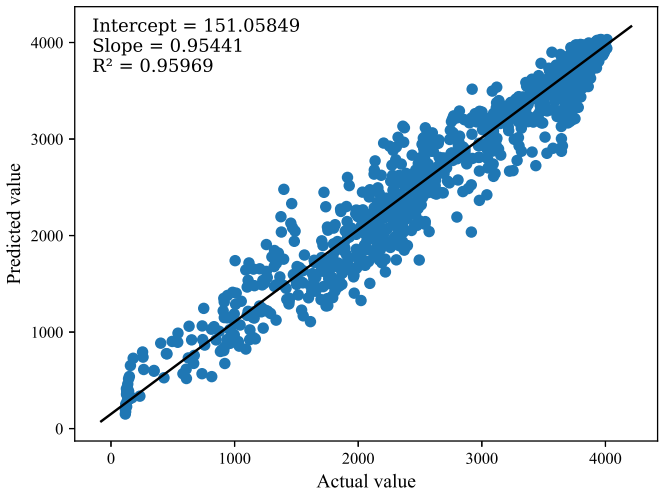


Figure 5.10: The static sensor simulators predicted vs actual sensor values

Parameters

There are several parameters to the BNS algorithm beyond those relevant to the closed-loop adaptations, and their ideal values were not apparent for this experiment. The effect of nine of these parameters on the performance of the algorithm was, therefore, also evaluated. This parameter evaluation was performed using a fake real world, as described in Chapter 4. This is represented by item 6.1 in Figure 6.1.

The parameters which were evaluated are:

- The range of values between which the controller's neural weights are initialised.
- The size of the tournaments used to select individuals for reproduction and real-world evaluation.
- The number of hidden neurons in the motion and sensor simulators.
- The size of the population of controllers.
- The mutation rate and magnitude.
- The number of neurons in the hidden layer of the neural network controller.
- Whether recurrent connections are used in the neural network controllers.
- The crossover operator used in reproduction.

Default parameter configuration

In order to evaluate different adaptations and configurations, a default BNS configuration was chosen. Individual parameters in the configuration could then be varied, and the effect of these changes on the algorithm's performance evaluated. The possible interaction of parameters was, therefore, not evaluated in this investigation. The default parameter values were chosen because they were found to perform well in preliminary experimentation. These choices were made in order to reduce the adverse effects caused by other parameters on the parameter being evaluated. The default parameter configuration is given in Table 5.1.

Table 5.1: Default parameter configuration

Parameter	Value
RW Evaluation Limit	100
RW Tournament Size	0.1
Controller Tournament Size	0.1
Population Size	100
Mutation Rate	0.1
Mutation Magnitude	0.05
Controller ANN Hidden Neurons	5
Weight Initialisation Range	[-5,5]
Simulator ANN Hidden Neurons	25
Recurrent Controller Network	True

Statistical analysis

Every parameter configuration under investigation was evaluated thirty times. For each of these thirty experiments, statistical observations were obtained using equation (5.3).

$$O_\tau = \sum_{i=1}^{100} f_\tau(i) \qquad \tau = 1, 2..30 \tag{5.3}$$

where $f_\tau(i)$ is the fitness of run τ at real-world evaluation i in the controller's execution. A Mann-Whitney U Test can then be performed to test the statistical significance of the difference between the distributions of the performance values of any two configurations.

Simulator performance presentation

This investigation was the first in which sensor SNNs were trained concurrently, using BNS. The accuracy of these simulators is, therefore, of interest. The results for simulator performance were obtained by comparing the final dynamic simulator's output to the expected values, given the robot's position and orientation. These values were then compared to assess simulator accuracy; ideally, the dynamic simulator should have learned to predict values as close to the fake-real-world values as possible.

5.3 Results and Discussion

Each set of results plotted in this section follows the same format as was followed in investigation A. Each configuration was used for thirty simulated experiments, and the

mean performance of the fittest individuals at each fake-real-world evaluation was plotted. The vertical axis represents the fake-real-world fitness, while the horizontal axis represents the number of real-world evaluations.

Section 5.3.1 presents the dynamic simulator's accuracy. Sections 5.3.3 to 5.3.12 then give the results of the evaluation of the other parameters to the BNS algorithm: crossover operators, data augmentation, weight initialisation ranges, islands, controller ANN topologies, Headless Chicken mutation, intermittent population resets, tournament sizes, mutation parameters, SNN topologies, and population size.

5.3.1 Simulator Performance

Figure 5.11 shows the performance of the left sensor simulator in the case where no adaptations were implemented (Figure 5.11). The left and right sensor simulators perform comparably; therefore, the accuracy of the left sensor simulator is used as representative of the accuracy of the sensor SNN as a whole. The expected sensor values are shown along the horizontal axis, and the values predicted by the dynamic simulator are shown along the vertical axis. A line of best fit is also included, with its slope and y-intercept values and the coefficient of determination, R^2. The dynamic simulator learned the trend of the data and was able to predict values very well, given the noise levels of the sensor readings. The number of data points around the middle of the range values is low because of the steep gradient of values returned by the sensors, which is visible in Figure 5.5.

5.3.2 Crossover Operators

Figure 5.12 shows the results of the evaluation of multiple crossover operators. SBX offered a significant performance improvement over the other operators ($p < 0.05$) (Table 5.2). It is thus an appropriate choice for this problem and can be recommended for considering for all BNS applications.

Table 5.2: p-values for Figure 5.12

	SBX	**SinglePoint**
SinglePoint	0.00868	
Uniform	0.01076	0.43764

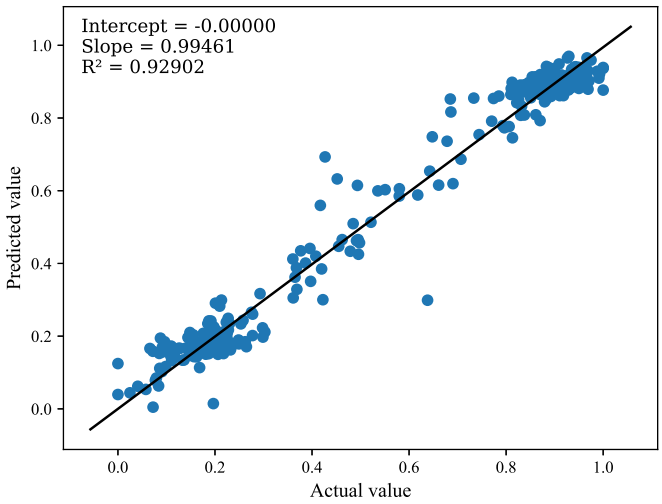


Figure 5.11: Sensor SNN predicted vs actual values with no adaptations

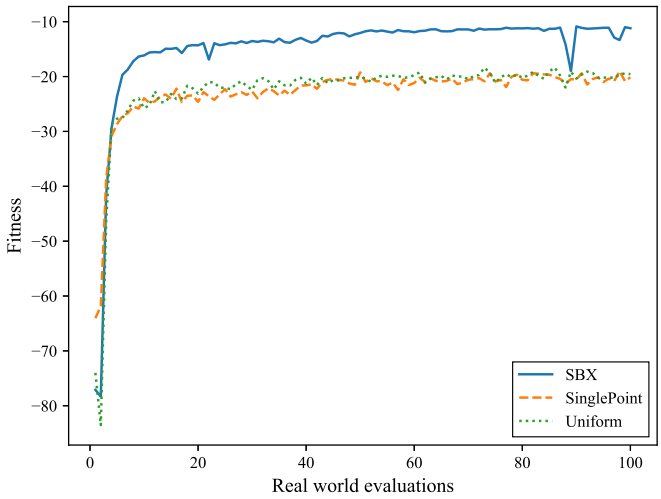


Figure 5.12: Fitness over time with different crossover operators

5.3.3 Data Augmentation

Data augmentation (Figure 5.13) had a significant positive impact on performance ($p < 0.01$), likely due to the more rapid improvement of the simulator allowing for the creation of more transferable controllers earlier in the evolution process. It is therefore suggested that, if realistic assumptions can be made with regards to the symmetry of the robot, data augmentation be used.

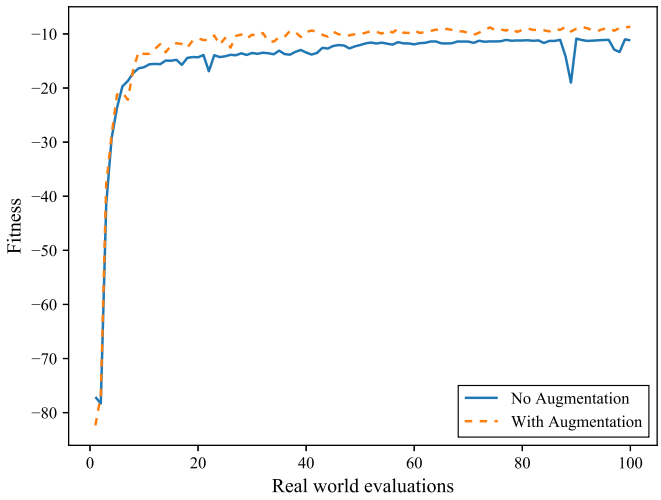
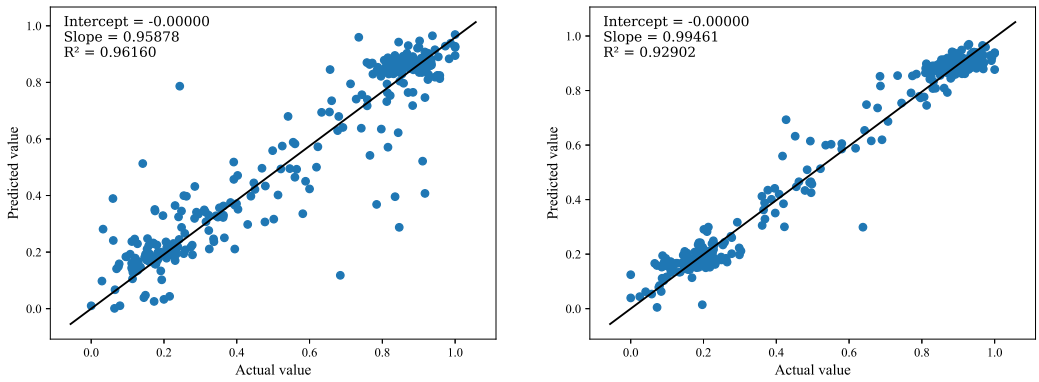


Figure 5.13: Fitness with and without data augmentation

Interestingly, Figure 5.14 shows that the simulator's predictions were slightly less reliable when data augmentation is used; the R^2 value is higher, but the slope of the line is further from the value of 1, a value that is desired since a perfect dynamic simulator would predict the same values as were observed in the real world, resulting in a slope of 1. This result points to the fact that the assumption of the robot's symmetry was not entirely accurate. Such a result may not seem to correspond with the improvement in performance seen in Figure 5.13, but it is likely that the early improvement caused by having extra training data caused a substantial improvement in performance, even if this data was not entirely accurate, since it allowed more transferable controllers to evolve earlier. These slight inaccuracies may also have resulted in the evolution of more robust and noise-tolerant controllers.



(a) With data augmentation (b) Without data augmentation

Figure 5.14: Sensor SNN predicted vs actual values

5.3.4 Weight Initialisation Range

Figure 5.15 shows the performance of BNS with the neural weights of the population of controllers initialised in different ranges. There was a significant benefit to the use of wider ranges of values, with a range of $[-10; 10]$ far outperforming $[1; 1]$ ($p < 0.00001$) (Table 5.3). This result shows the importance of the diversity and distribution of the initial population throughout the search space; a population initialised with neural weights in a larger range of values is likely to be more diverse than one with a smaller range.

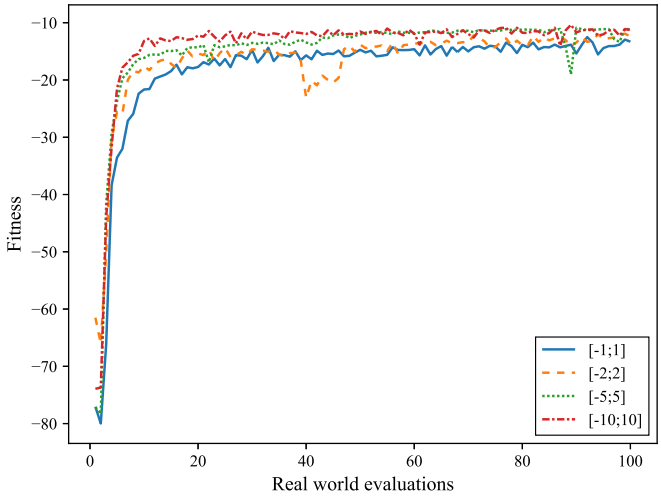


Figure 5.15: Fitness over time with different weight initialisation ranges

Table 5.3: p-values for Figure 5.15

	[-1;1]	**[-2;2]**	**[-5;5]**
[-2;2]	0.00403		
[-5;5]	0.02416	0.95873	
[-10;10]	0.00000	0.04207	0.11882

5.3.5 Island Evolutionary Algorithms

For the evaluation of each Island EA parameter, the other island parameters were held constant. These values at which the parameters were held are shown in Table 5.4. In each set of results, the performance of BNS without an Island EA is shown for reference.

Table 5.4: Default Island EA values

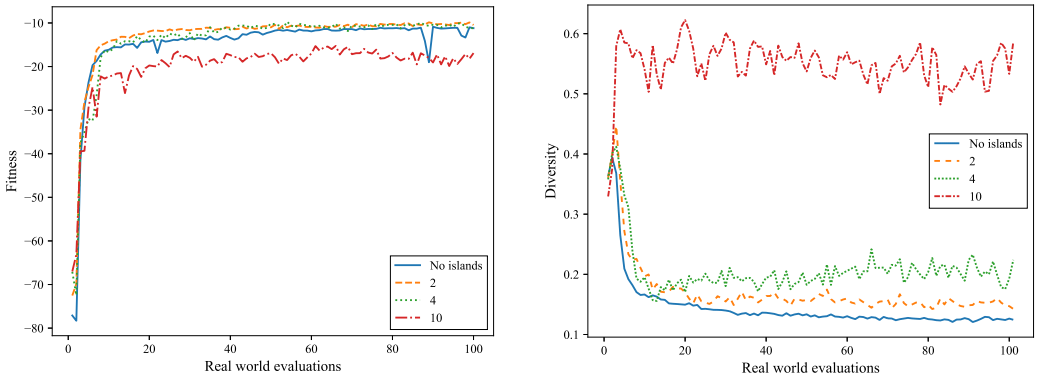
Parameter	**Value**
Interval	10
Number Islands	2
Number Migrants	1

Figure 5.16a shows that while two and four islands performed significantly ($p < 0.05$) better than an implementation without islands, ten islands caused the algorithm to perform significantly worse ($p < 0.005$) (Table 5.5). The likely reasons for these changes can be seen in Figure 5.16b. Two or four islands caused a small increase in diversity, leading to the discovery of better solutions and subsequent improved performance. Ten islands, however, was too many and caused the population to become unable to converge, likely due to the random replacement of individuals in each population by migrant individuals. The upper limit on improvements in performance based on the number of islands will not always be clear. Two is, therefore, a safe number of islands as it offers a significant, though small, improvement with the lowest risk.

Table 5.5: p-values for Figure 5.16a

	No islands	**2**	**4**
2	0.01327		
4	0.00583	0.56922	
10	0.00160	0.00000	0.00000

The frequency of migration (Figure 5.17) needs to be low enough that each island has



(a) Fitness over time with an Island EA with different numbers of islands

(b) Diversity over time with different numbers of islands

Figure 5.16: The effects of the number of islands

the opportunity to develop good controllers. When the time between migrations was too low, there was no significant performance benefit to the use of an Island EA (Table 5.6). Once this interval was increased to longer than five real-world evaluations, there was a significant ($p < 0.05$) positive impact on performance. The risk of choosing a value for this parameter, which negatively affects performance, is low; no values were found which negatively affected performance. It is, however, suggested that longer migration intervals be chosen to allow sufficient time for convergence within each island.

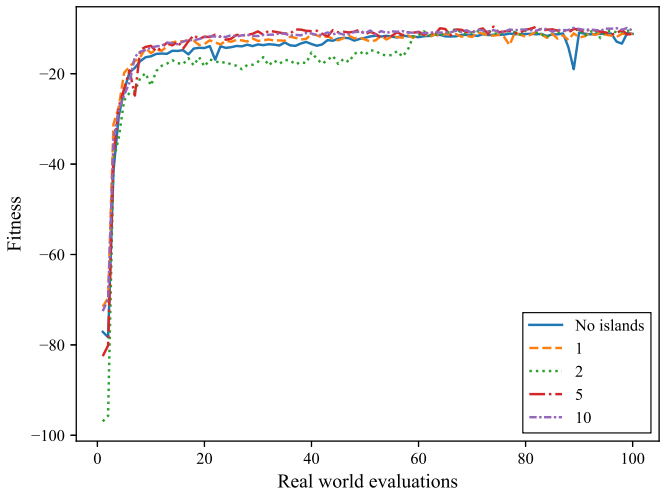


Figure 5.17: Fitness over time with an Island EA with different migration intervals

Table 5.6: p-values for Figure 5.17

	No islands	1	2	5
1	0.05188			
2	0.13345	0.52014		
5	0.04060	0.95873	0.62040	
10	0.01327	0.59969	0.36322	0.67350

Finally, as with the migration interval, there is little risk of choosing a value for the number of migrants which negatively affects performance. All evaluated values, except five, offered significant ($p < 0.05$) (Table 5.7) improvements.

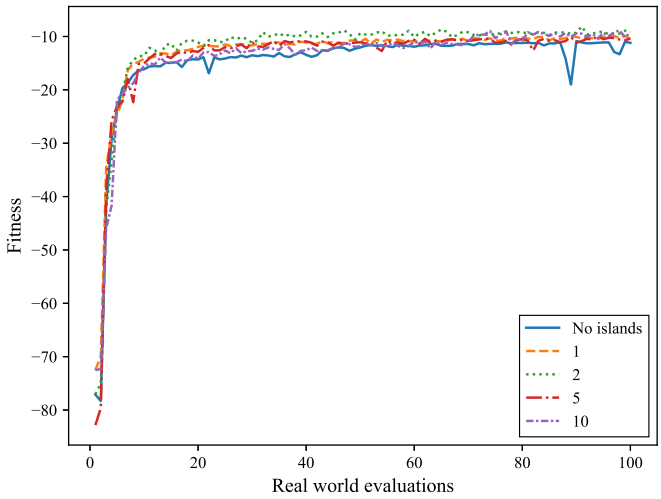


Figure 5.18: Fitness over time with various island migration sizes

Table 5.7: p-values for Figure 5.18

	No islands	1	2	5
1	0.01327			
2	0.00045	0.34783		
5	0.05012	0.50114	0.06353	
10	0.01123	0.57929	0.54933	0.29727

5.3.6 Controller Network Topology

The performance of a non-recurrent controller was evaluated (Figure 5.19) and found to perform significantly ($p < 0.005$) better than a recurrent network with a hidden layer

of equal size. It may be beneficial to investigate other recurrent topologies, such as an LSTM (Hochreiter, 1997), for controller implementation for more complex problems, but for simple navigation tasks, a simpler feedforward controller appears to offer the greatest performance benefits.

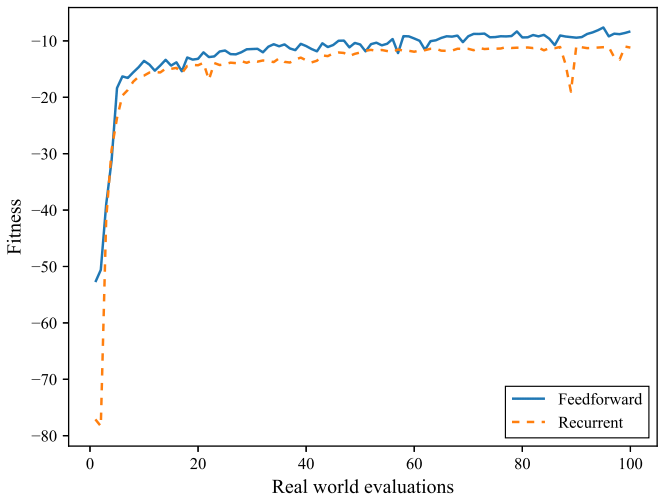


Figure 5.19: Fitness over time with recurrent and feedforward controllers

When using recurrent neural networks, there was no significant difference between the performance with different controller network sizes (Figure 5.20).

5.3.7 Headless Chicken

The results of experimentation using the Headless Chicken adaptation (Figure 5.21) show that the diversity introduced to the population had a significant ($p < 0.05$), but small, positive effect when introduced with any probability $\in (0, 0.25]$ (Table 5.8). This was no longer the case when the probability was increased to 0.5, likely because the high probability of mutation prevented the population from converging reliably, which can be seen in Figure 5.21b.

Based on the results presented here, it is suggested that Headless Chicken probability values smaller than 0.1 are used to ensure that the amount of diversity introduced does not negatively impact the population's rate of convergence.

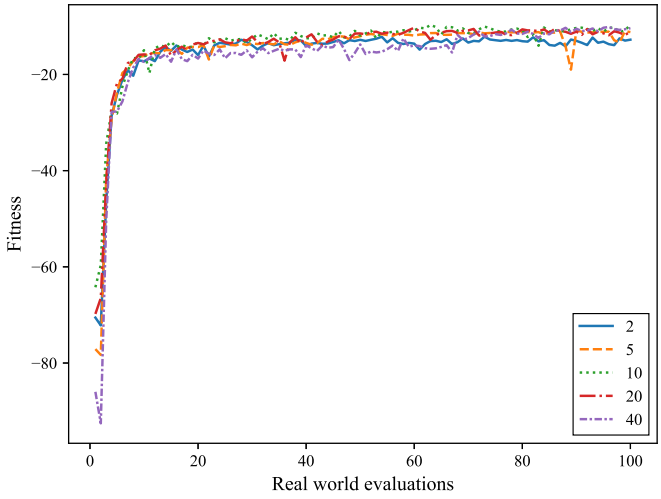
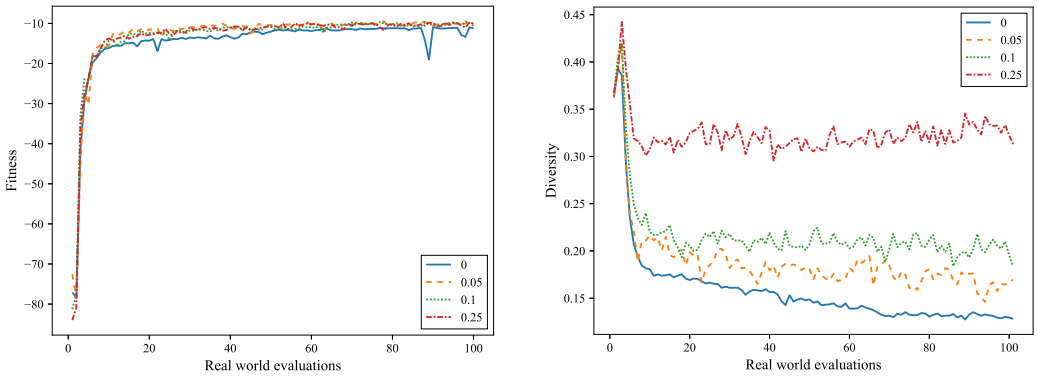


Figure 5.20: Fitness over time with different numbers of hidden neurons in the controller ANNs



(a) Fitness over time with various Headless Chicken probabilities

(b) Diversity over time with various Headless Chicken probabilities

Figure 5.21: The effects of various Headless Chicken probabilities

Table 5.8: p-values for Figure 5.21a

	0	**0.05**	**0.1**
0.05	0.00610		
0.1	0.01765	0.70617	
0.25	0.01765	0.63088	0.98231

5.3.8 Intermittent Population Reset

The results of experiments evaluating the effects of different Reset-Elitism and interval values are shown in Figures 5.22 and 5.23, respectively. For reference, the performance of BNS without intermittent population reset is also shown. For each of these parameter evaluations, the other parameter was held at a default value. These default values are shown in Table 5.9.

Table 5.9: Default intermittent reset values

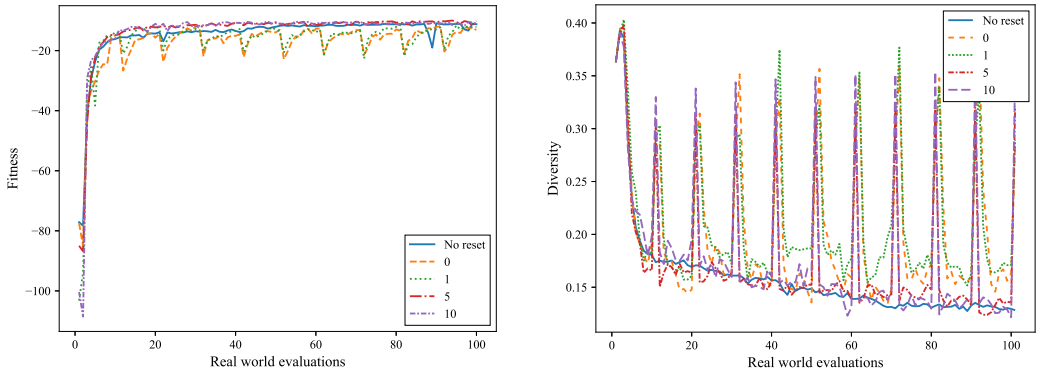
Parameter	**Value**
Interval	10
Elitism	5

The use of Reset-Elitists had a significant ($p < 0.05$), but small positive effect on the final fitness of controllers evolved with BNS (Table 5.10). The adaptation does have a clear impact on fitness over time; each time the population is reset, the fitness of the best controller drops and recovers again. This effect is reduced as the number of REs increase as more *good* controllers are transferred to the new population.

The performance benefit was greatest with five Reset-Elitists, representing 5 percent of the population. The magnitude of improvement was small; further research should be conducted to investigate the use of this adaptation before general guidelines for its use are suggested.

Table 5.10: p-values for Figure 5.22a

	No reset	**0**	**1**	**5**
0	0.16687			
1	0.14945	0.90000		
5	0.00667	0.00001	0.00001	
10	0.05746	0.00009	0.00008	0.34783



(a) Fitness over time with intermittent population reset and different Reset-Elitism values

(b) Diversity over time with intermittent population reset and different Reset-Elitism values

Figure 5.22: The effects of different Reset-Elitism values

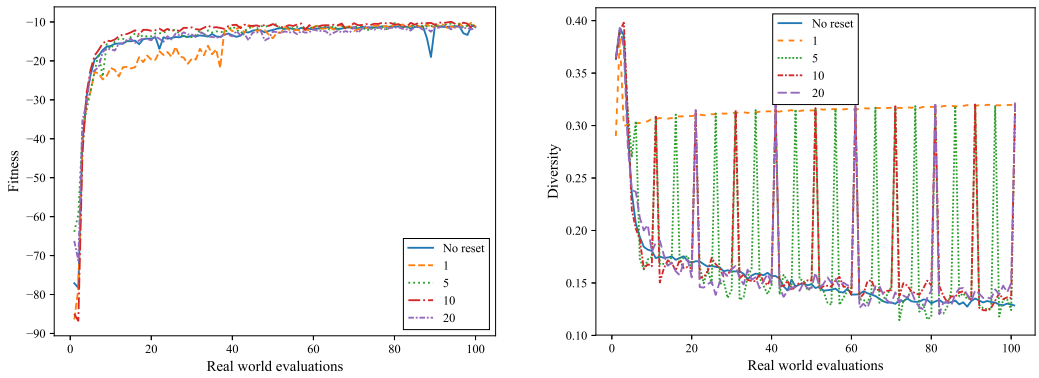
Similar to the Reset-Elitists parameter, only small ranges of Reset Interval values had a significant ($p < 0.01$) (Table 5.11) impact on the performance of the algorithm. In this case, a Reset Interval of ten real-world evaluations offered the most improvement, while an interval of one had a visibly negative effect on the initial training process, due to the population being unable to converge at all (Figure 5.23b). Once again, since the improvement was so small and the range of values which were beneficial so narrow, further research should be conducted on the use of this adaptation for specific BNS applications before guidelines can be proposed.

Table 5.11: p-values for Figure 5.23b

	No reset	**1**	**5**	**10**
1	0.10233			
5	0.08236	0.91171		
10	0.00667	0.42896	0.32553	
20	0.59969	0.31119	0.17145	0.02068

5.3.9 Tournament Sizes

Figures 5.24 and 5.25 show the results of experiments with different tournament sizes for the selection of individuals for reproduction and real-world evaluation, respectively. The



(a) Fitness over time with intermittent population reset and different reset interval values

(b) Diversity over time with intermittent population reset and different reset interval values

Figure 5.23: The effects of different reset interval values

tournament sizes are represented as proportions of the total population size.

It was found to be important that the *reproduction tournament* was small enough to apply sufficient reproductive pressure to the population in order to promote convergence of the population. A tournament 50 percent the size of the population failed to do this and negatively affected the performance of the algorithm. However, the size of the tournament used to select individuals for real-world evaluation had minimal impact on the performance of BNS. This was since controllers evaluated in the real world do not directly influence the evolution process; they serve only to gather training data for the simulators, which can be achieved through the evaluation of any individual in the population.

Table 5.12: p-values for Figure 5.24

	0.05	**0.1**	**0.2**
0.1	0.08500		
0.2	0.70617	0.05012	
0.5	0.00289	0.16238	0.00056

Figure 5.26 shows very clearly that the effect of the size of the real-world tournament had no significant impact on simulator performance. The accuracy of the dynamic SNN was comparable, regardless of the chosen real-world tournament size.

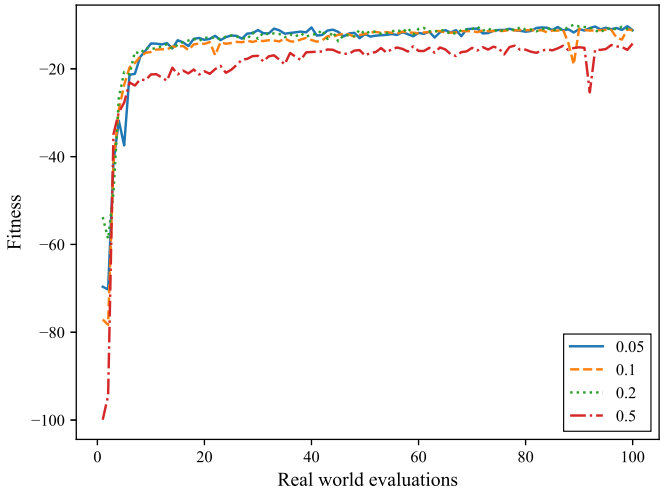


Figure 5.24: Fitness over time with different controller tournament sizes

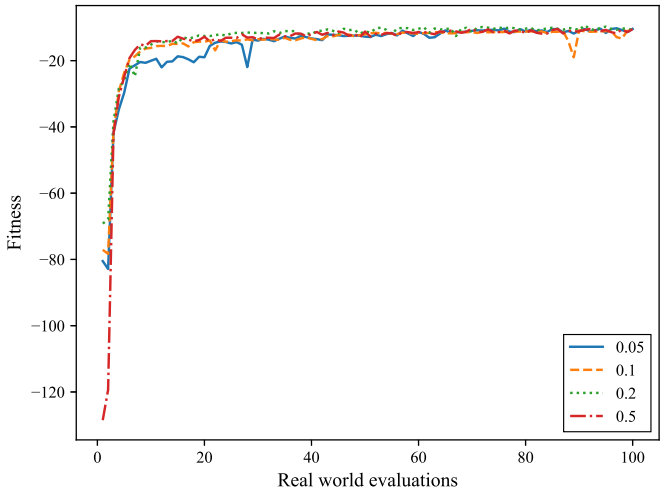
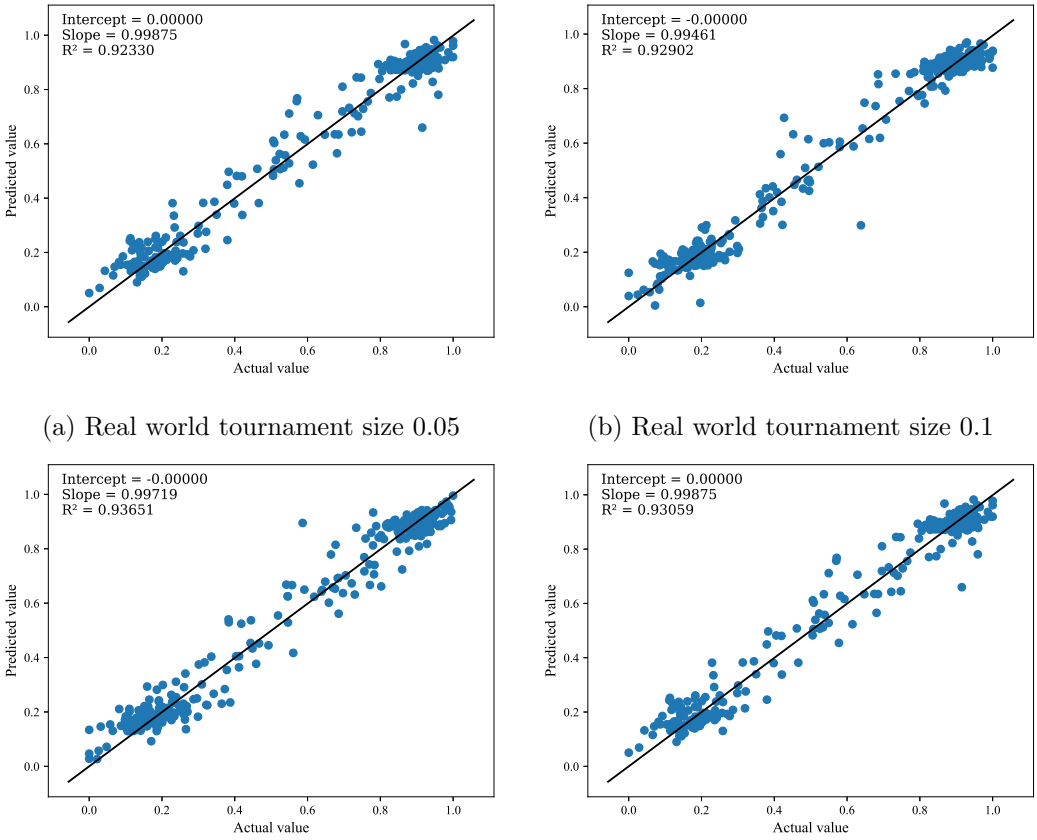


Figure 5.25: Fitness over time with different real-world tournament sizes



(a) Real world tournament size 0.05

(b) Real world tournament size 0.1

(c) Real world tournament size 0.2

(d) Real world tournament size 0.5

Figure 5.26: Simulator predicted vs actual values with different real-world tournament sizes

Table 5.13: p-values for Figure 5.25

	0.05	**0.1**	**0.2**
0.1	0.11199		
0.2	0.45530	0.01988	
0.5	0.43764	0.42896	0.11536

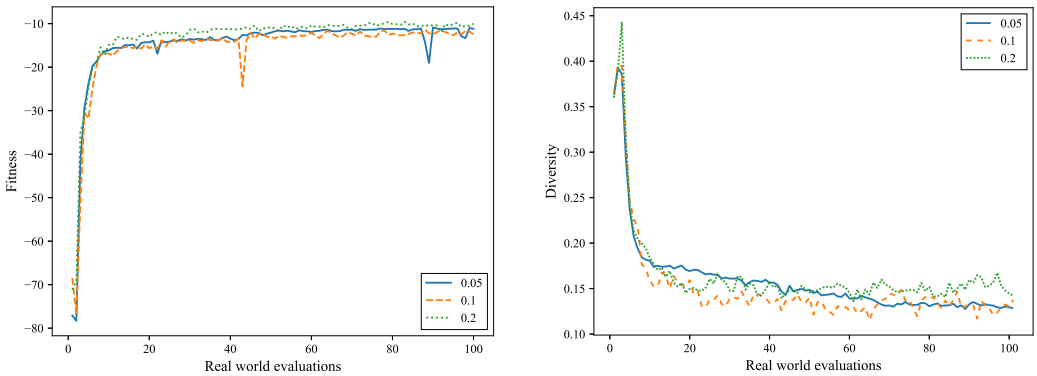
5.3.10 Mutation

While evaluating the effects of changes to either the mutation rate or magnitude, the other parameter was held constant. These default values are shown in Table 5.14.

Table 5.14: Default mutation values

Parameter	**Value**
Mutation rate	0.1
Mutation magnitude	0.05

An increase in the mutation magnitude from 0.05 to 0.2, shown in Figure 5.27a, had a significant ($P < 0.05$) impact on performance (Table 5.15). This was likely due to the prolonged increased diversity that this provided, during the later real-world evaluations (Figure 5.27b). A moderately large mutation magnitude is, therefore, suggested for use with BNS.



(a) Fitness over time with different mutation magnitudes

(b) Diversity over time with different mutation magnitudes

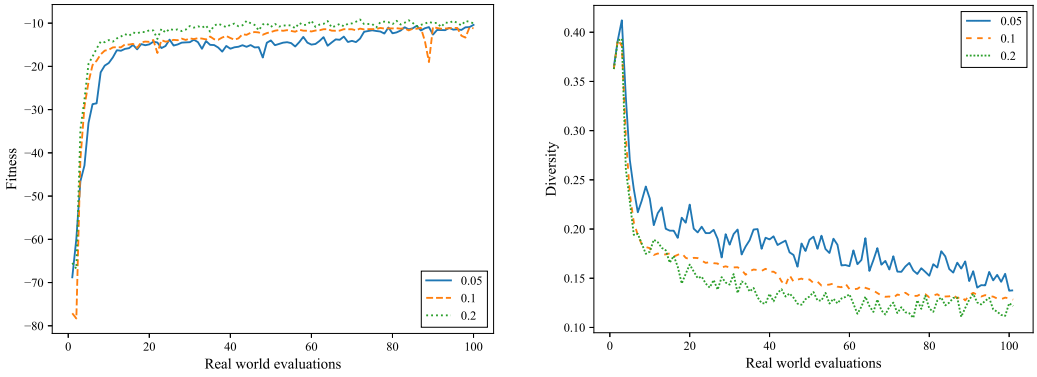
Figure 5.27: The effects of different mutation magnitudes

A larger mutation rate led to significantly ($p < 0.05$) improved performance (Table

Table 5.15: p-values for Figure 5.27a

	0.05	**0.1**
0.1	0.97052	
0.2	0.03514	0.01765

5.16). Interestingly, in this case, the base mutation rate appears to have been lower than was ideal. Increasing the mutation rate increased the rate of convergence of the population, likely allowing better solutions to be discovered than was possible with the lower mutation rate. For more complex problems, such high levels of mutation may prevent sufficient convergence from occurring, with adverse effects on BNS's performance, though in general, increased mutation is beneficial for the algorithm's performance.



(a) Fitness over time with different mutation rates

(b) Diversity over time with different mutation rates

Figure 5.28: The effects of different mutation rates

Table 5.16: p-values for Figure 5.28a

	0.05	**0.1**
0.1	0.34783	
0.2	0.02416	0.00205

5.3.11 Simulator Network Topology

Figure 5.29 shows the results of the use of different sizes of hidden layers for the simulators. There was very little significant difference between the performance of BNS with different

hidden layer sizes (Table 5.17), though it appears that choosing a larger hidden layer for the simulators is beneficial. There was no downside, and it would avoid potential issues that could be encountered where small simulators are unable to accurately simulate the real world, especially for more complex problems. Such an issue is visible in the early performance of BNS when using a simulator with five hidden neurons, as it is unable to capture the complexity of the real world.

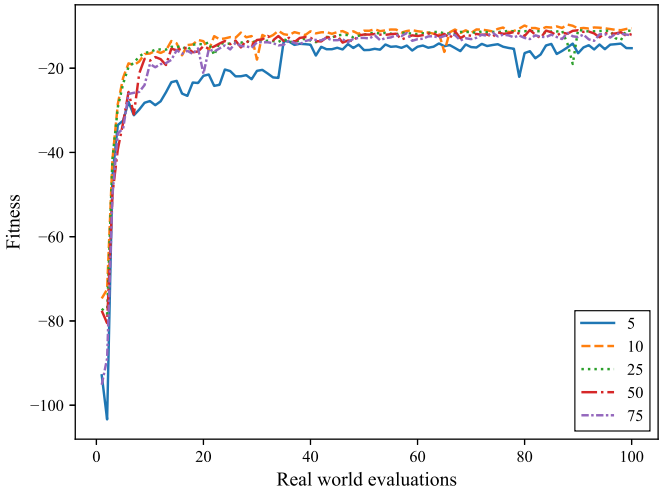
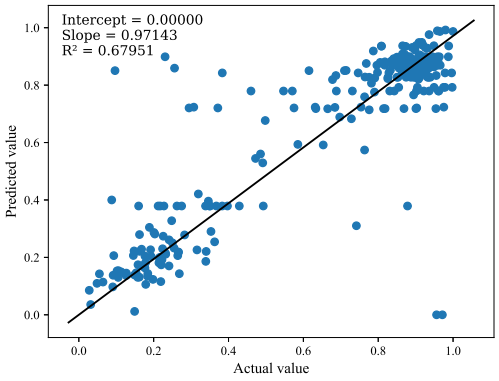


Figure 5.29: Fitness over time with different simulator hidden layer sizes

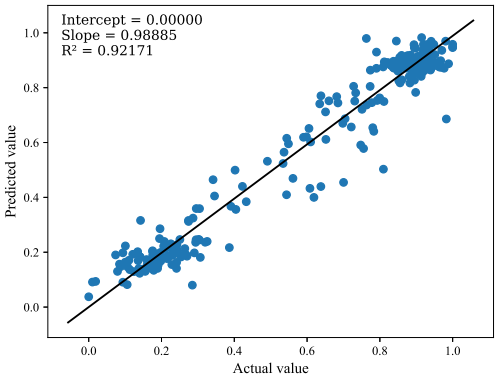
Table 5.17: p-values for Figure 5.29

	5	**10**	**25**	**50**
10	0.28378			
25	0.59969	0.07483		
50	0.71719	0.11199	0.92344	
75	0.42039	0.01911	0.66273	0.34783

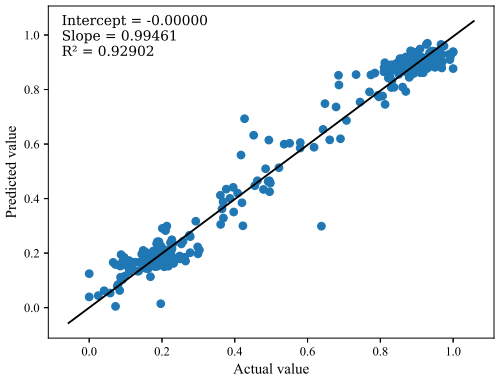
The difficulty encountered by SNNs with smaller hidden layers attempting to simulate complex environments can be seen in Figure 5.30. In Figure 5.30a, the accuracy of the simulator was far lower than for every other hidden layer size, clearly indicated by the R^2 value. This result reinforces the hypothesis that, while BNS may be able to train controllers effectively with a slightly under-performing simulator, if simulator accuracy is desired, it is vital that the SNN be large enough to handle the complexity of the simulation.



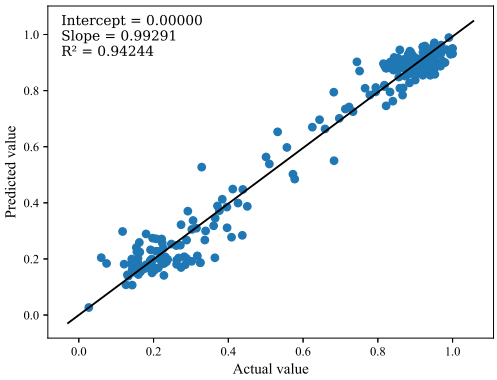
(a) 5 neurons in the hidden layer



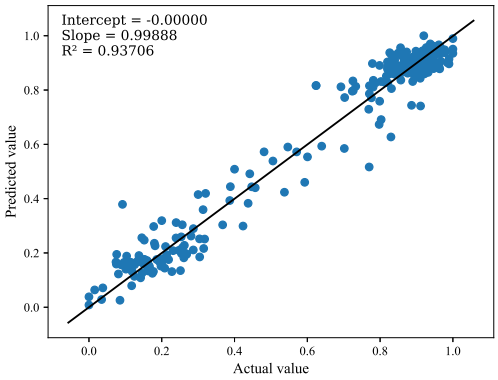
(b) 10 neurons in the hidden layer



(c) 25 neurons in the hidden layer



(d) 50 neurons in the hidden layer

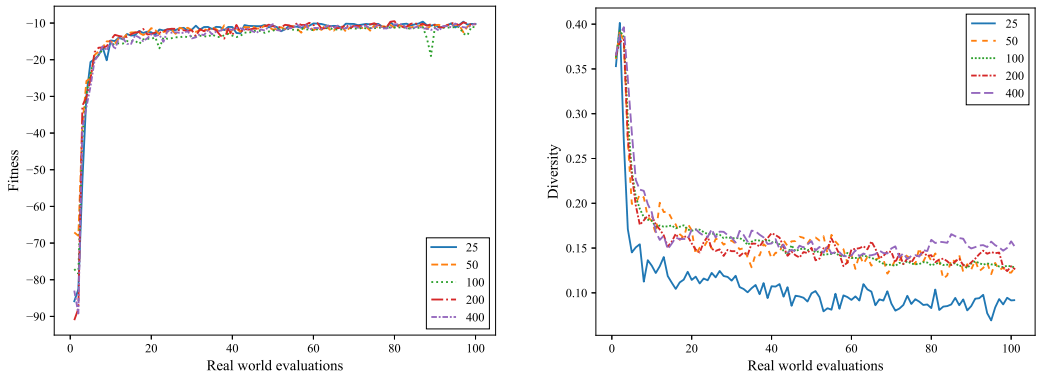


(e) 75 neurons in the hidden layer

Figure 5.30: Sensor SNN predicted vs actual value with different hidden layer sizes

5.3.12 Population Size

Figure 5.31a shows that, for this problem, the size of the population had little bearing on the performance of BNS (Table 5.18). The algorithm can, therefore, be implemented with any population size, though the use of a larger population may be beneficial as the complexity of problems and size of the solution space increases. As can be expected, small populations cause diversity within the population to be decreased (Figure 5.31b).



(a) Fitness over time with different population sizes

(b) Diversity over time with different population sizes

Figure 5.31: The effects of various population sizes

Table 5.18: p-values for Figure 5.31a

	25	**50**	**100**	**200**
50	0.17613			
100	0.25805	0.02324		
200	0.40354	0.58945	0.04841	
400	0.97052	0.36322	0.16238	0.52014

5.4 Real-World Results

This section presents plots of the paths followed by evolved controllers in the first real-world experiments using BNS to evolve closed-loop controllers. These results are represented by item 6.2 in Figure 5.2. The purpose of these results is to validate that the approaches are able to transfer from the fake real world to the real world. The results in

this section were obtained using the parameter values shown in Table 5.19.

Table 5.19: Parameters for real-world evaluation

Parameter	Value
Population size	100
Mutation rate	0.1
Mutation magnitude	0.05
Controller type	Feedforward
Controller ANN hidden neurons	10
Weight initialisation range	[-10,10]
Crossover operator	SBX
Islands	None
Augment data	True
Headless Chicken probability	0.25
Intermittent population reset	False

Figure 5.32 shows the paths followed by five controllers, evolved using BNS, in the real world, each starting at the position marked with a circle. It is clear that many controllers developed a *handedness*, with several preferring to turn in only one direction. This behaviour is very similar to that observed by Pretorius et al. (2013).

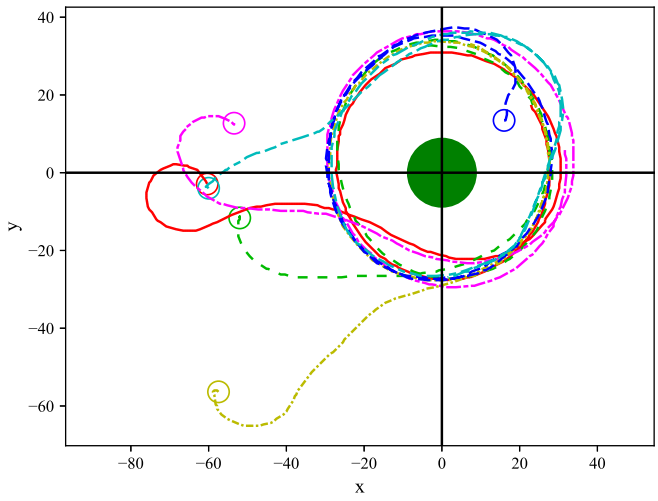


Figure 5.32: Real-world paths followed by evolved controllers

One major difference between these and previous results is the time taken for the behaviour to emerge. Previous experiments required data to be collected in order to build a simulator before training could begin. In addition to this, the evolutionary process was

allowed to run in simulation for 2000 generations (Pretorius et al., 2013). BNS not only developed a simulator without any prior data gathering but, in some cases, was able to evolve controllers exhibiting the correct, though not perfect, behaviour after only three real-world evaluations, which takes around three minutes. Figure 5.33 shows the path followed by such a controller on the fourth real-world evaluation.

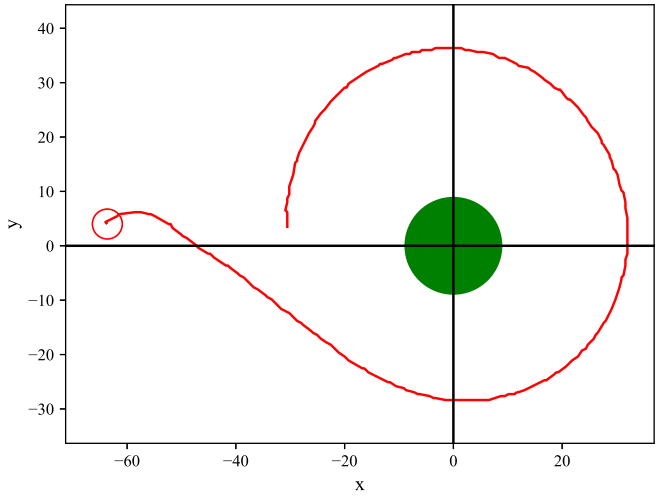


Figure 5.33: Real-world path followed by robot on the fourth real-world evaluation

Figure 5.34 shows composite images of the robot performing its task in the real world, discovering a solution by the fourth real-world evaluation. Each image shows one real-world evaluation.

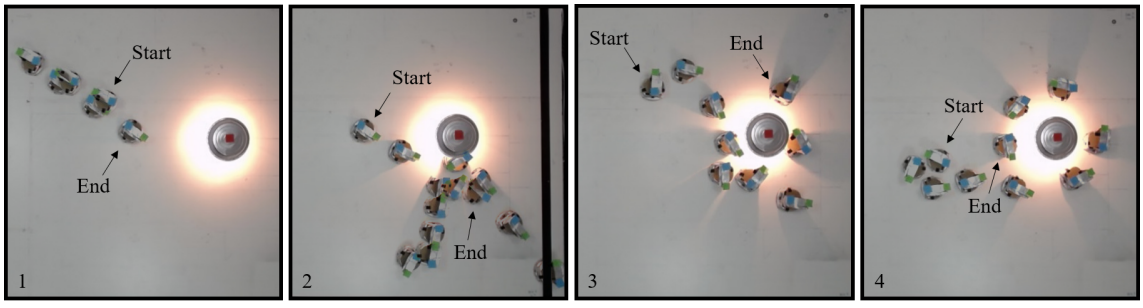


Figure 5.34: Composite images of robot in the real world

5.5 Conclusion

In this chapter, BNS was shown, for the first time, to be effective in evolving closed-loop controllers for a light-following robot. This shows that BNS may be effective for use in evolving more complex controllers. While this is an important step towards proving BNS's applicability to complex tasks, the task solved in this investigation was not particularly complex. It would thus be prudent for further research to be conducted on the evolution of controllers tasked with solving more complex problems.

In this investigation, diversity within the population of controllers has been shown to be an important factor for the overall performance of BNS; this is likely because higher diversity allows a larger area of the search space to be searched for viable solutions. There are various methods of introducing this diversity. Headless Chicken mutation and large weight initialisation ranges do this by directly introducing wider ranges of values into the population, while Island EAs aim to evolve multiple solutions independently. These three methods were all shown to have positive effects on performance. Intermittent population resets had a very clear impact on performance over time, but almost none on the final performance.

The accuracy of the simulator is also central to the performance of the algorithm. Section 5.3.3 showed that performance is improved when data is augmented to increase the rate of training data acquisition. Interestingly, the improved performance was not due to higher overall simulator accuracy; augmentation caused decreased final accuracy. The improved performance early in the evolution process caused by data augmentation likely meant that the simulator's accuracy improved rapidly earlier in the evolution process, allowing it to evolve more transferable controllers earlier.

The size of the SNN did not have a significant impact on the rate of improvement for this problem, but a larger SNN is suggested as a safe choice since it allows the simulator to learn to simulate the real world more accurately. In this case, however, the simulator's inaccuracy did not have too great an impact on performance, possibly due to the simplicity of the problem. The size of the real world tournament also had no significant effect on performance since real-world evaluations only gather data for the dynamic simulator and do not affect the evolution process in any other way.

In most cases, the number of hidden neurons in the controllers did not have a significant impact on performance; however, the use of recurrent connections was shown to have a negative impact. The choice of network architecture is likely highly problem dependent, and an appropriate structure should be chosen for the problem to be solved.

Simulated Binary Crossover was shown to offer far improved performance over both of the other evaluated methods of crossover. It should be at least considered for use in all BNS applications.

Finally, the real world results showed for the first time that BNS is able to rapidly evolve closed-loop controllers that transfer effectively to the real world and bridge the reality gap, as was done previously using a static SNN. Controllers capable of solving the problem were able to evolve in only four real-world evaluations. The behaviours exhibited by the controllers was similar to that seen in previous research by Pretorius et al. (2013); robots circled the light and often developed a handedness by preferring to turn only in one direction.

Chapter 6

Damage Recovery for Closed-loop Controllers

6.1 Introduction

The investigation presented in this chapter followed directly from investigations A (Chapter 4) and B (Chapter 5). Investigation A showed that BNS is able to recover from damage to the real-world robot for which controllers were being developed, and investigation B showed that it is possible for BNS to be used to evolve closed-loop controllers. Investigation C then aimed to evaluate the damage recovery ability of closed-loop controllers evolved using BNS. Figure 6.1, item 7 shows the steps in evaluating BNS's damage recovery ability.

Section 6.2 discusses the types of damage that were evaluated and the adaptations that were evaluated for their ability to improve BNS's damage recovery, followed by Section 6.3, which discusses their implementation. Sections 6.4 and 6.5 present the results of evaluation in simulation and the real world, respectively. Finally, Section 6.6 concludes the chapter.

6.2 Damage Types and Adaptations

This section discusses the types of damage which were inflicted on the robot, as well as the adaptations that were evaluated. It is represented by item 7.1 in Figure 6.1. Since

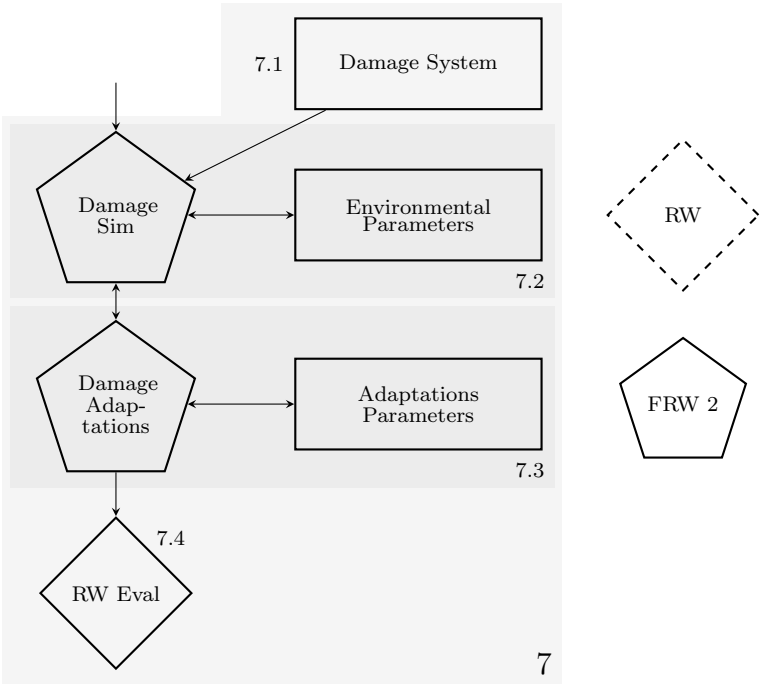


Figure 6.1: Investigation C focus

closed-loop controllers are able to make *decisions* based on information gathered from the world around them, the types of damage from which such a controller may be able to recover are much more varied than the simple damage evaluated in investigation A, where an open-loop controller was used. It was also possible to inflict damage on the sensors, which the robot uses to gather data from its environment.

6.2.1 Types of Damage

The types of damage to the robot's wheels which were evaluated are those where one wheel:

- is slowed to a percentage of its intended speed,
- stops spinning intermittently,
- is only able to spin at the minimum wheel speed, and
- is only able to spin at the maximum wheel speed,

or where both wheels are slowed to a percentage of their intended speeds.

The types of damage inflicted on the sensors are sensors which:

- always return the maximum sensor value,
- always return the minimum sensor value,
- intermittently return the minimum sensor value, and
- have more noise than an undamaged sensor.

6.2.2 Adaptations

Four adaptations previously evaluated in investigation A, introduced in Chapter 3, were evaluated for the evolution of closed-loop controllers (Figure 6.1, item 7.3):

- A sliding window of training data.
- Reset the population of controllers when damage occurs.
- Reset the simulator when damage occurs
- Increase the mutation rate and magnitude when damage occurs.

6.3 Implementation Details

For this investigation, all implementation details not explicitly discussed, and the problem to be solved, were implemented as in investigation B (Chapter 5). The discussion in this section focuses solely on the implementation of damage for this investigation.

6.3.1 Damage

Incorrect value damage

For all cases when damage causes motor or sensor values to be incorrect, the values are changed in software without informing the BNS process. This allows the effects of damage to be evaluated without inflicting actual damage on the physical robot.

Intermittent damage

For cases of intermittent damage, once every m motor commands or sensor readings the value is set to the minimum value, where m is a parameter of the experiment.

Increased sensor noise

When gathering data from the sensors for the construction of a static sensor simulator, the distribution of noise in the sensors was calculated. This data was then used to double the amount of noise experienced by the robot's sensors.

6.3.2 Time of Damage Application

It was observed that, in general, BNS was able to evolve a solution to the problem in fewer than ten real-world evaluations. Therefore, damage was applied after twenty real-world evaluations to allow ample time for the discovery of solutions to the problem. This step is represented by item 7.2 in Figure 6.1.

6.3.3 Parameter Evaluation

Parameters

Parameter configurations in this investigation were evaluated as in investigation A. The parameters investigated included those for the damage recovery adaptations, as well as those for the Headless Chicken, intermittent population reset, island, and data augmentation adaptations. The parameters from the previous investigation were evaluated to establish any impact they may have had on damage recovery for closed-loop controllers.

Default parameter configuration

As with investigation B, a default parameter configuration was chosen and varied in order to evaluate the effects of the parameter's performance.

Presentation and analysis

As with investigation B, each parameter configuration was run thirty times and a representative statistic calculated using equation (3.1) on page 58.

Table 6.1: Default parameter configuration

Parameter	Value
RW evaluation limit	100
RW tournament size	0.1
Controller tournament size	0.1
Population size	100
Mutation rate	0.1
Mutation magnitude	0.05
Controller ANN hidden neurons	5
Weight initialisation range	[-5,5]
Simulator ANN hidden neurons	25
Controller network type	Recurrent

6.4 Results and Discussion

In this section, the results of evaluations of BNS's damage recovery are presented. A large number of configurations were evaluated and their results aggregated, and a representative selection of these results presented. Results in this section are presented identically to those in Chapter 5.

6.4.1 Performance without adaptations

Figure 6.2 shows the performance of BNS without any adaptations, over time, with damage inflicted on the robot's wheels. Slowing down both of the robot's wheels had the smallest impact on performance, which was to be expected since the robot's symmetry was maintained. Slowing the robot's left wheel to either the minimum speed or some proportion of its desired speed had similar effects, though it took longer to recover when the motor was set to the minimum speed. Setting one of the wheels to the maximum speed had by far the largest impact of any damage type, likely because controllers evolved to favour turning counter-clockwise, meaning that the left wheel was required to spin slower. In cases when the left wheel was forced to the maximum speed, the robot's handedness and preference for turning left meant that it was unable to turn towards the light and drove directly away from it. Finally, intermittent damage caused only a minor initial drop in performance, but performance degraded over time. This decreasing performance is likely because the SNNs did not have any recurrent connections and were, therefore, unable to

simulate temporal damage, such as intermittent damage. As the simulated controllers were increasingly finely tuned, the intermittent damage had an increasing impact on performance. The negative effects of intermittent damage highlighted the importance of the selection of an appropriate topology for SNN construction so that all aspects of reality are able to be simulated.

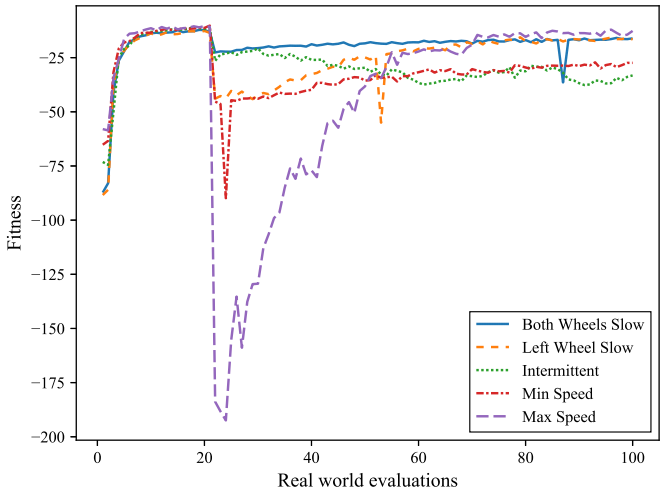


Figure 6.2: Performance over time using BNS for recovery from different wheel damage types

Figure 6.3 shows the performance of BNS over time, with damage inflicted on the robot's sensors. BNS was able to recover from each type of damage. The minimum reading and noisy reading damage types had the most substantial impact on performance. Max reading damage had a much smaller impact, and intermittent sensor damage even less so. The final fitness value for the noisy sensor damage was the lowest, likely since the controller was performing as close to optimally as possible, given the amount of noise being introduced into the sensor readings.

Figures 6.4a and 6.4b show the effects of a slow left wheel and noisy sensors, respectively - both without any BNS adaptations implemented and in the real world. Each figure shows the path followed by the robot after having learned to solve the problem, after damage is inflicted, and once the system had *attempted* to recover from damage. In every plot of a robot's real-world recovery from damage, the number of real-world evaluations taken for recovery is given in parentheses.

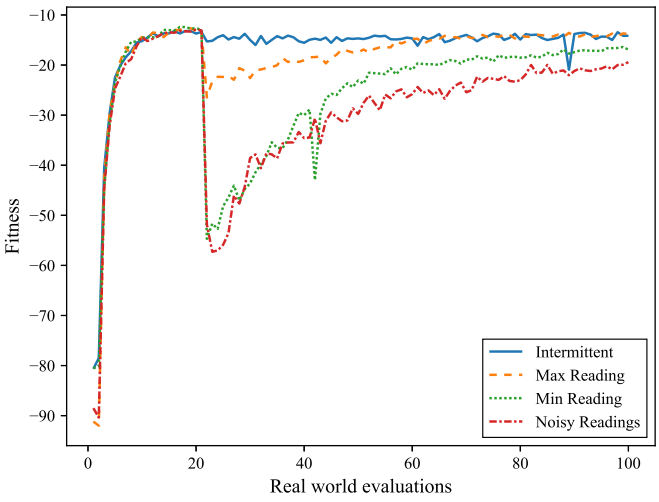


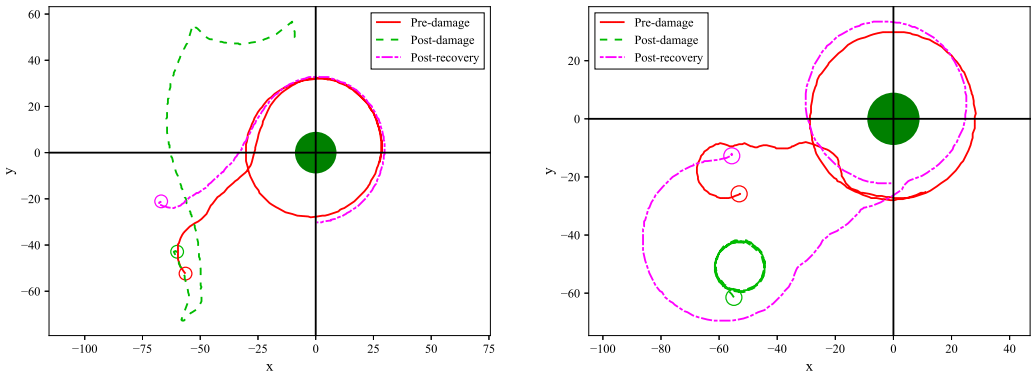
Figure 6.3: Performance over time using BNS for recovery from different sensor types

Feed-forward controllers were used for the real-world evaluations presented in this section since they were found, in investigation B, to offer significant performance benefits over recurrent controllers. These controllers allowed a clearer comparison to be drawn between the damage recovery performance of BNS with and without the adaptations discussed in this investigation, while minimising the influence of the poor selection of controller topology.

Post-damage during its execution, the robot in Figure 6.4a reached the boundaries of the testing area. When this occurred, the robot was rotated to a random angle towards the centre of the testing area and allowed to continue executing, which led to the three sharp angles visible in the path. The system did recover from the damage, but the post-recovery path followed by the robot was inefficient; the left wheel of the robot was slowed, yet the controller continued to prefer clockwise rotation, which required the left wheel to spin faster than the right. This preference was likely due to a lack of diversity in the population which rendered it unable to discover new counter-clockwise behaviours.

Noisy sensor damage had a very large impact on the path followed by the robot. Figure 6.4b shows that the robot drove in a tight circle, unable to move towards the light immediately after damage was applied. The algorithm was able to recover from this damage in twenty-three real-world evaluations. The final path did not make the tight

turns that were made before damage. This may have been because faster turning requires individual readings to be considered reliable, whereas, with large amounts of noise, more readings need to be taken before decisions are made in order to increase the confidence in those decisions.



(a) Paths followed by a robot experiencing left wheel damage (16 RW evaluations for recovery)

(b) Paths followed by a robot experiencing noisy sensor damage (23 RW evaluations for recovery)

Figure 6.4: Paths followed in the real world by robots experiencing different damage types

These results show that BNS has the ability to recover from damage, even before any adaptations have been made to the algorithm; the same as was found in investigation A. This inherent damage recovery is a powerful property of the algorithm; while other methods of ER require the implementation of specific damage detection and recovery systems, systems evolving robot controllers using BNS have this built in from the start.

6.4.2 Sliding Window

Investigation A showed that the use of a sliding window is beneficial for damage recovery with BNS. The results in this section show that this held when BNS was used to evolve closed-loop controllers. It is visible in Figure 6.2 that, of the types of damage evaluated, the *maximum wheel speed* damage had the largest impact on performance. Figure 6.5 shows that the use of a window of size 150 or 300 offered a highly significant ($p < 0.005$) (Table 6.2) improvement in recovery performance over no window, allowing BNS to recover almost entirely within ten real-world evaluations.

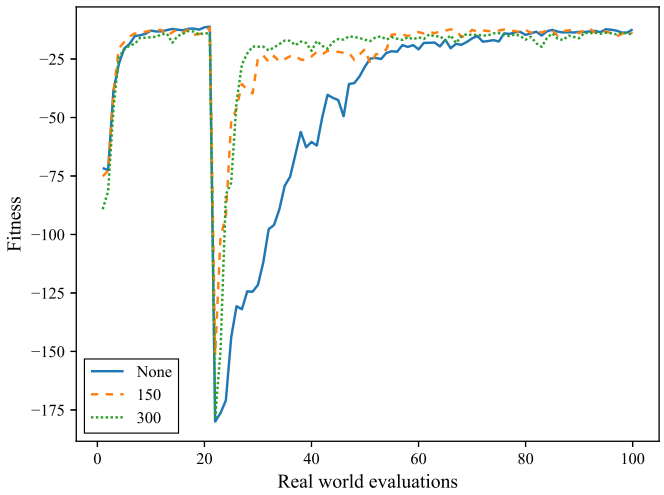


Figure 6.5: Performance over time with a wheel spinning the maximum speed and different sliding window sizes

Table 6.2: p-values for Figure 6.5

	None	**150**
150	0.00027	
300	0.00168	0.19073

The ideal size of this window was not obvious. Several sizes were, therefore, evaluated for each type of damage. Figure 6.6a shows the effect of different window sizes on a robot which sustained *noisy sensor* damage. Figure 6.6b shows the effect of different window sizes on a robot which sustained *left wheel* damage.

Table 6.3: p-values for Figure 6.6

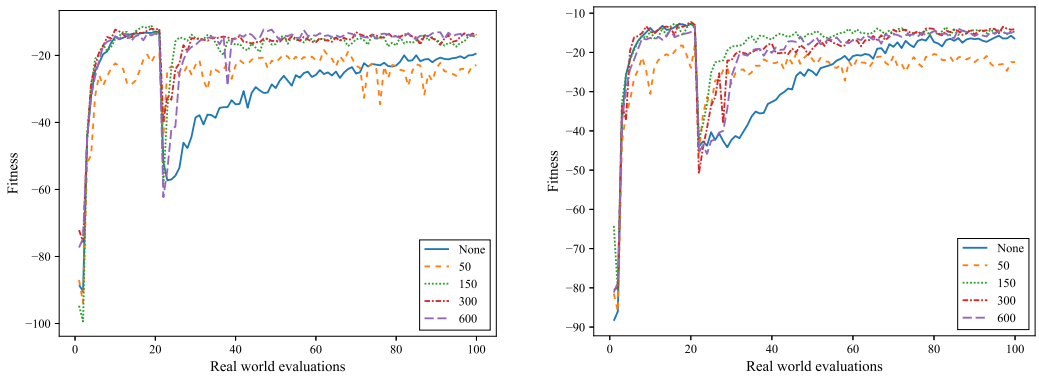
(a) p-values for Figure 6.6a

	None	**50**	**150**	**300**
50	0.48252			
150	0.00007	0.00000		
300	0.00002	0.00000	0.23399	
600	0.00005	0.00000	0.78446	0.52014

(b) p-values for Figure 6.6b

	None	**50**	**150**	**300**
50	0.47335			
150	0.00025	0.00000		
300	0.00275	0.00045	0.84180	
600	0.00868	0.00250	0.68432	0.61001

It is clear from these results that there is a lower bound to the size of the sliding window which is able to provide a benefit to BNS. A window of size 50 allowed BNS to recover more quickly than it was able to without a window, but caused the algorithm's



(a) Noisy sensor readings

(b) Damage to left wheel

Figure 6.6: Performance over time with different sliding window sizes

overall performance to be significantly ($p < 0.005$) worse than windows of any larger size. As the size of the window tends towards the size of the full set of data, the performance benefits will gradually decrease until the window is the same size as the data set. Since this fall-off in performance is likely to be more gradual than that which is found when the window is too small, there is less risk of choosing a window size which is too large, than one which is too small.

The ideal window size is likely problem-dependent since, in investigation A, a window of size 150 was found to perform better than 300, whereas the opposite was true in this case. The range of values which did offer significant improvement was large, however, and in general, the use of a window improved BNS's performance.

It should be noted that a sliding window was not effective for all types of damage and could not improve performance in situations where it is impossible for the simulator to simulate the robot accurately, such as with intermittent wheel damage. In the case of intermittent wheel damage, a sliding window had a very significant ($p < 0.00001$) negative impact on performance. Figure 6.7 shows the algorithm attempting to recover from intermittent damage with and without a sliding window.

BNS's recovery from every evaluated type of damage, other than intermittent, was improved through the use of a sliding window. Intermittent damage likely caused an increased level of noise in the training data. When using a sliding window, this noise makes up a larger portion of the training data being used to train the simulator.

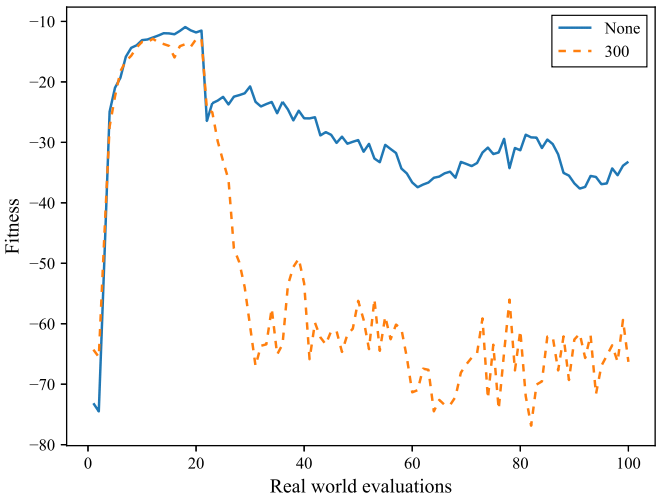


Figure 6.7: Performance over time with intermittent damage to the left wheel with a sliding window of size 300

6.4.3 Population and Simulator Reset

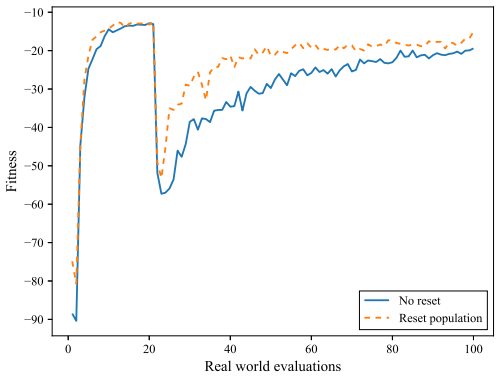
Population Reset

Figure 6.8 shows that resetting the controller population significantly ($p < 0.05$) improved BNS's ability to recover from damage. This was likely due to the increased diversity, as seen in Figure 6.8c, which caused the EA to explore more of the search space to identify viable solutions. This was not the case for every experiment, but the worst-case performance of BNS using a population reset was not significantly different from BNS without the reset, meaning that this adaptation is unlikely to cause a decrease in performance; at worst it causes no change.

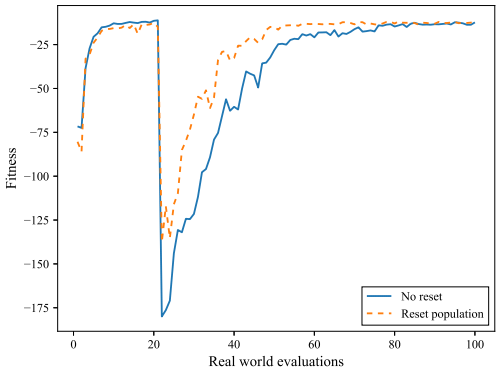
In cases where damage does not require a large change in controller behaviour, resetting the population had a momentary negative effect on performance, though this was quickly rectified (Figure 6.9).

Simulator Reset

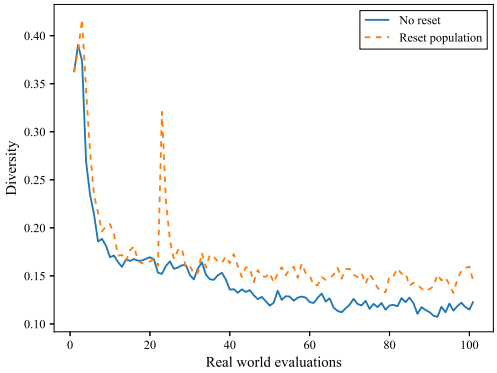
Resetting the simulator did not have a significant impact on the damage recovery performance of BNS (Figure 6.10). This result matches similar results obtained in investigation A. It was likely because the trained simulator, which may have converged to a solution,



(a) Performance over time with noisy sensor readings



(b) Performance over time with the left wheel spinning at the maximum speed



(c) Diversity over time

Figure 6.8: The effects of resetting the population after damage

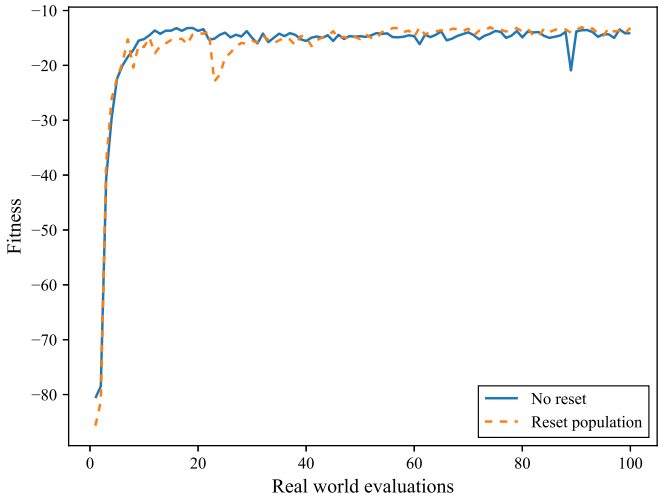


Figure 6.9: Performance over time with intermittent sensor damage and a population reset after damage

was no less incorrect than a random simulator in cases of extreme damage. Both the random and trained simulators, therefore, needed to change significantly to be able to simulate the new state of the real world.

6.4.4 Mutation Changes

Mutation Magnitude

The only significant ($p < 0.05$) impact found through increasing the mutation magnitude was a negative one when compared to the default configuration (Figure 6.11a). This was likely because a too-large increase in the mutation magnitude reduced the rate of convergence of the population enough that it had an adverse effect on performance. Figure 6.11b shows that the increased mutation magnitude was unable to increase the diversity of the population, which is likely the reason that the algorithm did not exhibit any improvement.

Table 6.4: p-values for Figure 6.11a

	0.05	**0.1**
0.1	0.12235	
0.2	0.47335	0.03917

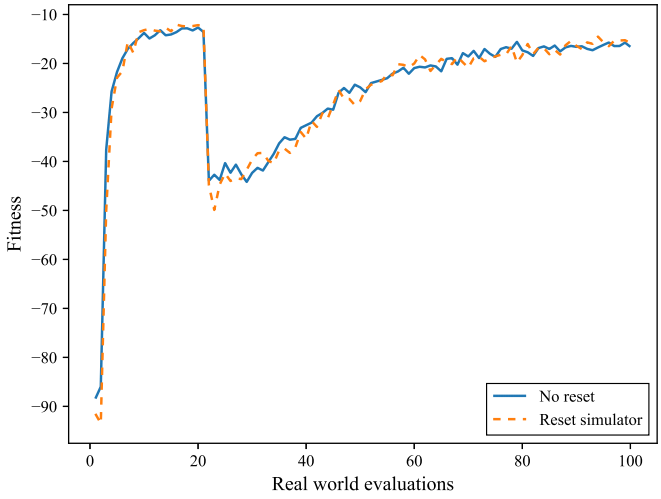
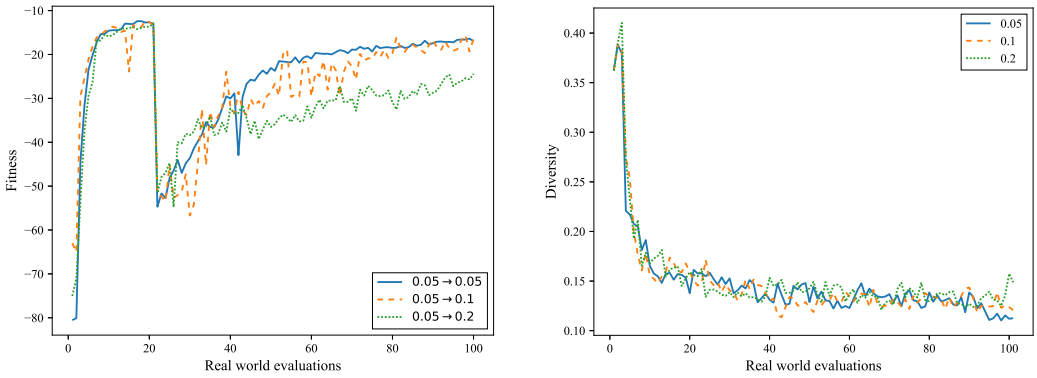


Figure 6.10: Performance over time with left wheel damage and a simulator reset after damage



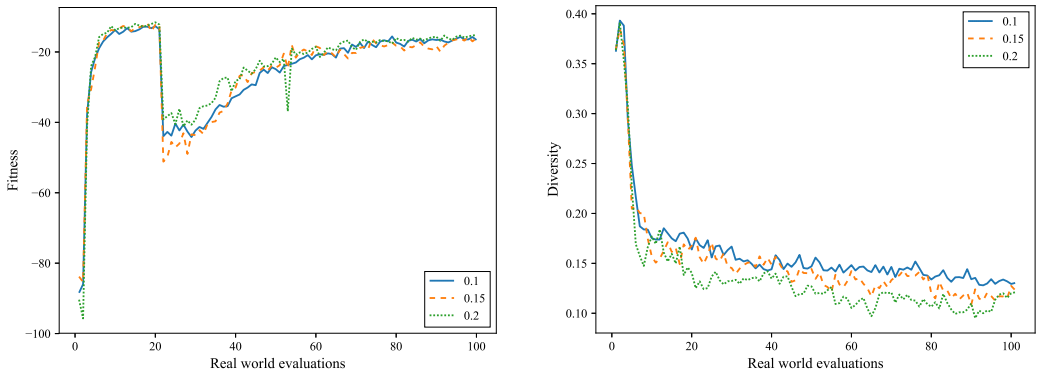
(a) Performance over time with a sensor returning the minimum value

(b) Diversity results

Figure 6.11: The effects of a mutation magnitude that increases after damage

Mutation Rate

Increasing the mutation rate (Figure 6.12a) after damage had occurred had no significant impact on BNS's damage recovery performance (Table 6.5). This was likely because the increased rate had no effect on the rate of convergence of the population (Figure 6.12b).



(a) Performance over time with left wheel damage

(b) Diversity over time

Figure 6.12: The effects of a mutation rate that increases after damage

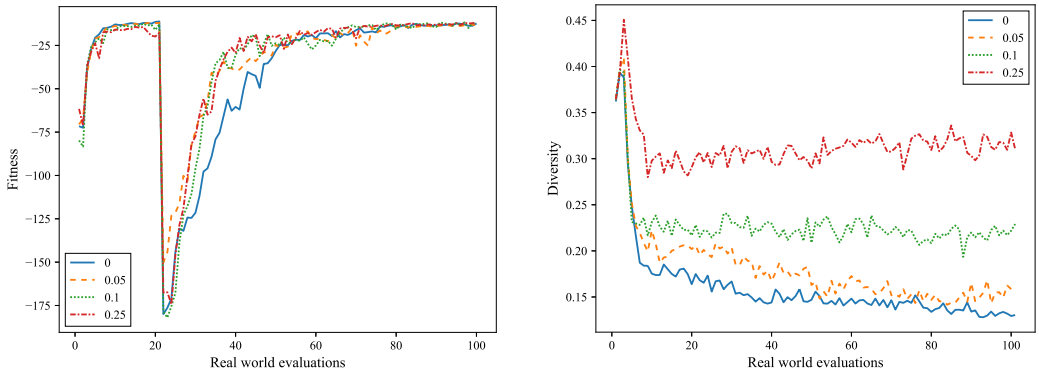
Table 6.5: p-values for Figure 6.12a

	0.1	**0.15**
0.15	0.71719	
0.2	0.84180	0.75059

6.4.5 Other parameters

Headless Chicken

The use of a Headless Chicken probability of 0.05 caused a marginal improvement in performance (Figure 6.13a). This result was similar to those found in investigation B, where only small Headless Chicken probabilities were beneficial. The additional diversity introduced through this adaptation (Figure 6.13b) likely allows a higher degree of exploration, leading to the faster discovery of better solutions.



(a) Performance over time with the left wheel spinning at the maximum speed

(b) Diversity over time

Figure 6.13: The effects of Headless Chicken mutation on damage recovery

Table 6.6: p-values for Figure 6.13a

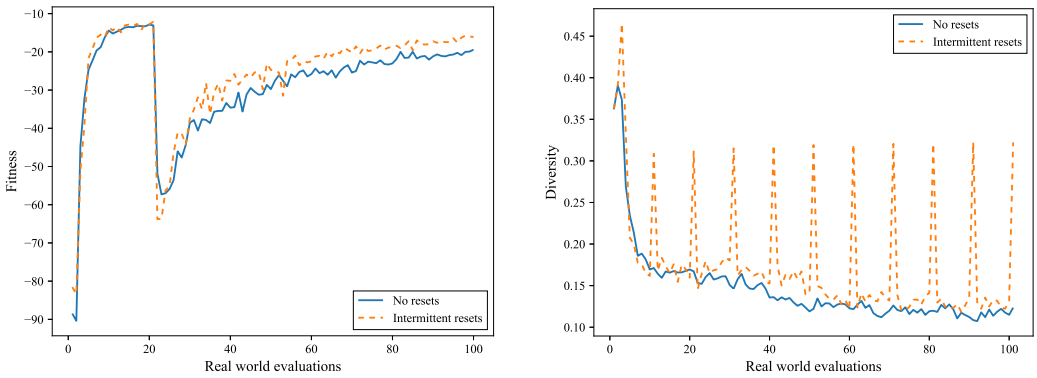
	0	**0.05**	**0.1**	**0.25**
0.05	0.07978			
0.1	0.30418	0.65204		
0.25	0.15367	0.59969	0.57929	

Intermittent Population Reset

The intermittent reset of the controller population was evaluated and found to offer a non-significant improvement (Figure 6.14a). The increases in diversity are clearly visible in Figure 6.14b, but the population rapidly converges to pre-reset levels of diversity between each reset. In investigation B, this adaptation was found to be particularly volatile, so there may exist a specific combination of reset parameters that cause an improvement, but the small benefits exhibited do not warrant the effort required to identify these problem-dependent values.

Island Evolutionary Algorithms

No significant improvement to damage recovery was found when using Island EAs over a single population (Fig 6.15a). There was, however, no evidence of Island EAs reducing BNS's damage recovery performance. Since the adaptation was found in investigation A to improve performance for BNS in general, it is recommended that an island is used, even

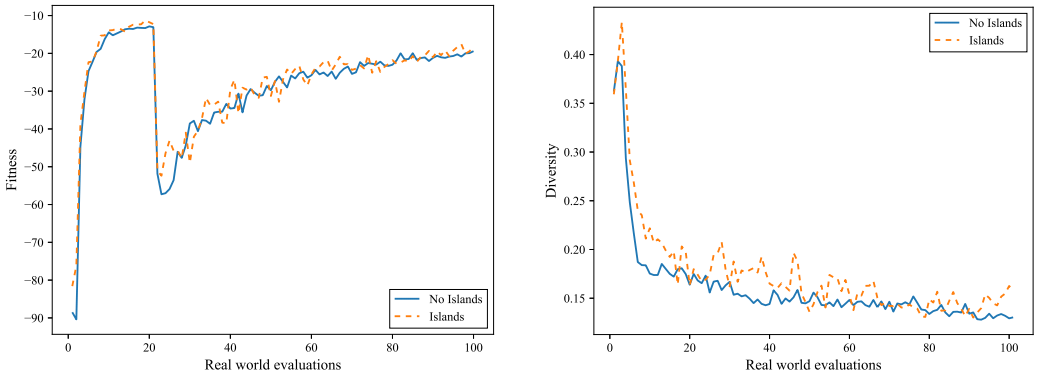


(a) Performance over time with noisy sensor damage

(b) Diversity over time

Figure 6.14: The effects of intermittent population resets

if it offers no direct advantage in the case of damage to the robot, since the adaptation marginally increases diversity in the population (Figure 6.15b).



(a) Performance over time with noisy sensor damage

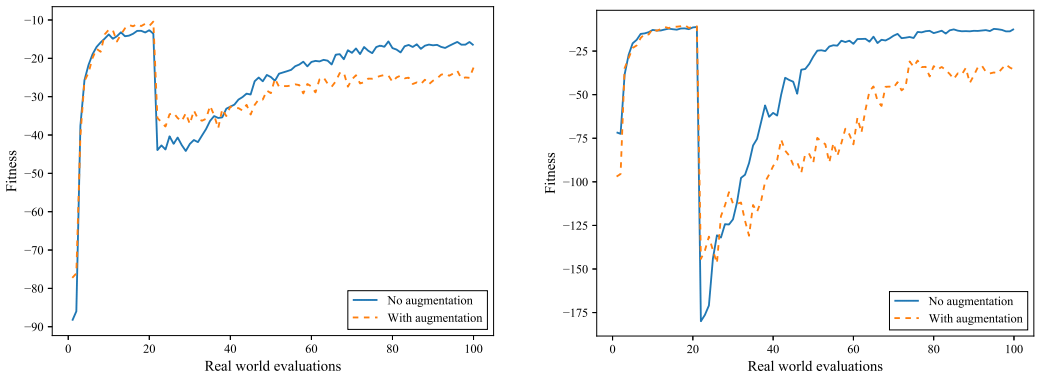
(b) Diversity over time

Figure 6.15: The effects of an Island EA

Data Augmentation

Data augmentation had significant ($p < 0.05$) adverse effects on BNS's damage recovery (Figure 6.16). This was likely because the use of data augmentation is based on the assumption of the robot's symmetry. Any damage that disrupts this symmetry makes this

assumption false.



(a) Performance with a damaged left wheel

(b) Performance with the left wheel spinning at the maximum speed

Figure 6.16: The effects of data augmentation on performance

If data augmentation is used but stopped when damage is detected, a sliding window should be a requirement (Section 6.4.2) since data augmentation increases the rate of data acquisition. If augmentation is stopped after damage occurs, and a sliding window is not used, it would take longer for the simulator to improve again than if augmentation had been used. This is because if augmentation allowed each pattern to be augmented into k patterns for t time periods, it would take tk time periods to collect even the same amount of data once augmentation is disabled.

6.5 Real-World Results

The results presented in this section follow the same format as those in Section 6.4.1; they show the path followed by the robot before and after damage, as well as its path after having recovered from damage, with each of these paths starting at the position marked with a circle. Each plot also includes the number of real-world evaluations taken for recovery, in parentheses. These results are the output of item 7.4 in Figure 6.1.

The three adaptations which were found to offer benefits were Headless Chicken mutation, a sliding window, and a population reset. Therefore, the real-world experiments presented in this section used these adaptations, with the parameter values given in Table

6.7 in addition to those presented in Table 6.1.

Table 6.7: Real-world parameter configuration

Parameter	Value
Recurrent controller network	False
Headless Chicken probability	0.05
Sliding window size	600
Population reset upon occurrence of damage	True

Figure 6.17 consists of three composite images. Each shows one real-world evaluation of the robot. The evaluations shown are the same as with all other damage plots: the pre-damage, post-damage, and post-recovery evaluations. In this case, the robot's left sensor began to return only the minimum sensor value, and the algorithm was able to adapt and re-learn the problem in eight real-world evaluations.

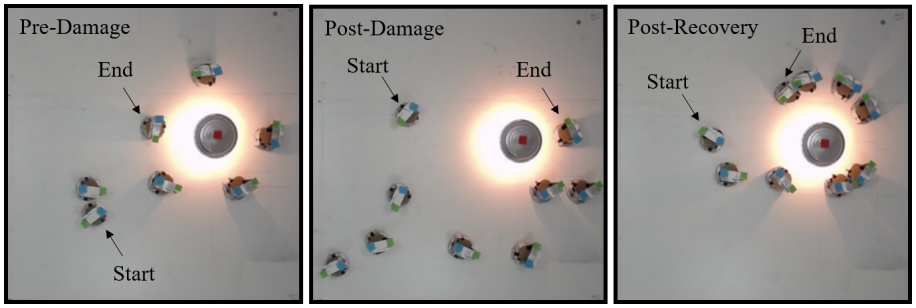


Figure 6.17: Composite images of the robot in the real world, recovering from damage (8 RW evaluations for recovery)

The case where the left sensor returned, only the maximum sensor value had extreme effects on the robot's path (Figure 6.18a). Immediately after the damage was inflicted, the robot was only able to drive in circles. The algorithm was able to recover rapidly from the damage and learn to navigate using only one sensor. The path followed was not as consistently circular, but is impressive given that, in this case, it took only seven real-world evaluations for the algorithm to recover.

Figure 6.18b again shows BNS's ability to recover from damage by discovering new solutions. One of the robot's sensors was damaged and returned only the minimum sensor value. The robot's path then appeared to avoid light, rather than approach it. Once recovered, though, the algorithm discovered that, by looping towards the light source, it

was able to use a single sensor to circle the light successfully.

Figure 6.18c shows a robot which experienced the same type of damage as in Figure 6.4a. In this case, when the robot was damaged it avoided the light and remained far away from it. Once it was allowed time to recover, it developed a new handedness. Where before damage it circled the light in a clockwise direction, afterwards it circled counter-clockwise at a closer distance than before.

Finally, Figure 6.18d shows a robot which experienced the same damage as Figure 6.4b, this time with adaptations implemented. The algorithm took only four real-world evaluations to recover, rather than twenty-three. The robot once again developed a new handedness and circled the light in the opposite direction to before.

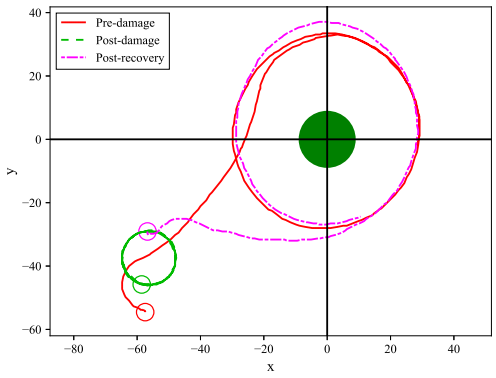
6.6 Conclusion

In this investigation, BNS was shown to be able to recover from damage automatically while evolving closed-loop controllers. This recovery is possible due to BNS's always-learning nature; it is able to make observations about its environment and the robot, and adapt based on these observations.

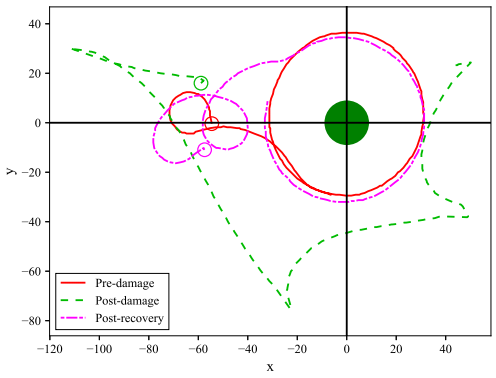
Several adaptations and parameter values were evaluated, but few offered significant positive effects on BNS's damage recovery. The use of a sliding window, a population reset, and a small Headless Chicken probability were the only ones to do so.

As in previous research, a sliding window of training data was shown to be by far the most effective method of improving damage recovery. The use of a window which was too small had an adverse effect on performance; it is, therefore, important that the window be large enough. Several sliding window sizes were evaluated, and it was shown that the risk involved with the selection of a window that is too large is less than the risk involved with selecting one that is too small. It was also found that in cases where the simulator is already unable to simulate the real world, a sliding window may cause performance to become significantly worse. A sliding window is still recommended, however, since the cause of this issue is more likely the simulator topology. It is vital that the chosen SNN topology be appropriate for the aspects of reality that are to be simulated.

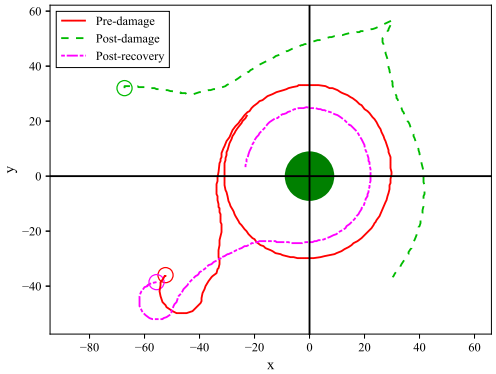
In previous research, resetting the population when damage occurred was shown to



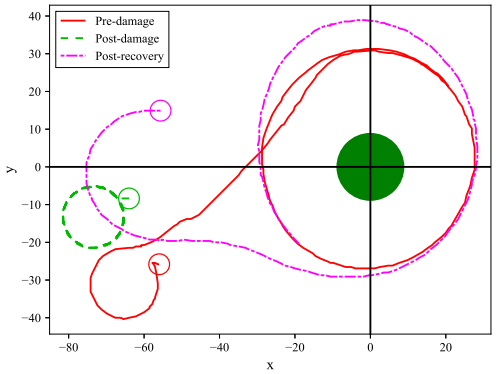
(a) Paths followed by a robot experiencing maximum sensor damage with adaptations implemented (7 RW evaluations for recovery)



(b) Paths followed by a robot experiencing minimum sensor damage with adaptations implemented (9 RW evaluations for recovery)



(c) Paths followed by a robot experiencing damage to its left wheel with adaptations implemented (10 RW evaluations for recovery)



(d) Paths followed by a robot experiencing noisy sensor damage with adaptations implemented (4 RW evaluations for recovery)

Figure 6.18: Paths followed in the real world by robots experiencing different damage types

have no effect on the performance of BNS. In the results presented in Section 6.4.3, it was shown that for specific problems, resetting the population offered a significant performance improvement. More substantial improvements were observed in cases of damage which required very different behaviour from the robot before and after damage; the discovery of new behaviours through exploration of the search space was assisted through

the increase in diversity offered by resetting the population. Since no configuration was found where a population reset negatively affected performance, it is an adaptation that can be recommended for inclusion in a BNS implementation. Resetting the population when damage occurs does, however, require damage to be detected, so the use of the adaptation necessitates the implementation of damage detection systems, which increases the complexity of BNS's implementation.

The use of a Headless Chicken mutation offered a small improvement. Similar results were found in investigation B. As the Headless Chicken probability increases, this improvement was diminished. It is, therefore, suggested that a small Headless Chicken probability is used with BNS implementations since it assists with the introduction of additional diversity into the population, allowing for the discovery of new solutions.

Chapter 7

Damage Recovery for Complex Robots

7.1 Introduction

Investigations A and C have shown that BNS is able to recover from damage not only for simple robots with simple controllers, but also for those with more complex, closed-loop controllers. Investigation D, the focus of this chapter, now aims to evaluate BNS's ability to recover when damage occurs to a robot of much greater complexity than the simple differentially steered robot used in previous investigations.

While the robot is more complex that used in investigation A, the procedure followed to conduct the two investigations is very similar. Figure 7.1, which shows the process followed for this investigation, is therefore the same as the matching diagram in investigation A. Item 2 represents the BNS system being evaluated. Item 3.1 represents the system responsible for simulating damage, and items 3.2 and 3.3 represent the evaluation of different environmental and adaptation parameters in simulation, respectively. Finally, Item 3.4 represents the real-world evaluation of the BNS algorithm.

Research on the evolution of controllers for advanced robots has been conducted by Woodford (2018), and many implementation decisions were based on that research. The BNS implementation (Figure 7.1, item 2) was developed by Woodford (2018) and augmented with a damage system by the author.

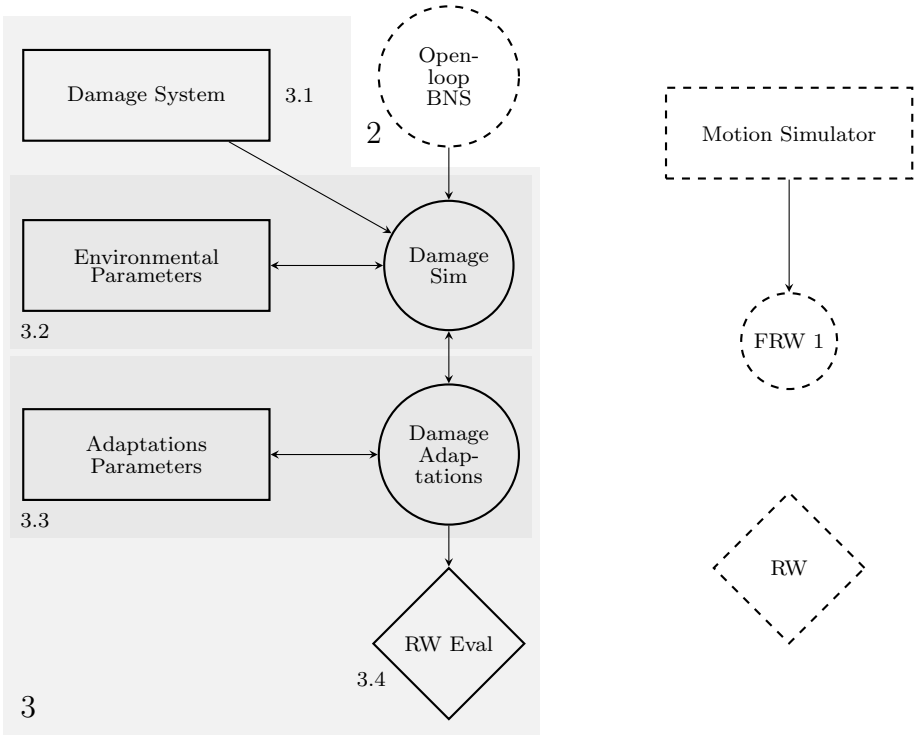


Figure 7.1: Investigation D focus

Section 7.2 discusses the implementation details while Section 7.3 presents the results of initial experiments, determining the effects of damage on the hexapod. Sections 7.4 and 7.5 discuss the results of damage-recovery experiments in simulation and in the real world, respectively. Finally, Section 7.6 concludes the chapter.

7.2 Implementation Details

In this investigation, BNS was implemented to evolve controllers for a hexapod robot, with the objective of moving the hexapod as far in any direction as possible. This section discusses the implementation that was done in order to conduct this investigation. Research on the evolution of controllers for the hexapod was conducted by Woodford (2018). The results of that research were used as the starting point for this investigation, and are discussed further in this section.

Section 7.2.1 discusses the specific hexapod robot which was used, Section 7.2.2 discusses the types of damage the robot can experience, Section 7.2.3 discusses the testing area, and Section 7.2.4 discusses the construction of a controller. Sections 7.2.5 and 7.2.6

discuss the implementation of BNS's EA and hexapod motion simulator. Lastly, Section 7.2.7 discusses details of the parameter evaluation and the analysis of the results.

7.2.1 PhantomX AX Hexapod

The robot used for this investigation was the PhantomX AX Hexapod Mark II (Figure 7.2). This robot is significantly more complex than the Khepera. The Khepera's movement depended only on the speed of its two motors. The hexapod has eighteen independent servo motors, each with a unique ID (Figure 7.3).

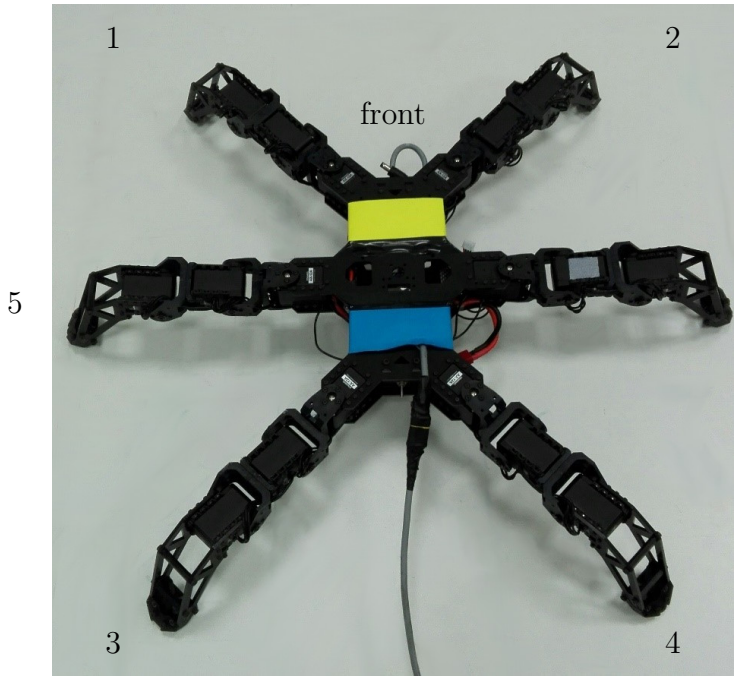


Figure 7.2: PhantomX AX Hexapod Mark II, with leg numbers marked

7.2.2 Damage

Damage for this investigation took the form of broken legs on the hexapod. This was simulated by altering commands to raise the leg to a position where it could not contact the ground (Figure 7.1, item 3.1). The robot's legs are numbered in Figure 7.2. The damage *types*, and their corresponding broken legs, are shown in Table 7.1; these damage types represent all possible damage types, when the symmetry of the robot is taken into

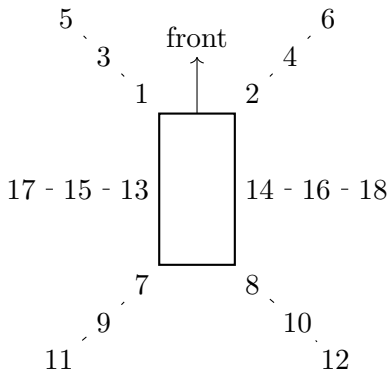


Figure 7.3: Hexapod Servo Configuration

account. Types of damage with more damaged legs are likely to present a greater challenge and have more adverse effects than those with fewer damaged legs.

Table 7.1: Damage types

Damage Type	0	1	2	3	4	5	6	7
Damaged Legs	1	5	1,2	5,6	1,5	1,4	1,6	1,3,5
Working Legs								
Damage Type	8	9	10	11	12	13	14	15
Damaged Legs	1,5,6	1,2,4	1,4,5	1,3,6	1,3,4,5	1,3,5,6	1,4,5,6	1,3,4,6
Working Legs								

Figure 7.4 shows the effects of damage on the paths followed by three evolved controllers (1 - 3), and one hand-designed tripod controller, in the real world robot. The tripod controller walks with a *tripod gait*, which always has at least three of the robot's legs in contact with the ground. The controllers were each executed on an undamaged robot, and one with a broken leg (damage type 0), and their paths were observed. Controllers 1 and the tripod controller experienced the worst decrease in performance, while controller 2 experienced very little change. Controller 3 was unable to move as far, but still exhibited the same side-to-side swaying motion as it exhibited before damage.

It is clear that a single type of damage can have very varied results, depending on the type of controller affected by the damage. This means that the performance of the algorithm and the adaptations are also likely to vary to a greater degree than was observed

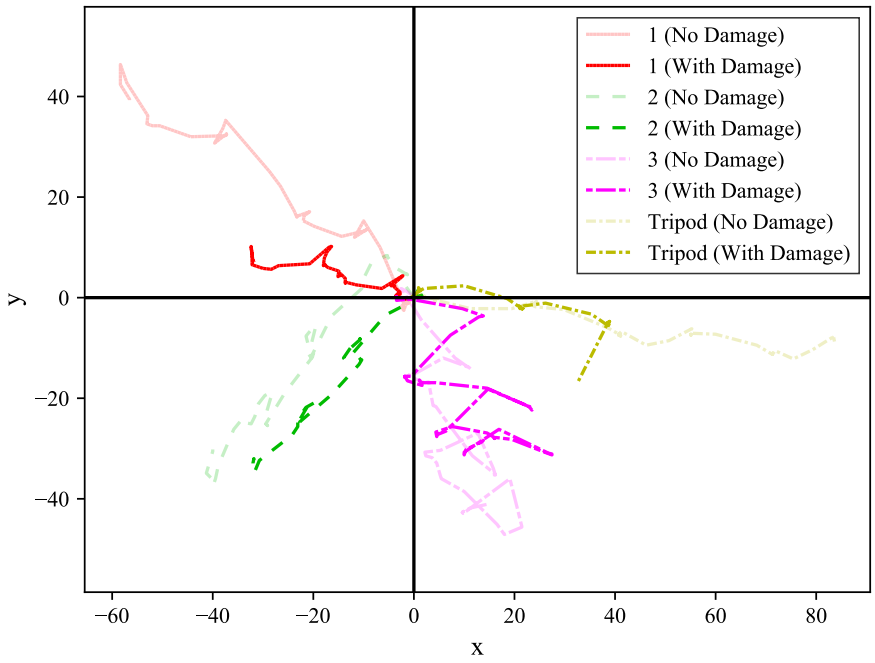


Figure 7.4: The paths followed by different controllers before and after damage

in previous experiments, where the effects of specific damage types on the Khepera robot were more consistent.

7.2.3 Testing Area

The only data required is the robot's change position and orientation based on the commands it executes. The same tracking system as in the previous investigations (Section 4.2.6, page 72) was, therefore, able to be used. Figure 7.2 shows the hexapod affixed with coloured tracking markers.

7.2.4 Controllers

The hexapod robot's servo motor positions are set by executing a *command*. Each command consists of eighteen values which represent the desired position of the robot's servos.

Investigations B and C used closed-loop controllers on a simple robot. This investigation no longer used closed-loop controllers, but rather open-loop controllers, as in investigation A. The goal of the controllers is to cause the robot to move the hexapod as

far as possible in any direction, by walking, or with another form of legged locomotion. This objective is the same as was used by Woodford (2018).

Each open-loop hexapod controller consists of eleven hexapod commands. The robot starts in its default standing stance, with each servo at its half-way position. Each command is then executed in order. Sufficient time is allowed between the execution of each command for the servos to reach their desired positions and for the robot to come to rest.

The values to which each servo can be set must be constrained, since the physical construction of the robot prevents the servos from rotating too far. Mutation of the controllers is therefore also constrained to within these limits.

7.2.5 Motion Simulator

The SNN used to simulate the hexapod robot has a similar topology to those used to simulate the Khepera (Figure 7.5). Instead of two previous and current motor speeds, as was the case with the Khepera, the hexapod simulator takes as input the current and next positions for each of the hexapod's eighteen servo motors. There is no *time* input, as was the case with the Khepera simulator, since the next command is only executed once the desired motor position has been reached. The hexapod also had no requirement for a sensor simulator, since the closed loop controllers utilised for this experiment did not use any sensors on the robot during the execution.

The most notable change, however, is the number of ANNs used for simulation. In previous experiments, one ANN was used for the simulation of each of the robot's changes in x and y positions, and its rotation. This was because, for the Khepera, investigation had shown the configuration to offer good results (Pretorius et al., 2013). In the case of the hexapod the opposite was shown to be true. Therefore, a single network with a hidden layer containing 200 hidden neurons was used to predict all three of the robot's change in x, y, and rotation (Woodford, 2018). Additionally, an *ensemble* of networks was used, which allowed an *uncertainty* value to be determined for each prediction. These values were then used as a part of the evaluation of controllers in simulation (Section 7.2.6).

The SNN also has an additional *dropout* layer. This layer is responsible for causing dropout of the neurons in the previous layer, thereby reducing the risk of overfitting, which is important, given that the set of training data starts very small at the beginning of the

BNS execution.

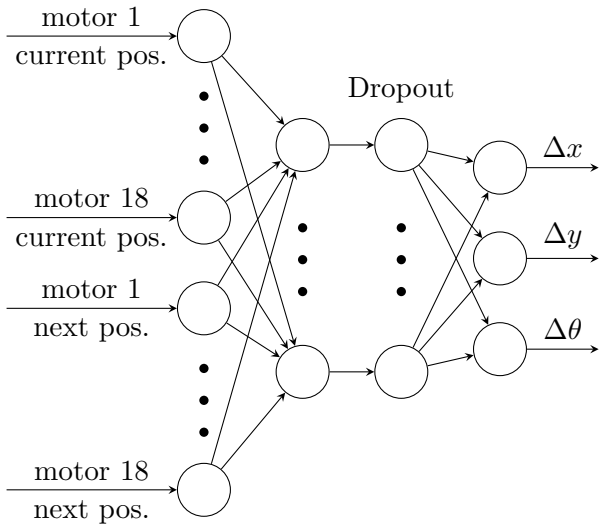


Figure 7.5: Simulator neural networks for the Hexapod

7.2.6 Evolutionary Algorithm

The EA for this investigation was created in a similar manner to those for the others, and all choices in terms of operators and parameter values were based on those found to be effective by Woodford (2018). Tournament selection was used for the selection of individuals for reproduction, which were then combined using simulated binary crossover and mutated with a normally distributed random variable.

The controller fitness function was designed to reflect the goal of achieving the maximum displacement in any direction. The fitness of a hexapod controller is given by:

$$f_h = -\frac{V}{\|p_e - p_s\|_2} \tag{7.1}$$

where p_e and p_s are the robot's final and initial positions, respectively, and V is the penalty value. The controller's penalty is calculated as:

$$V = \sum_{i=1}^{n_c} \sigma_i^2 \tag{7.2}$$

where n_c is the number of commands per controller and σ_i^2 is the variance of the predictions given by the ensemble of simulators for the i^{th} command. The penalty value serves to

reward controllers that have lower uncertainty. The behaviours of such controllers are likely to be simulated more accurately and transfer to the real world more effectively since the networks in the ensemble *agree* on how the controller will behave. This directs evolution towards controllers that are able to transfer well, similar to the transferability approach (Koos et al., 2013a).

As in previous investigations, a method of measuring the population's diversity was required. The population's average individual was calculated as in investigations A, B, and C, and the diversity of the population, d_p, was calculated using equation (7.3):

$$d_p = \ln(\sum_{i=1}^{n_p} \sum_{j=1}^{198} |c_{ij} - \bar{c}_j|) \tag{7.3}$$

where n_p is the number of individuals in the population, each individual has 198 genes (eleven commands with eighteen servo positions per command), c_{ij} is the j^{th} gene of the i^{th} individual, and $\bar{c}_j$ is the j^{th} gene of the average individual. The natural logarithm is taken to make the results more easily interpretable, since they would otherwise fall in a very wide range, and differences between populations with low diversity would be indistinguishable.

7.2.7 Parameter Evaluation

The parameter evaluation in this investigation was carried out as described in Chapter 4. A default parameter configuration was chosen and each parameter then varied and evaluated thirty times in the fake real world (Figure 7.1, item 2). The results of the evaluations in the fake real world were used to determine the overall impact of the parameter on the algorithm's performance.

The fake real world for this experiment (Figure 7.1, FRW 1) was implemented by Woodford (2018). As in previous experiments, the static SNN was constructed as per the standard method of SNN construction: random commands were executed on the real world robot. The data from these experiments were then used to train the SNN.

Parameters

The default parameter configuration for all experiments is presented in Table 7.2. The adaptation and environment parameters investigated, as well as the values evaluated for each, are then presented in Tables 7.3 and 7.4.

Table 7.2: Default parameter configuration

Parameter	Value
RW tournament size	0.7
Controller tournament size	0.5
Population size	400
Mutation rate	0.1
Mutation magnitude	20
Intermittent simulator reset	True

Table 7.3: Adaptation parameter values

Parameter	Values
Sliding window size	None, 150, 300, 600, 1000
Mutation rate	0.1, 0.15, 0.25
Mutation magnitude	20, 30, 40
Reset controller population	True, False
Reset simulator	True, False
Complete restart	True, False
Training data reset	True, False

Table 7.4: Environment parameter values

Parameter	Values
RW evaluation at which damage applied	100
Total RW evalautions	200
Damage type	0,1,..,15

Presentation and analysis

When presenting results, instead of fitness values, the square of the displacement of the fittest controller is shown. This metric is more easily visualised and understood. The BNS algorithm is run for more real-world evaluations when evolving controllers for the hexapod, than when evolving controllers for the Khepera; the results are more difficult

to visualise if all real-world evaluations are shown. The performance is, therefore, shown from the first real-world evaluation after damage occurs (real-world evaluation 101).

The observations for each configuration for the Mann-Whitney U test were calculated using:

$$O_\tau = \sum_{i=101}^{200} (p_{e,i\tau} - p_{s,i\tau})^2 \qquad \tau = 1, 2..30 \tag{7.4}$$

where $p_{e,i\tau}$ is the final location of the robot after the i^{th} real-world evaluation during the τ^{th} run using the parameter configuration. Likewise, $p_{s,i\tau}$ is the starting location for the i^{th} real-world evaluation during the τ^{th} run. This equation is based on equation (3.1) and is once again designed to take into account not only the final performance, but also the rate of recovery of the algorithm.

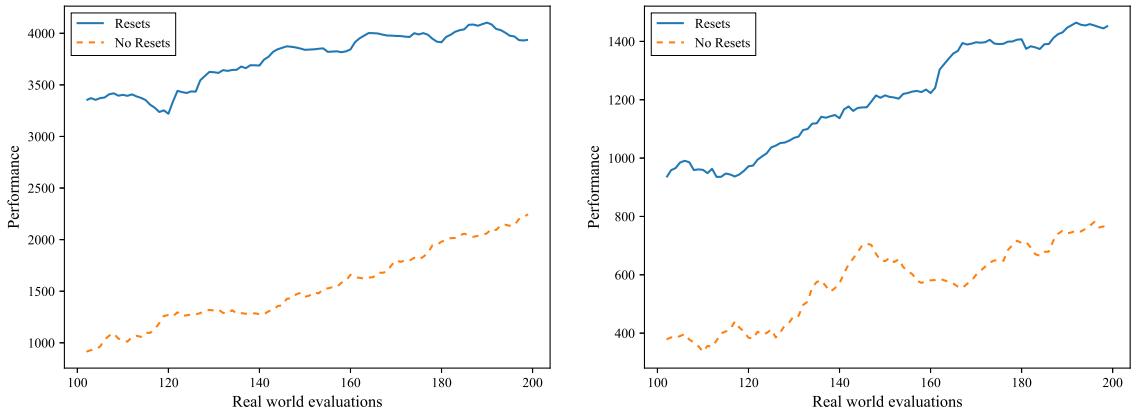
7.3 Initial Results and Discussion

This section presents the results of evaluation of BNS without any damage adaptations (Figure 7.1, item 3.2). These evaluations were performed to discover more about the impact of damage on the hexapod, the impact of the intermittent simulator resets proposed by Woodford (2018) on damage recovery performance (Section 7.3.1), and BNS's ability to recover from damage before damage adaptations were implemented (Section 7.3.2).

7.3.1 Intermittent Simulator Reset

Woodford (2018) found that for more complex robots, intermittently resetting the simulator brought about a significant improvement in performance. Figure 7.6 shows that this is also true when considering damage recovery. Resetting the simulator once every twenty fake-real-world evaluations had an immediate and very significant ($p < 0.0005$) positive impact, possibly due to the simulator reset which occurred at the same real-world evaluation as when damage was inflicted.

These results, along with those found by Woodford (2018) indicate strongly that intermittent simulator resets should be implemented for more complex robots. They bring about improved performance even when damage is not a concern, and allow for improved



(a) Damage Type 0

(b) Damage type 5

Figure 7.6: Post-damage performance when the simulator is reset intermittently with different damage types

recovery when it is. Since BNS continues to function in the same way before and after damage, it makes sense that the same adaptation that improved performance before damage would do so afterwards. All results in Section 7.3.2 and 7.4 were obtained using BNS with intermittent simulator resets.

7.3.2 No Adaptations

This section presents the results of BNS attempting to recover from damage in simulation with no changes made to the algorithm. Figure 7.7 shows these results for each of the sixteen evaluated damage types, which were obtained by aggregating the results of thirty fake real world experiments for each damage type, that is, thirty controllers. The evolution of controllers begins after five real-world evaluations.

At real-world evaluation 100, damage is inflicted. The result of this is clearly visible in the plots where it causes the performance to decrease suddenly. It is clear from the performance plots that the effects of different damage types have differing *severity*. Damage types 0, 1, and 3 have very little effect on the performance of the robot; it continues to improve without a large decrease in performance.

Damage types 4 and 6 - 15 all perform very poorly, and their performance is far worse after damage than before; while their performance does improve over time, the post-

damage performance of the robot in the case of these damage types is too poor for it to be realistically considered to have *recovered*. This group of damage types likely represents those that are severe enough that the robot is unable to recover effectively. The robots experiencing these damage types do begin to recover very slowly but are not able to attain levels of performance near to those achieved before damage. Intuitively, this makes sense, as many of these damage types represent the robot having lost control of three or more of its legs.

Finally, damage types 5 and 2 fall between the other two groups of damage types. The damage has a large impact on performance, but the effects are not as severe as for damage types 6 - 15.

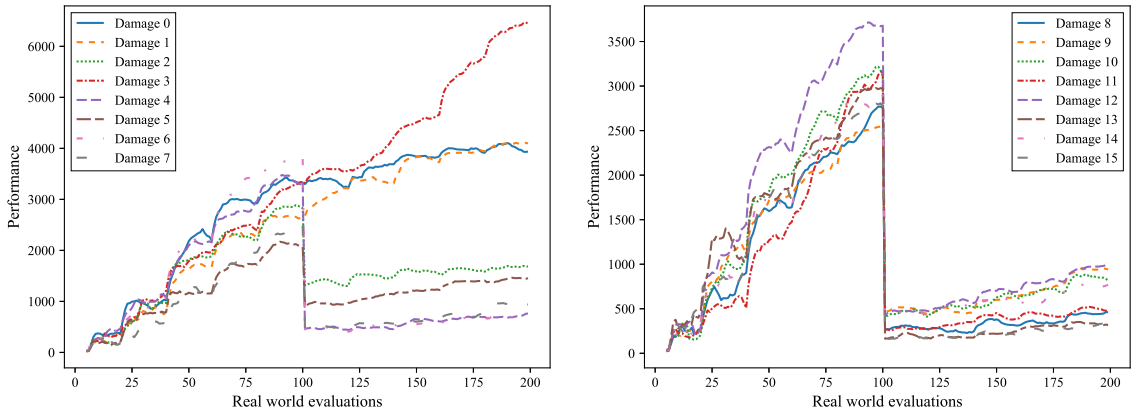


Figure 7.7: Performance over time when different damage types are applied

Since it would be infeasible to present the results of the evaluation of adaptations on every type of damage presented here, four types of damage have been chosen for further evaluation. Damage type 0 was chosen as it is similar to damage inflicted on the robot in other damage recovery research (Cully et al., 2015) and represents a damage type which has only a small effect on the robot's performance. Damage type 14 was chosen since it is one of the damage types with the largest negative effect on performance, and while the algorithm is unable to recover to pre-damage levels of performance, it is still of interest to investigate which adaptations are able to cause *larger* improvements in performance. Finally, damage types 2 and 5 were chosen as their performance represents a middle-ground between the other two damage types.

Plots of the performance of all four damage types will not necessarily be presented for each adaptation. Only those that contribute useful information to the presentation of the results are shown.

Figure 7.8 shows the effects of the four chosen damage types on the hand-designed tripod controller. It is clear that different damage types have very different effects on the controller. Damage types 0 and 5 reduced the distance that the robot was able to walk slightly, but their impact was much smaller than the impact of damage types 2 and 14. The types of damage that have the most negative impact on the robot's performance are likely to be different for other controllers; the paths presented in Figure 7.4 are evidence of this.

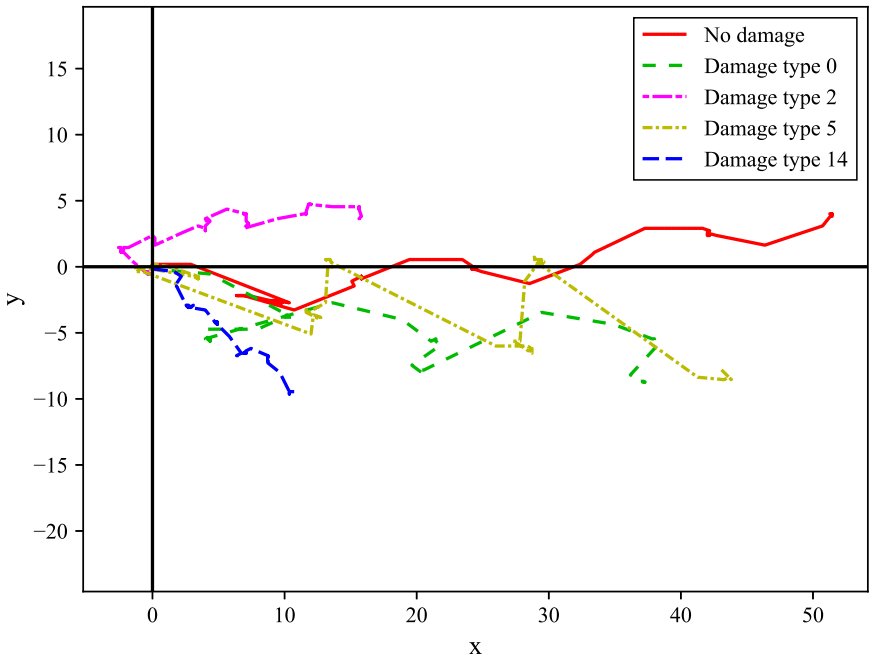


Figure 7.8: The paths followed by a tripod controller experiencing different damage types

7.3.3 Identified Issues

Two major issues were identified with regard to the effects of damage on the hexapod. Firstly, each evolved controller utilises its own method of locomotion. Each of these will depend on different combinations of the robot's legs. This, in turn, means that damaging a

specific leg on the robot will have different effects depending on the controller being used. The effects of different types of damage on a single controller were also found to be very diverse. These varied effects increase the variance of the algorithm's performance after damage, making the analysis of results more challenging than in previous investigations. Results which may appear significant when considering only the mean performance are often not, due to their high variance. This is seen in a number of results in Section 7.4.

The second issue is that some damage types cause the robot to be unable to move very far at all, even after recovering from damage. An example is damage type 14, which leaves the robot with only two functioning limbs. With so few limbs, the robot's performance does improve very slightly, but it is unable to reach levels of performance near those of the undamaged robot. The goal of the implementation of adaptations is therefore simply to improve the performance of BNS's damage recovery from its base rate, since it is unlikely that the robot can fully recover from every type of damage.

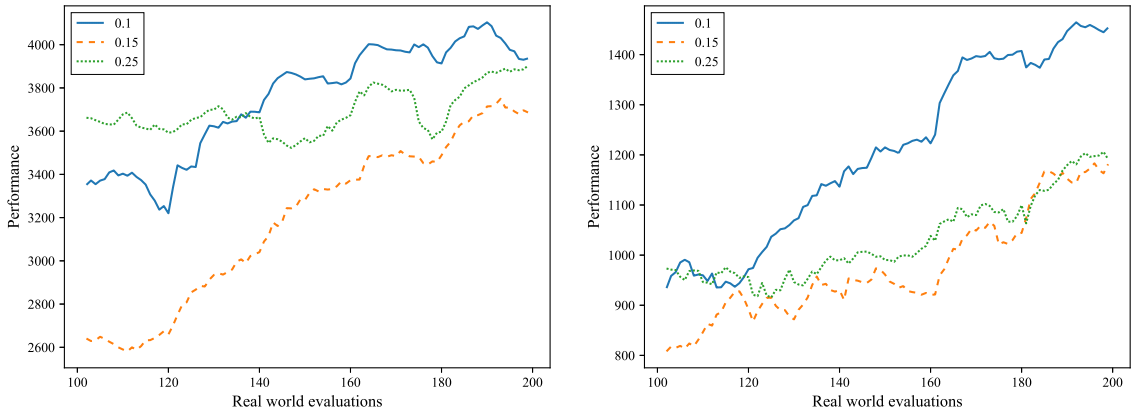
7.4 Adaptation Results and Discussion

In the following sections, the results of experiments using BNS with various configurations (Figure 7.1, item 3.3: Section 7.2.7) are shown. Section 7.3.2 presents the results with the default configuration, and sections 7.3.1 to 7.4.5 show the results when using each of the adaptations.

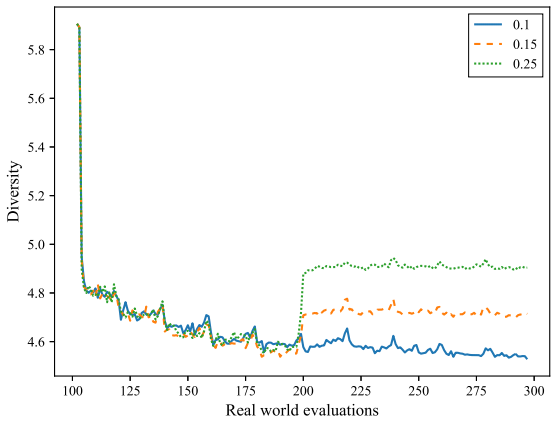
7.4.1 Mutation Changes

As was found in the previous investigations, increasing the mutation rate after damage occurred (Figure 7.9) had no significant, positive impact on performance (Table 7.5). It can be seen in Figure 7.9c, however, that an increased mutation rate does increase the diversity in the population, though this increase was insufficient to bring about a significant improvement in performance.

As with the mutation rate, increasing the mutation magnitude after damage (Figure 7.10) had no significant impact on performance for any type of damage (Table 7.6). Once again, Figure 7.10c shows that this was the case in spite of an increase in population diversity.

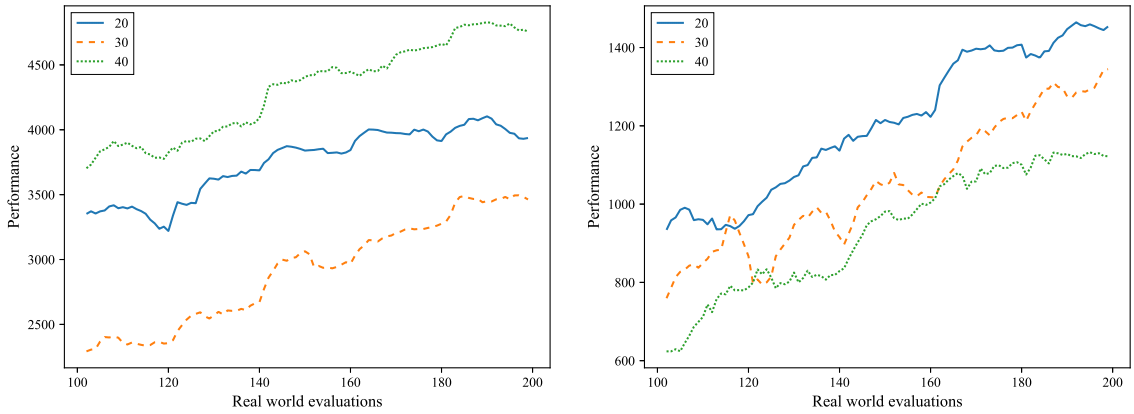


(a) Post-damage performance for damage type 0 (b) Post-damage performance for damage type 5

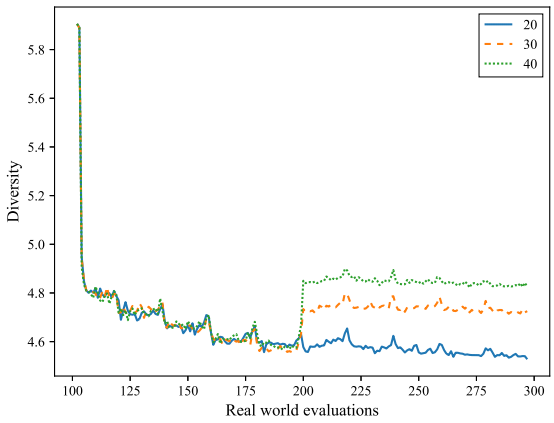


(c) Diversity over time

Figure 7.9: The effects of different mutation rate increases after damage occurs



(a) Post-damage performance for damage type 0 (b) Post-damage performance for damage type 3



(c) Diversity over time

Figure 7.10: The effects of different mutation magnitude increases after damage occurs

Table 7.5: p-values for Figure 7.9

(a) p-values for Figure 7.9a

	0.1	**0.15**
0.15	0.36322	
0.25	0.52014	0.80727

(b) p-values for Figure 7.9b

	0.1	**0.15**
0.15	0.22823	
0.25	0.34783	0.87663

Table 7.6: p-values for Figure 7.10

(a) p-values for Figure 7.10a

	20	**30**
30	0.23985	
40	0.87663	0.27719

(b) p-values for Figure 7.10b

	20	**30**
30	0.24581	
40	0.14128	0.64142

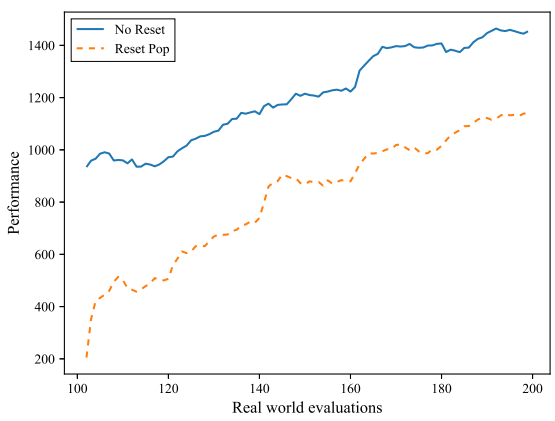
Finally, increasing both the mutation rate and magnitude together also had no significant impact on performance. This is, once again, similar to the results of the previous investigations.

The fact that these adaptations did not improve performance despite the increases in population diversity indicate that, at least for the hexapod, population diversity was not as large an issue as anticipated. At the same time, the adaptation did not negatively affect performance. There is, therefore, no reason not to suggest a marginal increase in the mutation parameters if damage can be detected; the increases may offer performance benefits in cases where diversity is more of an issue than was the case in these experiments.

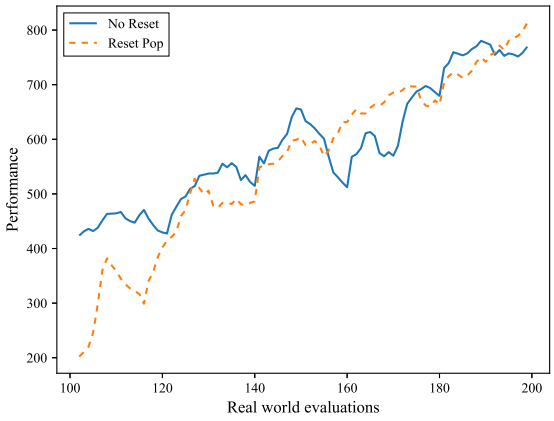
7.4.2 Population Reset

When evolving hexapod controllers, resetting the population (Figure 7.11) was found to have no significant effect on performance in the majority of cases (Figure 7.11b). In those cases where the adaptation had an effect, such as Figure 7.11a, performance was significantly worse ($p < 0.05$) than without the adaptation. The effect of the adaptation on population diversity can be seen very clearly in Figure 7.11c.

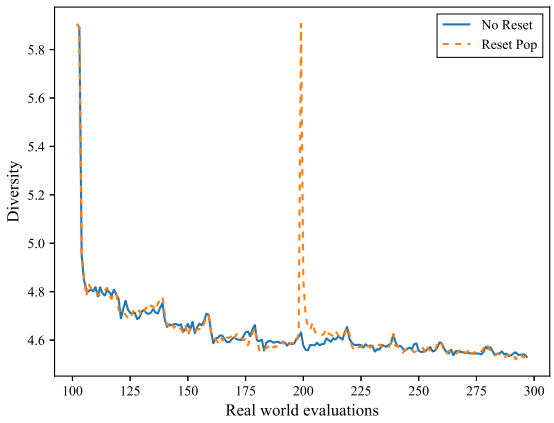
The apparent contradiction between these and previous results can be explained by the large difference in complexity between the robots. For simple robots, such as the Khepera, the evolution of an effective controller occurs much more rapidly than for the hexapod. In previous investigations only thirty or twenty real-world evaluations were needed to evolve



(a) Post-damage performance with damage type 5



(b) Post-damage performance with damage type 14



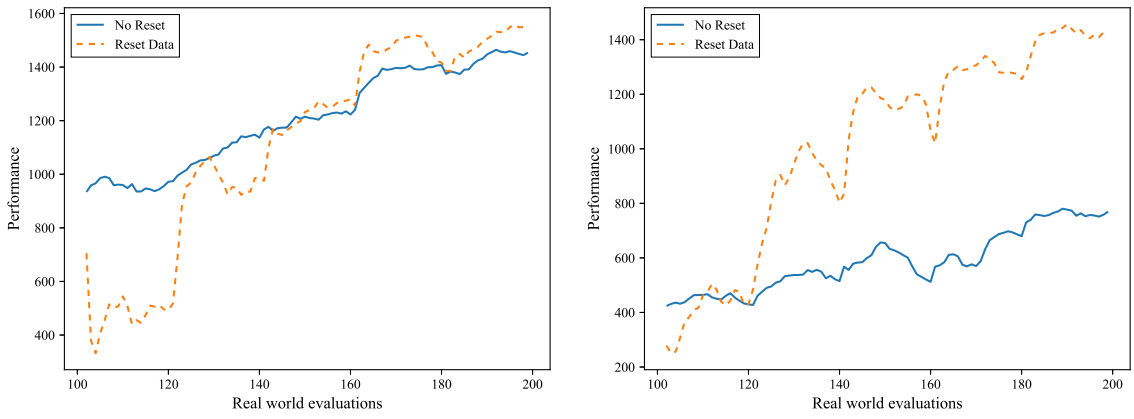
(c) Diversity over time

Figure 7.11: The effects of resetting the controller population after damage occurs

working controllers for investigations A and C, respectively. These low numbers meant that it was often better to evolve an effective controller from a completely new, random, population than to attempt to evolve the new controller starting with the population of controllers which had been evolved for the undamaged robot. This is not the case for the hexapod robot. The controllers' eighteen dimensional search space is much more complex than the Khepera's two-dimensional one. One hundred real-world evaluations were required to evolve controllers which could be considered to adequately complete the task. This is so much longer than was required for the Khepera that, if the damaged robot operates at all similarly to the undamaged robot, it becomes faster to evolve performant controllers from the existing population than to start from a new one.

7.4.3 Training Data Reset

The worst-case performance, when resetting the training data after damage occurred, was not significantly different to the performance without the reset (Figure 7.12a). In cases of extreme damage, such as damage type 14, the adaptation had a very significant ($p < 0.0005$) positive effect on performance.



(a) Post-damage performance with damage type 5

(b) Post-damage performance with damage type 14

Figure 7.12: The effects of resetting the training data after damage occurs

This adaptation is able to improve the algorithm's performance because resetting the training data means that after damage occurs, the simulator no longer trains on the old

incorrect training data and is able to adapt to the new state of the robot more rapidly.

The downside to this adaptation's use is that it requires knowledge of the occurrence of damage so that the data can be reset. The sliding window adaptation does not have this requirement, which makes it a more appealing choice if damage detection systems are not already in place for a given robot.

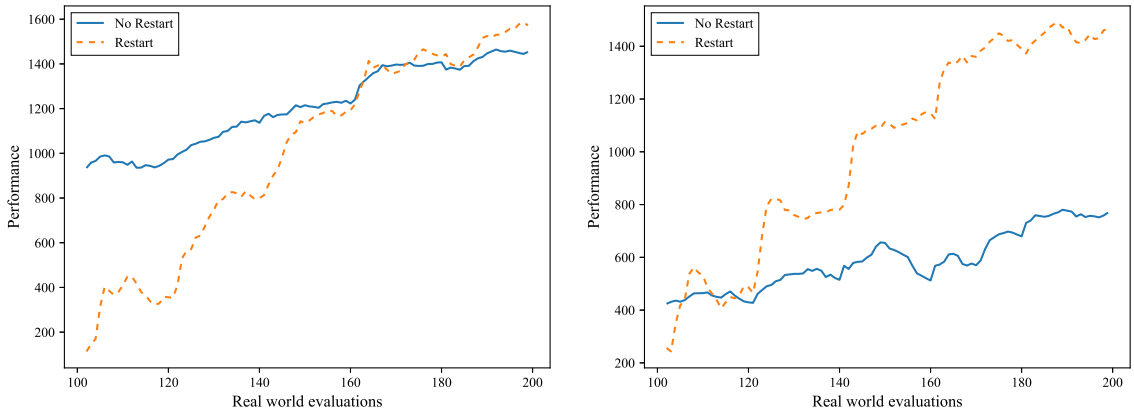
7.4.4 Complete Restart

As with resetting the training data, the worst case performance when restarting the entire BNS process was no different than with no adaptations (Figure 7.13a). In other cases, the adaptation had a significant ($p < 0.0005$) positive impact on performance. The effects of the reset on the population's diversity can be seen in Figure 7.13c. The rate of convergence after the reset was slower than when resetting only the population. This was likely because when starting BNS anew, the new simulator changes rapidly, causing sudden changes to the evaluated fitness of individuals in the population; an individual that was evaluated as the best in the population may no longer perform well when the simulator is updated. This causes convergence to be erratic early in the evolution process.

When resetting the entire BNS algorithm, three aspects of the system must be reset: the simulator, the training data, and the population of controllers. Since the simulator is already reset once every twenty real-world evaluations (Section 7.3.1), the only difference between resetting the entire algorithm and resetting the training data is the reset of the controller population. The results in Section 7.4.2 showed that for complex robots, the population reset had either no impact at all, or a negative impact. Figure 7.14 shows the performance with a complete restart and a training data reset. The results are not significantly different, once again reiterating that the use of the population reset adaptation, even in conjunction with other adaptations, does not offer any benefit to the algorithm's ability to recover from damage for complex robots.

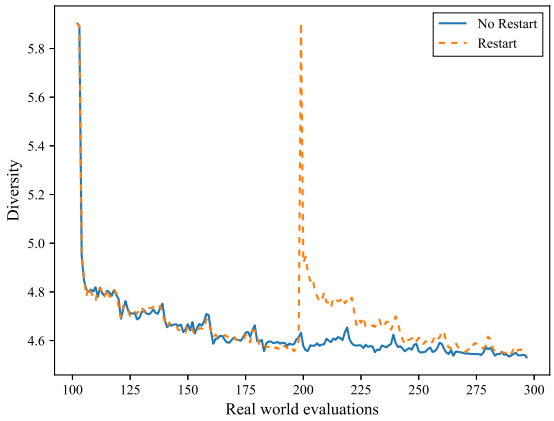
7.4.5 Sliding Window

Figure 7.15 shows the effects of different window sizes on the algorithm's damage recovery. It was found that only in cases of more extreme damage, such as damage type 14, did



(a) Post-damage performance with damage type 5

(b) Post-damage performance with damage type 14



(c) Diversity over time

Figure 7.13: The effects of completely restarting BNS after damage occurs

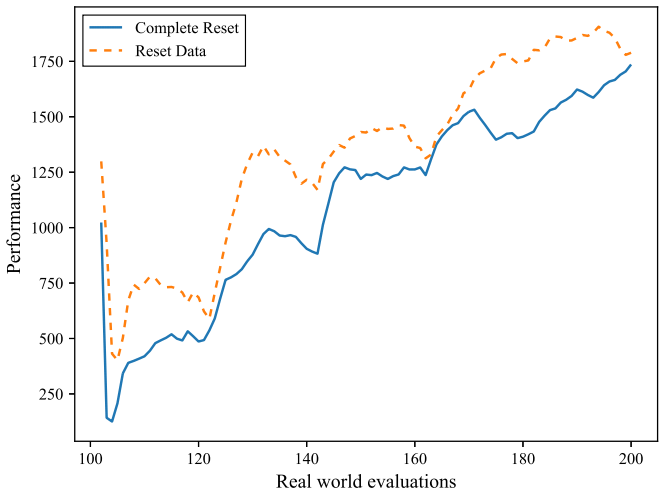
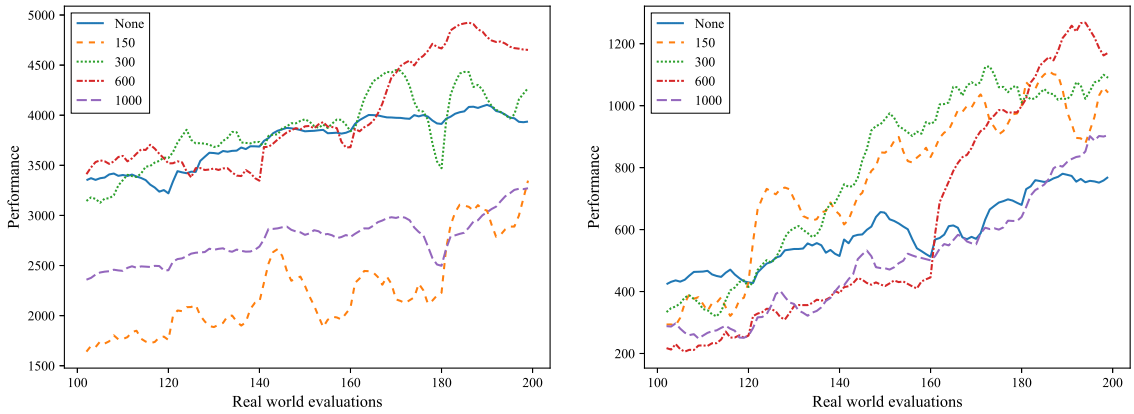


Figure 7.14: Post-damage performance when using the complete reset and reset data adaptations for damage type 2

the windows bring about an improvement in performance, as was the case for the training data reset. For damage types that are less severe, such as damage type 0, the windows did not cause any significant improvement. This is likely because in the case of damage type 0, and other mild damage types, the difference in the robot's behaviour before and after damage was not large enough to make the older training patterns entirely invalid. The advantage of a much larger training set consisting of only slightly incorrect training data outweighed the advantages of ceasing training using those training patterns. For damage type 14, on the other hand, the drastic change in behaviour meant that the sliding window assisted the simulator in adapting more rapidly after the damage had occurred.

In many cases, such as with damage type 14 (Figure 7.15b), the performance when using a window of size 600 or 1000 is not significantly different from no window at all. This corresponds with the findings of the previous investigations which found that, as the size of the window increases towards the size of the full set of training data, the benefits of a window's use become smaller. The window of size 150 performed significantly worse than no window in the case of damage type 0, once again highlighting the importance of a large enough window size. A sliding window of the same size did perform significantly better than no window in the case of damage type 14, but a moderately large window can once again be recommended, since the worst case performance for a window of size 300

was not significantly different from no window and at best offered the best improvement of any window size.



(a) Post-damage performance for damage type 0 (b) Post-damage performance for damage type 14

Figure 7.15: The effects of different sliding window sizes

(a) p-values for Figure 7.15a

	None	**150**	**300**	**600**
150	0.02068			
300	0.92344	0.01765		
600	0.98231	0.03917	0.99410	
1000	0.09334	0.57929	0.10233	0.12597

(b) p-values for Figure 7.15b

	None	**150**	**300**	**600**
150	0.03147			
300	0.04676	0.79585		
600	0.48252	0.07483	0.11882	
1000	0.64142	0.00403	0.00697	0.16687

It was considered that the sliding window's beneficial performance contributions may be overshadowed by the use of intermittent simulator resets. Experiments were therefore conducted to evaluate the impact of a sliding window in the absense of intermittent simulator resets, but the effects of the use of a sliding window were found to be similar whether or not intermittent simulator resets were used.

7.5 Real-World Results

This section presents the results of BNS executed on the hexapod in the real world (Figure 7.1, item 3.4). Only the paths followed by robots experiencing damage types 0 and 2 are shown here, since preliminary experiments found that after experiencing more extreme damage types, the robot is unlikely to recover to move far at all.

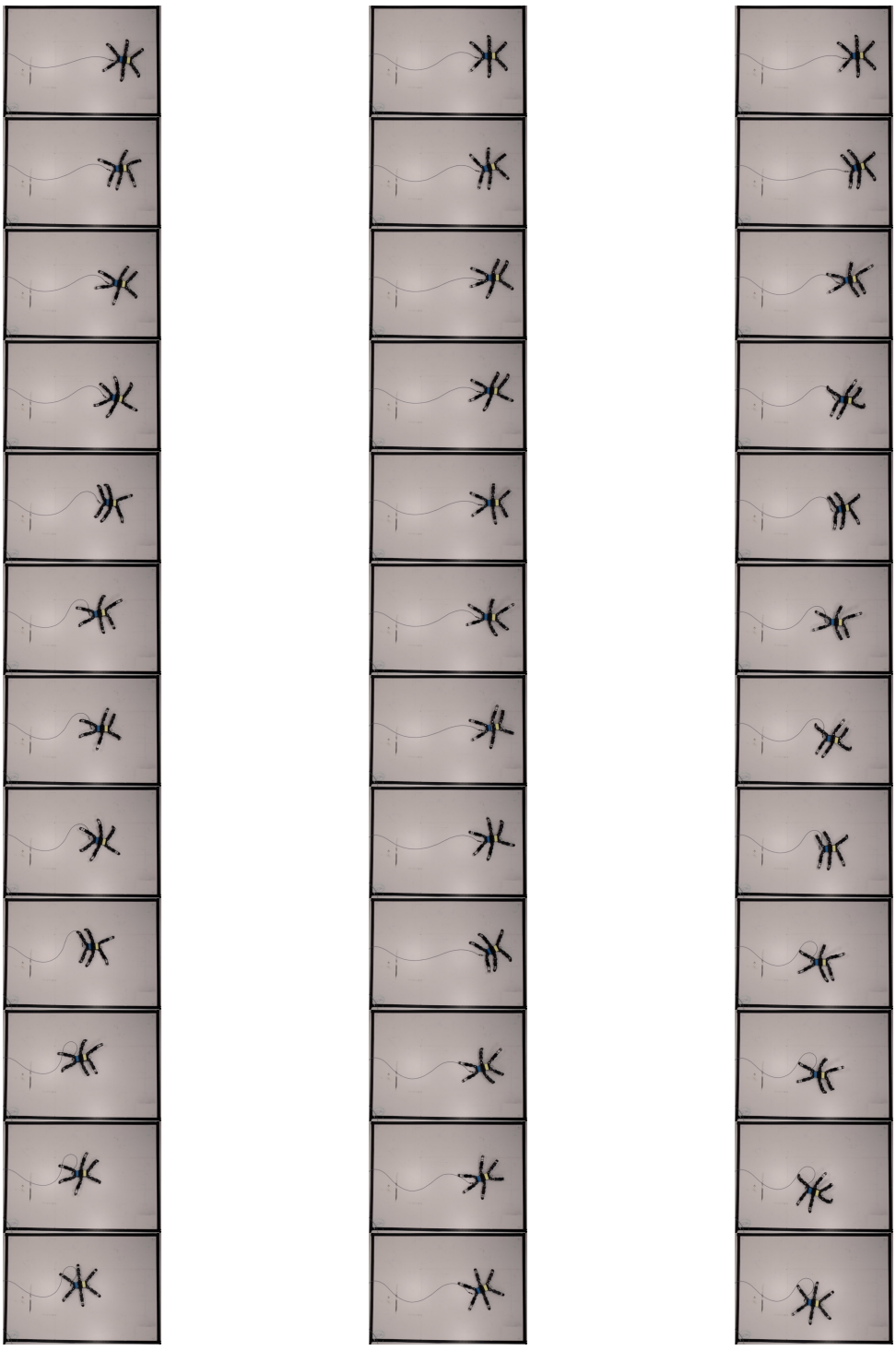
The first figure, Figure 7.16, shows images of the hexapod's motion in the real world. Figure 7.16b shows the first real-world evaluation after damage occurred. The robot's gait is the same as before damage, which can be seen by comparing the positions of the robot's legs in each frame of Figure 7.16a and 7.16b, but the hexapod's displacement was much smaller than in Figure 7.16a. BNS was then able to recover and evolved a new controller, which was adapted to the damage, shown in Figure 7.16c. The new controller used a new gait which had a greater reliance on the left middle leg, in order to compensate for the damaged left front leg. This can be seen in the figure as the number of front-to-back movements of the middle leg, which are responsible for pushing the robot forward, is increased.

Further results in this section represent a range of possible post-damage levels of performance; one was unable to attain pre-damage levels of performance (Figure 7.17a), one performed comparably before damage and after recovery (Figure 7.17b), and one performed much better after it had recovered from damage (Figure 7.17c).

Figure 7.17a shows the paths followed by a robot that experienced damage type 0. After damage was inflicted, the BNS process was completely restarted. In the case of this experiment, the system was then unable to recover. The best post-damage recovery was observed thirty-six real-world evaluations after damage occurred. This result highlights the risk of completely restarting the BNS algorithm: since the variability of BNS's results are so high, completely restarting the process may mean that effective controllers are unable to be evolved for the damaged robot in a reasonable period of time. That being said, the same variability means that the restarted BNS process may cause far better controllers to be evolved.

Figure 7.17b shows the paths followed by a robot that experienced damage type 2 while utilising a sliding window of size 300. The post-damage path of this robot more closely resembled its pre-damage path than the paths shown in Figure 7.17a. This is because the population was not reset when damage occurred, but was allowed to adapt to the damage. The post-damage controller therefore causes the robot to move in the same direction as the pre-damage controller, and they share a similar side-to-side method of locomotion.

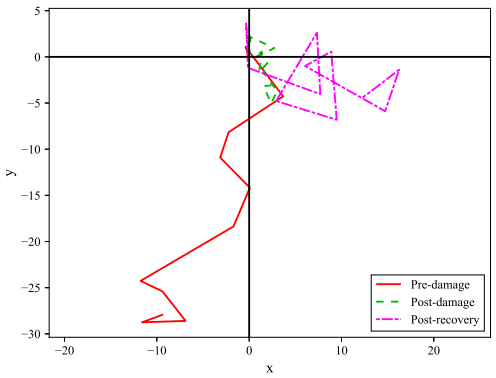
Finally, Figure 7.17c shows the paths of a robot which experienced damage type 0 when the BNS training data set was reset after the damage occurred. As with the sliding



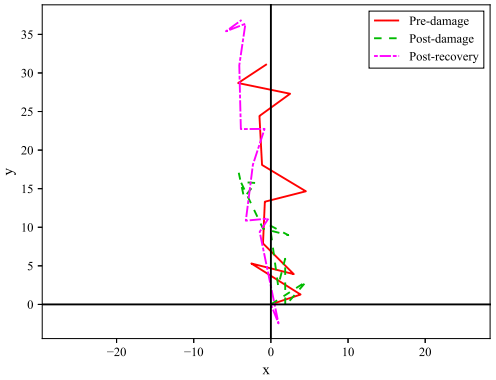
(a) Pre-damage execution (b) Post-damage execution (c) Post-recovery execution

Figure 7.16: Images captured after each hexapod command, with the robot experiencing and recovering from damage to one leg (damage type 0).

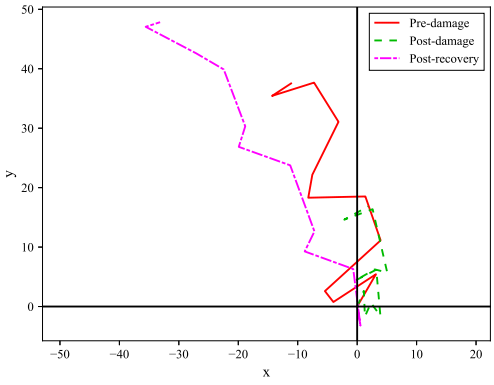
window, the pre- and post-damage paths in Figure 7.17c move in the same direction and have similar locomotion patterns. Unlike previous experiments, the robot was able to move much further, since it was able to evolve a new solution which did not utilise the damaged leg.



(a) Damage type 0 and a complete BNS restart after damage (36 RW evaluations for recovery)



(b) Damage type 2 with a sliding window of size 300 (28 RW evaluations for recovery)



(c) Damage type 0 with a training data reset after damage (46 RW evaluations for recovery)

Figure 7.17: Paths followed by a hexapod robot experiencing various damage types with different BNS adaptations implemented

These results show clearly that BNS is capable of recovering from damage to the hexapod robot in the real world. While the variability of the results does mean that in some cases the robot may be unable to evolve an effective solution, in many others it is

able not only to recover, but to perform even better than it could before damage.

7.6 Conclusion

In this investigation, BNS's ability to recover from damage to a robot with a complex morphology was evaluated. The algorithm was successful in improving its post-damage performance for sixteen types of damage which ranged from minimal to extreme, though in the case of the extreme damage, the improvement was insufficient for the robot to recover fully. The poor performance of the algorithm in cases of more extreme damage types can be attributed to the fact that when experiencing those damage types, very few of the robots' legs still worked. As an example, a robot that only has the use of three of its six legs is unlikely to be able to reach levels of performance anywhere near those of a robot with six fully functional legs. The algorithm was, however, found to recover from the less extreme damage types effectively, both in simulation and in the real world. After the robot experienced damage, the algorithm developed new gaits with less reliance on the robot's damaged limbs, allowing the robot to regain a great deal of its pre-damage functionality.

Woodford (2018) found that intermittently resetting the dynamic simulator led to an improvement in controller evolution when using BNS with complex robot morphologies. The resets were therefore implemented as a part of the default configuration in the investigation. It was hypothesised, however, that the resets would cause an improvement in performance both before and after damage. For this reason, evaluations were performed with and without intermittent simulator resets. These evaluations showed that simulator resets had the most significant impact of any adaptation evaluated in this investigation.

Changes to the mutation parameters were evaluated again. As in previous investigations, no combination of parameter values for the mutation rate and magnitude was able to bring about an improvement in performance over the default configuration. If damage can be detected with relative ease, however, the adaptation can be recommended, since in certain circumstances, the additional diversity introduced may be beneficial to the algorithm's recovery from damage.

Resetting the population was found to have the opposite effect when evolving con-

trollers for complex robots as it does for simple robots. With a simple robot morphology, evolving a functional controller is a fast enough process that resetting the population to a random state allows new solutions to be found more quickly than if the existing population is used. For a complex robot, such as the hexapod, the search space has too many dimensions and is too complex for this to be the case. Choosing not to reset the population, therefore, expedites the training process since the existing individuals are likely to be relatively nearer in the search space to good solutions for the damaged robot than was the case for the Khepera robot.

Resetting the simulator training data was found to have a significant positive effect on the algorithm's performance, as was restarting the entire BNS algorithm from the beginning. Restarting the algorithm requires resetting three aspects of the system: the simulator, the training data, and the population. Since the simulator is already reset once every twenty real-world evaluations, the only difference between the complete restart and the training data reset is a reset of the population. The training data reset adaptation and the complete restart adaptation were therefore compared, and their performance was found not to be significantly different from each other. This reaffirmed that the population reset did not have a significant positive impact on performance.

The results of the evaluation of the training windows were very similar to those for a training data reset. This is likely because both adaptations aim to rid the dataset of incorrect training patterns. A moderately-sized window, such as 300 training patterns, can once again be recommended since it did not cause a decrease in performance under any circumstances. A window of size 1000 did, however, cause performance to decrease. This is likely because as the window grows larger, it takes longer for the old incorrect training patterns to be discarded. There is potentially also an advantage to having more training data, even if that data is slightly incorrect, though this advantage lessens as damage becomes more severe. A window of size 1000 was likely too large to bring the benefits usually associated with a sliding window of training data, but also prevented the entire set of data from being used for training, thereby negatively affecting performance on both fronts.

Chapter 8

Conclusions and Future Work

8.1 Introduction

This research evaluated the ability of the BNS algorithm to recover from damage to the physical robots for which it had evolved controllers. The outcomes of the research objectives are discussed in Section 8.2. The contributions of this research are detailed in Section 8.3, while its limitations are discussed in Section 8.4. Finally, Section 8.5 recommends future avenues of research while Section 8.6 summarises the chapter.

8.2 Overview of Results and Outcomes of Research Objectives

This section discusses how each of the research objectives, which were used to focus this research, was addressed.

RO1: Investigate existing ER damage-recovery methods

Before research could be conducted to investigate damage recovery for a specific algorithm, it was important to investigate and review the literature for the underlying evolutionary robotics concepts, as well as other existing methods of damage recovery. A literature review was conducted (Chapter 2) to achieve this goal.

The literature review investigated ANNs and EAs, including their use for evolutionary

robotics. The use of simulators in ER, their implementation, and the reasons for their use, were discussed. Simulator neural networks and Bootstrapped Neuro-Simulation were then investigated, as they are the methods upon which this research was based. Finally, existing methods of damage recovery were reviewed.

RO2: Determine BNS's damage-recovery capabilities

It was hypothesised that BNS would have some inherent ability to recover from damage, due to the algorithm's always-learning nature. The algorithm is continuously gathering new data about the robot and its environment and is thus able to continue gathering data after damage occurs, allowing it to learn the new state of the robot and adapt to the damage.

Investigation A was a preliminary investigation, which showed that this hypothesis was true. The BNS algorithm was able to evolve controllers for a differentially-steered robot, capable of completing a simple navigation task. The robot was then damaged, and the system was able to continue running and recover from the damage.

RO3: Propose, implement, and evaluate adaptations to improve BNS's damage-recovery capabilities

BNS had been shown to be able to recover from damage without any changes to the algorithm. This did not mean that the algorithm was unable to perform better. Four *adaptations* were therefore proposed and hypothesised to be capable of improving the performance of the algorithm:

- The controller population was reset after damage occurred.
- The mutation rate and magnitude were varied after damage occurred.
- A sliding window of training data was used to train the SNN.
- The SNN was reset after damage occurred.

Investigation A included experiments that sought to evaluate these adaptations both in simulation and reality. The use of a sliding window was found to be the most advantageous

as, in addition to offering the largest improvement in performance, it does not require the knowledge of the occurrence of damage, and can simply be applied for the entire BNS execution.

RO4: Propose, implement, and evaluate a method of evolving complex controllers using BNS

Prior to this research, BNS had only been used to evolve simple controllers, which were unable to make decisions based on observations about their environment. Investigation B addressed this issue by evolving closed-loop, neural-network-based controllers to solve a light-following problem. The results of the investigation showed that BNS is capable of evolving such a controller alongside simulators for both its motion and sensors. This BNS implementation was the first that had simultaneously evolved two entirely independent simulators.

Several new adaptations to the BNS algorithm were proposed and evaluated since no such research had been conducted for closed-loop controllers. The investigated adaptations were alternative crossover operators, data augmentation, alternative weight initialisation strategies, Island EAs, differing controller ANN topologies, Headless Chicken mutation, intermittent population resets, differing tournament sizes, and various population sizes. A parameter study was conducted to investigate both the parameters for these new adaptations and the optimal values of the existing parameters.

The two most important factors were found to be population diversity and simulator accuracy, with the adaptations that directly affected these two parameters having the largest overall impact on the algorithm's performance. Headless Chicken mutation, large neural weight initialisation ranges, and Island EAs increased diversity, while data augmentation allowed the SNN to improve its accuracy earlier in the evolution process, leading to the creation of improved controllers.

RO5: Propose, implement, and evaluate adaptations for BNS to allow for damage recovery of complex controllers

Upon completion of investigation B, the groundwork had been laid for the investigation of damage recovery for complex controllers using BNS. In investigation C, experiments were conducted on the same robot and problem as had been investigated in investigation B, but the robot was damaged in order to evaluate the algorithm's damage recovery.

The use of a complex controller meant that, in addition to the simple wheel damage that had been evaluated in investigation A, it was possible to evaluate the effects of damage to a robot's sensors. The types of damage inflicted on the robot's wheels could also be made more complex since the controllers were able to adapt in real time to the effects of the damage.

BNS was found, once again, to perform excellently and recovered from the damage. It was found, however, that the algorithm is unable to recover from damage that is impossible for the dynamic simulator to simulate, such as temporal-based damage, which occurs intermittently. In the case of such damage, the real-world performance of the robot is unable to improve, causing it to become worse as the simulators become more refined.

In this investigation a sliding window was shown once again to be the most effective adaptation; it offered significant improvements in performance while not requiring any knowledge of damage. Adaptations that moderately increased diversity in the population were also found to improve performance.

RO6: Demonstrate and investigate the transferral of BNS's damage recovery to a more complex robot morphology

Finally, BNS needed to be evaluated on a complex robot morphology. Investigation D investigated BNS's ability to recover from damage to a hexapod robot, which has a far more complex morphology than the Khepera. The latter had been used for all previous experimentation.

BNS was once again shown to have the ability to recover from damage, though the increased complexity of the robot meant that recovery was slower than in previous experiments. Unlike in previous investigations, resetting the controller population when damage

occurred was found to have a negative impact on post-damage performance. This is likely due to the greater complexity of the hexapod controller search space when compared to that of the Khepera controllers. Finding candidate solutions for the Khepera problems is simple enough that resetting the population allows the algorithm to rapidly discover new solutions, unhindered by biases caused by the pre-damage controllers. This is not the case for the hexapod, where finding potential solutions is far more challenging. The benefits of prior information, such as the pre-damage controller population, therefore grow alongside the complexity of the search space, causing the algorithm to perform better when the population is not reset.

Intermittent simulator resets were shown to be extremely beneficial for damage recovery with complex robot morphologies. Sliding windows, on the other hand, were found to offer a less significant performance improvement with complex robots. There are, however, still benefits to their use in some cases, provided that a large enough window is used.

8.3 Contributions

This research's theoretical contribution is the recommendation of adaptations and parameter configurations that improve the BNS algorithm's damage-recovery performance (in the case of investigations A, C, and D) or its performance in general (in investigation B). Additionally, investigation B (closed-loop controller evolution) was the first instance of BNS being used to evolve closed-loop controllers.

The controller reset adaptation causes the population of controllers to be reset when damage is detected, and the mutation rate and magnitude adaptations alter the mutation parameters once damage is detected. Both of these adaptations aim to increase diversity in the population to aid the search for new solutions. The simulator reset adaptation aims to prevent the pre-damage simulator from negatively impacting the training of the simulator after damage by resetting the SNN to a random state. Finally, the sliding window adaptation, which consistently showed the most promising improvement in performance, restricted the SNN's training data to only the newest training patterns, thereby allowing the system to *forget* the old, irrelevant training data and focus only on the new accurate data when damage occurs.

In order to conduct the investigations to evaluate the adaptations, an existing BNS implementation was used as a starting point and adapted specifically for the evaluation of damage recovery. For each of investigations A, C, and D, the existing systems were augmented with damage systems that were able to inflict simulated damage on the robot, affecting the robot in the same way that real damage would without the additional costs that would be involved with inflicting real damage on the robot.

The algorithm and adaptations were evaluated both in simulation and in the real world. This allowed for confirmation that the conclusions drawn in simulation do transfer to reality. The simulated evaluation of the algorithm made use of a *fake real world*. This fake real world was constructed from existing motion SNNs and a new sensor SNN, which was a contribution of this research. The sensor SNN was trained using real-world data and allowed the behaviour of the Khepera robot's sensors to be simulated for investigation C (closed-loop controller evolution). The fake real world allowed for the evaluation of far more parameter combinations than would have been possible had all testing been done in the real world.

8.4 Limitations

Recovery from damage is not guaranteed in all situations, even when using the newly proposed adaptations. In each investigation, situations were found where the severity of the damage was great enough that the robot was unable to recover to a level of performance anywhere near its pre-damage performance. That is not to say that the algorithm could not improve performance after damage occurred, but that it could only do so to a reasonable level.

A number of the adaptations require knowledge of the occurrence of damage, such as the controller population reset, which was found to perform well in investigation D. In order to be implemented in real-world applications, systems would need to be implemented in order to detect damage and *trigger* these adaptations.

The types of damage evaluated in these investigations were chosen to provide a wide range of situations from which the algorithm needed to recover. While many types of damage were evaluated in this research, in real-world applications, there would be far

more possible types of damage that could occur. The assumption is that the conclusions drawn in these investigations will also apply to the unknown damage types.

8.5 Recommendations for Future Research

The adaptations proposed in this research focused solely on recovery from damage and not any methods of damage detection. Since many of the adaptations require damage detection to work in the real world, future research should focus on methods of damage detection using BNS.

The same uncertainty calculations that are used with the ensemble of dynamic SNNs could be used to detect damage. When a robot becomes damaged, and the SNNs begin training on training patterns from the damaged robot, the SNNs will start to produce predictions which are more diverse than before. The level of disagreement among SNNs in the ensemble could therefore be monitored and used to detect damage to the robot without implementing specific damage detection systems on the physical robot.

While it would require changes to the fundamental BNS algorithm, combinations of BNS and other trial-and-error methods of damage recovery discussed in Chapter 2 may be beneficial. These new algorithms could serve to improve the rate of damage recovery further by refining BNS's ability to identify potential solutions when recovering from damage.

8.6 Summary

Damage recovery is an important area of research in the field of evolutionary robotics because as robots' applications become more complex, so does their risk of damage. This damage may happen in environments that make it impossible for humans to intervene and repair the robots. As such, robots that are able to adapt to the damage and continue functioning without assistance hold an advantage over those that cannot.

In this research, BNS was shown to recover when damage was inflicted on a differentially-steered robot controlled by an open-loop controller. Novel adaptations to the algorithm were then proposed in the interest of improving this damage recovery. The adaptations

were based on hypotheses about factors of the algorithm which could have negatively impacted performance. A sliding window of training data was found to be the most effective.

BNS was then used to evolve closed-loop controllers by training two simulator neural networks simultaneously. Since no previous research had been conducted investigating BNS's ability to evolve closed-loop controllers, additional adaptations to the algorithms were proposed and found to be successful in improving BNS's performance.

Evaluations were then carried out to investigate BNS's ability to recover from damage when using closed-loop controllers. Once again, the algorithm was shown to be able to recover before the implementation of any adaptations. The sliding window adaptation was also shown to offer the largest improvement over the base performance of the algorithm.

In the final investigation, BNS's ability to recover from damage to a more complex robot morphology was investigated. Controllers were evolved for hexapod locomotion. The robot then had different combinations of its legs damaged and BNS's recovery was evaluated. BNS was shown to have the ability to recover, but the recovery was slower than that seen in previous investigations due to the increased complexity of the hexapod robot. Adaptations were once again evaluated; intermittently resetting the simulator, a method proposed for use with complex robots, and a sliding window, were found to offer significant improvements to the algorithm's performance.

Bibliography

E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, 2002.

P. J. Angeline, P. J. Angeline, G. M. Saunders, G. M. Saunders, J. B. Pollack, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks." *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 54–65, 1994. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/18267779

I. A. Basheer and M. Hajmeer, "Artificial neural networks: Fundamentals, computing, design, and application," *Journal of Microbiological Methods*, vol. 43, no. 1, pp. 3–31, 2000.

D. Beasley, D. R. Bull, and R. R. Martin, "An overview of genetic algorithms : Part 1, fundamentals," *University Computing*, vol. 2, no. 15, pp. 56–69, 1993. [Online]. Available: http://www.geocities.ws/francorbusetti/gabeasley1.pdf

J. Bongard, V. Zykov, and H. Lipson, "Resilient Machines through Continuous Self-Modelling," *Science*, vol. 314, no. 5802, pp. 1118–1121, 2006. [Online]. Available: http://www.jstor.org/stable/20032832

J. C. Bongard, "Evolutionary Robotics," *Communications of the ACM*, vol. 56, no. 8, pp. 74–83, 2013.

J. C. Bongard and H. Lipson, "Nonlinear system identification using coevolution of models and tests," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 4, pp. 361–384, 2005.

V. Braitenberg, *Vehicles: Experiments in synthetic psychology.* MIT Press, 1986.

R. a. Brooks, "Artificial Life and Real Robots," in *Proceedings of the First European Conference on Artificial Life.* MIT Press, 1992, pp. 3–10. [Online]. Available: http://www.mitpressjournals.org/doi/abs/10.1162/106454600568393

K. Chatzilygeroudis, V. Vassiliades, and J.-B. Mouret, "Reset-free Trial-and-Error Learning for Robot Damage Recovery," *Robotics and Autonomous Systems*, vol. 100, pp. 236–250, 2016. [Online]. Available: http://arxiv.org/abs/1610.04213

N. Cheney, R. Maccurdy, J. Clune, and H. Lipson, "Unshackling Evolution: Evolving Soft Robots with Multiple Materials and a Powerful Generative Encoding," in *The Genetic and Evolutionary Computation Conference*, Amsterdam, The Netherlands, 2013, pp. 167–174.

M. Črepinšek, S.-H. Liu, and M. Mernik, "Exploration and Exploitation in Evolutionary Algorithms: A Survey," *ACM Computing Surveys*, vol. 45, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2480741.2480752

A. Cully, J. Clune, D. Tarapore, and J. B. Mouret, "Robots that can adapt like animals," *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

K. Deb, S. Pratab, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NGSA-II," *IEEE Transactions on Evolutionary Computing*, vol. 6, no. 2, pp. 182–197, 2002.

K. Deb and R. B. Agrawal, "Simulated Binary Crossover for Continuous Search Space," *Complex Systems*, vol. 9, no. 3, pp. 115–148, 1995.

S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods for Software Engineering Research," in *Guide to Advanced Empirical Software Engineering*, 2008, pp. 285–311. [Online]. Available: http://www.springerlink.com/index/n815725515063p2m.pdf

A. P. Engelbrecht, *Computational Intelligence*, 2nd ed. Chichester: John Wiley & Sons, Ltd, 2007.

D. Floreano and C. Mattisussi, "Bio-inspired artificial intelligence: theories, methods, and technologies," *MIT Press*, 2008.

D. Floreano and F. Mondada, "Automatic creation of an autonomous agent: Genetic evolution of a neural-network driven robot," in *Third International Conference on Simulation of Adaptive Behavior*. MIT Press, 1994, pp. 421–430.

P. P. J. B. Hancock, "Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification," *COGANN92 International Workshop on Combinations of Genetic Algorithms and Neural Networks*, vol. 1, pp. 108–122, 1992. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=273944

I. Harvey, P. Husbands, D. Cli, I. Harvey, P. Husbands, and D. Cli, "Issues in Evolutionary Robotics," in *The Second International Conference on Simulation of Adaptive Behaviour*, 1993, pp. 364–373.

F. Herrera, M. Lozano, and J. L. Verdegay, "Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis," *Artificial Intelligence Review*, vol. 12, no. 4, pp. 265–319, 1998. [Online]. Available: http://link.springer.com/article/10.1023/A:1006504901164

J. Hiller and H. Lipson, "Automatic design and manufacture of soft robots," *IEEE Transactions on Robotics*, vol. 28, no. 2, pp. 457–466, 2012.

S. Hochreiter, "Lstm Can Solve Hard Long Time Lag Problems," in *Neural Information Processing Systems*, 1997, pp. 473–479.

D. Hunter, H. Yu, M. S. Pukish, J. Kolbusz, and B. M. Wilamowski, "Selection of Proper Neural Network Sizes and Architectures A Comparative Study," *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 228–240, 2012.

N. Jakobi, P. Husbands, and I. Harvey, "Noise and the Reality Gap: The Use of Simulation in Evolutionary Robotics," in *European Conference on Artificial Life*. Springer, 1995, pp. 704–720.

James G. Bellingham and Kanna Rajan, "Robotic in Remote and Hostile Environments," *Science*, vol. 318, no. 5853, pp. 1098–1102, 2007.

K-team Corporation, "Khepera III." [Online]. Available: http://www.k-team.com/mobile-robotics-products/old-products/khepera-iii

N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, 2004, pp. 2149–2154. [Online]. Available: http://ieeexplore.ieee.org/document/1389727/

S. Koos, A. Cully, and J. B. Mouret, "Fast damage recovery in robotics with the T-resilience algorithm," *International Journal of Robotics Research*, vol. 32, no. 14, pp. 1700–1723, 2013.

S. Koos, J. B. Mouret, and S. Doncieux, "The transferability approach: Crossing the reality gap in evolutionary robotics," *IEEE Transactions on Evolutionary Computation*, vol. 17, no. 1, pp. 122–145, 2013.

Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

J. Lehman, J. Clune, D. Misevic, C. Adami, J. Beaulieu, P. J. Bentley, S. Bernard, G. Beslon, D. M. Bryson, P. Chrabaszcz, N. Cheney, A. Cully, S. Doncieux, F. C. Dyer, K. O. Ellefsen, R. Feldt, S. Fischer, S. Forrest, A. Frénoy, C. Gagné, L. L. Goff, L. M. Grabowski, B. Hodjat, F. Hutter, L. Keller, C. Knibbe, P. Krcah, R. E. Lenski, H. Lipson, R. MacCurdy, C. Maestre, R. Miikkulainen, S. Mitri, D. E. Moriarty, J.-B. Mouret, A. Nguyen, C. Ofria, M. Parizeau, D. Parsons, R. T. Pennock, W. F. Punch, T. S. Ray, M. Schoenauer, E. Shulte, K. Sims, K. O. Stanley, F. Taddei, D. Tarapore, S. Thibault, W. Weimer, R. Watson, and J. Yosinksi, "The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities," in *eprint arXiv:q-alg/9503002*, 2018, pp. 1–31. [Online]. Available: http://arxiv.org/abs/1803.03453

H. Lipson, J. Bongard, V. Zykov, and E. Malone, "Evolutionary Robotics for

Legged Machines: From Simulation to Physical Reality," in *Intelligent Autonomous Systems*, 2006, pp. 11–18. [Online]. Available: http://books.google.com/books?hl=en&lr=&id=D4QYV3D7JWoC&oi=fnd&pg=PA11&dq=Once+more+unto+the+breach:+Co-evolving+a+robot+and+its+simulator&ots=aZj9HvUgI3&sig=4GIt9oh_ugZS6WsWigmtp4rNm7g

H. Lund and O. Miglino, "From simulated to real robots," in *Proceedings of IEEE International Conference on Evolutionary Computation*, 1996, pp. 362–365.

A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 30, 2013, p. 6. [Online]. Available: https://web.stanford.edu/~awni/papers/relu_hybrid_icml2013_final.pdf

V. Papaspyros, K. Chatzilygeroudis, V. Vassiliades, and J.-B. Mouret, "Safety-Aware Robot Damage Recovery Using Constrained Bayesian Optimization and Simulated Priors," in *30th Conference on Neural Information Processing Systems*, 2016, pp. 1–5. [Online]. Available: http://arxiv.org/abs/1611.09419

G. B. Parker, "Co-evolving model parameters for anytime learning in evolutionary robotics," *Robotics and Autonomous Systems*, vol. 33, no. 1, pp. 13–30, 2000.

C. J. Pretorius, M. C. du Plessis, and C. B. Cilliers, "Simulating robots without conventional physics: A neural network approach," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 71, no. 3-4, pp. 319–348, 2013.

C. J. Pretorius, M. C. du Plessis, and C. B. Cilliers, "Towards an artificial neural network-based simulator for behavioural evolution in evolutionary robotics," *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on - SAICSIT '09*, pp. 170–178, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1632149.1632171

C. J. Pretorius, M. C. du Plessis, and C. B. Cilliers, "A neural network-based kinematic and light-perception simulator for simple robotic evolution," in *2010 IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.

C. J. Pretorius, M. C. du Plessis, and J. W. Gonsalves, "A comparison of neural networks and physics models as motion simulators for simple robotic evolution," in *IEEE Congress on Evolutionary Computation*, 2014, pp. 2793–2800.

C. J. Pretorius, M. C. du Plessis, and J. W. Gonsalves, "Neuroevolution of Inverted Pendulum Control: A Comparative Study of Simulation Techniques," *Journal of Intelligent and Robotic Systems*, vol. 86, no. 3-4, pp. 419–445, 2017. [Online]. Available: http://link.springer.com/10.1007/s10846-017-0465-1

C. Pretorius, "Artificial neural networks as simulators for behavioural evolution in evolutionary robotics," Master's Thesis, Nelson Mandela Metropolitan University, 2010.

K. Sanderson, "Mars rover Spirit (2003-10)," *Nature*, vol. 463, no. 7281, p. 600, 2010.

H. T. Siegelmann and E. D. Sontag, "On the computational power of neural nets," *Journal of computer and system sciences*, vol. 50, no. 1, pp. 132–150, 1995.

Sinno Jialin Pan and Q. Yang, "A Survey on Transfer Learning," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, 2010, pp. 1345–1359.

D. A. Sofge, M. A. Potter, M. D. Bugajska, and A. C. Schultz, "Challenges and Opportunities of Evolutionary Robotics," in *International Conference on Computational Intelligence, Robotics and Autonomous Systems*, 2003. [Online]. Available: http://arxiv.org/abs/0706.0457

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/106365602320169811

V. Verma, G. Gordon, R. Simmons, and S. Thrun, "Real-time fault diagnosis," *IEEE Robotics and Automation Magazine*, vol. 11, no. 2, pp. 56–66, 2004.

D. D. Wackerly, W. Mendenhall, and R. L. Scheaffer, *Mathematical Statistics with Applications*, 7th ed. Belmont: Brooks/Cole, 2008.

D. Whitley, *Genetic algorithms and neural networks*, 3rd ed., G. Winter and J. Periaux, Eds. John Wiley & Sons Ltd, 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.2120&rep=rep1&type=pdf

G. W. Woodford, "Personal Communication," 2018.

G. W. Woodford, M. C. du Plessis, and C. J. Pretorius, "Evolving snake robot controllers using artificial neural networks as an alternative to a physics-based simulator," in *IEEE Symposium Series on Computational Intelligence*, 2015, pp. 267–274.

G. W. Woodford, C. J. Pretorius, and M. C. du Plessis, "Concurrent controller and Simulator Neural Network development for a differentially-steered robot in Evolutionary Robotics," *Robotics and Autonomous Systems*, vol. 76, pp. 80–92, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2015.10.011

G. W. Woodford, M. C. du Plessis, and C. J. Pretorius, "Concurrent controller and Simulator Neural Network development for a snake-like robot in Evolutionary Robotics," *Robotics and Autonomous Systems*, vol. 88, pp. 37–50, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2016.11.018

X. Yao and Y. Liu, "A new evolutionary system for evolving artificial neural networks." *IEEE transactions on neural networks*, vol. 8, no. 3, pp. 694–713, 1997. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=572107

J. C. Zagal and J. Ruiz-del Solar, *UCHILSIM: A Dynamically and Visually Realistic Simulator for the RoboCup Four Legged League*, D. Nardi, M. Riedmiller, C. Sammut, and Santos-Victor, Eds. Springer, 2005, vol. 3276. [Online]. Available: http://www.springerlink.com/content/kj5k1vgr7u6lk5dv

J. C. Zagal and J. Ruiz-Del-Solar, "Combining simulation and reality in evolutionary robotics," *Journal of Intelligent and Robotic Systems*, vol. 50, no. 1, pp. 19–39, 2007.

Appendix A

Mann-Whitney U Test Statistic

The Mann-Whitney U Test Statistic is calculated as:

$$U = n_1 n_2 + \frac{n_1(n_1+1)}{2} - W \tag{A.1}$$

where n_i is the number of observations in sample i and W is the rank sum for sample 1. The rank sum is the sum of the ranks of all observations from sample 1, in the ordered list of all observations, where the rank of an observation is its position in the ordered list. For example, given two samples of observations, $x_1..x_4$ and $y_1..y_4$, with the following values:

25	26	27	28	29	30	31	32
x_1	x_2	x_3	y_1	y_2	x_4	y_3	y_4

$$W = \sum_{i=1}^{4} \text{rank}(x_i) = 1 + 2 + 3 + 6 = 12 \tag{A.2}$$

The rejection region for the two-tailed test is $U \leq U_0$ or $U \geq n_1 n_2 - U_0$ where U_0 is the value such that $P(U \leq U_0) = \alpha$ where α is the probability of a Type I error.

Appendix B

Publications

B.1 ICCSIT 2018 & JCP

A damage system was implemented for a simple differentially steered robot evolved using Bootstrapped Neuro-Simulation. BNS was shown to possess damage recovery capabilities. Additional enhancements to the method were proposed, and parameters to these methods evaluated.

This study was presented at the 11^{th} International Conference on Computer Science and Information Technology and published in the Journal of Computers (JCP).

B.2 Robotics and Autonomous Systems

A closed-loop controller evolution system was implemented for BNS. The transferability of controllers evolved using this system was demonstrated and thereafter a damage system implemented. This damage system allowed for the evaluation of BNS and its ability to recover from damage to a robot using a closed-loop controller. Adaptations and improvements to BNS were proposed for both closed-loop controller evolution and damage recovery. The paper was submitted to the journal of Robotics and Autonomous Systems.