UNIVERSITY OF LIVERPOOL

DOCTORAL THESIS

Manifolds & Memory

Improving the Search Speed of Evolutionary Algorithms

Author:
James BUTTERWORTH

Supervisors:
Prof. Karl TUYLS
Prof. Rahul SAVANI

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Computer Science

June 4, 2023

Declaration of Authorship

I, James BUTTERWORTH, declare that this thesis titled, "Manifolds & Memory: Improving the Search Speed of Evolutionary Algorithms" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 04/06/2023

"We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time."

- Little Gidding (1942)
T. S. Eliot

UNIVERSITY OF LIVERPOOL

Abstract

Faculty of Science and Engineering
Department of Computer Science

Doctor of Philosophy

Manifolds & Memory: Improving the Search Speed of Evolutionary Algorithms

by James BUTTERWORTH

Evolutionary Algorithms (EA) are a set of algorithms inspired by Darwin's theory of Natural Selection that are well equipped to perform a wide variety of optimisation tasks. Due to their use as a derivative-free continuous value optimisation algorithm, EAs are often compared to gradient based optimisation techniques, such as stochastic gradient descent (SGD). However, EAs are generally deemed subpar to gradient based techniques, evidenced by the fact that none of the most commonly used Deep Learning frameworks implement EAs as a neural network optimisation algorithm, and that the majority of neural networks are optimised using gradient based techniques. Nevertheless, despite often cited as being too slow to optimise large parameter spaces, such as large neural networks, numerous recent works [123, 138] have shown that EAs can outperform gradient based techniques at reinforcement learning (RL) control tasks.

The aim of this work is to add more credence to the claim that EAs are a competitive technique for real valued optimisation by demonstrating how the search speed of EAs can be increased. We achieve this using two distinct techniques.

Firstly, knowledge from the optimisation of a set of source problems is reused to improve search performance on a set of unseen, target problems. This reuse of knowledge is achieved by embedding information with respect to the location of high fitness solutions in an indirect encoding (IE). In this thesis, we *learn* an IE by training generative models to model the distribution of previously located solutions to a set of source problems. We subsequently perform evolutionary search within the latent space of the generative part of the model on various target problems from the same 'family' as the source problems. We perform the first comparative analysis of IEs derived from autoencoders, variational autoencoders (VAE), and generative adversarial networks (GAN) for the optimisation of continuous functions. We also demonstrate for the first time how these techniques can be utilised to perform transfer learning on RL control tasks. We show that all three types of IE outperform direct encoding (DE) baselines on one or more of the problems considered. We also perform an in-depth analysis into the behaviour of each IE type, which allows us to suggest remediations to some of the pathologies discovered.

The second technique explored is a modification to an existing neuroevolutionary (the evolution of neural networks) algorithm, NEAT [137]. NEAT is a topology and weight evolving artificial neural network, meaning that both the weights and the architecture of the neural network are optimised simultaneously. Although the original NEAT algorithm includes recurrent connections, they typically have trouble memorising information over long time horizons. Therefore, we introduce a novel algorithm, NEAT-GRU, that is capable of mutating gated recurrent units (GRU) into

the network. We show that NEAT-GRU outperforms NEAT and hand coded baselines at generalised maze solving tasks. We also show that NEAT-GRU is the only algorithm tested that can locate solutions for a much harder navigational task where the bearing (relative angle) towards the target is not provided to the agent.

Overall we have introduced two novel techniques that have successfully achieved an increase in EA search speed, further attesting to their competitiveness compared to gradient based techniques.

Acknowledgements

I would first like to give my heartfelt thanks to my supervisors. **Karl Tuyls**, you believed in me enough to give me the opportunity to pursue this PhD. Throughout it you have provided me with a lot of freedom to conduct the research that was both pertinent and personally intrigued me. Thank you for introducing me to the field of Evolutionary Algorithms and robotics, and welcomingly me so warmly into the smARTLab.

Rahul Savani, you took on the responsibility as my second supervisor half way through my PhD and immediately got to work providing in-depth valuable technical advice. Your ability to grasp the bigger picture all the way down to the miniature details over such a wide variety of areas has always impressed and inspired me. Your stoic approach to problems has taught me how to be calm in the face of seemingly impossible tasks. Along with **Karl**, you have taught me how to both strive for excellence, and how to treat people for whom you are professionally responsible with respect and compassion.

I would never have gotten through this PhD without the advice, support and many laughs afforded by my colleagues at the University of Liverpool. **Greg Palmer**, we began this journey together as undergraduate students with no knowledge of AI. Together we discovered how fascinating it could be by using evolutionary algorithms to evolve simulated fighting agents for our second year group project. I remember the delight on your face when you excitedly presented me with the results of an overnight evolutionary run, which had produced an agent with an ability to snipe other agents from long distances. It was safe to say from that point, we were hooked. **Jacopo Castellini** your joy is infectious and it has helped me greatly through a number of really tough times. You were also a source of many interesting and helpful discussions. **Tom Spooner** thank you for the many conversations where we spoke about how much we love programming, and the many times that we threw AI ideas around. Others including **Kimberley McGuire**, **Daniel Fernandes Gomes**, **Bastian Broecker**, **Shan Luo**, **Paolo Paoletti**, **Paul Dunne**, **Frans Oliehoek**, **Martin Gairing**, and **Joe Jerome** provided lots of technical advice and support. Thank you to the numerous support staff over the years for providing a safe space to escape when the stresses of the PhD all got too much: **Lindsay**, **Rebekah**, **Jamie**, **Helen**, and **Alison**.

I would like to thank **Benjamin Schnieders**, a great friend who is unfortunately no longer with us. We shared a lot of great times together, the trials and tribulations of doing a PhD, and a particularly memorable trip to Germany to partake in the RoboCup @ Work competition. You taught me so much about about programming, C++, and the joys of robotics. We lost you at the end of 2018, at which time I knew you still viewed me as a C++ beginner, but I hope now you would be proud of how much I have learnt. Not a day goes by that there isn't a little bit of what you taught me in everything that I build.

Between October 2019 and April 2020 I undertook an internship at NNAISENSE in Lugano, Switzerland. I am so grateful for this opportunity, for which I thank **Faustino Gomez**. I learnt so much from my colleagues there, especially my good friend **Vojtěch Micka**. Vojta, thank you for looking after me throughout my time in Lugano, and for being such a great friend during my time there and since then also.

I would finally like to thank my friends and family. My parents, **Neal** and **Lynne**, for everything. Dad, thank you for instilling in me an enthusiasm for life and the confidence and abilities to achieve what I set my mind to. Mum, thank you for your unwavering belief in me and all your love and support throughout my entire life.

I would also like to thank my sister, **Win**, for always being a constant source of joy and tomfoolery.

Finally, I would like to thank **Katie** for the insurmountable support and guidance both during this PhD and personally. This PhD has been more stressful for you than it has been for me; partly because of my laid-back nature, and partly because you so greatly want me to achieve everything that you believe me capable of. You have taught me so much in such a short space of time about the type of person I want to be, and for that I am eternally grateful.

Contents

List of Figures

List of Tables

List of Abbreviations

AE	**A**uto**e**ncoder
BC	**B**ehaviour **c**haracteristic
BW	**B**ipedal **W**alker
CMAES	**C**ovariance **M**atrix **A**daptation **E**volution **S**trategy
CMC	**C**ontinuous **M**ountain **C**ar
CNN	**C**onvolutional **n**eural **n**etwork
CPPN	**C**ompositional **P**attern **P**roducing **N**etwork
CTRNN	**C**ontinuous **T**ime **R**ecurrent **N**eural **N**etwork
DDE	**D**ata-**d**riven **e**ncoding
DE	**D**irect **e**ncoding
DEIS	**D**irect **e**ncoding with an **i**nformed **s**tart
DNA	**D**eoxyribo**n**ucleic **a**cid
EA	**E**volutionary **A**lgorithm
EC	**E**volutionary **C**omputation
EDA	**E**stimation of **d**istribution **a**lgorithm
ENTM	**E**volvable **N**eural **T**uring **M**achine
EP	**E**volutionary **P**rogramming
ES	**E**volution **S**trategy
FFPSO	**F**orce **F**ield **P**article **S**warm **O**ptimisation
FL	**F**rozen **L**ake
FSM	**F**inite-**s**tate **m**achine
FST	**F**inite-**s**tate **t**ransducer
GA	**G**enetic **A**lgorithm
GAN	**G**enerative **A**dversarial **N**etwork
GPU	**G**raphics **p**rocessing **u**nit
GP	**G**enetic **P**rogramming
GRU	**G**ated **R**ecurrent **U**nit
IE	**I**ndirect **e**ncoding
KL	**K**ullback-**L**eibler
LSTM	**L**ong **S**hort-**t**erm **M**emory
MAV	**M**icro-**A**rial **V**ehicle
NE	**N**euro**e**volution
NEAT	**N**euro**E**volution of **A**ugmenting **T**opologies
NS	**N**ovelty **S**earch
NTM	**N**eural **T**uring **M**achine
PCA	**P**rincipal **C**omponent **A**nalysis
PSO	**P**article **S**warm **O**ptimisation
RL	**R**einforcement **L**earning
SGD	**S**tochastic **G**radient **D**escent
SLAM	**S**imultaneous **L**ocalisation **a**nd **M**apping
TWEANN	**T**opology and **W**eight **E**volving **N**eural **N**etwork
UAV	**U**nmanned-**A**rial **V**ehicle

UC	**U**niversal **c**ontroller
VAE	**V**ariational **A**uto**e**ncoder

Chapter 1

Introduction

Evolutionary algorithms (EA) are a suite of algorithms that can automatically find solutions to a wide variety of problems. EAs are inspired by the process of natural selection, whereby individuals that are comparatively fitter than the other members of a population have a larger chance of survival and thus, stand a higher chance of passing their genes onto subsequent generations. EAs emulate this process by subjecting each individual in the population to a fitness evaluation, and then selecting individuals for reproduction with a frequency directly proportional to their fitness values. Stochastic genetic operators, such as mutation and crossover, are subsequently applied to the genes of the selected individuals in order to further explore the search space. The processes of selection, crossover, and mutation working in tandem balances both exploitation and exploration of a search space, resulting in a powerful optimisation technique.

The nature of EAs allows them to be applied to many different types of optimisation problem, such as combinatorial optimisation and continuous optimisation. In the field of combinatorial optimisation, EAs have been applied to problems such as the travelling salesman problem [98], timetable scheduling [129] and job scheduling [101]. In the area of continuous optimisation, EAs have been applied to problems such as the optimisation of neural network controllers [17, 35, 80, 114, 123, 138], the design of optical fibre [87], and to the placement of wind turbines [150].

Given their applicability at optimising functions over $\mathbb{R}^n$, they are naturally compared to other continuous optimisation techniques that utilise the gradient. Using the gradient to find the global optima of a function is a wise strategy and can result in faster search than ignoring this information. It is for this reason that gradient based algorithms such as gradient descent or Newton's method are often preferred over EAs for continuous optimisation. However, if the search space is highly non-convex, or the gradients are very noisy or not available at all, derivative-free techniques, such as EAs, may be more fruitful.

The Deep Learning revolution of the 21^{st} century [44] is an example of where gradient based optimisation methods are more commonly used than the derivative-free counterparts, such as EAs. The backpropagation algorithm made it such that a many layer neural network is end-to-end differentiable. Therefore, most modern deep learning frameworks such as PyTorch and Tensorflow only implement gradient based optimisation algorithms under the assumption that they are superior to derivative-free alternatives. Faster search, combined with the fact that deep neural networks appear to have less local minima than previously assumed [21, 27, 75], provides the main argument for ignoring derivative-free techniques.

Despite this, experiments from the past decade have shown EAs to be comparative to gradient based techniques in terms of search speed and the fitness of the solution found. These examples come both from the area of reinforcement leaning

(RL) [123, 138], and from function approximation and prediction using neural networks [97]. This is owed to the inherent parallelisability of EAs and interesting conclusions with respect to gradient based optimisation. In [138], it is shown that EAs locate higher fitness solutions compared to gradient based RL techniques in just under half the Atari games they are trained to play. Furthermore, it is also shown that *random search* outperforms RL techniques, DQN and A3C in 3 and 6 games respectively. This suggests that in certain optimisation tasks, such as RL tasks, following the gradient, which could be noisy and non-informative, can even be detrimental to performance.

In an attempt to further dispel claims that EAs are not scalable to Deep Learning sized parameter sets, this thesis introduces two distinct techniques to improve the search speed of EAs. The promising results shown here illustrate that there are certainly more performance gains that can be achieved from EAs on top of the mounting evidence that they are just as powerful as gradient based techniques. As a result, it is hoped that EAs start to be recognised as a viable competitor, especially in the field of Deep Learning, and to convince the community that they should be included in modern Deep Learning libraries.

1.1 Research Questions

The aim of this thesis is the ask whether there exist techniques that can improve the search speed of EAs in a number of different domains, without compromising on the quality of the solutions found.

Firstly, we ask whether we can improve the performance of EAs in transfer learning tasks. Transfer learning is where knowledge that has been gained on some source problems can be reused to improve learning on some target problems. This reuse of knowledge results in a search speed increase by bootstrapping learning on new problems. We ask whether this is the case on a wide variety of optimisation tasks. We consider whether continuous function optimisation can benefit from EAs reusing knowledge. Given that many modern machine learning problems can be formulated as the minimisation or maximisation of some continuous function, $f : \mathbb{R}^n \rightarrow \mathbb{R}$, we believe this to be a sensible place to start. We subsequently ask whether new transfer learning techniques can assist in a number of control environments typically tackled by reinforcement learning algorithms.

We approach transfer learning in EAs by embedding knowledge from source problems in an indirect encoding (IE). An IE is a static (through the evolutionary search process) function that maps a genotype to a phenotype. A genotype is a vector of genes on which evolutionary operators are applied and the phenotype is that which is subject to fitness evaluation. We ask whether IEs can embed knowledge from previous source problems to improve search speed on unseen but similar target problems. In order to construct the aforementioned IEs, we use generative models to learn a model of previously found good solutions and subsequently perform evolutionary search in its latent space.

These questions, coupled with an approach on how to achieve it, leads to a first set of research questions:

RQ1: Can we use generative models to construct indirect encodings for evolutionary algorithms to perform transfer learning for the optimisation of continuous functions?

RQ2: Which generative model type produces an indirect encoding that improves search performance the most on continuous optimisation problems, and under what conditions?

RQ3: Can we use generative models to construct indirect encodings for evolutionary algorithms to perform transfer learning in reinforcement learning control tasks?

Secondly, we consider whether the inclusion of the long term memory mechanism, the gated recurrent unit (GRU) [22] in the seminal EA algorithm, NEAT [137], improves search speed on a number of generalised maze solving tasks. The first task consists of solving randomly generated mazes where the inputs to the agent are: the distance to the goal, the relative angle (the bearing) towards the goal, and proximity sensors to prevent crashing against walls. The second task consists of simpler environments where only the distance to the goal is provided as input to the agent. The second task is a much harder navigational task that requires the memorisation of previous distance readings in order to navigate successfully.

These considerations lead to a second set of research questions:

RQ4: Does the inclusion of gated recurrent units into NEAT result in improved evolutionary search performance on generalised maze solving tasks?

RQ5: Does the inclusion of gated recurrent units into NEAT result in improved evolutionary search performance on a much harder navigational task whereby bearing information is not available?

1.2 Contributions

This thesis contributes to its overall aim by presenting two separate techniques for increasing the search speed of evolutionary algorithms.

In Chapter 3, we answer **RQ1** and **RQ2**, and:

- We show that autoencoders, variational autoencoders (VAE), and generative adversarial networks (GAN) all have the ability to produce indirect encodings that improve evolutionary search compared to baselines on unseen continuous optimisation problems.
- We show that the performance of the indirect encodings derived from the generative models is highly dependent on the target problem being optimised and its relation to the source problems for which the indirect models were trained.
- We compare the performance of the three generative models and highlight pathologies that affect each of them and show under which conditions these occur. We subsequently suggest remediations to some of the pathologies.

In Chapter 4, we answer **RQ3**, and:

- We show that the techniques presented in Chapter 3 can be reformulated as a way to perform efficient transfer learning on RL control tasks.
- We show that autoencoders, VAEs, and GANs all have the ability to produce indirect encodings that improve evolutionary search compared to baselines on unseen RL control problems. We illustrate this on three OpenAI Gym benchmark environments.

- We compare the performance of all three generative models at producing IEs, highlight certain failure modes, and suggest remediations.

In Chapter 5, we answer **RQ4** and **RQ5**, and:

- We introduce a new algorithm, NEAT-GRU, that integrates GRU cells into NEAT networks.
- We show that NEAT-GRU reliably produces polices that outperform NEAT and hand-designed baselines at generalised maze solving tasks.
- We show that NEAT-GRU greatly outperforms NEAT at a harder navigational task whereby bearing information is not provided to the agent. In fact, NEAT is never able to produce solutions to this task, whereas NEAT-GRU produces a solution on every run.

1.3 Published Material

Throughout the course of this PhD a number of publications were produced. Chapters 4 and 5 are based around the following publications:

> **Chapter 4:** James Butterworth, Rahul Savani, Karl Tuyls. "Generative Models over Neural Controllers for Transfer Learning". In: *Parallel Problem Solving from Nature – PPSN XVII*. PPSN 2022, pp. 400-413 [16].
>
> **Chapter 5:** James Butterworth, Rahul Savani, Karl Tuyls. "Evolving Indoor Navigational Strategies Using Gated Recurrent Units in NEAT". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO 2019, pp. 111-112 [15].

There were also publications that were produced during the course of this PhD that were not included in this thesis. Work conducted during my masters was extended and published:

> James Butterworth, Bastian Broecker, Karl Tuyls, Paolo Paoletti. "Evolving Coverage Behaviours For MAVs Using NEAT". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS 2018. pp. 1886–1888 [17].

This work explored how NEAT could be used to evolve controllers for micro-aerial vehicles (MAV) performing dynamic coverage. Dynamic coverage is the process of maximally and continuously monitoring an area of interest. In this work controllers were evolved in simulation and then transferred to real MAVs.

Another publication that I was heavily involved with was the following:

> Lauren Parker, James Butterworth, and Shan Luo. "Fly Safe: Aerial Swarm Robotics using Force Field Particle Swarm Optimisation". In: CoRR abs/1907.07647 2019 [100].

In it we modified particle swarm optimisation (PSO) to be used as a goal finding algorithm for use on real MAVs. Each MAV represents a particle in PSO, which share information regarding distance to the goal with one another. The original PSO

algorithm allows the particles to move within close proximity to one another. However, this would be dangerous on a real swarms of MAVs. Therefore, we introduced a new algorithm, Force Field Particle Swarm Optimisation (FFPSO), which induces repellent force fields around the particles in PSO. We showed the applicability of this new algorithm in simulation and on a real swarm of MAVs.

1.4 Structure

The structure of this thesis is as follows. Chapter 2 introduces preliminary material needed to further understand the context of this thesis. Topics covered include evolutionary algorithms, neural networks, and generative models. Where previous algorithms have been used in our work, their descriptions have been moved to Chapter 2. Chapter 3 applies indirect encodings derived from generative models to continuous function optimisation. A comparison of three generative models at producing IEs is provided and remediations are suggested for the individual pathologies. Chapter 4 applies the techniques from Chapter 3 to the area of transfer learning in reinforcement learning control problems. All three models are compared on three OpenAI Gym environments and analyses of the results is carried out. Finally, in Chapter 5 we propose NEAT-GRU, an extension to NEAT that includes GRUs. We show how NEAT-GRU outperforms baselines and NEAT at both generalised maze solving and a harder navigational task where the relative angle to the goal is not provided to the agent.

Chapter 2

Preliminaries

2.1 Evolutionary Computation

Evolutionary Computation (EC) is a term that was coined in 1991 [36], which at the time referred to the field consisting of Genetic Algorithms, Evolution Strategies, and Evolutionary Programming. Since then, the field has expanded to include other sub areas such as, Genetic Programming, Differential Evolution, and Particle Swarm Optimisation. All the algorithms studied in the field of EC, referred to as Evolutionary Algorithms (EA), draw inspiration from the Darwinian theory of natural selection - the key mechanism of biological evolution.

2.1.1 Evolution by Natural Selection

Evolution by natural selection is a theory first proposed by Charles Darwin and Alfred Wallace [26], and subsequently expanded by Darwin in his seminal work, *On the Origin of Species* [25]. The theory aimed to explain the wide diversity of life on Earth, as well as the apparent observation that animals often seem highly adapted to the conditions of the environment that they inhabit. Darwin and Wallace suggested that the observed adaptation is due to a process called natural selection.

All life undergoes a “struggle for existence". This occurs due to the finite amount of resources available to sustain an exponentially growing population indefinitely. Due to this struggle for existence, many organisms do not survive long enough to reproduce. Variations naturally occur between individuals in a population. Some of these variations lead to advantages in this struggle for existence, thereby increasing the chance that a particular individual reproduces. These variations are passed on to the offspring of the individuals that were successful in reproducing. This process results in the observed change of a population through time, often in a way that renders the individuals more adapted to their environment. It is hypothesised, and widely accepted, that this process of evolution by natural selection has given rise to some of the most complicated and intelligent systems that exist, such as, flocks of birds, the eye, and the human brain.

When first proposed, Darwin’s theory of natural selection lacked a rigorous theory of heredity - a required component for any system undergoing natural selection [111]. It was not until the integration of Mendel’s theories of inheritance with the Boveri-Sutton chromosome theory of inheritance - which identifies chromosomes as the carriers of genetic material - and subsequently, the integration of these ideas with the theory of natural selection by Fisher in 1930, that evolution by natural selection as a theory could stand up to rigorous scrutiny. It was later discovered that chromosomes are made up of a molecule called deoxyribonucleic acid (DNA). DNA copying errors and recombination (crossover) of the DNA during meiosis result in the natural variation alluded to in Darwin’s original work.

The full collection of genetic material of an organism is known as the **genotype**. DNA carries this genetic information and is the means by which this information is passed between organisms through the generations. The **phenotype** is the set of all the observable characteristics of the organism itself. The phenotype can be thought of as the *realisation* of the genotype having gone through a developmental procedure and then subject to effects of its environment. The phenotype is greatly influenced by the genotype that it is derived from. For example, in humans the HTT gene is responsible for the creation of a protein called Huntingtin. A mutated HTT gene can cause Huntington's disease, a fatal neurodegenerative disease with no known cure.

2.1.2 Evolutionary Algorithms

A Canonical Evolutionary Algorithm

Most evolutionary algorithms have a set of core similarities making them distinct compared to other search algorithms.

- A population of individuals or solutions is maintained which is altered through a number of discrete time steps, known as a generation.
- Each solution is subject to a fitness evaluation, the result of which has an influence on some selection procedure.
- Genetic operators are applied to selected members of the population. Many types are used, mainly crossover and mutation.
- Each individual is typically represented by a vector of values known as the genotype. These values can take on many forms, and do, depending on the type of problem being solved.

One should be able to see the similarities to biological systems beginning to appear. Populations of solutions are maintained at any one time. Each individual in a *finite* population is subject to a fitness evaluation determining whether its genes are passed onto the next generation; this is akin to the 'struggle for existence' alluded to by Darwin. Variations are induced in individuals via genetic operators such as crossover (similar to meiosis) and mutation. The genotype of the digital individuals is a vector of values akin to the genes held in DNA.

Figure 2.1 demonstrates a typical evolutionary algorithm. A population of solutions is randomly initialised and subsequently subject to a fitness evaluation. A fitness evaluation consists of each individual in the population being evaluated according to some environment-dependent fitness function. In the case of a mathematical optimisation problem, the fitness would be the value $f(\mathbf{x})$ where $\mathbf{x}$ would be the genotype of the individual. Whereas, if the problem of interest is a routing problem, the fitness function may consist of the length of the route obtained by the individual.

When the entire population has been evaluated, the population is then subjected to a selection procedure. The selection procedure consists of selecting individuals for the application of genetic operators. The majority of selection algorithms give precedence to individuals with a larger fitness score. A wide variety of selection procedures exist, some of the more common ones are:

- **Truncation selection**. The population is ranked according to fitness score and only the top k individuals are selected.

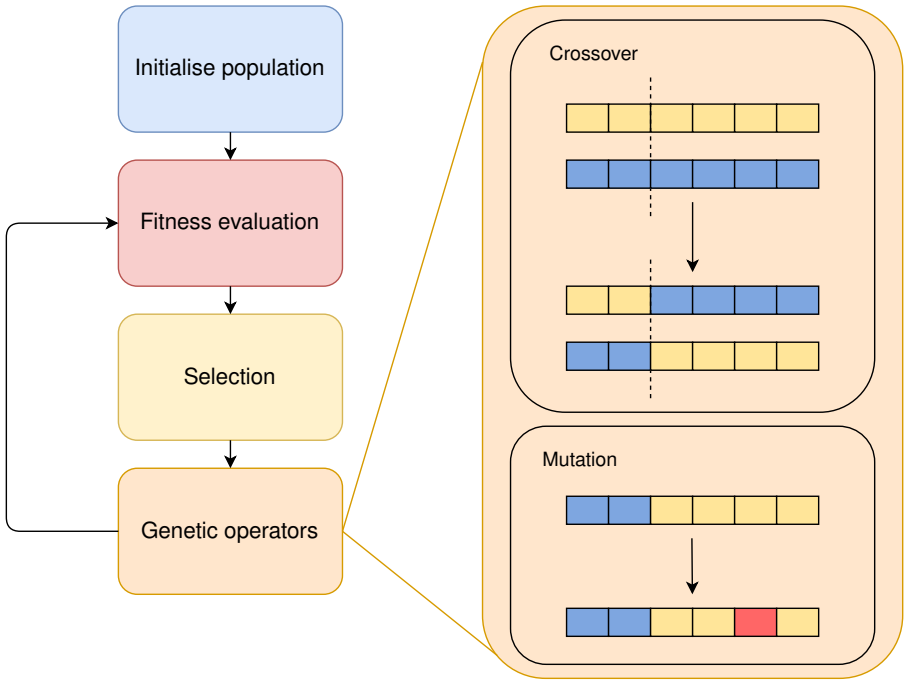


FIGURE 2.1: A canonical evolutionary algorithm. The population of individuals is randomly initialised (blue). Each individual is subject to a fitness evaluation (red). Individuals are selected to reproduce based on their newly acquired fitness values (yellow). Genetic operators, such as mutation and crossover, are applied to the genes of the individuals (orange). The new child individuals form the new population, which are in turn subject to fitness evaluations. This process repeats until a certain fitness is achieved or a maximum number of generations is attained.

- **Roulette wheel selection**. The individuals of the population are selected stochastically where the probability of selection for individual i is $p_i = f_i / \sum_{j=1}^{n} f_j$, where n is the population size and f_i is the fitness of individual i.
- **Tournament Selection**. A set of k individuals in randomly chosen from the population and the individual with the highest fitness is selected.

Once individuals have been selected, they undergo a number of genetic operators. Similar to selection, there are a wide variety of operators but the most common ones are **crossover** and **mutation.**

One-point crossover consists of randomly splitting the genotypes from two selected parents at the same location, as shown in Figure 2.1. Once split, two children are created, each with a subset of genes from both parents. Other crossover techniques exist such as two-point crossover, in which the random splitting occurs twice on each genotype - this technique can be generalised to k-point crossover for k number of splits. Another common crossover technique is uniform crossover in which each individual gene of the child is selected from either parent with equal probability.

Mutation consists of randomly altering the value of some gene, as shown in Figure 2.1. Mutation operators take on different forms depending on the data type of the gene. For a binary genotype, mutation would take the form of a bit flip. However, for a floating point, genotype mutation might consist of adding a random number derived from a gaussian distribution with zero mean and some variance,

σ, commonly known as the mutation power. Each gene has a certain probability of undergoing mutation, known as the mutation rate. Both the mutation rate and the mutation power are hyperparameters of the EA, however, as we will see later, many algorithms do adjust these values at runtime.

Once genetic operators are applied, the produced children form the population of the next generation, which go on to be evaluated, selected and modified themselves. This process will continue until some stopping criteria is met. Typical stopping criteria are: a maximum number of generations has been reached, a threshold fitness score has been achieved, or a time limit has been passed.

Brief history

Ideas of algorithms that draw inspiration from evolution, Evolutionary Algorithms, first started to appear in the 1940s and 1950s. Genetical or evolutionary search was first considered in 1948 as a search method to "organise" Turing's unorganised machine, which can be considered as one of the first randomly connected binary neural network designs [145]. Turing hypothesised that a system of genes required to represent his proposed unorganised machines would not be very complex, and hinted at the fact that search similar to natural selection could be used to optimise such a machine.

In 1958, Friedberg designed and implemented a system called a 'Learning Machine' in which a computer program was represented as a sequence of instructions [39]. This program was subject to operators akin to mutation and crossover, and instructions that are present in successful programs remain for subsequent evaluations. Soon after, in 1962, Bremermann [11] embedded the ideas from evolution firmly in the area of mathematical optimisation. He solved systems of linear equations and linear programming problems by applying mutation and crossover to genotypes consisting of binary genes.

As time progressed, three separate evolutionary algorithmic ideas emerged, each with slightly different details and target applications. These algorithmic areas were that of Genetic Algorithms (GA), Evolution Strategies (ES), and Evolutionary Programming (EP).

Genetic Algorithms

Genetic Algorithms, pioneered and made popular by John Holland [54], focused attention on the development of application-independent evolutionary algorithms. This was realised by adopting a binary string as the genotype, and, if required, mapping this genotype to an application appropriate phenotype. In the case of a problem such as the Knapsack Problem, a binary string representation is very natural, a 1 at position i of the string indicates the presence of the i^{th} item in the bag and a 0 represents its absence. However, in the case of problems such as continuous function optimisation, an appropriate mapping between the binary genotype representation and the floating point application appropriate representation must be designed. An inadequate genotype-phenotype mapping can lead to a large performance degradation [119] and issues such as Hamming Walls, where a large number of synchronised mutations are required to escape a local fitness peak.

The application-independent binary string representation has the advantage that it requires little modification when being applied to different problems and the same genetic operators can be reused. Genetic Algorithms typically employ a significant

amount of crossover and very little mutation, which is seen as a "background operator, assuring that the crossover operator has a full range of alleles so that the adaptive plan is not trapped in a local optima" [54].

Evolution Strategies

Ideas of evolution strategies began in the 1960s and 1970s with the work of Rechenberg and Schwefel [7][1]. The focus of evolution strategies is on the optimisation of real-valued functions [28]. Due to this, mutation typically consists of the addition of a random number drawn from a gaussian distribution with a mean of 0 and a variance, σ, known as the mutation strength. Mutation plays a much greater role in evolution strategies than in genetic algorithms. Crossover, or recombination, consists of either averaging the genes of k parents (intermediate recombination) or randomly selecting each gene of the child from the corresponding gene of either of the two parents (dominant recombination). However, unlike in a genetic algorithm where two parents produce two offspring, recombination in an evolution strategy produces only one child. A key feature of most evolution strategies is the runtime adaption of the mutation step sizes, this is performed in order to "adapt to the properties of the fitness landscape" [7].

The (1 + 1)-ES is one of the first, and most conceptually simple to understand, ES algorithms. The 1971 dissertation of Rechenberg [110] was the first work to analyse the (1 + 1)-ES algorithm on real valued functions. (1 + 1)-ES has a population of 2, one parent and one child. Each generation the parent produces a child via mutations of its genes using a gaussian distribution with a mean of 0 and a standard deviation, σ. If the fitness of the child is the same as or greater than that of the parent, the child is used as the parent in the next generation, otherwise the child is discarded.

Algorithmic variations of (1 + 1)-ES continued to be explored, such as the more generic (μ + λ)-ES, where μ and λ represent the number of parents and children in the population respectively. In order to keep the population size constant, the worst performing λ members of the population are removed after each generation. With a value of $\mu > 1$ recombination could be applied as described above, which can speed up evolution substantially [7].

It was discovered early into the development of evolution strategies that the performance of the algorithm was highly dependent on the choice of the mutation strength, σ [28]. Therefore, the mutation strengths (one for each gene) were coevolved with the parameters to be optimised, resulting in a genome of double the original size. This resulted in mutation strengths that responded to the specific underlying fitness landscape, leading to faster optimisation.

The multivariate gaussian distribution used to mutate the genes in the evolution strategies discussed so far is defined as $\mathcal{N}_k(\mu, \Sigma)$, where μ is the mean vector of zeros of size k, and Σ is the covariance matrix of size $k \times k$. In the case of the (μ + λ)-ES set of algorithms Σ is defined in the following way:

$$\Sigma = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k^2 \end{pmatrix}$$

[1] I would prefer to cite the original papers but they are in German. Instead, I cite an introduction to evolution strategies co-authored by Schwefel in which he describes the origins.

Where σ_k is the mutation strength for the k^{th} gene. The covariance matrix, Σ, is diagonal in this case. However, the covariance matrix of a gaussian distribution can be any positive definite matrix, therefore it can have elements other than 0 in the non-diagonal entries. This results in a normal distribution with a *rotated* hyperellipsoid, as opposed to that with a diagonal covariance matrix, which has a non-rotated hyperellipsoid shape. A more expressive gaussian distribution can lead to much quicker convergence to optima and greater adaptation to the particular fitness landscape.

A more recent evolution strategy known as covariance matrix adaptation evolution strategy (CMAES) [50] does exactly that. It adapts the entirety of Σ throughout search resulting in one of the most powerful algorithms for finding global solutions in complex real-valued optimisation problems, outranking other search algorithms on standard benchmarks [51]. CMAES is covered in much more detail in Section 2.1.3.

CMAES is an estimation of distribution algorithm (EDA). An EDA alters the parameters of the probability distribution that produces the individuals each generation, rather than the individuals themselves, as is the case for GAs. An alternative family of EDAs are those known as Natural Evolution Strategies (NES) [149]. NES algorithms approximate the natural gradient of the parameters of the search distribution and perform gradient ascent to move the search distribution into a region of higher expected fitness. An NES known as xNES uses a multivariate gaussian as its search distribution and therefore has many similarities to CMAES, however, all the parameter updates in xNES are derived using the principles of gradient ascent, whereas other heuristics are used to derive the parameter updates in CMAES. A simplified version of xNES is used in [123] to show that evolution strategies can outperform standard reinforcement learning techniques on modern RL benchmarks.

Evolutionary Programming

Evolutionary programming was introduced in the 1960s by Fogel as a technique for the optimisation of finite-state machines (FSM), in particular, finite-state transducers (FST). An FST is a model of computation that accepts inputs from a finite input alphabet and generates output from a finite output alphabet. It does this by moving between a finite number of states as a result of the input and subsequently generating an output symbol with every transition. An FST is seen as a more general model of computation than an FSM due to the fact that it additionally *outputs* symbols.

In the work of Fogel [38], the evolutionary algorithm proceeds as those discussed previously, with the main differences being the structure to which they are applied. Fogel originally only used mutations as a method of variation, however, he did suggest the concept of recombination on FSTs. Five different types of mutation were considered [37]:

- Add a state and randomly assign all the input-output and input-transition pairs for this state.
- Alter the starting state.
- Delete a state.
- Change an output symbol of a particular state.
- Alter a particular state-transition associated with a single input in a single state.

Genetic Programming

Genetic Programming (GP) is a more recent subfield of evolutionary computation that aims to evolve computer programs using evolutionary algorithms [69]. This is achieved by representing individual programs as syntax trees and applying evolutionary operators directly to the trees. Similarly to genetic algorithms, mutation plays a much smaller role in GP than recombination. A significant problem that arises during GP is 'bloat' in which the syntax trees grow to very large sizes as search proceeds. A number of techniques have been introduced to prevent this, such as setting a maximum tree depth or applying a penalty in the fitness function for larger trees [32].

2.1.3 CMAES

Covariance matrix adaptation evolution strategy (CMAES) [50] is a powerful evolution strategy that is used extensively in Chapter 4, as such we describe it in detail here[2]. It is known to be one of the leading algorithms for the optimisation of complex real-valued functions [31], even when the search space is rugged. The superior performance of CMAES as compared to other algorithms was the main motivation for its use in this thesis, where speed of convergence was an important metric to maximise.

Each generation λ individuals are sampled from a multivariate normal distribution:

$$\mathbf{x}_k^{(g+1)} \sim \mathcal{N}(\mathbf{m}^{(g)}, (\sigma^{(g)})^2\mathbf{C}^{(g)}) \qquad \text{for } k = 1, ..., \lambda \tag{2.1}$$

where $\mathbf{x}_k^{(g+1)} \in \mathbb{R}^n$ is the k^{th} individual (search point) from generation $g+1$; $\mathbf{m}^{(g)} \in \mathbb{R}^n$ is the mean of the distribution at generation g; $\sigma^{(g)} \in \mathbb{R}_{>0}$ is the step size, or "overall" standard deviation of the distribution at generation g; $\mathbf{C}^{(g)} \in \mathbb{R}^{n\times n}$ is a covariance matrix at generation g, such that $(\sigma^{(g)})^2\mathbf{C}^{(g)}$ is the full covariance matrix of the multivariate normal distribution, $\mathcal{N}$.

Similar in nature to other EDAs, the question becomes where in search space the population producing distribution should be located and what shape should it take in order to locate higher fitness solutions? Firstly, the centroid, or mean, $\mathbf{m}$, of the distribution is altered. The new mean, $\mathbf{m}^{(g+1)}$, is calculated as a weighted average of the μ highest fitness points in the population:

$$\mathbf{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(g+1)} \tag{2.2}$$

where $\mathbf{x}_{i:\lambda}^{(g+1)}$ is the i^{th} highest fitness solution such that $f(\mathbf{x}_{1:\lambda}^{(g+1)}) \geq f(\mathbf{x}_{2:\lambda}^{(g+1)}) \geq ... \geq f(\mathbf{x}_{\lambda:\lambda}^{(g+1)})$ and f is the function to be maximised; $w_i \in \mathbb{R}_{>0}$ are positive weight coefficients such that $\sum_{i=1}^{\mu} w_i = 1$ and $w_1 \geq w_2 \geq ... \geq w_\mu > 0$.

Equation 2.2 points to the fact the CMAES uses a truncation selection variant as its selection procedure by only considering the top μ individuals when calculating the new mean. The final mean update equation is rewritten as an update of m and incorporates a step size hyperparameter, c_m:

$$\mathbf{m}^{(g+1)} = \mathbf{m}_{(g)} + c_m \sum_{i=1}^{\mu} w_i (\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}) \tag{2.3}$$

[2]The majority of our description of CMAES is an adaptation of [49].

where $c_m \leq 1$, but is usually set to 1. Equation 2.3 is equal to 2.2 when the default setting, $c_m = 1$ is used.

As with other evolution strategies, CMAES also modifies the 'mutation rates' at run-time. For CMAES, this takes the form of modifying both σ and $\mathbf{C}$. A larger σ results in individuals being sampled from a larger portion of the search space, and modifying $\mathbf{C}$ alters the shape of the multivariate distribution. The covariance matrix, $\mathbf{C}$, is updated every generation as follows:

$$\mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu \sum w_j)\mathbf{C}^{(g)} + c_1 \mathbf{p}_c^{(g+1)} \mathbf{p}_c^{(g+1)\top} + c_\mu \sum_{i=1}^{\lambda} w_i \mathbf{y}_{i:\lambda}^{(g+1)} \mathbf{y}_{i:\lambda}^{(g+1)\top} \tag{2.4}$$

where

$$\mathbf{y}_{i:\lambda}^{(g+1)} = (\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)})/\sigma^{(g)} \tag{2.5}$$

and $\mathbf{p}_c^{(g+1)}$ is the evolutionary path and is calculated as:

$$\mathbf{p}_c^{(g+1)} = (1 - c_c)\mathbf{p}_c^{(g)} + \sqrt{c_c(2 - c_c)\mu_{eff}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{2.6}$$

where c_1, c_μ, and c_c are hyperparameters that are typically set to default values and are rarely altered by the practitioner. μ_{eff} is known as the *variance effective selection mass* and is calculated as:

$$\mu_{eff} = \frac{1}{\sum_{i=1}^{\mu} w_i^2} \tag{2.7}$$

The step size, σ, is also updated at every generation as follows:

$$\sigma^{(g+1)} = \sigma^{(g)} exp\left[\frac{c_\sigma}{d_\sigma}\left[\frac{||\mathbf{p}_\sigma^{(g+1)}||}{\mathbb{E}||\mathcal{N}(\mathbf{0}, \mathbf{I})||} - 1\right]\right] \tag{2.8}$$

where c_σ and d_σ are hyperparameters, and $\mathbf{p}_\sigma^{(g+1)}$ is calculated as:

$$\mathbf{p}_\sigma^{(g+1)} = (1 - c_\sigma)\mathbf{p}_\sigma^{(g)} + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}} \mathbf{C}^{(g)\,-\frac{1}{2}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{2.9}$$

and $\mathbb{E}||\mathcal{N}(\mathbf{0}, \mathbf{I})||$ is the expected norm of a vector of size n drawn from $\mathcal{N}(\mathbf{0}, \mathbf{I})$, which can be approximated as:

$$\mathbb{E}||\mathcal{N}(\mathbf{0}, \mathbf{I})|| \approx \sqrt{n}(1 - \frac{1}{4n} + \frac{1}{21n^2}) \tag{2.10}$$

For more information regarding the reasoning behind the complex update rules, and suggested default values for the hyperparameters, we refer the reader to [50] and [49].

2.2 Indirect Encodings

The main basis of Chapters 3 and 4 is that one can capture knowledge from source problems in an Indirect Encoding (IE), what is an indirect encoding? An IE is a mapping from a genotype to a phenotype. A genotype is the structure that genetic

operators are applied to, and the phenotype is that which is subject to fitness evaluation.

An illustrative example comes from biology, where the complex structure of our brain with its 100 trillion connections and 100 billion neurons is represented in a mere 30 000 genes in DNA. Our DNA is our genotype which undergoes processes such as mutations and meiosis (akin to crossover), and our brains are part of our phenotype, resulting in our behaviours that are subject to the forces of natural selection (akin to fitness evaluations). The developmental process that transforms DNA into the complex organism can be viewed as an indirect encoding because there is not a one-to-one mapping between the genes and the different traits of the phenotype. It is clear that a vast amount of information is compressed and stored within DNA, and it is this impressive feat that EA practitioners wish to emulate.

In the EA literature IEs are referred to in a number of ways, such as: Artificial Embryogeny [136], Developmental Encodings [144], Artificial Development [68], Maturation Functions [52], Genotype-Phenotype Maps [96], and Representations [119]. There are a plethora of forms that an artificial IE can take, such as: a continuous function, $f : \mathbb{R}^n \to \mathbb{R}^m$; a binary representation of integers, $f : \{0,1\}^n \to \mathbb{Z}$; a formal grammar that applies a set of production rules turning starting symbols into complex character strings; a bijective function mapping each individual gene to each individual trait in the phenotype (a direct encoding). Constraints tend to be imposed by the domain on the phenotype form, however the practitioner can choose whatever genotype representation and IE they wish, as long as the IE produces the appropriate phenotype form.

There are many performance advantages of using IEs over direct encodings. HyperNEAT [135] is an example of an algorithm that uses an indirect encoding to evolve neural networks that are typically larger than those optimised by EAs (neural networks are explained in more detail in Section 2.3). HyperNEAT does this by determining the weights of a large neural network (the phenotype) by querying a small neural network with the coordinates of the neurons of the large network. The small network, known as a Compositional Pattern Producing Network (CPPN), maps the coordinates of two neurons to the weight value between those neurons. Figure 2.2 illustrates how the weight of each connection of the substrate (the larger network) is determined by querying the smaller sized CPPN.

The evolutionary process of HyperNEAT involves evolving the weights of the CPPN, *not* the larger network. Due to the fact that the CPPN has a much smaller number of parameters to evolve than the larger network, this evolutionary process can proceed much faster than if one was evolving the weights of the larger network directly. This is an example of an IE of the form $f : \mathbb{R}^n \to \mathbb{R}^m$ where $n << m$. Using the HyperNEAT algorithm can lead to increased speed of search and a higher fitnesses of the located solutions [24, 135], especially when there is a high level of regularity required in the phenotype.

The CPPN, introduced in [134], is a neural network that has activation functions other than the typical sigmoid function. Activation functions used include gaussian, sine, $|x|$, and more (Figure 2.3). These additional functions assist in the creation of repetitive, symmetric, and regular weight patterns in the larger phenotypic network. This can help exploit the intrinsic geometry of the task to which the large phenotypic network is applied to. [134] explores the advantages of HyperNEAT on a visual discrimination and a robotic food gathering task.

Another example where using a modified IE can assist with search is when representing integers using boolean values, i.e. $f : \{0,1\}^n \to \mathbb{Z}$. Apart from the usual binary numeral system that can be used to represent integers, one can also construct

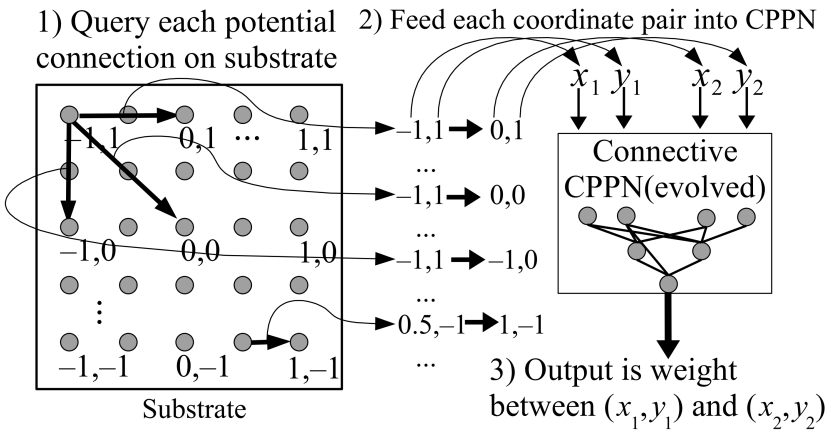


FIGURE 2.2: The neurons of the larger network are defined by their coordinates with respect to one another. For every permutation of neuron pairs in the substrate of the larger network, the x and y values of the coordinates are input into the CPPN. The CPPN outputs a weight value for the connection between every neuron pair. *Image source:* [135]

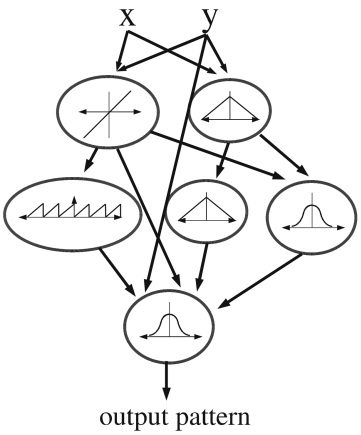


FIGURE 2.3: An example CPPN that takes substrate coordinates as input and outputs a connection weight. The neuron activation functions consist of a number of functions such as gaussian, sine, sigmoid, amongst others. *Image source:* [134]

a different mapping from boolean symbols to integers, one example being Gray code [46]. Table 2.1 illustrates the difference between Gray code and the binary numeral system for representing integers 0 to 8. Gray code was designed such that adjacent integers differ by only a single bit flip in their Gray code representation. This is particularly useful when performing evolutionary search over a boolean genotypes because certain fitness functions can contain Hamming cliffs. This occurs when the movement between neighbouring integers, for example 7 and 8, requires the flip of many bits in binary representation, 0111 and 1000, respectively. If 8 has a higher fitness value than 7, then only the specific rare combination of 4 bit flip mutations is required to achieve a higher fitness. On the other hand, in many problems, Gray code creates a much smoother fitness function by requiring only a single bit flip to move between adjacent integers. Binary and Gray encodings can be interpreted as IEs, by providing different mappings between boolean vectors and integers.

Recent works [6, 18, 40, 57, 58, 96, 106] use machine learning techniques to *learn*

Integer	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100

TABLE 2.1: Binary and Gray code representations of integers 0 - 8.

an IE. These IEs typically take the form $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ because many are represented by neural networks. Employing these IEs in search often results in much faster search speed and, because these IEs are automatically created, they are bespoke to the problem at hand. Learnt IEs represented by neural networks are the type of IE that we explore in greater detail in Chapters 3 and 4.

2.3 Neural Networks

An artificial neural network is a computational model that was originally inspired by neural networks in the brain. It consists of 'neurons' and connections between neurons that mimic synapses in the real brain. Signal travels through layers of neurons across the connections to an output. As signals travel through the network they are modified by real values associated with each connection, these are commonly referred to as weights. Like synapses in the brain, these weights can be altered such that the same input results in a different output. An artificial neural network learns by altering these weights such that a set of desired outputs is realised for some particular inputs.

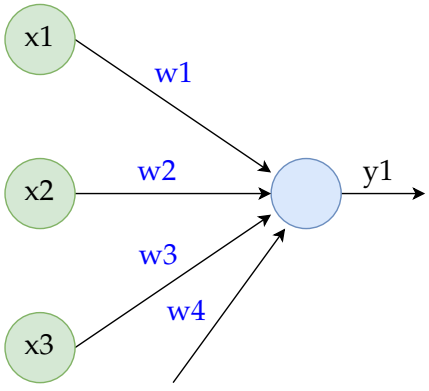


FIGURE 2.4: A model of a neuron in an artificial neural network. Inputs are in green, weights are in blue. Inputs get multiplied by weights and summed at the output neuron, shown in blue.

An example neuron is shown in Figure 2.4. It shows inputs in green boxes which are real numbers, weights in blue belonging to each connection and an output neuron in which all inputs are aggregated to produce output y_1. The mathematical function computed by the neuron in Figure 2.4 is:

$$y_1 = \phi(w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4) \tag{2.11}$$

where ϕ is typically some non-linear function commonly known as the activation function. It is clear that the output of the neuron for a specific input is significantly influenced by the values of the weights. Each neuron also typically has a bias input, represented as the w_4 connection in Figure 2.4.

The general form for a neuron of this type would be:

$$y = \phi(\boldsymbol{w}^T \boldsymbol{x} + b) \tag{2.12}$$

where $\boldsymbol{x}$ is a vector of inputs, $\boldsymbol{w}^T$ is the transpose of a vector of weights, and b is the bias input.

One can arrange a number of neurons in a single layer such that each neuron in the layer computes a different non-linear function on the inputs in the following way:

$$\boldsymbol{y} = \phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{2.13}$$

where $\boldsymbol{x}$ is still a vector of inputs, $\boldsymbol{W}$ is now a *matrix* of real valued weights, and $\boldsymbol{b}$ is now a *vector* of bias inputs.

Deep learning is a term referring to the situation in which a number of neural network layers are stacked one after another. An example of a 3 layered deep neural network is as follows:

$$\boldsymbol{y} = \phi(\boldsymbol{W_3}\phi(\boldsymbol{W_2}\phi(\boldsymbol{W_1}\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}) + \boldsymbol{b_3}) \tag{2.14}$$

This is known as a deep feedforward neural network. At this abstraction, Equation 2.14 little resembles the biological neuron abstraction from Figure 2.4, however it is important to understand the origins of these models.

The learning problem for a feedworward neural network consists of altering the weight matrices $\boldsymbol{W_1}, \boldsymbol{W_2}, ..., \boldsymbol{W_n}$ and the bias vectors $\boldsymbol{b_1}, \boldsymbol{b_2}, ..., \boldsymbol{b_3}$ (for which we will now collectively refer to as $\boldsymbol{\theta}$) such that one maps $\boldsymbol{x}$ to a desired $\boldsymbol{y}$.

2.3.1 Brief history

Models similar to that described above were first conceptualised in the 1940s with the seminal work of McCulloch and Pitts [88] and also in Turing's 1948 work on unorganised machines [145]. In 1958 Frank Rosenblatt invented the perceptron [118], which is the same model as that of Equation 2.12, where ϕ was set to be the heavyside step function (outputs 1 if some threshold is reached, otherwise 0 is outputted). This resulted in a binary classifier designed for image recognition. It was later built in hardware resulting in a machine called the 'Mark 1 Perceptron'.

Despite promising initial results, it was subsequently proven that the perceptron had limitations, in that it could not classify data points that were not linearly separable according to their classes [91]. This was because the perceptron was a single layer neural network. It was not until 1975 that the backpropagation algorithm was invented, resulting in a method for the training of neural networks with multiple layers [148], thereby resolving the limitations of single layered neural networks like the perceptron.

A number of neural network variants were invented in order to more efficiently process different types of data. Convolutional neural networks (CNN) [74] are particularly suited to processing images and videos, whereas long short-term memory (LSTM) [53] enhances performance on sequences of data.

At the turn of the century, the computational cost to train neural networks was significant compared to rival algorithms such as support vector machines. CNNs had therefore fallen out of favor at computer vision conferences until the early 2010s when two key works implemented fast CUDA kernels to train CNNs on the GPU, that of Ciresan [23] and Krizhevsky [70]. This resulted in dramatic improvements in vision benchmarks and with AlexNet [70] winning the ImageNet competition by a significant margin. The availability of lower cost graphics processing units (GPU) coupled with them being used to train large neural networks has lead to the deep learning revolution, whereby deep learning has begun to penetrate many different industries.

2.3.2 Backpropagation

The backpropagation algorithm is that which is used to calculate the gradients of each weight of the neural network with respect to the loss function. The loss function outputs a single real value that represents the distance between the current output of the neural network and the desired output. The gradient of this with respect to the weights is informative because it suggests which direction to alter the weights in order to reduce the loss, thereby resulting in a network output closer to the desired output.

Once the gradients have been computed, one can then use an optimisation algorithm that takes advantage of gradient information to compute the new value of each weight. Stochastic gradient descent (SGD) was initially used to perform this optimisation step, which consists of taking a small step in the opposite direction of the gradient. Recently, more powerful optimisation algorithms have been proposed and have become the default for training neural networks, such as ADAM [63]. ADAM has a number of additional features such as computing adaptive learning rates for each individual weight and keeping an exponentially decaying average of gradients to reduce gradient variance.

2.3.3 Gated Recurrent Units (GRU)

A recurrent neural network is one which contains cycles. This means that it can reuse information from previous inputs to the network to inform its current decision, which in turn gives the network memory.

Figure 2.5 shows a vanilla recurrent neuron for which the output is computed as:

$$y_t = \phi(w_1 x_t + w_2 y_{t-1} + w_3) \tag{2.15}$$

where x_t and y_t are the input and output at time step t respectively; w_1, w_2 and w_3 are adjustable weights; and y_{t-1} is the output from the previous time step.

This type of neuron (or layer of neurons) is able to memorise inputs over short periods of time but struggles to learn longer term dependencies. The problem is that the gradient of the loss with respect to the weights can explode or vanish when the length of the sequence inputted into the network is large. This is because a small change in the recurrent weight, w_2, can have a large effect on the output as the number of time steps inputted into the network gets larger.

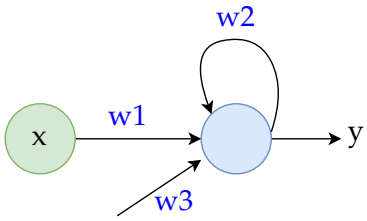


FIGURE 2.5: A model of a recurrent neuron. Input is in green, weights are in blue. Input gets multiplied by w_1 and summed with w_2 multiplied by the previous output y_{t-1} and the bias w_3.

Vanishing and exploding gradients rendered the vanilla recurrent network impractical for learning longer sequences, thereby incentivising a new architectural design, the LSTM [53]. The LSTM consists of a separate memory cell that can open and close depending on the current input and previous hidden state. This meant that information can be isolated without contamination from the rest of the input until it is required at some point in the distant future. The LSTM architecture circumvented the exploding and vanishing gradient problems of previous recurrent networks, resulting in a network that was capable of learning over much longer sequences.

A more recently invented architecture, similar in nature to the LSTM, is the gated recurrent unit (GRU) [20]. The GRU performs just as well as, and sometimes better than, the LSTM, whilst simultaneously having less weights to optimise. Given that it is a key component of Chapter 5, we will describe it in more detail here.

The full calculation performed by a GRU is given in equations 2.16 - 2.19, resulting in an output, h_t, known as the hidden state, at every time step, t.

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \tag{2.16}$$
$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \tag{2.17}$$
$$\tilde{h}_t = \phi(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{2.18}$$
$$h_t = z_t \odot \tilde{h}_t + (1 - z_t) \odot h_{t-1} \tag{2.19}$$

where W_r, W_z, W_h, U_r, U_z, U_h are weight matrices; b_r, b_z, and b_h are vectors of weights; σ and ϕ are the sigmoid and hyperbolic tan activation functions respectively, and $\odot$ is the Hadamard product, also known as the element-wise product. Figure 2.6 illustrates this computation.

A GRU consists of two main operations or 'gates', the reset gate and the update gate. The reset gate (equation 2.16) determines how to combine the new inputs with the previous hidden state and subsequently suggests a new candidate value, $\tilde{h}_t$, to store. The update gate (equation 2.17) decides how much of the previous hidden state information should be allowed to remain and how much of the new candidate information to store instead. This provides a mechanism by which the current hidden state can be kept around indefinitely as long as the update gate chooses not to change it. Depending on the new inputs, the update gate may determine that it is appropriate to store the new information instead.

2.4 Neuroevolution

Neuroevolution (NE) is a field of evolutionary algorithms concerned with the evolution of neural networks [133]. Unlike the neural networks in Section 2.3, which

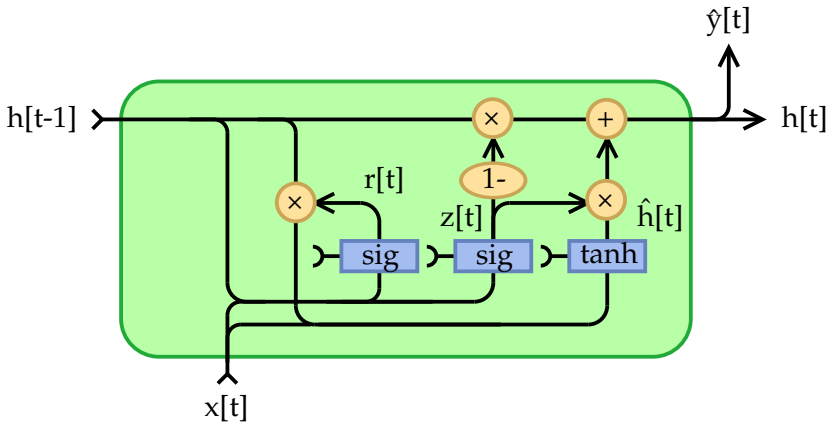


FIGURE 2.6: Illustration of a GRU cell. Sigmoid and hyperbolic tan activation functions are represented by 'sig' and 'tanh' respectively. The multiplication symbol, $\times$, represents the Hadamard product. *Image source:* https://en.wikipedia.org/wiki/Gated_recurrent_unit

use gradient descent to optimise the weights, the weights of neural networks undergoing neuroevolution are optimised using evolutionary algorithms. Additionally, given that EAs do not require that which they are optimising to be differentiable, they can optimise additional parts of the neural network such as the architecture [90, 109, 137] and the activation functions [8, 72].

Results from the field of NE in the last 5 years have shown that they can often be competitive with state of the art reinforcement learning techniques [123, 138]. One of the reasons is due to the increased speed of processing compared to traditional RL techniques. EAs are inherently parallelisable meaning that one can provide more CPU cores and continue to get speed improvements. In [138] it is shown that a genetic algorithm can train in ∼4 hours compared to ∼7-10 days for a state of the art RL technique, the Deep Q-Network.

Another reason NE shows promise compared to gradient based methods is that following the gradient can often lead to search getting stuck in local optima. Results in [138] suggest that following the gradient can be extremely detrimental to performance compared to evolutionary based techniques. In fact, an entire field of work called Novelty Search [76] promotes that optimising the fitness function using any optimisation technique can be detrimental and instead searches for *novel* behaviours. Using novelty search subsequently results in higher fitness solutions due to the lack of local optima in which search can become trapped.

2.4.1 NEAT

A seminal neuroevolutionary algorithm used extensively throughout Chapter 5 is NeuroEvolution of Augmenting Topologies (NEAT) [137]. NEAT is an example of a Topology and Weight Evolving Artificial Neural Network (TWEANN), which is a class of neural networks in which both the weights *and* the topology of a neural network are optimised using evolutionary algorithms. Optimising the topology as well as the weights can be useful for a number of reasons. Firstly, one does not have to spend additional time performing an architectural hyperparameter search, which is often the case when using fixed topology neural networks. Secondly, certain TWEANN algorithms, such as NEAT, can result in significant improvements in learning speed compared to fixed topology neural networks. Finally,

many TWEANN algorithms aim to produce minimally sized networks resulting in reduced computational and memory demands.

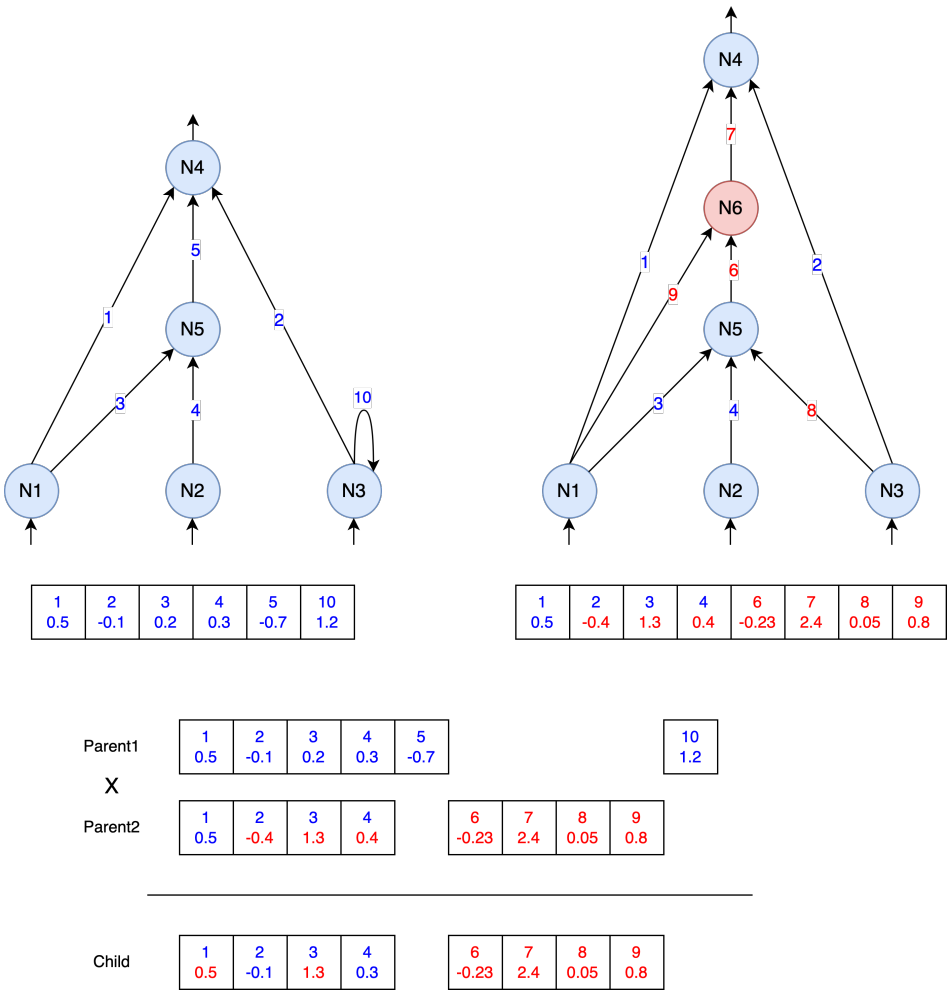


FIGURE 2.7: The crossover process in NEAT. Each parent has a set of genes representing the connections between nodes. Each gene has a historical marking integer which is incremented each time a new gene is mutated into a network, and a weight value. The blue genes represent those that originate from parent 1 and the red genes represent those that are new in parent 2, including weight mutations on the genes that have the same historical marking as those from parent 1. During crossover the child inherits common (matching) genes randomly and the disjoint and excess genes from the most fit parent (here parent 2 has the highest fitness).

A large problem with augmenting topology neural networks is that when a new structural item is added, for example, a new hidden node or a new connection between nodes, it is highly likely that the new respective weights are unoptimised. It would often take some number of generations for the new weights to optimise such that the network as a whole is competitive with the rest of the population. Due to this, networks with new additional structure are often not preserved in the population for long enough to optimise their weights and achieve a competitive advantage. This can be particularly problematic when new structural mutations coupled with enough time to optimise the surrounding weights are what is *needed* to achieve

higher levels of fitness. NEAT solves this problem through a process called speciation.

The speciation mechanism in NEAT assigns each individual in the population to a group (species) of other individuals whose genomes are similar to its own. The similarity between genomes is calculated according to the number of genes that are common between them, as well as the difference between the weights of the respective genes. Common genes are tracked using 'historical markings' which are unique integers assigned to each new gene that arises through mutations into the population. Once speciated, a number of mechanisms allows newer topological structures to be preserved in the population. These mechanisms include: explicit fitness sharing between members of a species, boosting the fitness of younger species, and ensuring the survival and reproduction of at least one member of the species [3]. Explicit fitness sharing helps to protect genomes with innovative topological structures within the same species by allowing them to borrow the higher fitness of others in its species, thereby buying time to optimise the weights associated with the new structure. It also helps protect genomes different enough that they form a new species because newer species are given a fitness boost and at least one member of each species is allowed to reproduce (for a limited time [3]).

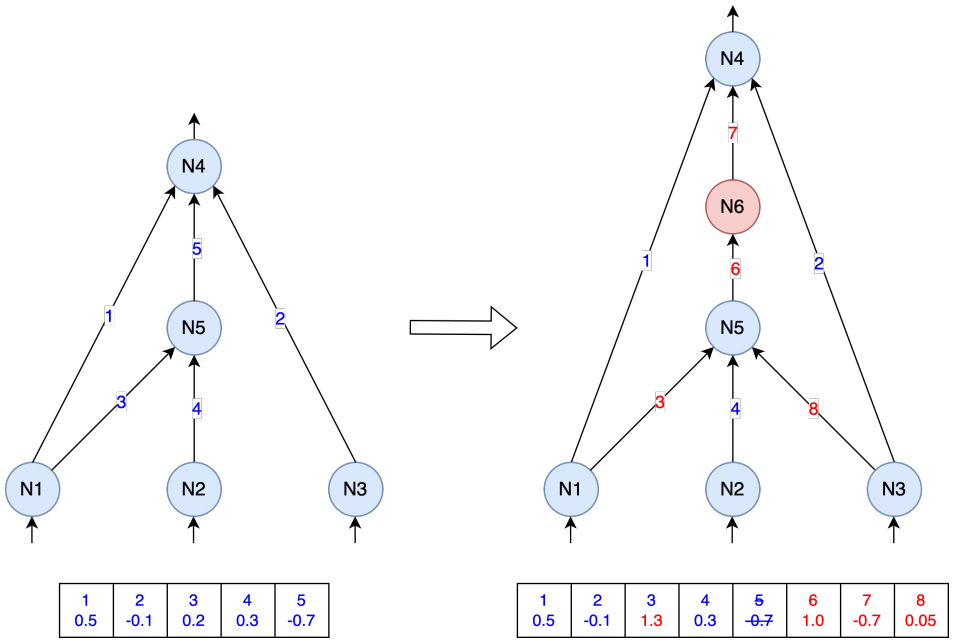


FIGURE 2.8: A representation of the 3 types of mutations that can occur in NEAT. A weight mutation perturbs the weight on connection number 3. A structural *add connection* mutation adds connection number 8. A structural *add node* mutation adds node number 6 and in turn connections 6 and 7.

The historical markings that are used to calculate the similarity between genomes also assists with providing a useful crossover procedure (Figure 2.7). For each gene common to both parents (has the same historical marking), the child inherits randomly from either parent. The child inherits disjoint and excess genes from the fittest of the two parents. This allows genes that have historically played a similar role to

[3] If no members of a species have improved their fitness within a set number of generations (default 30), the species is removed from the population. This prevents species with one poorly performing genome surviving indefinitely.

be placed in the same position in the offspring genome. This crossover procedure prevents the replacement of one functional sub network with a completely different functional sub network; this is a common occurrence in fixed topology network crossover, which can have a dramatic negative effect on fitness.

There are 3 types of mutation that can occur in NEAT networks, shown in Figure 2.8. A weight mutation that randomly mutates the weight of a connection according to some mutation power, and two structural mutations, an add node mutation and an add connection mutation. The add connection mutation adds a connection between two previously unconnected nodes, as is the case with connection number 8 in Figure 2.8. The add node mutation adds a new node into the network (node 6 in Figure 2.8). When a new node is added the previous connection is disabled and two new connections are added. The connection that runs into the new node takes on a weight value of 1 and the connection running out of the new node takes on the weight value of the disabled connection; this reduces the impact on the output due to the new node.

2.5 Generative Models

A generative model is that which learns to model a probability distribution, $P(\boldsymbol{x})$, over training data. One is then able to sample from $P(\boldsymbol{x})$ to generate new instances similar to that of the training data, hence the term 'generative' model. Generative models feature heavily in Chapters 3 and 4, in particular, autoencoders, VAEs, and GANs, as such we describe them in detail here.

2.5.1 Generative Adversarial Networks (GAN)

A GAN [45] employs an innovative method to learn $P(\boldsymbol{x})$, which consists of a zero-sum game between two neural networks, the generator and the discriminator. The generator generates data points in an attempt to fool the discriminator into believing that this fake data is derived from the true training set. The discriminator attempts to discriminate the fake data generated by the generator from the true training data.

This is achieved by setting up a two player minimax game between the discriminator, D, and the generator, G, with value function, $V(D, G)$ (Equation 2.20). The discriminator outputs a single scalar, which represents the probability of the input being real or fake data.

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})}[ln(D(\boldsymbol{x}))] + \mathbb{E}_{\boldsymbol{z} \sim p_z(\boldsymbol{z})}[ln(1 - D(G(\boldsymbol{z})))] \tag{2.20}$$

The aim of the discriminator is to *maximise* equation 2.20, which consists of maximising the log likelihood of the training data, $p_{data}(\boldsymbol{x})$, corresponding to the first term of equation 2.20, and to minimise the probabilities it associates to the fake generated data, corresponding to the second term of equation 2.20. The aim of the generator is to *minimise* equation 2.20, which is the opposite of the aim of the discriminator. Samples from a prior distribution, $p_z(\boldsymbol{z})$, are used as input to the generator. In practice (and in our work), $p_z(\boldsymbol{z})$ is set to be the unit normal distribution. Typically, the dimension of $\boldsymbol{z}$ is much less than the dimension of $\boldsymbol{x}$. Backpropagation is used to train both networks.

Despite being proven that the global optima of the minimax game defined in equation 2.20 is when the probability distribution produced by the generator, p_g, is equal to p_{data} [45], common pathologies of GAN training regimes exist, preventing

convergence. Firstly, if the discriminator is too good with respect to the generator, the generator can struggle to produce convincing data points because gradients that inform it how to improve vanish. This problem can be alleviated by either using a Wasserstein loss [2] or implementing an alternative loss function introduced in the original GAN paper [45]. A second pernicious pathology, known as mode collapse, occurs when the generator produces the same data point, or small set of data points, as oppose to a diverse selection of data points representing p_{data}. This can happen when the discriminator gets stuck in a local optima where it is unable to penalise the generator for adopting this strategy. The Wasserstein loss is similarly useful in preventing mode collapse.

2.5.2 Autoencoders

An autoencoder [121] is a neural network that is optimised to reconstruct its input after being passed through a bottleneck layer. An encoder is used to map the input, $\boldsymbol{x}$, to a latent variable, $\boldsymbol{z} = e_{\theta_e}(\boldsymbol{x})$, via a neural network parameterised by θ_e. A decoder is used to reconstruct $\boldsymbol{x}$ from $\boldsymbol{z}$ resulting in $\hat{\boldsymbol{x}} = d_{\theta_d}(e_{\theta_e}(\boldsymbol{x}))$, via a neural network parameterised by θ_d. An illustration of this process is shown in Figure 2.9. It does this by minimising the mean squared error loss between $\boldsymbol{x}$ and $\hat{\boldsymbol{x}}$ (Equation 2.21) by backpropagating through the entire autoencoder and subsequently optimising the parameters using gradient descent techniques highlighted in Section 2.3.2.

$$l(\boldsymbol{x}; \theta_e, \theta_d) = (\boldsymbol{x} - d_{\theta_d}(e_{\theta_e}(\boldsymbol{x})))^2 \tag{2.21}$$

The dimensionality of $\boldsymbol{z}$ is, more often than not, less than that of $\boldsymbol{x}$ and as such the autoencoder is often used as a dimensionality reduction technique. A number of hidden layers can be used in both the encoder and decoder part of the autoencoder (highlighted in yellow in Figure 2.9) enabling the representation of complex non-linear mappings.

An autoencoder is not technically a generative model because it is not optimised to model $P(\boldsymbol{x})$, however it does have some properties that allow it to behave similar to one. Once trained, one can sample the latent code, z, (in the bounds attributed to the code, for example, if a sigmoid activation function is used in the pre code layer the code is bounded between 0 and 1) and the decoder will often map the codes to data similar to the training data. However, there is no requirement that the entirety of the code be mapped to respective training data values, which means that there are often sampled codes that are mapped to data points far from the training data distribution.

2.5.3 Variational Autoencoders (VAE)

The variational autoencoder [65] is structurally similar to the autoencoder in a number of ways (Figure 2.10). Both the VAE and the autoencoder encode input data, $\boldsymbol{x}$, as a lower dimensional latent code, $\boldsymbol{z}$, and subsequently use a decoder to attempt to reconstruct $\boldsymbol{x}$ as accurately as possible, $\hat{\boldsymbol{x}}$. The encoding process of the VAE differs slightly in that the output is two vectors, $\boldsymbol{\mu}$ and $ln(\boldsymbol{\sigma}^2)$. The latent code, $\boldsymbol{z}$, is subsequently sampled from a multivariate isotropic gaussian, $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2 \boldsymbol{I})$. The decoding process proceeds in the same way as the autoencoder. The forward pass of the VAE is illustrated in Figure 2.10.

The loss function of the VAE (Equation 2.22) has an additional regularisation term compared to that of the autoencoder. Similar to the mean squared error loss of the autoencoder (Equation 2.21), the first term of the VAE loss, $\mathbb{E}_{z \sim p_{\theta_e}(z|\boldsymbol{x}_i)}[ln(p_{\theta_d}(\boldsymbol{x}_i|\boldsymbol{z}))]$,

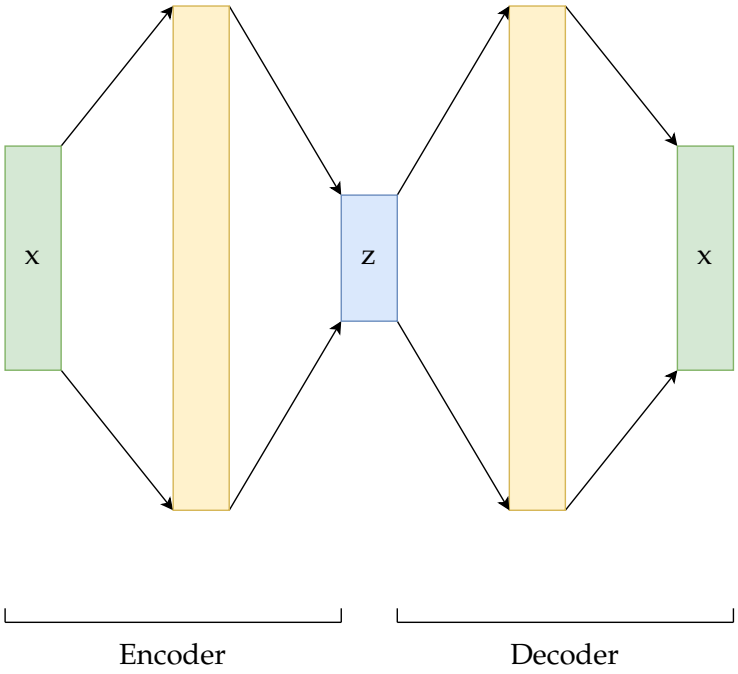


FIGURE 2.9: A representation of an autoencoder. The encoder takes input, $\boldsymbol{x}$, (in green) and passes it through hidden layers (yellow) to a lower dimensional latent code, $\boldsymbol{z}$. The decoder attempts to reconstruct $\boldsymbol{x}$ from $\boldsymbol{z}$.

penalises the VAE for a poor reconstruction. This term represents a probability distribution defined by the decoder for training data point i, which in the case of a continuous valued output would likely be the gaussian, $\mathcal{N}(\hat{\boldsymbol{x}}_i, \boldsymbol{I})$. By optimising the probability distribution $p_{\theta_d}(\boldsymbol{x}|\boldsymbol{z})$ to increase the likelihood of the training data, the reconstruction, $\hat{\boldsymbol{x}}$, improves in turn.

$$l_i(\boldsymbol{x}_i; \theta_e, \theta_d) = -\mathbb{E}_{\boldsymbol{z}\sim p_{\theta_e}(\boldsymbol{z}|\boldsymbol{x}_i)}[ln(p_{\theta_d}(\boldsymbol{x}_i|\boldsymbol{z}))] + \mathcal{D}_{KL}(p_{\theta_e}(\boldsymbol{z}|\boldsymbol{x}_i)||p(\boldsymbol{z})) \tag{2.22}$$

The second term in Equation 2.22 represents the Kullback-Leibler (KL) divergence between the probability distribution over $\boldsymbol{z}$ given the training data (the encoder) and a prior distribution over $\boldsymbol{z}$, $p(\boldsymbol{z})$. This prior is defined to be the unit normal distribution with a mean of 0 and standard deviation of 1. One way to conceptualise the KL divergence is as a distance between two probability distributions. Therefore, the minimisation of the KL divergence results in the distribution over $\boldsymbol{z}$ conditioned on the training data, $p_{\theta_e}(\boldsymbol{z}|\boldsymbol{x}_i)$, approaching the unit normal. This has the effect of 'spreading out' the training data over the latent code, z, resulting in a code that, when enumerated over, results in all outputs resembling the training data; this is not a feature of a normal autoencoder. It discourages sections of the latent code that do not code for data points that have a high likelihood of coming from the training data distribution. The generative procedure of a VAE consists of sampling from a unit normal distribution in its latent space and observing the output.

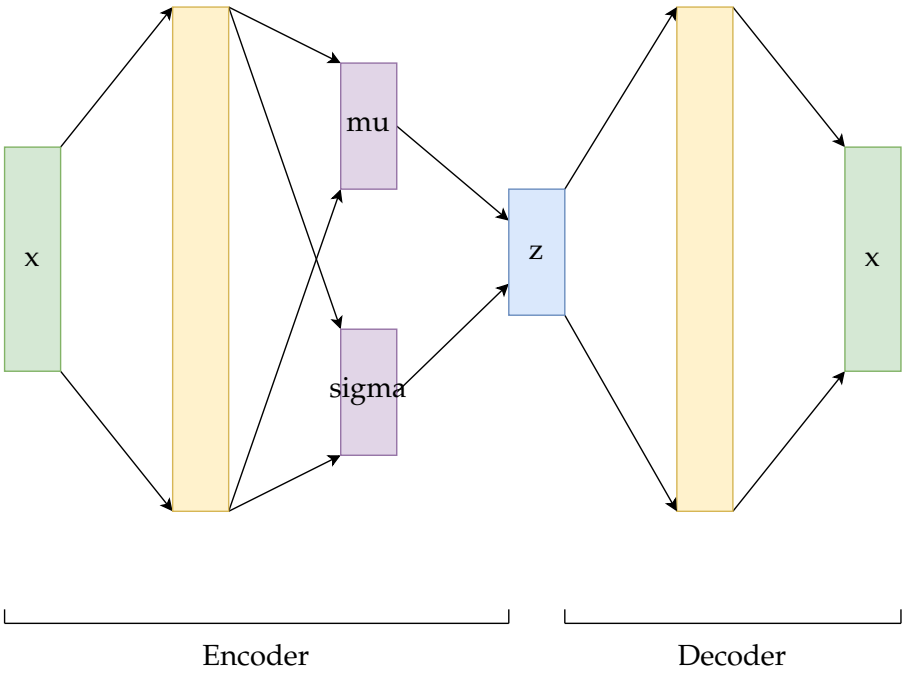


FIGURE 2.10: A representation of a variational autoencoder. The encoder maps the input, $\boldsymbol{x}$, to two vectors $\boldsymbol{\mu}$ and ln $\boldsymbol{\sigma}$. The latent variable $\boldsymbol{z}$ is sampled from the multivariate isotropic gaussian $\mathcal{N}(\boldsymbol{\mu}, \sigma_2 \boldsymbol{I})$. The decoder attempts to reconstruct input $\boldsymbol{x}$ from sampled latent code $\boldsymbol{z}$.

Chapter 3

Indirect Encodings for Continuous Optimisation

Indirect encodings (IE) map the genotype space to the phenotype space, and have shown promising results in certain optimisation tasks. They have resulted in greatly improved search speed, and in some instances, the accuracy of the solutions found. Recently, data-driven encodings (DDE), which *learn* an indirect encoding by modelling previously found solutions to a problem using a generative model, have been proposed [40]. In [40], it is demonstrated that previously located solutions to a set of source problems can be used to train a VAE to produce an IE. The VAE derived IE can subsequently be used to solve novel, but similar, optimisation problems to the source problems in a much shorter amount of time than a direct encoding.

Despite the impressive results of [40], a VAE is the only generative model considered to construct the IE. Autoencoders (AE) and generative adversarial networks (GAN) are also promising algorithms that can be used to model distributions of solutions and act as an IE in subsequent evolutionary search. Other works do use autoencoders [96, 106] and GANs [18, 57, 58] to learn IEs, however none of them individually perform comparative analyses between the three different generative model types. It continues to be unclear as to which generative model produces the most successful IE for a particular task; a successful IE would be that which can locate an accurate solution faster than a direct encoding on a variety of unseen target problems.

To this end, in this chapter we perform a comparative empirical analysis of the ability of autoencoders, VAEs, and GANs to learn IEs and use them to subsequently find solutions on similar, but unseen target optimisation problems. It is this ability to leverage knowledge about the search space of the source problems on unseen target problems that we believe to be the most valuable application of DDEs. In order to train the generative model, previous solutions to optimisation problems must be located using a direct encoding. Despite significant search speed gains demonstrated in work such as [96], to then subsequently use this IE to solve the same problem faster seems slightly redundant after it has already been solved many times previously.

As a result, a significant focus of this chapter is concerned with the comparative interpolative and extrapolative abilities of the learnt IE types on target problems different to the source problems whose solutions are used to train the IE. This has not been a significant aspect of previous works [40, 57, 58], in that it has not been clear how different the target functions have been to the source functions. Of course, we do not expect our IEs to incur advantages when the source and target problems are completely different, in this case our methodology would most likely produce no advantages. Thus, we restrict our investigations to source and target functions that are part of the same *family* of functions, for which we assume only a small part of the

full search space to contain optima. Despite this sounding artificially constrained, we believe this is a good proxy for real world problems where slight changes in the environment result in a similar but different optimisation problem of the same 'family'.

We compare the performance of autoencoders, VAEs, and GANs of code sizes 1 and 2 at producing IEs for the optimisation of simple bivariate quadratic functions. We consider 2 different families of bivariate quadratics, one in which the maxima of the source and target functions are linearly related and another in which the maxima are not linearly related. The 2 dimensional nature of these functions means that we can clearly visualise the manifolds imposed on the search space by the IE and use this to inform us of the reasons for the behaviours we observe during evolutionary search. It also means that we can run optimisation rapidly, comparing many different techniques and target functions without having to rely on very powerful computers. We additionally compare against two baselines that use a direct encoding.

Contributions

1. We perform a comparative empirical analysis of autoencoder, VAE, and GAN derived IEs on two families of bivariate quadratic functions.

2. We show that for certain target problems all learnt IEs at some point outperform two baselines in terms of search speed *and* accuracy of the solutions found.

3. We show that the performance of the IEs with respect to the direct encoding (DE) baselines is highly dependant on the particular target function being optimised.

4. We show that the training procedures for AEs and VAEs with a code size of 1 are susceptible to local minima and overfitting such that their respective IEs extrapolate poorly. This is not the case for the GAN.

5. We highlight a speed accuracy trade off for the code size 1 GAN derived IEs and suggest an algorithm that retains the speed advantages of the IE and the accuracy advantages of the DE. This involves running the IE until a fitness plateau is reached and then switching search back to using a direct encoding.

6. We show that under certain conditions mode collapse occurs during the GAN training procedure and illustrate that this results in very poor performance of the respective IE.

7. We show significant manifold folding in the case of the VAE and GAN derived IEs of code size 2 and hypothesise that this results in lower than optimal accuracy by inducing local maxima in the search space. We suggest a number of regularisation techniques that may alleviate this effect.

3.1 Mathematical Framework

This section introduces the mathematical framework for which all indirect encoding instances can be embedded into and highlights how the techniques used in this chapter fit within it. Most of this framework and notation have been conceptualised

before [52, 68, 119], therefore the notation here will be kept as similar to previous work as possible.

Let us begin by recalling an instance of mathematical optimisation. Given a solution space, $\mathcal{S}$, and a function, $f : \mathcal{S} \to \mathbb{R}$, that maps instances from the solution space to the real numbers, the aim of an optimisation procedure is to find a solution:

$$\mathbf{x}^* \in \mathcal{S} \quad s.t.$$

$$f(\mathbf{x}^*) \leq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{S} \quad \text{'minimisation'}$$
$$f(\mathbf{x}^*) \geq f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathcal{S} \quad \text{'maximisation'}$$

Adopting the nomenclature from the field of evolutionary algorithms, the above optimisation problem is formulated in the following way. Given a genotype space, Γ, and a fitness function, $f : \Gamma \to \mathbb{R}$, an EA aims to find a:

$$\mathbf{g}^* \in \Gamma \quad s.t.$$
$$f(\mathbf{g}^*) \geq f(\mathbf{g}) \quad \forall \mathbf{g} \in \Gamma$$

It does this by applying biologically inspired operators such as: selection, crossover and mutation to the elements of a subset of Γ, known as a population $P \subseteq \Gamma$, resulting in a new $P' \subseteq \Gamma$.

In order to integrate IEs into the above formulation one needs to introduce an intermediate space, Φ the phenotype space s.t.

$$\delta : \Gamma \to \Phi$$
$$f : \Phi \to \mathbb{R}$$

We adopt the notation used in [52] and use δ to refer to the maturation function or the indirect encoding. Given this new space we can reformulate the above maximisation problem. Given a genotype space Γ, a phenotype space Φ, an indirect encoding $\delta : \Gamma \to \Phi$, and a fitness function $f : \Phi \to \mathbb{R}$, an EA aims to find a:

$$\mathbf{g}^* \in \Gamma \quad s.t.$$
$$f(\delta(\mathbf{g}^*; \boldsymbol{\theta})) \geq f(\delta(\mathbf{g}; \boldsymbol{\theta})) \quad \forall \mathbf{g} \in \Gamma$$

For reasons which will become clear, we have conditioned the IE δ on an additional vector of parameters $\boldsymbol{\theta}$.

The elements of Γ are vectors of objects. These objects can take a number of different forms: real numbers, integers, binary values, characters etc. The EA applies the evolutionary operators to the elements of Γ only, it does not apply operators to Φ. This may sound odd if one is not explicitly using an IE in their EA, however we would argue that in this case the identity encoding, δ_I, was being used resulting in a one-to-one mapping from Γ to Φ, resulting in $\Gamma = \Phi$.

The form of Φ is dictated by the domain in which the EA is being applied. For example, if the aim is to evolve a neural network controller for an agent, then Φ is forced to be a vector of real numbers, which act as weights for said neural network. For this reason, an EA practitioner does not have much choice over the form of Φ, however they do have freedom over the form of Γ and δ as long as the range of δ is Φ.

In this chapter we are performing continuous mathematical optimisation such that $\Phi = \mathbb{R}$ and we set $\Gamma = \mathbb{R}$. However, Γ could also take other forms such as $\mathbb{Z}$, but one would then have to use an appropriate, $\delta : \mathbb{Z} \rightarrow \mathbb{R}$. For the indirect encoding, δ, we use a neural network, hence $\boldsymbol{\theta}$ would equate to the weights of the neural network. These weights are optimised before the evolutionary search begins and are static throughout search. $\boldsymbol{\theta}$ is not altered by the evolutionary operators, only the elements of Γ are.

In Section 3.2, we are concerned with the optimisation of *families* of functions. We can view a *family* of functions as the set of functions, $f(\delta(\mathbf{g};\boldsymbol{\theta});\boldsymbol{\tau})$, where $\boldsymbol{\tau}$ is a vector of function parameters. For example, a quadratic function of the form $f(x) = ax^2 + bx + c$ can be written in a parameterised form, $f(x;a,b,c) = ax^2 + bx + c$, where each setting of the values a, b, and c is a different function, but still of the same quadratic family. In the quadratic example, it is this vector of parameters a, b, c that is represented by $\boldsymbol{\tau}$.

3.2 Methodology

We compare the performance of three generative models, an autoencoder, a VAE, and a GAN at producing IEs for solving mathematical optimisation problems. We do this by training the models on a set of solutions to a subset of problem instances, the source problems, $p_s \subset \mathcal{P}_f$, of a *family* of functions, $\mathcal{P}_f$. A solution here is that which meets some threshold fitness value. The source solutions are located using a standard genetic algorithm with a direct encoding. A gaussian mutation with a mutation rate of 0.1 and a mutation power of 0.1, truncation selection with a selection percentage of 10%, and an initial population distributed according to a uniform distribution bounded between -1 and 1 was used. A population size of 100 was run for 100 or 250 generations for the linear and non-linear experiments, respectively. The fitness function for all the experiments is simply the output value of the optimisation problem, $f(\mathbf{x})$, with the aim being to maximise this function.

The architectures of the generative models vary slightly depending on the experiment, however, if a hidden layer was used, it had 64 hidden nodes with a ReLU activation function. Experiments are performed using both a code size of 1 and 2. Each model is trained for 20 000 or 30 000 epochs for the linear and non-linear experiments respectively. An ADAM optimiser is used with a learning rate of 1^{-3} and 5^{-4} for the AE and VAE, respectively. For the GAN, RMSprop with respective learning rates of 2^{-4} and 5^{-4} for the generator and discriminator was used. RMSprop was shown to attain better performance compared to ADAM for the GAN in initial experiments. For each model type, 5 models were trained for each experiment.

We then test the performance of the trained IEs on a set of target problems, $p_t \subset \mathcal{P}_f$, from the same family of functions as p_s, however we adhere to the restriction: $p_s \cap p_t = \emptyset$. By doing this, we demonstrate the ability of the IE to locate solutions on functions whose solutions it has not been trained upon.

To test the performance of the trained IEs on the target problem instances, p_t, we ran a standard genetic algorithm in the latent space of the decoder, in the case of the AE and the VAE, or the generator, in the case of the GAN. Five evolutionary runs were performed for each of the 5 trained models for each model type. Truncation selection with a selection percentage of 10% was used on a population of size 100 for 100 or 200 generations for the linear and non-linear experiments respectively. For the VAE and GAN derived IEs gaussian mutation with a mutation rate of 0.5 and a mutation power of 0.1 was used. For the AE derived IE the same mutation

rate was used, however the mutation power was reduced to 0.01666 ($\frac{0.1}{6}$). A unit gaussian distribution was used to initialise the first population for the VAE and GAN derived IEs, whereas a uniform distribution with a lower and upper bound of 0 and 1, respectively, was used for the AE derived IE.

The modifications to mutation rate and initial distribution for the AE derived IEs were due to the fact that the code processed by an AE is different to that of a VAE and GAN. The encoders of the AEs used in these experiments end with a sigmoid layer, meaning that all codes are bounded between 0 and 1. This is in contrast to a VAE and GAN where the code is distributed according to a unit gaussian distribution. It is therefore unwise to initialise a population using a unit gaussian distribution for an AE derived decoder because many of the genomes would fall outside the 0 to 1 bound. Given that the AE has not been trained to process codes outside of this range, the output of the decoder is unlikely to fall within the desired modelled distribution. We also chose to divide the size of the mutation rate by 6 because the range of a unit gaussian, [-3, 3] (99.7% of the sampled points) is 6 times larger than the range of the [0, 1] bounded uniform distribution. It is therefore likely that the same sized change in code value for an AE derived IE will have a much larger change in the output space compared to the VAE and GAN derived IEs. We scaled the mutation power in this way to ensure a fairer comparison between all three types of IE.

For baselines, we used a DE and a DE with an informed start (DEIS). The DEIS is an alternative way to bootstrap the evolutionary process of the direct encoding using information from p_s. An alternative starting point (in our case this takes the form of an alternative mean value of the gaussian distribution used to initialise the genomes of the first population) can be computed by performing an evolutionary search that attempts to maximise the average fitness of all the source problems, p_s, together. This results in a solution that is located at a point with the minimal summed distance to all the maxima in p_s. It is this solution that is used as the mean for the initial starting gaussian distribution for the DEIS baseline.

Both the DE and DEIS baselines used a standard genetic algorithm with gaussian mutation and truncation selection. The initial population for the DE baseline was generated from a unit gaussian distribution. A mutation rate of 0.5 and a mutation power of 0.1 was used. A selection percentage of 10% was used for truncation selection and a population of 100 was ran for 100 or 200 generations for the linear and non-linear experiments respectively. For simplicity, no crossover was performed on any evolutionary runs. This process was repeated 5 times on the target problems, p_t, and the best fitness so far in each of the generations was recorded.

The training of the IEs and the initialisation of the DEIS have additional preparatory overheads compared to evolution using a DE only. To compare techniques according to the total number of FLOPS, including pre-training, would be particularly meticulous and, more importantly, implementation dependent. We have therefore decided to evaluate with respect to the number of generations inline with evaluation methods used in [40] (number of generations) and [34] (number of gradient steps). However, future work could perform timed tests for the entire process and use this as an additional metric of evaluation.

3.3 Experiments

The following experiments illustrate the performance of an indirect encoding derived from an autoencoder, a VAE, and a GAN at optimising 2 families of bivariate quadratic functions.

The general form of a bivariate quadratic function is defined as

$$f(x,y) = ax^2 + by^2 + cxy + dx + ey + f \tag{3.1}$$

where, a, b, c, d, e, f are constants, the modification of which result in a different quadratic function.

Here, we consider the following family of bivariate quadratic functions:

$$f(x,y) = -(x-\alpha)^2 - (y-\beta)^2 \tag{3.2}$$

$$\Rightarrow f(x,y) = -x^2 - y^2 + 2\alpha x + 2\beta y - \alpha^2 - \beta^2 \tag{3.3}$$

where α and β are constants. This family can be encapsulated by the general form (eq. 3.1) where $a = -1, b = -1, c = 0, d = 2\alpha, e = 2\beta, f = -\alpha^2 - \beta^2$. The family of functions described by equation 3.2 have a unique maxima at (α, β).

3.3.1 Bivariate quadratic linear

The first set of experiments aim to optimise a specific subset of functions, $\mathcal{P}_{lin}$, of the form of equation 3.2: those where α and β are linearly related, i.e. $\alpha = k_1\beta + k_2$. We set $k_1 = 1$ and $k_2 = 0$, resulting in $\alpha = \beta$. We consider the case where $k_1 = 1$ and $k_2 = 0$, however, we expect that very similar behaviour would be observed for any values of k_1 and k_2. It is also worth noting that training a generative model on solutions of the family where $\alpha = \beta$ ($k_1 = 1$ and $k_2 = 0$) will not work as an effective IE on the family where $\alpha \neq \beta$ even if α and β are still linearly correlated.

In order to train the generative models we generate training data by optimising three source functions, p_s, of the family, $\mathcal{P}_{lin}$, given in equation 3.2 where $\alpha = \beta$ using a direct encoding:

$$(\alpha = 4, \beta = 4), (\alpha = 0, \beta = 0), (\alpha = -4, \beta = -4)$$

10 solutions were found for each of the 3 source problems, resulting in a training data size of 30. For these experiments, 0 hidden layers were used in the encoders, decoders, generators and discriminators of the generative models.

To test the ability of the IEs to generalise and optimise functions for which solutions had not been present in the training data, the following target problem functions, $p_t \subset \mathcal{P}_{lin}$, were used:

$$(\alpha = 2, \beta = 2), (\alpha = -2, \beta = -2)$$

Code size 1 IEs

Figure 3.1 shows the comparative performance of the two baselines and both the AE and VAE derived IEs of code size 1 on the $(\alpha = 2, \beta = 2)$ target problem. It shows the significant search speed advantage attributed to using the IEs compared to the two baselines, despite the IEs never having had access to information regarding the particular target function being optimised. The IEs were able to interpolate between the source training functions in order to bootstrap the evolutionary process on a new target function. This resulted in both IEs finding a well-performing solution in 2 generations compared to 10 generations in the case of the baselines. The difference in performance between the AE and VAE derived IEs of code size 1 is negligible. In this experiment the GAN derived IE performed much worse than the baselines and as a result its respective comparison plot has been moved to the appendix (Figure

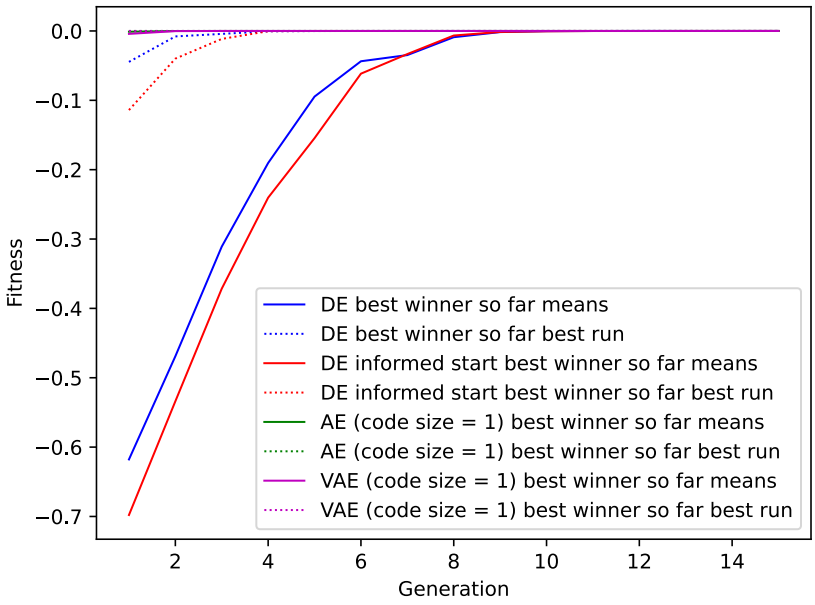


FIGURE 3.1: 5 evolutionary runs for a DE, a DEIS, an AE derived IE, and a VAE derived IE with code sizes of 1 on the target problem, $(\alpha = 2, \beta = 2)$. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

A.1) in order to more clearly visualise the advantages attributed to the AE and VAE. The plot for the other target problem is very similar and therefore not shown.

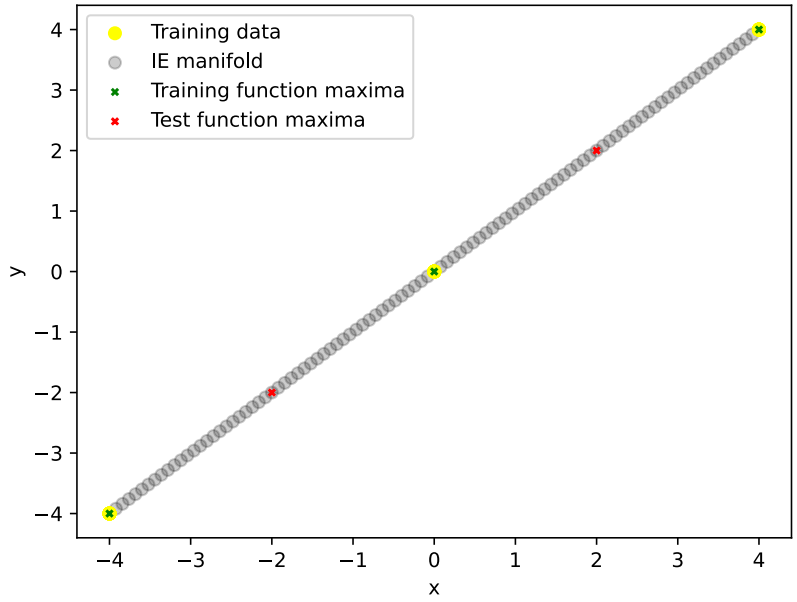


FIGURE 3.2: Enumerated manifold of the best performing AE derived IE of code size 1. Enumeration occurs over the [0, 1] latent space at increments of 0.01. Maxima for the source and target problems are shown as green and red crosses respectively. Training data for the generative models is shown in yellow.

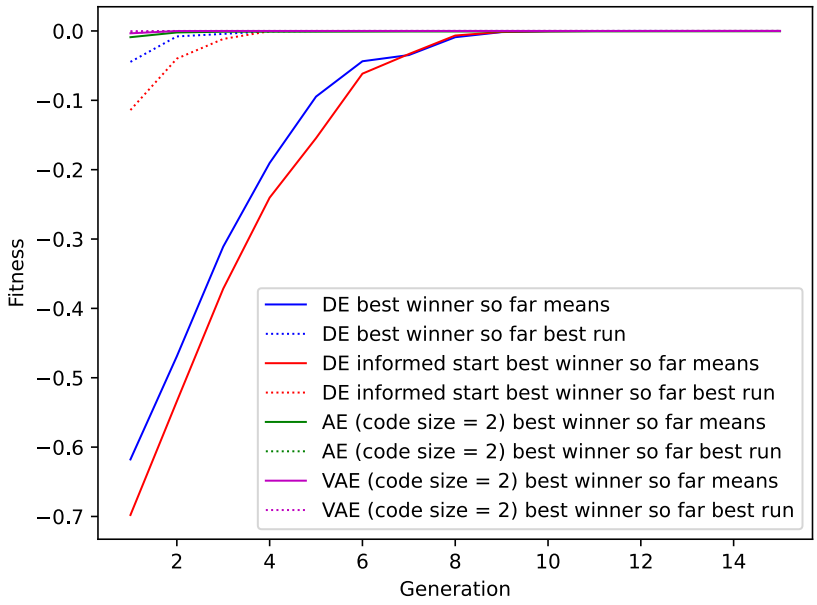


FIGURE 3.3: 5 evolutionary runs for a DE, a DEIS, an AE derived IE, and a VAE derived IE with code sizes of 2 on the target problem, $(\alpha = 2, \beta = 2)$. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

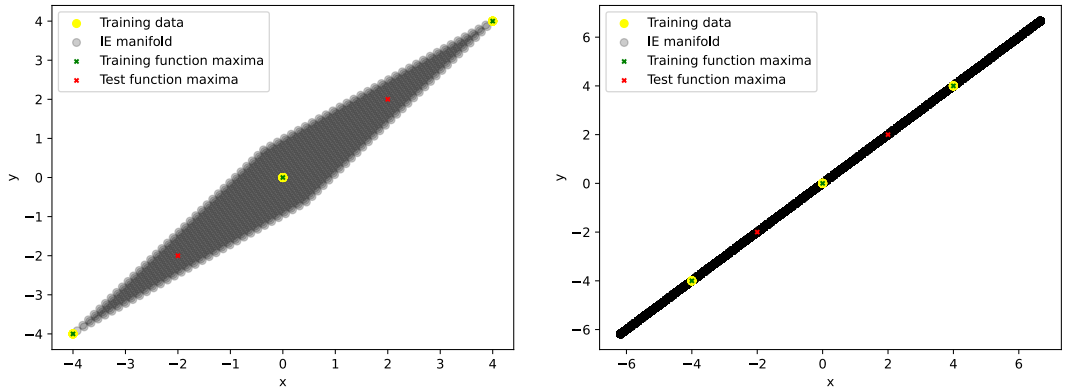
Figure 3.2 provides insight into the performance improvements offered by the AE derived IE by illustrating the search space manifold of the best performing AE derived IE of code size 1. To generate the image we enumerated across the latent space of the decoder (between 0 and 1) in increments of 0.01. Figure 3.2 shows how the search space is greatly reduced (constrained along a 1 dimensional manifold) but it does intersect the maxima of the target problems (red crosses). The AE is trained to encapsulate the source function maxima (green crosses), however it in turn encapsulates the target function maximas also. The manifold of the VAE derived IE is very similar to that of the AE but is still illustrated in the appendix (Figure A.2).

Figure A.3 illustrates the manifold enumerated by the code size 1 GAN derived IE and provides clear evidence that its very poor evolutionary performance can be attributed to mode collapse occurring at one of the source targets maxima. This mode collapse results in a manifold that does not intersect with either of the maxima of the target functions, thereby rendering it impossible to locate an accurate solution. The inability to locate an accurate solution results in the fitness plateau attributed to the GAN derived IE in Figure A.1.

Code size 2 IEs

Figure 3.3 shows the comparative performance of the two baselines and both the AE and VAE derived IEs of code size 2 on the $(\alpha = 2, \beta = 2)$ target problem. Similar to the code size 1 experiments, the AE and VAE significantly outperform the baselines and the GAN performs considerably worse (Figure A.4). Illustrating the respective AE and VAE manifolds in Figure 3.4 again shows how both the training and test function maxima are encapsulated by the manifolds. It is interesting to observe the

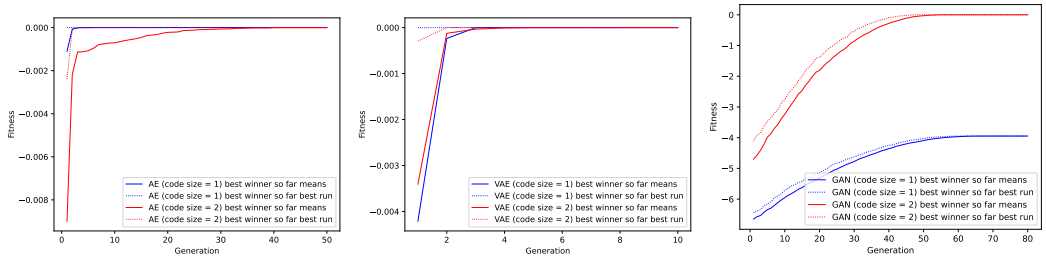
larger spread of the AE derived manifold compared to that of the VAE derived manifold. We hypothesise that this is due to the regularisation effect of the training procedure of the VAE, which results in a compression of the two dimensional manifold around the training points. This compression subsequently results in slightly faster search for the code size 2 VAE derived IE compared to the AE derived IE (Figure 3.3) *and* slower search for the code size 2 AE derived IE compared to its code size 1 counterpart (Figure 3.5a). This is because a larger proportion of the AE manifold contains poor solutions, which takes time to be traversed in order to locate a well-performing solution. Due to the compression of the code size 2 VAE manifold we do not see as large of a fitness differential between the code sizes for the VAE IEs (Figure 3.5b) as we do for the equivalent AE comparison.



(A) AE. Enumeration occurs over the [0, 1] latent space at increments of 0.03.

(B) VAE. Enumeration occurs over the [-3, 3] latent space at increments of 0.05.

FIGURE 3.4: Enumerated manifold of the best performing AE and VAE derived IEs of code size 2. Maxima for the source and target problems are shown as green and red crosses respectively. Training data for the generative models is shown in yellow.



(A) AE

(B) VAE

(C) GAN

FIGURE 3.5: Code size 1 and 2 comparisons for each IE type (5 runs each) on the $(\alpha = 2, \beta = 2)$ target problem.

The poor performance of the code size 2 GAN derived IE shown in Figure A.4 is again attributed to mode collapse at one subset of training points (Figure A.5). However, in contrast to the GAN derived IE with code size 1, the code size 2 counterpart did locate an optimal solution eventually (Figure 3.5c). This is because the dimensionality of the manifold is the same as that of the search space, therefore the solution *is* located within the manifold of the IE.

3.3.2 Bivariate quadratic non-linear

The second set of experiments aims to optimise another subset of functions, $\mathcal{P}_{nonlin}$, of the form of equation 3.2: those where α and β are quadratically related, i.e. $\beta = k_1\alpha^2 + k_2\alpha + k_3$. We set $k_1 = 1$, $k_2 = 0$ and $k_3 = 0$, resulting in $\beta = \alpha^2$.

In order to train the generative models we generate training data by optimising nine source functions, p_s, of the family, $\mathcal{P}_{nonlin}$, given in equation 3.2 where $\beta = \alpha^2$ using a direct encoding:

$$(\alpha = -4, \beta = 16), (\alpha = -3, \beta = 9), (\alpha = -2, \beta = 4),$$
$$(\alpha = -1, \beta = 1), (\alpha = 0, \beta = 0), (\alpha = 1, \beta = 1),$$
$$(\alpha = 2, \beta = 4), (\alpha = 3, \beta = 9), (\alpha = 4, \beta = 16)$$

10 solutions were found for each of the 9 source problems, resulting in a training data size of 90. For each of the three generative models considered, a code size of both 1 and 2 were used. For these experiments, 1 hidden layer with 64 hidden neurons was used in the encoders, decoders, generators and discriminators of the generative models. A hidden layer was essential to capture the non-linear quadratic relationship between the maxima of the source problems.

To test the ability of the IEs to generalise and optimise functions for which solutions had not been present in the training data, the following target problem functions, $p_t \subset \mathcal{P}_{nonlin}$, were used:

$$(\alpha = 0.5, \beta = 0.25), (\alpha = -3.5, \beta = 12.25), (\alpha = 5, \beta = 25)$$

The inclusion of the $(\alpha = 5, \beta = 25)$ target problem exhibits the extrapolative capabilities of the IEs.

Code size 1 IEs

We first compare the performance of the three types of IE with a code size of 1 with the baselines for each of the 3 target problems, p_t. The performance of each technique varies considerably depending on the target problem in question.

For the first target problem, $(\alpha = 0.5, \beta = 0.25)$, the DE performed the best of all the techniques, as shown in Figure 3.6. This is likely because the initial population of the DE was initialised according to a unit gaussian into which the first target problem firmly falls. Therefore, a number of solutions in the first generation will already have been significantly close to this maxima. In contrast, the DE with an informed start performs significantly worse than all the other techniques (Figure A.6), locating a solution after 60 generations. This is due to the fact that the initial population for this baseline is distributed according to a gaussian with $\mu = (0.67, 12.50)$ and a standard deviation of 1.0. It therefore requires many generations for the population to move towards the $(0.5, 0.25)$ maxima.

All 3 IEs perform well and locate a good solution within the first 4 generations, in the case of the AE and GAN, or 15 generations in the case of the VAE (Figure 3.6). Despite initially finding a well-performing solution, the fitnesses of the solutions found by the IEs was not as high as those found by the DE, this is evidenced by the plateauing of their fitnesses between the -0.05 and 0.0 fitness range. Enumerating over the manifolds of the 3 IEs (Figure 3.7) illustrates why this is the case. Each 1-dimensional manifold constructed does not *directly* intersect with the target maxima $(0.25, 0.5)$, although they do get close. Search using these manifolds can

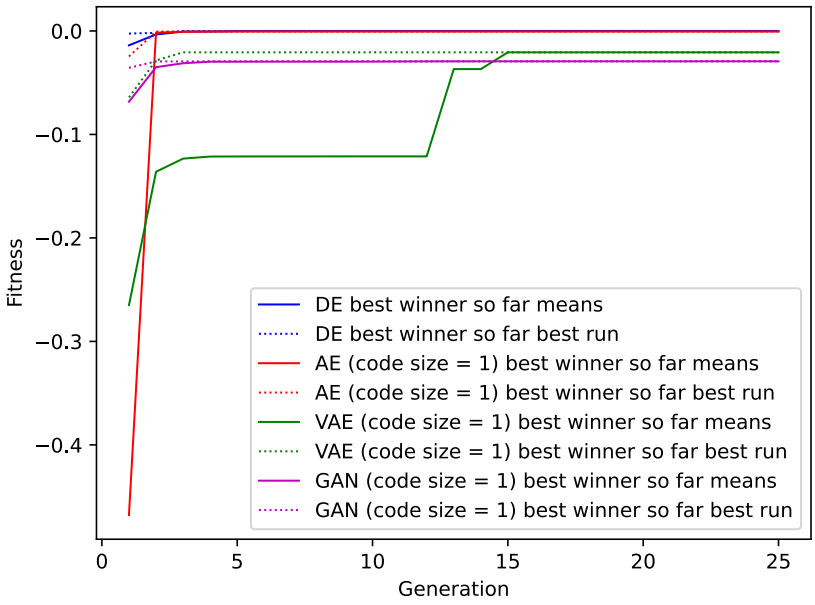


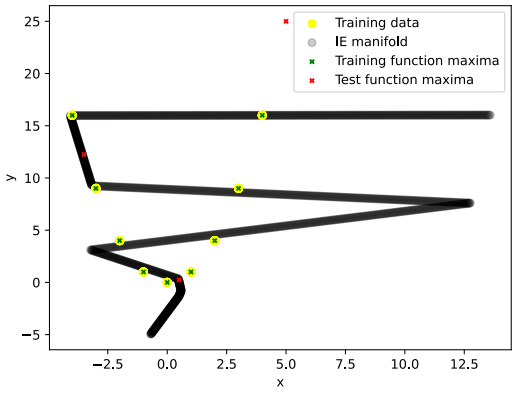
FIGURE 3.6: 5 evolutionary runs for a DE, and all 3 IEs with a code size of 1 on the target problem, $(\alpha = 0.5, \beta = 0.25)$. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

therefore never find a solution closer than the manifold permits, thereby explaining the plateauing of fitnesses of the IEs.

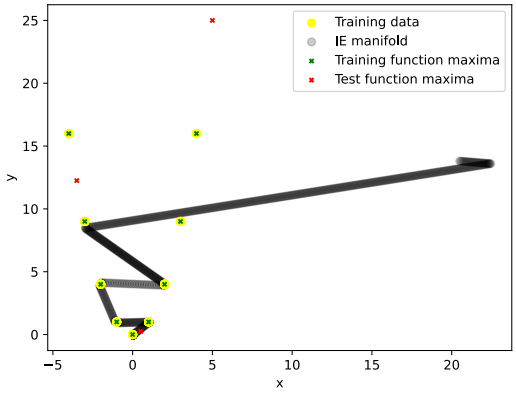
Another interesting difference between the manifolds created by the AEs and VAEs compared to those created by the GANs is that they are locally optimal and do not accurately reflect the quadratic nature of the underlying training distribution. Figure 3.7a and 3.7b show an attempt to fit a 1-D manifold to the training data but the training procedure has become trapped in a local minima. However, the manifold created by the GAN (Figure 3.7c) is a good approximation of a quadratic function; the approximation very nearly intersects all the training points. This behaviour is seen repeatedly throughout the code size 1 non-linear experiments and has downstream effects on the extrapolation performance of the IEs as will be shown for training problem, $(\alpha = 5, \beta = 25)$.

Target problem $(\alpha = -3.5, \beta = 12.25)$ also tests the interpolative capabilities of the IEs. Figure A.7 shows that the DE performs particularly poorly on this problem, despite performing the best on the previous target problem. It takes nearly 70 generations for the DE to find a well performing solution. This is because (-3.5, 12.25) is significantly further from the origin (the starting location of the DE) than (0.5, 0.25). Although a DE with an informed start performs significantly better than the DE, it is still outperformed by all 3 IEs.

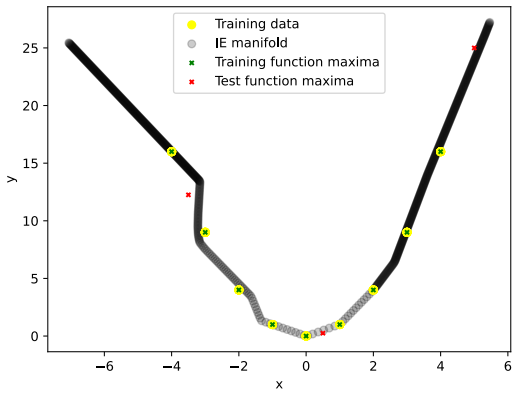
Figure 3.8 compares the performance of the 3 IEs on the $(\alpha = -3.5, \beta = 12.25)$ target problem. Despite the fact that the AE and VAE find a solution on average faster than the GAN, Figure A.10 illustrates that the particular local minima entered into by the AE and VAE training procedure resulted in manifolds that closely bypass the (-3.5, 12.25) target maxima by fortunate chance. If the manifolds were like that of Figure 3.7b where (-3.5, 12.25) is a significant distance from the manifold, performance would have been significant impaired. This is in contrast to the manifold created by the GAN which again accurately models the quadratic relationship in the



(A) Autoencoder, enumeration range = [-0.2, 1.2] in increments of 0.001.



(B) VAE, enumeration range = [-3, 3] in increments of 0.01.



(C) GAN, enumeration range = [-3.5, 3.5] in increments of 0.01.

FIGURE 3.7: Enumerated manifolds over the latent space of all 3 code size 1 IEs on the target problem, $(\alpha = 0.5, \beta = 0.25)$. Maxima for the source and target problems are shown as green and red crosses respectively. Training data for the generative models is shown in yellow.

training data (Figure A.10c).

The target problem, $(\alpha = 5, \beta = 25)$, tests the extrapolative capabilities of the IEs because the point (5, 25) is beyond the distribution of the training data. Figure 3.9 shows that the GAN derived IE found a solution within 10 generations on average, which is the fastest of the all of the techniques considered. The DE performed slower than every one of its counterparts, finding a solution in 150 generations on average. The DEIS benefits from its modified initial distribution and found a solution within 75 generations on average. Despite the AE and VAE derived IEs starting evolution with better performing solutions than the baselines, the fitness quickly plateaus at a point significantly lower than that of an acceptable solution.

This behaviour can be explained by considering the manifolds in Figure 3.7. Due to the misrepresentation of the underlying training distribution by the IEs of the AE and VAE, the resultant manifolds diverge away from the (5, 25) point. This means that the point closest to (5, 25) on these manifolds is still very far from the (5, 25) maxima. This large distance results in a poor fitness that is impossible to improve upon, which subsequently causes the observed fitness plateau. However, the manifold created by the GAN derived IE (Figure 3.7c) captures the quadratic nature of the underlying training data and therefore creates a manifold that closely passes the

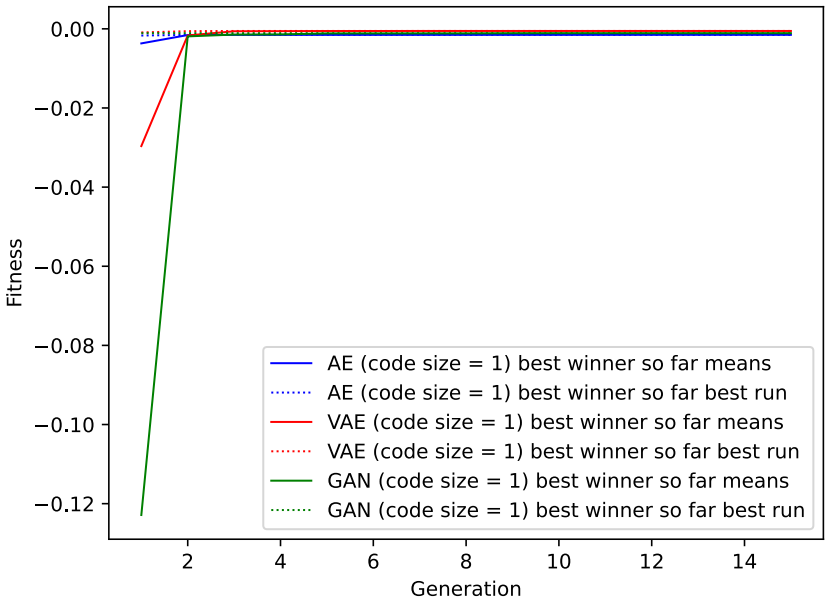


FIGURE 3.8: 5 evolutionary runs for each of the 3 IEs with a code size of 1 on the target problem, $(\alpha = -3.5, \beta = 12.25)$. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

point (5, 25). The GAN derived IE is therefore able to find a solution quickly (its search only occurs on its 1-D manifold rather than the full space) and accurately.

Although the solution is found much quicker using the GAN derived IE, the solution is less accurate than those found by the baselines due to the fact that the manifold does not intersect *exactly* with the point (5, 25). The best solution found by the GAN derived IE is (5.1531, 24.9785), resulting in a fitness of -0.02389. Whereas, the solutions found by the baselines are much closer to (5, 25), resulting in fitness values of -2.6584^{-8} and -2.2737^{-11} for the DE and DEIS respectively. It is therefore clear that there is a speed accuracy trade off when using an IE with a dimensionality less than that of the original search space. It would be the choice of the practitioner to determine how to balance this trade off.

Code size 2 IEs

Figure 3.10 shows that on the target problem, $(\alpha = 0.5, \beta = 0.25)$, the DE still outperforms all the IEs in terms of finding a solution the fastest and the DEIS performs the worst (Figure A.8). The AE derived IE locates a solution faster than the other IEs, despite starting the evolutionary process with the lowest average fitness. Figure 3.11 suggests that the AE derived IE initialises the population with an even spread across the manifold (Figure 3.11a), which contains many poor performing solutions. Whereas, the VAE manifold results in an initial population spread sparsely across areas that do not contain any maxima (3.11b). It is this property of the manifolds that result in the initial population of the AE having a lower average fitness than that of the VAE derived IE.

Another interesting result is that the solutions found by the GAN derived IE plateau below the optimal fitness. This is unexpected because the dimension size of the manifold is the same as that of the search space, therefore the manifold of the IE

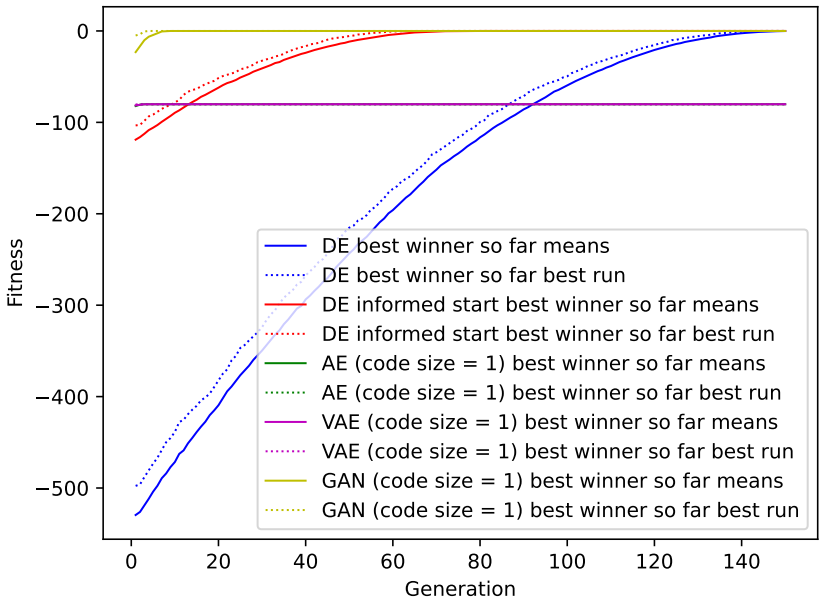


FIGURE 3.9: 5 evolutionary runs for a DE, a DEIS, and each of the 3 IEs with a code size of 1 on the target problem, $(\alpha = 5, \beta = 25)$. The performance of the AE and VAE are so similar that their separation is not clear on this plot. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

does cover the entire search space. Figure 3.12 zooms into the manifold created by the GAN derived IE from Figure 3.11. It shows that folding of the manifold occurs, which could have the effect of creating local maxima in the search space, this may be the cause of the GAN derived IE fitness plateauing.

Figure 3.13 shows that for the target problem, $(\alpha = -3.5, \beta = 12.25)$, again all 3 IEs outperformed both baselines in terms of search speed (Figure A.9 includes the DE performance also). We similarly observe the fitness plateauing with the GAN derived IE but Figure A.11c suggests that this could be because the point (-3.5, 12.25) falls outside of the manifold drawn from a code distributed by a unit gaussian and as such may take a large number of mutations to locate.

Unlike with the code size 1 IEs (Figure 3.9), all 3 code size 2 IEs beat both baselines in term of search speed for the interpolative target problem, $(\alpha = 5, \beta = 25)$, as shown in Figure 3.14. No plateauing is observed for the AE and VAE due to the fact that the dimension size of the IE manifold is the same as that of the search size. Therefore, the solution is located somewhere within the IE manifold, unlike in the code size 1 equivalents. The best solutions found by the AE and VAE are closer to the optima than that found by the DE. We do however see plateauing for the GAN derived IE for which the best solution found was (5.3515, 24.9453), which is slightly different to the (5, 25) optima. We again attribute this to the folding behaviour illustrated in Figure 3.12 creating local maxima in the search space. The dominant superiority of the VAE on this target problem is demonstrated by its ability to find a good solution within 10 generations with a higher accuracy that the other techniques.

Figure 3.15 compares code size performance for all 3 IE types for all 3 non-linear target problems. For the AE, it is clear that code size 1 IEs perform evolutionary

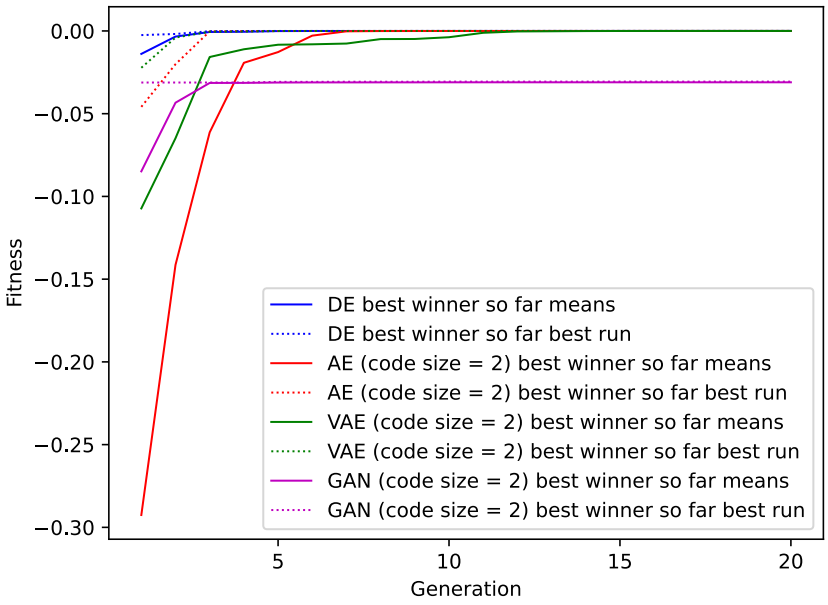
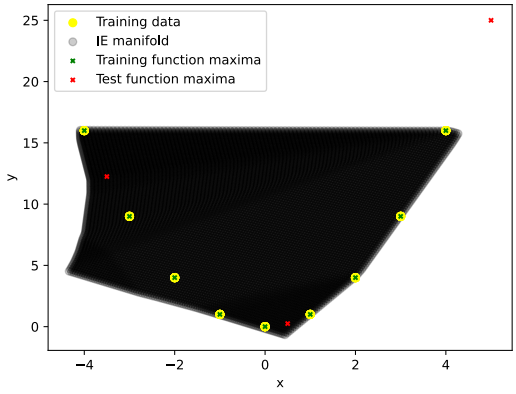
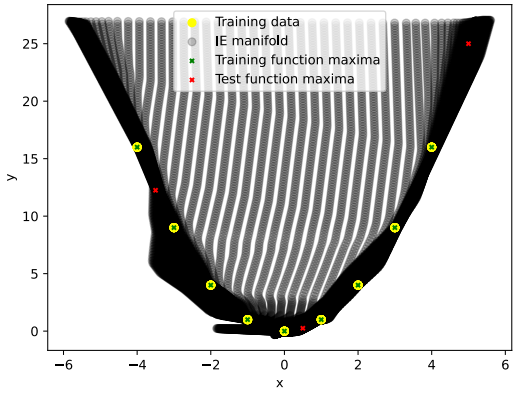


FIGURE 3.10: 5 evolutionary runs for a DE and all 3 IEs with a code size of 2 on the target problem, $(\alpha = 0.5, \beta = 0.25)$. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

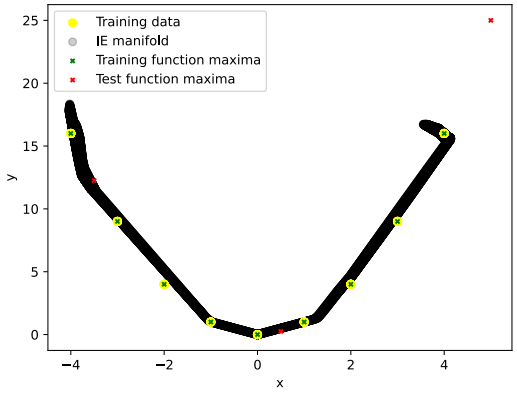
search faster than the code size 2 equivalents, but can plateau if the 1 dimensional manifold does not intersect with the target maxima. However, a more in depth analysis into the results (and a closer look at the graphs) reveals that the final solution found by the code size 2 AE derived IEs have a higher fitness. For the VAE, no clear pattern can be observed across the different target problems. For the GAN, a similar pattern to the AE derived IEs can be observed in that the code size 1 IEs find a solution faster than the code size 2 equivalents. However, in contrast to the AE, the code size 1 final solutions have a higher fitness than the code size 2 counterparts. Furthermore, the code size 1 GAN derived IEs do not suffer from such extreme plateauing because the derived manifold successfully extrapolates to unseen target problems.



(A) Autoencoder, enumeration range = [0.0, 1.0] in increments of 0.01.



(B) VAE, enumeration range = [-3.0, 3.0] in increments of 0.03.



(C) GAN, enumeration range = [-3.0, 3.0] in increments of 0.03.

FIGURE 3.11: Enumerated manifolds over the latent space of all 3 code size 2 IEs on the target problem, ($\alpha = 0.5, \beta = 0.25$). Maxima for the source and target problems are shown as green and red crosses respectively. Training data for the generative models is shown in yellow.

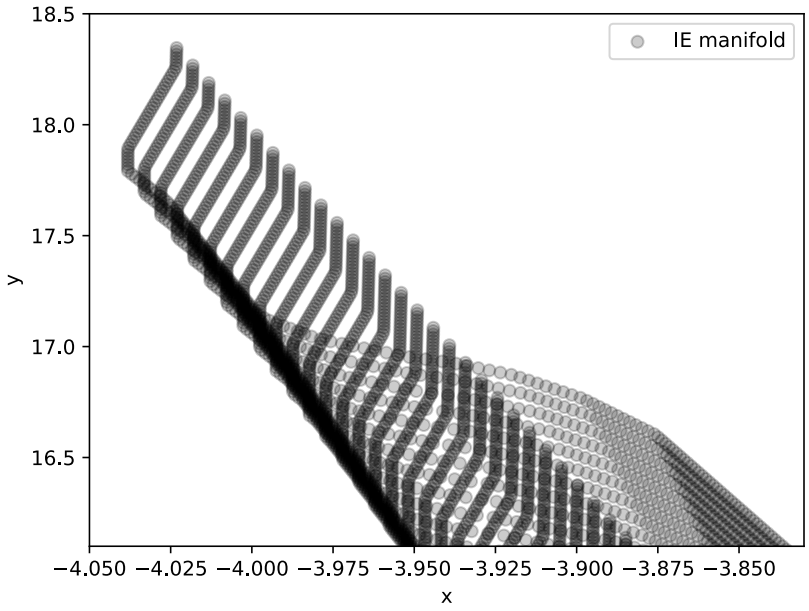


FIGURE 3.12: Zoomed enumerated manifold of the GAN derived IE from Figure 3.11c. It illustrates the folding of the manifold which might lead to local optima in search.

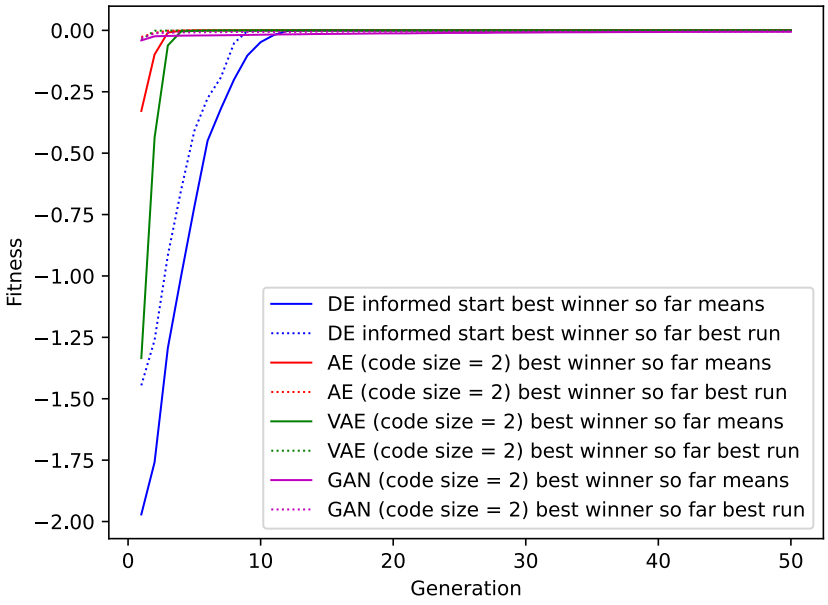


FIGURE 3.13: 5 evolutionary runs for a DEIS and each of the 3 IEs with a code size of 2 on the target problem, ($\alpha = -3.5, \beta = 12.25$).

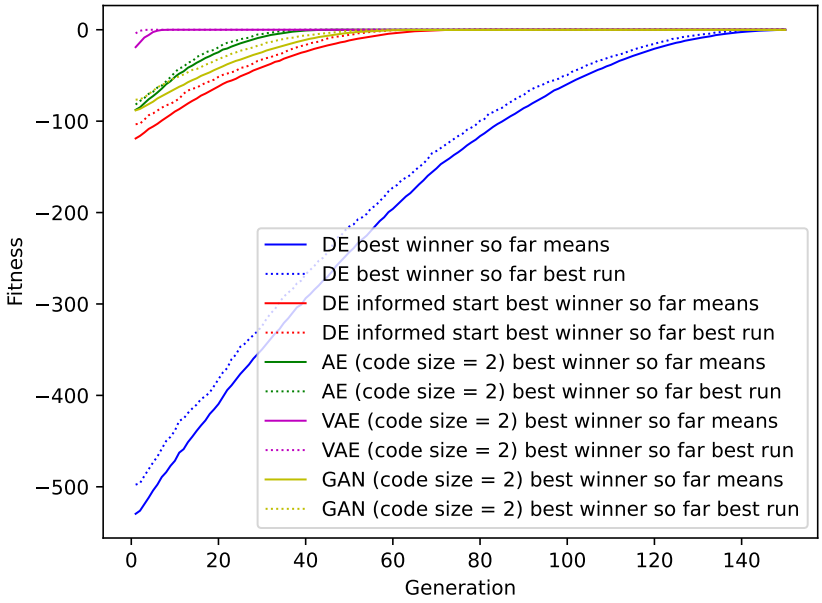
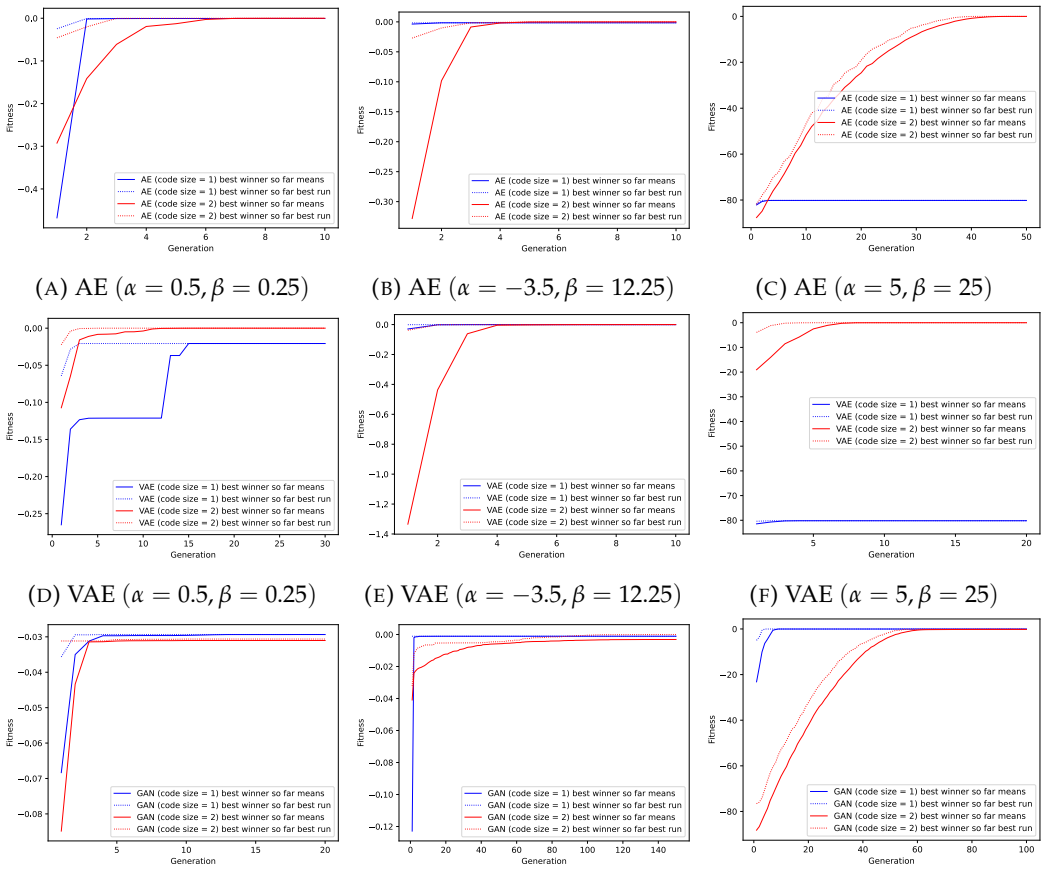


FIGURE 3.14: 5 evolutionary runs for a DE, a DEIS, and each of the 3 IEs with a code size of 2 on the target problem, $(\alpha = 5, \beta = 25)$.



(A) AE $(\alpha = 0.5, \beta = 0.25)$ (B) AE $(\alpha = -3.5, \beta = 12.25)$ (C) AE $(\alpha = 5, \beta = 25)$

(D) VAE $(\alpha = 0.5, \beta = 0.25)$ (E) VAE $(\alpha = -3.5, \beta = 12.25)$ (F) VAE $(\alpha = 5, \beta = 25)$

(G) GAN $(\alpha = 0.5, \beta = 0.25)$ (H) GAN $(\alpha = -3.5, \beta = 12.25)$ (I) GAN $(\alpha = 5, \beta = 25)$

FIGURE 3.15: IE code size comparisons for each of the 3 IE types on each of the non-linear target problems. Plots record the mean of the best winner so far in the evolutionary run and the best of the best winners so far.

3.4 Discussion

The experimental results show that IEs derived from autoencoders, VAEs, and GANs have the ability to outperform direct encoding baselines on unseen target functions from the same family as the source functions. These advantages take the form of both evolutionary search speed and accuracy of the solutions found. These advantages are observed when there are linear *and* non-linear relations between the maxima of the source and target problems in question. This bodes well for real world problems for which we assume non-linear relationships to exist between maxima in a particular family of functions. These results illustrate how information from source functions can be used to craft a search manifold that enables faster and sometimes more accurate search on similar, but unseen target functions. Given that many machine learning problems are mathematical optimisation problems under the hood, the techniques introduced here will be directly transferable to areas such as classification, regression, and reinforcement learning, as we show in Chapter 4.

We have also illustrated that the training procedure for AEs and VAEs of code size 1 is highly susceptible to local minima in the non-linear case, whereby the training points are not all accurately modelled (Figure 3.7b). The resultant manifolds therefore generalise to unseen maxima poorly, as evidenced by the poor performance of the code size 1 AE and VAE derived IEs on the extrapolative target function $(\alpha = 5, \beta = 25)$ (Figure 3.9). This results in extreme fitness plateaus with a fitness much less than an optimal solution because the manifolds do not intersect the maxima of the target problem. This is in contrast to the GAN training procedure, which does not become trapped in a local minima for a code size of 1, resulting in manifolds that extrapolate well to unseen target functions (Figure 3.7c); this has positive knock on effects on the performance of the respective IE (Figure 3.9). We expect poor generalisation to be an issue when using AE and VAE derived IEs beyond the code sizes of 1 and 2 to the general case where the code size is less than the dimensionality of the search space. Further experiments are required to gather additional evidence for this hypothesis, however, if true, could result in a heuristic such as, one should be cautious of malformed manifolds when using IEs derived from AEs and VAEs with a code size less than that of the search space. Despite the poor generalisation being a pernicious problem for a code size of 1, this problem disappears for a code size of 2 where the VAE derived IE of code size 2 extrapolates most successfully of all the techniques evaluated (Figures 3.11b and 3.14).

We observed a speed accuracy trade off occurring in the non-linear experiments for the GAN derived IE. Significant evolutionary search speed advantages are realised for the code size 1 IE compared to the baselines (Figure 3.9) and its code size 2 counterpart (Figure 3.15i). However, this comes at a cost to the accuracy of the final solution, whereby the solution closest to the (5, 25) maxima found is (5.1531, 24.9785). This is a limitation when the manifold with a dimension less than that of the search space does not intersect directly with the maxima of the target function (Figure 3.7c). A solution to this would involve switching back to a direct encoding when the search of the indirect encoding begins to plateau. This would have the advantage of the search speed gains of the IE without sacrificing the accuracy of the final solution.

One disadvantage observed when training GAN derived IEs is mode collapse (Figure A.5). This only occurred for the linearly related source problems, however it rendered the code size 1 IE unsuitable for locating new target solutions (Figure A.1) and caused the code size 2 IE to experience extremely long search times (Figure A.4). Currently, it is unclear why mode collapse occurred in this situation; it could

have been due to the architecture of the generator and discriminator, or it could have been because of the distribution of the training data. What is apparent though is the dramatic negative effects that mode collapse has on the ability of the derived IE to locate solutions to new target problems.

The GAN derived IEs of code size 2 seem to suffer from a more significant final solution accuracy problem compared to the AE and VAE derived IEs. On all three non-linear target problems, the fitness of the solutions found by the GAN derived IE of code size 2 plateaued at a fitness slightly lower than the optimal fitness (Figures 3.10, 3.13, and 3.14). The reason for this is not immediately clear because the manifold created by the code size 2 GAN derived IE is the same size as the search space and will therefore encapsulate the target maxima somewhere within itself. It would be understandable for search to be slower, however observing plateauing behaviour on a code size 2 IE was surprising. One explanation could be the fact that the manifold of the GAN derived IE seems to undergo a larger amount of folding, illustrated in Figure 3.12. This folding would be a natural by-product of the training procedure in which the GAN needs to compress its output solely around the training data. However, it may be the case that a large amount of folding can lead to local maxima in the search space, which in turn results in the fitness plateauing behaviour. Further experiments are required in order to ascertain whether this is the sole cause of these observations. If it is, there are certain regularisation techniques that can be employed during the training procedure of the GAN to alleviate this, such as weight decay or early stopping.

We saw no advantages of using an IE when the target problem in question has a maxima very close to solutions produced by the initial population of the DE. This was the case for the target problem $(\alpha = 0.5, \beta = 0.25)$ for which neither code size 1 or 2 IEs outperformed the DE (Figures 3.6 and 3.10 respectively). This provides evidence for the fact that the performance of the DE with respect to the IEs is highly dependant on the target problem. We therefore cannot say that IEs, if constructed correctly, will always outperform a DE because it is also fundamentally dependant on the location of the maxima of the function being optimised.

We provided an alternative baseline, the DEIS, in an attempt to start evolution in a location close to the maxima of the source problems. Without any additional information about the unseen target functions, and under the assumption that they are of the same family as the source functions, we can assume that this is the most sensible place to begin search. However, as was observed with the DE, the performance of the DE with informed search is again highly dependent on the location of the maxima of the target problem. If the target function maxima was closer to the centroid of the source function maximas than the mean of the initial distribution of the DE, the DE with informed search was shown to perform evolutionary search faster than the DE. As a result, the informed search brought large advantages over the DE in certain cases (Figure A.7), whereas in other cases it performed the worst of all the techniques (Figure A.6).

3.5 Future Work

This current work is greatly limited in that it only explores the proposed techniques on a small subset of bivariate quadratic functions. We hope that the advantages offered by generative model derived IEs and the insights gained here generalise to other function families. We want to test these techniques in other areas for which optimisation is a key component, such as classification, regression, and reinforcement

learning (explored in Chapter 4). We particularly believe advantages will be realised in transfer learning where knowledge from source domains (source problems) can be utilised via an IE on target domains (target problems).

The experiments carried out here were limited by computational power resulting in only 5 evolutionary runs for each technique, we would like to repeat the results with a larger number of runs. We would also like to repeat the experiments with a sweep over hyperparameters, such as generative model architectures and evolution parameters. Testing on more than 3 target problems would help us to ascertain a greater understanding of when IEs give advantages and at which point a direct encoding performs best.

We would also like to explore some of the remedies presented in 3.4. It would be interesting to test whether switching back to a direct encoding after realising the search speed gains of the indirect encoding results in a technique that is both faster and more accurate than using a DE or IE alone. We would also like to explore whether performing regularisation techniques on the GAN results in a smoother search manifold and in turn prevents fitness plateauing.

3.6 Conclusion

In summary, we have compared the performance of indirect encodings derived from autoencoders, VAEs, and GANs on unseen target functions for two families of bivariate quadratic functions. We have shown that all three techniques can produce IEs that, for certain target functions, outperform baselines. We have provided informative illustrations that highlight the differences between the different types of IE. We have shown that the comparative performance of the IEs compared to the DE baselines is highly dependant on the target function being optimised. We have highlighted pitfalls of AE and VAE derived IEs when using a code size of 1, which results in poor performing IEs. We have suggested remedies to improve the speed accuracy trade off observed when using a GAN derived IE. Finally, we have highlighted and suggested remediations to the manifold folding that occurs in the IEs derived from code size 2 VAEs and GANs, which we have hypothesised cause local maxima in the search space.

Chapter 4

Generative Models over Neural Controllers for Transfer Learning

Transfer learning refers to the situation where what has been learned on one task can be exploited in a different but related task [44]. An example from the supervised learning setting would be to use the initial layers of a classifier trained to detect pictures of cats in order to initialise or pretrain a classifier used to detect pictures of dogs. One would assume that the features learned in the first layers to detect cats such as edges, corners, changes in lighting etc. will also be useful in detecting dogs. Pretraining a model with a large auxiliary dataset that differs from the target dataset enhances object detection performance compared to a model trained *solely* on the target image dataset [41]; suggesting the features that were useful in classifying objects in the first dataset were also useful for classification in the second dataset.

In reinforcement learning (RL) tasks, transfer learning consists of leveraging prior knowledge from a set of source domains to improve learning on some target domains [156]. For example, the ability to walk is a prerequisite for many different tasks such as hunting, building and evading predators. However, when learning each of these tasks, animals do not have to relearn how to walk each time, they reuse the previously learnt walking behaviour that was required for some previous task (source domain). It would similarly be inefficient for RL agents to relearn primitive actions every time they learn a new, but related, task. To avoid this, there needs to be some mechanism by which *invariant* knowledge over the source domains is gathered and stored.

Evolutionary algorithms (EA) are a set of gradient-free search algorithms that can be used to find and optimise policies for RL control tasks. EAs also have the ability to store invariant domain knowledge via an indirect encoding (IE). One way of producing a domain-dependent IE is by learning a distribution over the parameters of previously found solutions using a generative model, this is known as a data-driven encoding (DDE) [40]. Data-driven encodings have been explored in a number of works referenced in Section 4.1. However, despite the proven link between evolutionary processes and learning theory suggesting evolution has the ability to generalise via their IEs [67, 147], to the authors knowledge, no work has yet applied DDEs to transfer learning in RL tasks. Consequently, in this work we use indirect encodings to capture the similarities in neural network controller parameter spaces for a set of source domains, and then reuse these IEs to evolve solutions on a set of unseen target domains with much greater speed.

We argue that the optimisation of some fitness function of an RL control task $f(\delta(\mathbf{g};\boldsymbol{\theta});\boldsymbol{\tau})$, with respect to, $\mathbf{g}$, the genotype, where $\delta(\mathbf{g};\boldsymbol{\theta})$ is the phenotype - which in this case would be the weights of a neural network controller - and $\boldsymbol{\tau}$ is a vector of some domain parameters, is exactly the same as optimising instances of families of functions as was done in Chapter 3. Thus, we can directly apply the techniques

explored in Chapter 3 to learn a generative model over solutions to some subset $d_s \subset \mathcal{D}$ of source domain instances, and subsequently use this model as an IE, δ, to optimise the fitness function to some target domain instances $d_t \subset \mathcal{D}$ where $d_s \cap d_t = \emptyset$ at a much greater speed than without an IE, thereby, performing efficient transfer learning in RL control tasks.

Three different IEs are explored: the decoders derived from a trained autoencoder (AE) and a variational autoencoder (VAE), as well as the generator derived from a trained generative adversarial network (GAN). Hence, we address the following questions, which are central to transfer learning:

1. What knowledge is relevant for generalisation to future tasks?
2. Which storage mechanism should be used for this knowledge?

For 1., we model the distribution of well-performing solutions in parameter space over a set of source domain; for 2., we store relevant knowledge in the IE. We demonstrate the ability of this technique on three OpenAI gym environments: Continuous Mountain Car, Frozen Lake and Bipedal Walker and give evidence that IEs trained in this way perform much better than two other baselines.

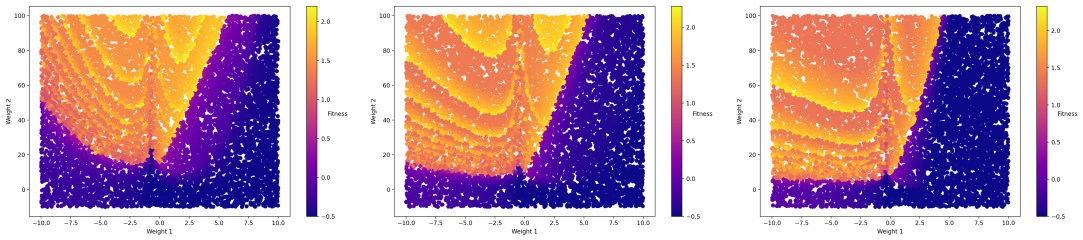
Contributions

1. We propose a way to perform transfer learning in RL control tasks by capturing invariant domain knowledge in the neural network controller parameter space using generative models. The generative part of the model is then used as an indirect encoding to reevolve on similar but unseen tasks.
2. We provide evidence that our method results in much faster learning speed compared to two baselines: a direct encoding and a direct encoding with an informed search start location.
3. We demonstrate these advantages for three different OpenAI gym RL control tasks.
4. We compare the performance of AEs, VAEs, and GANs on these three different control tasks.
5. We analyse the failure modes of some of the models and suggest remediations.

4.1 Related Work

An autoencoder and a VAE are used in [106] and [18], respectively, to learn a distribution over neural network controller parameters for benchmark RL tasks, such as Bipedal Walker and Cart Pole. Unlike our work, the focus is not on transferring gained knowledge to unseen target domains, rather, the enumeration and analysis of behaviours on a single domain instance. Furthermore, the fitness of discovered solutions is not used as a metric of evaluation in either [106] or [18], whereas our work considers the fitness of produced solutions of paramount importance.

DDE-Elites [40] uses a VAE to learn a distribution over joint angles for well-performing solutions to a 2D planar arm inverse kinematics environment. Similar to our work, evolutionary search is subsequently performed in the latent space of the VAE decoder, which acts as an IE, and is compared to a direct encoding. Also, optimisation on novel but similar tasks is performed, akin to the transfer learning



(A) Engine power = 0.0010. $f(\delta_I(\mathbf{g}); [0.0010])$ (B) Engine power = 0.0015. $f(\delta_I(\mathbf{g}); [0.0015])$ (C) Engine power = 0.0020. $f(\delta_I(\mathbf{g}); [0.0020])$

FIGURE 4.1: Fitness function plots for three different engine power settings in CMC. Plots were generated by producing 1000 random neural networks and assessing their fitnesses. The axes are the weight values of the neural networks and the colour represents the fitness.

experiments in our work. However, unlike our work, the technique is not evaluated on RL control tasks. Furthermore, in [40], the parameters of optimisation are joint angles as opposed to the parameters of neural network controllers.

Conditional GANs (cGAN) have also been used to construct IEs. The optimisation of *high-level* control policies for a robotic arm, and the design of buildings, an energy plant and the respective energy distribution network for an urban neighbourhood is assisted by a cGAN in [57] and [58] respectively. Similar to our work, [57] tests the generalisation capabilities of the trained generative model on unseen but related domains. Unlike our work, neither [57] or [58] consider neural network controllers, nor do they perform any form of search in the latent space of their respective generative models.

COIL [6] applies the ideas of [40] to the concept of constrained mathematical optimisation problems. AutoMap [96] uses the decoder of an autoencoder as an IE to re-evolve on a rugged fitness landscape resulting in *much* faster learning on the same task.

All of the works cited in Section 4.1 consider only a single type of IE, whereas our work compares three different types of IE.

4.2 Conceptual Overview

In RL, each domain typically has a reward function that gives a set reward for each state entered by an agent, an RL agent will try to maximise this cumulative reward by taking actions resulting in larger rewards. An agent whose policy is optimised by an EA typically will receive a *single* valued reward at the end of an environmental run, which is subsequently used to inform the selection procedure. Step-wise RL rewards can be converted into a single valued final reward by simply summing the rewards for all the steps of the episode.

One can imagine that for completely different domains, the fitness functions are vastly different. For example, a neural network trained on Continuous Mountain Car (CMC) would not work at all on Bipedal Walker. Aside from the fact that the number of inputs and number of outputs of the control networks are different, even if they were not, the likelihood is that a good solution on one domain would not be a good solution on another. However, for similar domains it may be the case that the fitness functions are somewhat similar. This similarity between fitness functions may result in much of the search space for a set of similar domains being of low fitness, and thus, need not be considered during optimisation.

For example, CMC can be solved with a very simple neural controller; there are 2 inputs and 1 output and a solution can be found using a linear controller with only 2 weights (no bias). The more complicated fitness function used in this work encourages the car to get as close to the goal as possible whilst also rewarding the car for getting there in a faster time *and* for having a lower speed at the goal. There are a number of adjustable domain parameters, $\boldsymbol{\tau}$, within CMC, such as gravity and engine power. For each of these parameterised domain instances, the resulting fitness function, $f(\delta(\mathbf{g};\boldsymbol{\theta});\boldsymbol{\tau})$, will be different.

Figure 4.1 illustrates the fitness functions for three different engine power settings (values of τ) and a fixed δ (in this case the identity mapping, δ_I). It can be observed that despite the fitness functions being different, they contain structural similarities. For example, no matter what the power value, all networks with a second weight value of less than 0, or a first weight value greater than 5, have poor fitness scores. Information of this kind could be very useful in future search because we can assume that for domains with similar power values, it will be fruitless to search within these low valued fitness areas. Similarly, we can see that the higher fitness solutions are located in a *ridge* on the top left of the fitness function. This ridge is at a different place for each power setting but still in the same area of the weight space.

Without prior information about where good solutions in the weight space are located, there would not be much reason to start search anywhere other than at 0 with an unknown starting distribution variance. If one were therefore going to constrain the search space to [-100, 100] (which is reasonable when searching over neural network weight space) the areas of interest in Figure 4.1 are a small proportion of the space. Therefore, without integrating previous information into search, a substantial amount time and computational power might be required to locate this region.

The main proposition of this chapter is that we can capture the area of good solutions in neural network controller weight space common to a set of different parameterised source domains, d_s using generative models. These generative models will then act as an indirect encoding, thereby mapping arbitrary genotype values to high fitness areas of the phenotype space. We can then evolve again on some unseen target domains, d_t, using this IE, resulting in a much more precise search. As we have shown in Chapter 3 this proposition can be extended much further than the neuroevolution experiments presented here and can be used for any parameterised optimisation problem where there are commonalities amongst the different search spaces.

4.3 Methodology

We prepared training data for the generative models by using CMA-ES [50] to evolve solutions on a set of parameterised source domains, d_s, using a direct encoding. The initial centroid of CMA-ES was set to $[0.0]^n$, where n is the number of weights in the neural network controller, and the initial σ was set to 1.0. Each element in the search space was bound between $[-100, 100]$ [1]. A solution was defined as a neural network controller that achieved some minimal fitness value.

Next, all three generative models were trained using the training data, which were vectors of neural network controller weights. For all three generative models a hidden layer of 64 neurons with ReLU activation was used in both the encoder and

[1] Often CMA-ES can discover solutions with very large values making it more difficult to train a generative model over.

decoder or the discriminator and generator. The hidden layer in the decoders and generators meant that the respective IE was able to capture non-linear relationships between the genotype and phenotype space. ADAM [64] was used as the optimiser for all generative models, with a learning rate of 1^{-3} used for the autoencoder and VAE, and learning rates of 2^{-4} and 5^{-4} used for the generator and discriminator of the GAN respectively. For the experiments in Section 4.4 a code size of 1 and 2 was used in the generative models. Time constraints prevented us from experimenting with a code size larger than 2, however, it is perfectly reasonable to assume that performance could be positively affected by larger code sizes and it certainly warrants further investigation. The number of epochs used were 10 000, 10 000, and 40 000 for the autoencoder, VAE, and GAN respectively. Each of the three types of model were trained 5 separate times.

We then performed evolution using CMA-ES over the latent space of the decoder, in the case of the AE and VAE, or the generator, in the case of the GAN. For the AE, the initial centroid was set to $[0.5]^m$, where m is the size of the code. This is because the learnt code was bound in the range $[0, 1]$ due to the sigmoid activation function in the pre-code layer. For the VAE and GAN, the initial centroid was set to $[0.0]^m$ because their training procedures are such that the typical code processed by the latent space is that derived from a unit normal distribution. An initial sigma of 1.0 was used for all the IE experiments. This evolutionary process was performed 5 times for each generative model trained.

For evaluation, we tested the speed of evolution and the fitness of the best winner found for the three IEs and two baselines on a set of target domains. The two baselines were: (1) a direct encoding, and (2) a direct encoding *starting* evolution from a controller pre-trained to maximise the average fitness over the full set of source domains. We call this pre-trained controller a universal controller (UC), which has been previously used as a baseline for MAML[34]. Each of the two baselines, the DE and the UC, were ran 10 and 5 times respectively.

4.4 Experiments

4.4.1 Continuous Mountain Car

Continuous Mountain Car (CMC) is a variation of Mountain Car in which a continuous force value is applied to the car instead of a discrete force value. The car begins the run in a trough of a valley and aims to reach a flag atop one of the two mountains. The simulation ends once 1000 time steps have elapsed or the car reaches the goal position.

In CMC one can modify the engine power of the car such that the force applied to the car by the controller, and thereby the acceleration of the car, will be different given the same control signal. We use this engine power as the modifiable domain parameter, τ, in these experiments. Each instance of engine power will result in a similar but related fitness function, as in Figure 4.1. A controller trained on one engine power instance will either overshoot or undershoot the flag.

In order to induce a larger difference in the fitness functions between each engine power setting, we modified the fitness function such that a greater reward is achieved for having a lower velocity at the moment the flag is reached [2] - shown in Equation 4.1. The default reward was not influenced by the velocity of the cart at

[2]The alternative would have been to use environments with larger engine power increments for training and testing, however, this had large unstable effects on the dynamics of the cart.

the flag. This has the effect of more drastically penalising an overshooting cart (an undershooting cart is penalised more greatly by the time component of the fitness function) as a result of a small increase in engine power. A set of fitness functions that are too similar fail to highlight that interpolative advantages gained by using the IE techniques. It would also mean the controller pre-trained to maximise the average fitness of the set of source domains (the UC) can achieve a much higher fitness, thereby, bootstrapping the informed start DE baseline more significantly; this subsequently erodes the comparative advantages of the IE techniques.

There are three aspects of the modified fitness function, Equation 4.1: a higher reward given for a cart achieving a smaller distance to the flag, a higher reward given for locating the flag in a lesser amount of time, and a higher reward given for achieving a lower velocity at the moment the flag is reached. The modified fitness function, f, is given by:

$$f = \begin{cases} p_c + (1 - \frac{t_f}{t_{max}}) + (1 - \frac{v_f}{v_{max}}), & \text{if flag reached,} \\ p_c, & \text{otherwise,} \end{cases} \tag{4.1}$$

where p_c is the closest position the cart has come to the flag over the course of the entire run (this value increases as the car gets closer to the flag, up to a maximum of 0.45); t_f is the number of time steps elapsed when the flag is reached; t_{max} is the maximum number of time steps, which is 1000 in this case; v_f is the velocity when the flag is reached; and v_{max} is the maximum velocity of the cart.

In order to collect training data for the generative models, CMA-ES was used (as described in Section 4.3) with a direct encoding to evolve 333 solutions for each of the three source domain instances, d_s, with an engine power in the following set: $\{0.0008, 0.0012, 0.0016\}$, giving a total of 999 solutions. A population size of 100 was evolved for 100 generations. A neural controller with 2 inputs, 1 output, no hidden layers, and no bias was used, resulting in a weight space size of 2. Once trained, we used the decoder or generator as the IE for another evolutionary process (as detailed in Section 4.3) on CMC target domains, d_t, with test engine power values in the set: $\{0.0010, 0.0014\}$.

Figure 4.2 shows the fitnesses of the best winner so far for the evolutionary runs for a test engine power of 0.0014. The plotted IE is derived from a VAE with a code size of 1 - the best performing of all 3 types of IE. The run using an IE begins the evolutionary run with a much higher fitness than both the DE and the UC, and repeatedly finds a solution that beats the baselines after 30 generations. For this particular IE, the mean of the best winner so far starts above 2.0 in the first generation, which is considered to be a full solution to this environment. This demonstrates that just *randomly* sampling in the latent space of the VAE's decoder results in neural network weights that are close to the maximum fitness of this unseen test domain. These results show that the IE and UC have the ability to integrate information from their own respective training procedures to bootstrap their evolutionary procedures.

Figure 4.3 compares the performance of an AE, a VAE, and a GAN with code size 1 for CMC with a test engine power of 0.0014. It shows that both the AE and VAE derived IEs outperform the UC, however the GAN plateaus at the same fitness as the UC despite starting with a larger fitness in the earlier generations. Figure 4.4 compares the performance of all 3 IEs for code sizes 1 and 2. It suggests that using a code size of 1 is preferable in this domain with the settings examined so far for all 3 IEs. It also suggests that using a code size of 2 might lead to premature convergence in a number of cases in which a higher fitness solution cannot be found despite one being found in a previous run.

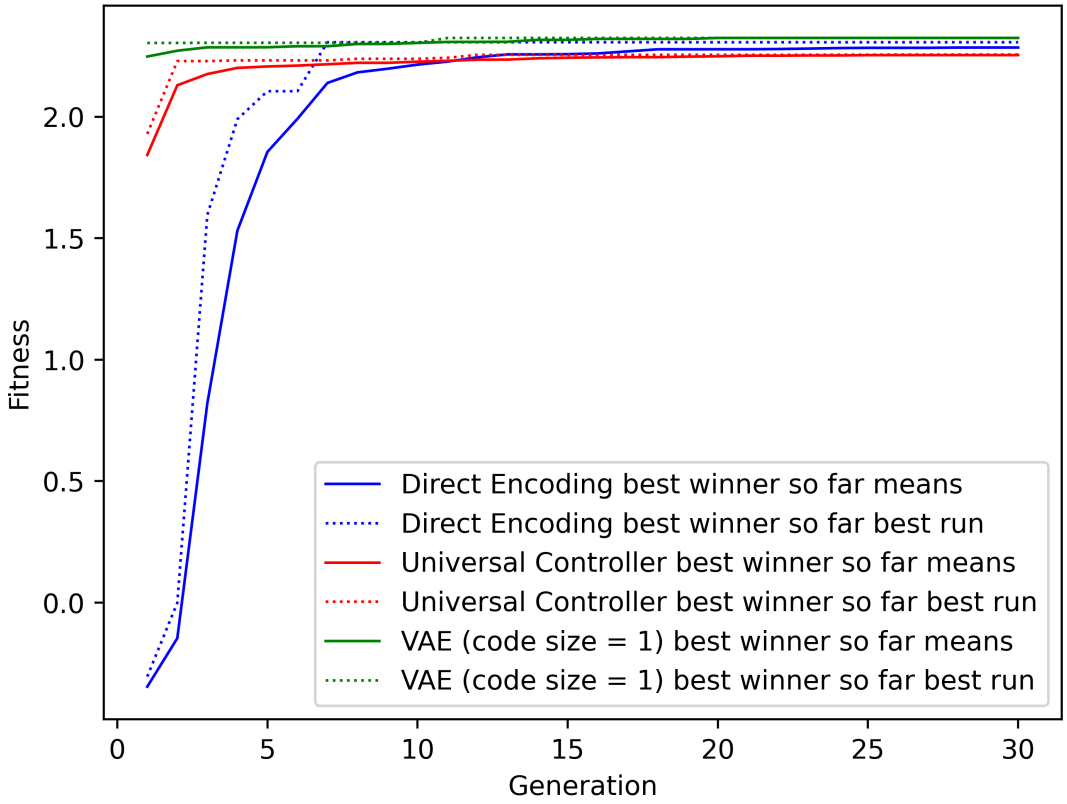


FIGURE 4.2: 10 evolutionary runs for a DE, 5 evolutionary runs for a UC, and 5 evolutionary runs for an IE derived from a VAE with a code size of 1 on CMC with an engine power of 0.0014. The fitnesses plotted are those of the best winner so far, this is the best solution found so far during the evolutionary run. The solid lines are the mean fitnesses of the runs and the dotted line is the best run according to the final generation fitness.

Figure 4.5 highlights some key information about the controller's weight space. The illustrated training data, which represents the weights of solutions to the source domains, is shown as lying in regular parabolic shapes for each particular engine power. The grey dotted line, which represents the enumeration over the latent space of the IE, shows how the decoder of the VAE learnt to map values from its latent space to the weight space such that it could reconstruct the training data as accurately as possible using a single dimension. It intersects the weight space near the center of the source domain solutions, which evidently happens to coincide with high fitness solutions of the target domains. It is interesting to see how this enumerated weight manifold is more sparse in areas of least interest and more dense in areas with a higher likelihood of finding a good solution, this is due to the regularisation effect induced by training procedure of the VAE.

Figure 4.5 also shows the starting positions of evolutionary search for the DE and the UC as red and green crosses, respectively. Without previous knowledge of the search space, there is no other information to suggest that the best place to start search is anything other than the origin; for this reason we start evolutionary search of the DE at $[0.0, 0.0]$. However, starting search at the origin results in a significant amount of time and compute expended as the search distribution maneuvers into the area of good solutions for every single new domain instance. Alternatively, the UC has been trained to maximise the average fitness over the three source domains.

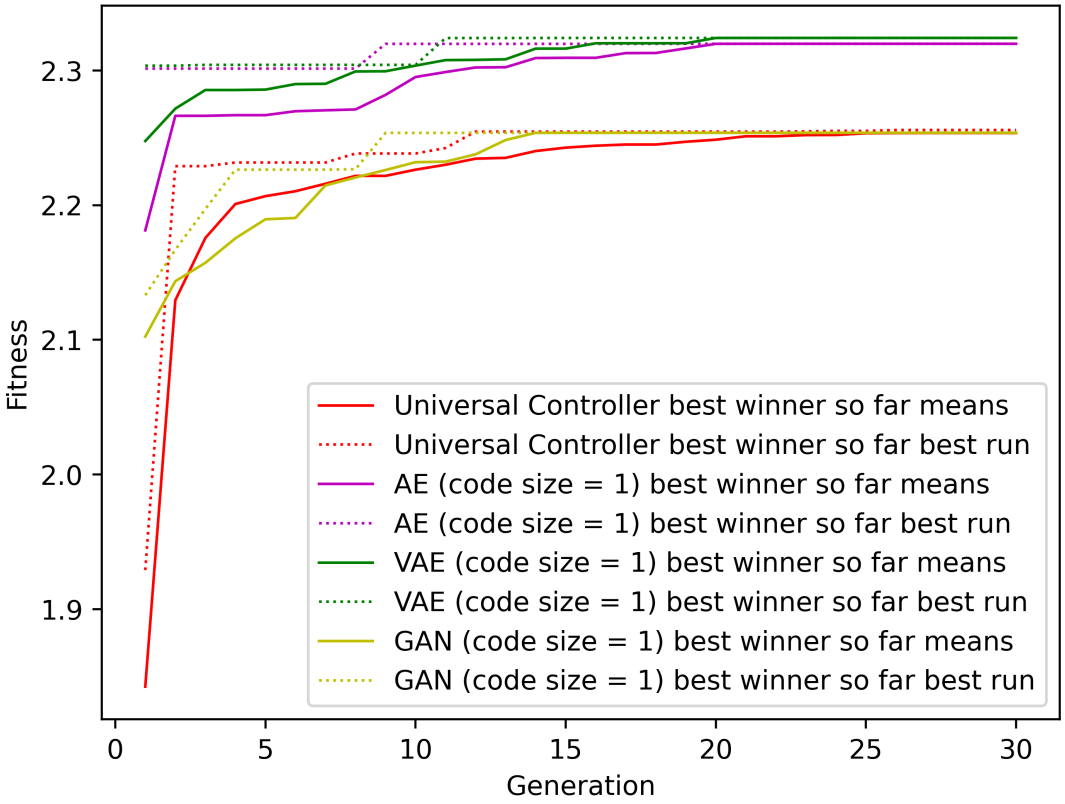


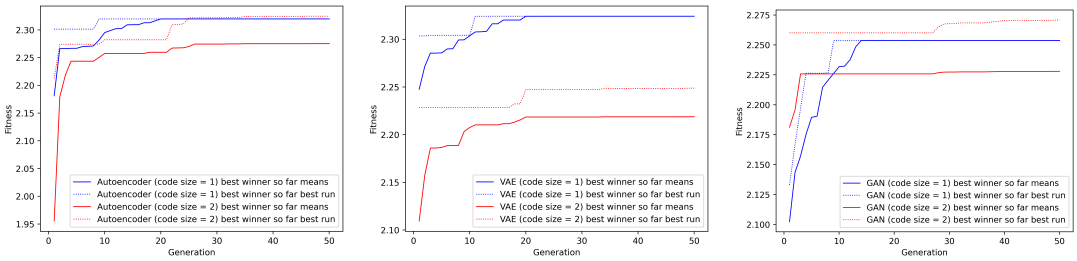
FIGURE 4.3: Comparison of IEs derived from an autoencoder, a VAE, and a GAN on CMC with test engine power 0.0014. All IEs use a code size of 1. Universal controller is also shown for comparison but a direct encoding is left out because its fitness values for the first few generations are much lower. The fitnesses plotted are those of the best winner so far, this is the best solution found so far during the evolutionary run. The solid lines are the mean fitnesses over 5 evolutionary runs and the dotted line is the best run according to the final generation fitness.

Even though the UC does not achieve perfect fitness on any of the source domains individually *or* the newly tested 0.0014 target domain, it allows search to start in a much more informed position, which leads to much faster convergence to a solution on the new target domain.

4.4.2 Frozen Lake

Frozen Lake (FL) is a simple, text-based, maze-like environment with a discrete state and action space. An agent aims to move from a start location to a goal location without falling through a hole into the lake within a designated time limit. By default, FL is a stochastic environment, however we modify it to be deterministic for these experiments so that the resulting fitness function is deterministic. The action space consists of four actions: north, east, south, west. The state space consists of one integer representing the current tile that the agent is located at.

A neural network controller with 1 input, 4 hidden units with a ReLU activation function, and 4 output units (one for each of the 4 discrete actions) with a sigmoid activation was used - this resulted in a neural network weight space size of 28. In order to select the action of the agent, the index of the output neuron with the largest



(A) Autoencoder. (B) VAE. (C) GAN.

FIGURE 4.4: Comparison of IE performance between code sizes 1 and 2 on CMC with test engine power 0.0014. The fitnesses plotted are those of the best winner so far, this is the best solution found so far during the evolutionary run. The solid lines are the mean fitnesses over 5 evolutionary runs and the dotted line is the best run according to the final generation fitness.

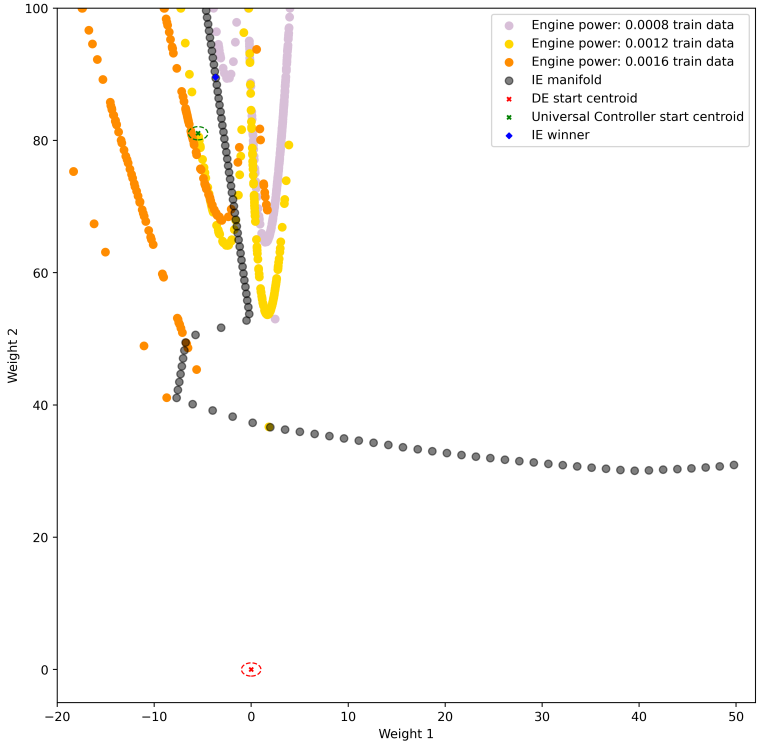


FIGURE 4.5: The weight space of the neural network controller for the CMC domain. The thistle, gold and dark orange points represent the training data used to train the generative models. The grey dotted line labelled 'IE manifold' represents an enumeration over the one dimensional latent space of the decoder derived from the VAE in Figure 4.2 mapped into the two dimensional weight space. The enumeration is over the range [-3,3] with increments of 0.05. The blue diamond at (-3.68, 89.58) represents the best winner found by the decoder. The red and green crosses represent the initial centroids of search for the DE and the UC, respectively, with the dotted circles representing the initial sigma of the search distributions.

output value was used. Not all positions in the maze can be located by a controller with zero hidden layers, it was for this reason that a hidden layer was included in the controller. FL is therefore an appropriate domain to demonstrate the ability of our

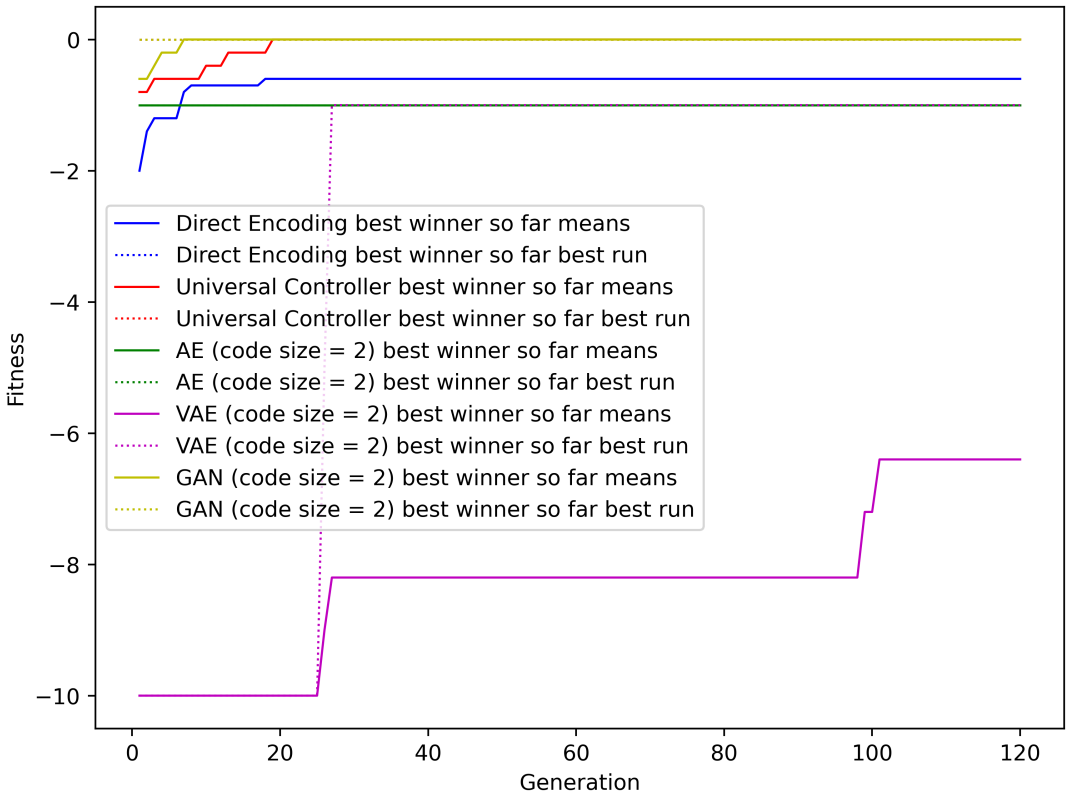


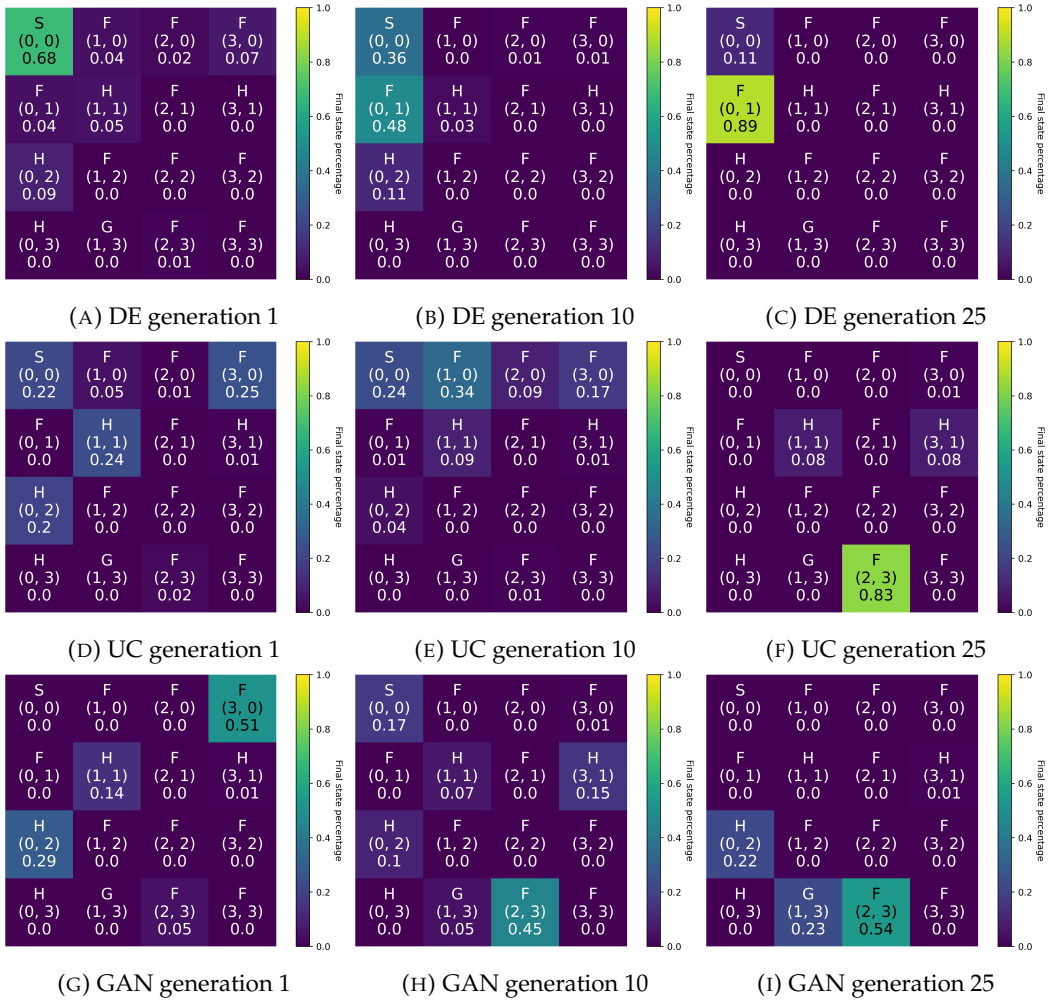
FIGURE 4.6: 10 evolutionary runs for the DE and 5 evolutionary runs for both the UC and all 3 IEs on Frozen Lake with an target goal position of (1,3). All IEs use a code size of 2. The fitnesses plotted are those of the best winner so far, this is the best solution found so far during the evolutionary run. The solid lines are the mean fitnesses over 5 evolutionary runs (10 for the DE) and the dotted line is the best run according to the final generation fitness.

learnt IEs to find neural network controllers with a hidden layer and with a larger number of weights than in the CMC experiments.

For FL, we use goal position as the modifiable domain parameter. The source domains, d_s, consists of those with goal positions in the set: {(1,2), (3,2), (3,3)} and the target domains, d_t, are those with goal positions in the set: {(1,3), (3,0)}. The reward given is inversely proportional to the manhattan distance between the end location of the agent and the goal, with an additional reward of -10 given if the agent falls in a hole.

The training data for the generative models was collected in the same way as in CMC: 999 solutions were found, 333 for each of the three source domains. These solutions were found using a random search. In practice, random search found solutions faster and more frequently than a directed search procedure. We hypothesise that this is because the fitness function used was deceptive [42].

Figure 4.6 shows the results on the (1,3) target goal position for all 3 IEs and the two baselines. The plot shows that the only IE to achieve a maximum fitness of 0 (locating the goal) is the GAN, which it does repeatedly over the 5 runs. The IE derived from the GAN has not been trained on controllers that locate this target goal but it is successfully able to interpolate over the weight space in order to quickly generate controllers capable of finding this new goal location. The DE encodes no previous domain knowledge and therefore takes a much longer time to converge to



(A) DE generation 1 (B) DE generation 10 (C) DE generation 25

(D) UC generation 1 (E) UC generation 10 (F) UC generation 25

(G) GAN generation 1 (H) GAN generation 10 (I) GAN generation 25

FIGURE 4.7: Representations of the Frozen Lake environment highlighting the percentage of the population in a single generation that ends the episode in a particular tile. Each tile in the 4x4 FL environment is labeled by one of the following types: S, the starting location; F, frozen tile (traversable); H, hole; and G, the goal location. The target domain with goal position (1,3) is shown. The coordinates of the tile and the aforementioned percentage are also shown. Each subfigure highlights the state of the environment at different generations for the DE, UC and GAN plotted in Figure 4.6.

a good solution.

Figure 4.6 also shows the very poor performance attributed to the VAE derived IE. This is interesting considering that the VAE was the highest performing IE in the CMC domain, however, this could be due to the presence of the hidden layer in the controller network (further analysis in the Section 4.5). Although the AE derived IE does not perform as poorly as the VAE derived IE, it never achieves maximum fitness in this domain, meaning it never locates the target goal of (1,3).

Figure B.1 in Appendix B.1 shows results comparing the performance code sizes 1, 2, and 3 for the AE, VAE, and GAN. The plots illustrate that a code size of 2 performs best for the GAN, the AE seems to gain advantages by using a code size of 3, and the code size best suited to the VAE is 1 - although performance remains very poor.

Figure 4.7 illustrates that without any information about the solution space the

DE chooses from the four actions equally in the starting tile resulting in half of the controllers in generation 1 not moving at all. Alternatively, the GAN derived IE from Figure 4.6 biases the solution space such that *zero* individuals in generation 1 choose either the action north or west and end the episode in the start state. This is because no goal position in the source domains was found by solutions that chose action north or west in the starting location, therefore, the IE biases search away from weight space that results in those behaviours. Figure 4.7c shows that as the number of generations increases the DE gets stuck in a local minima at the deceptive (0,1) location. In contrast, 23% of the controllers that the GAN derived IE produces find the goal (Figure 4.7i).

4.4.3 Bipedal Walker

Bipedal Walker (BW) is a simulation that requires the design of a controller that allows a two legged robot to walk as far as possible in a fixed time without falling over. It is more complicated than the previously tested domains due to the fact that there are 24 state inputs and 4 action outputs. In these experiments we use a controller network with no hidden layers resulting in 100 tunable weights. This demonstrates the capability of our technique to scale to a one hundred dimensional RL control problem.

For this domain, we use the knee speed as the modifiable domain parameter. We collected training data for the IEs using source domains, d_s, with the knee speeds: {2, 4, 6}. The default reward function for BPW is used. Target domains, d_t, with knee speeds of {3, 5} were used to evaluate performance.

Figure 4.8 shows that the best winners for both a GAN derived IE and the UC start the evolutionary run with a much higher fitness than the DE. Due to the difficulty of this domain, it takes a significant number of generations for the DE to generate a solution with fitness greater than 250, however, the IE and UC already achieve this in the first generation. The GAN finds solutions with a much higher fitness faster than the UC, however, it begins to plateau after a short amount of time, and is eventually overtaken by the UC. We hypothesis that this plateauing could be eased by using a larger code size. It may be the case that a code size of 2 is unable to fully capture the commonalities between the source domain fitness functions in weight space and therefore struggles to interpolate.

Figure 4.9 confirms that a GAN derived IE achieves the best performance out of the other two types of IE. Furthermore, it is the only IE type that achieves a higher fitness in the initial generations than the UC. However, both the AE and VAE are able to attain higher fitnesses in the initial generations than the DE which has been left out due to much lower fitnesses at the beginning of its evolutionary runs.

Comparing a code size of 1, 2, and 3 for the three types of IE - Appendix B.2 - does not reveal any general pattern. The VAE seems to benefit most greatly from a code size of 1 (not plotted in Figure 4.9) achieving performance comparable to a GAN with a code size of 2. It also highlights that the AEs performance is decreased by using a code size of 3 compared to code sizes 1 and 2.

The code for all the experiments described in Section 4.4 is available at `https://github.com/jamesbut/IEGymExps-PPSN2022`.

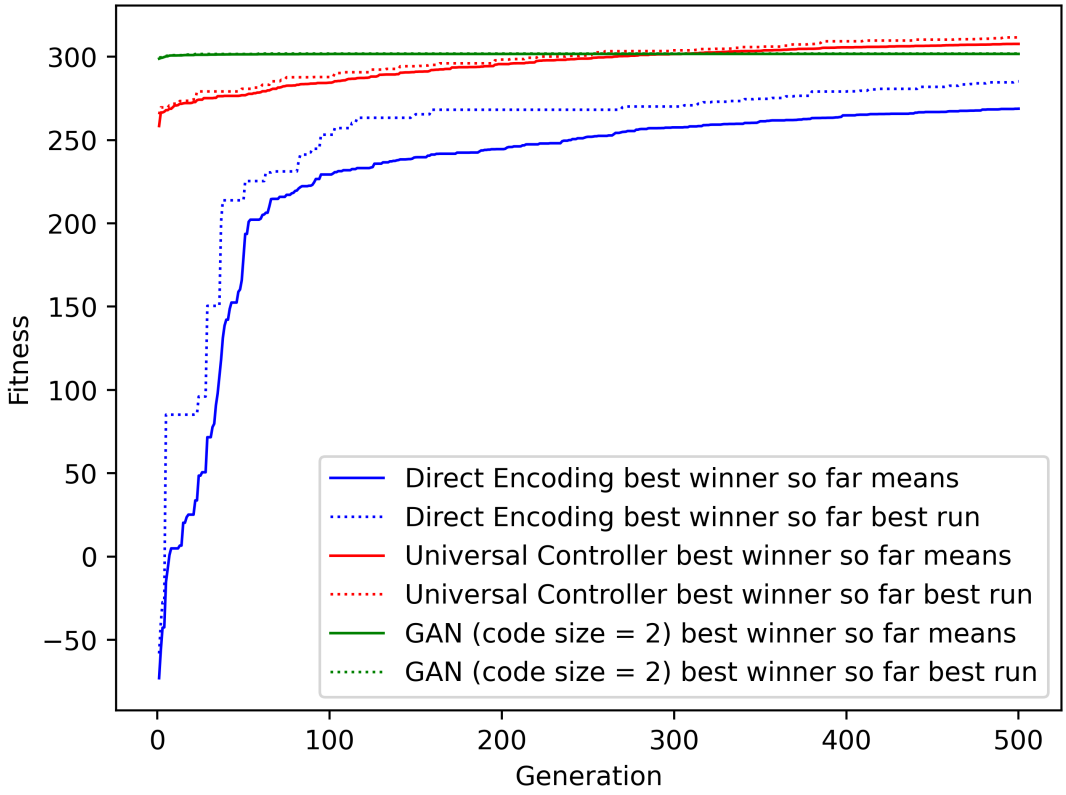


FIGURE 4.8: 10 evolutionary runs for the DE and 5 evolutionary runs for both the UC and a GAN with code size = 2 on Bipedal Walker with a knee speed of 5. The fitnesses plotted are those of the best winner so far. The solid lines are the means of the runs and the dotted line is the best run according to the final generation fitness.

4.5 Discussion

The results in Section 4.4 show clearly that under certain conditions we can learn indirect encodings that improve evolutionary search speed compared to two baselines on unseen target domains. Also, in some cases, such as the CMC experiments, the VAE derived IE locates a solution with a higher fitness overall than both baselines. This is the first time in the literature that generative models have been shown to produce IEs that can find neural network controller solutions for tasks whose solutions have not been used to train the IE (transfer learning). It has also demonstrated that one can perform evolutionary search in a much lower dimensional space and, not only locate viable solutions to known tasks, but interpolate to unseen tasks too.

Due to the improved search speed gains these techniques could provide a way to scale evolutionary algorithms to optimise neural network controllers with a much larger number of parameters, a feat EAs have traditionally struggled to achieve. However, there could be multiple challenges with this. For example, the time saved using these techniques may be less than the extra time taken to train the generative models on very large networks. Also, the minimum latent space size needed to capture all the relevant information in the search space is not known without an extensive hyperparameter sweep, which would become increasingly timely as the search space grows and again may erase the search speed gains of using an IE. Finally, the extrapolative accuracy of the IEs on the target functions may degrade in higher dimensional spaces without an exponential increase in the amount of training

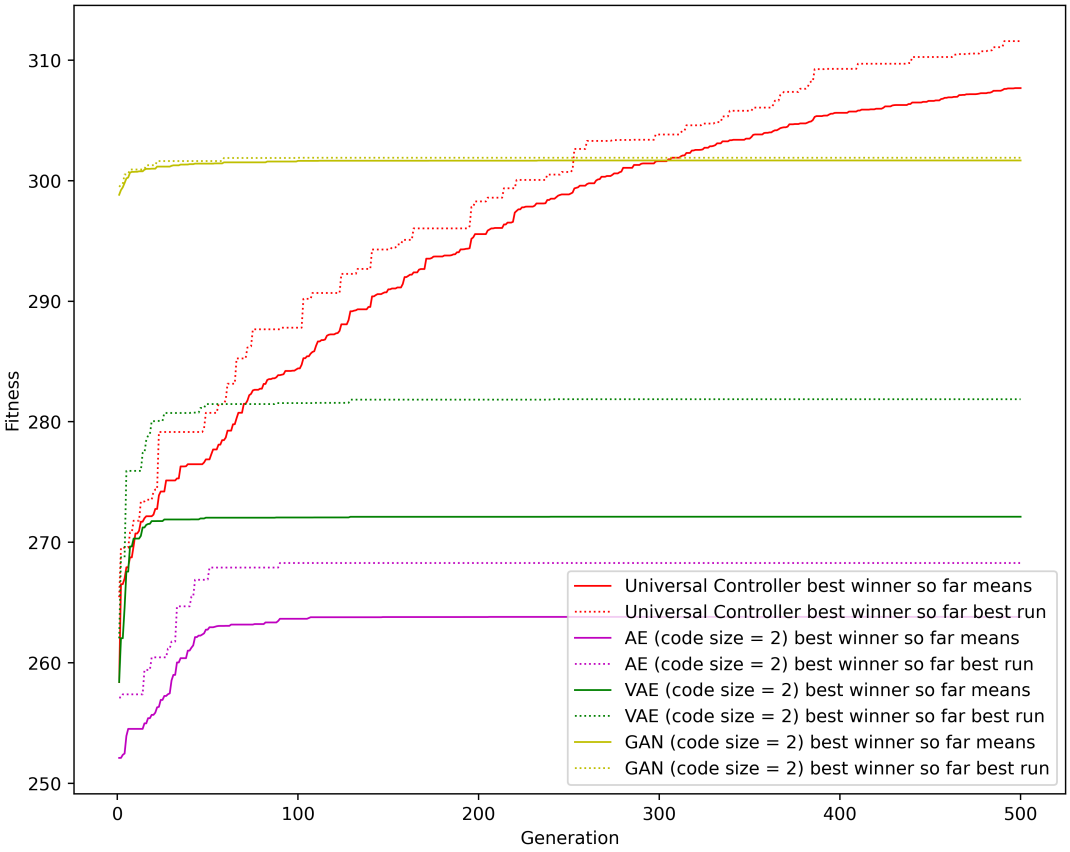


FIGURE 4.9: Comparison of IEs derived from an autoencoder, a VAE, and a GAN on BW with test knee speed 5. All IEs use a code size of 2. Universal controller is also shown for comparison but a direct encoding is left out because its fitness values for the first few generations are much lower. The fitnesses plotted are those of the best winner so far, this is the best solution found so far during the evolutionary run. The solid lines are the mean fitnesses over 5 evolutionary runs and the dotted line is the best run according to the final generation fitness.

data due to the curse of dimensionality. This training data is computationally expensive to generate because it requires the optimisation of similar functions to the target functions using a direct encoding. The challenges listed above are only hypothetical and would have to be confirmed with further experiments.

The experiments comparing the performance of different code sizes unfortunately do not provide evidence to prove hypotheses of the form:

(1) As code size increases evolutionary search speed decreases.

(2) As code size increases the fitness of the best performing solutions increases.

(1) would be a reasonable hypothesis to posit because a larger code size would mean a larger number of parameters to optimise. However, Figure B.2c provides evidence against this because the IE with the lowest code size does not achieve higher fitnesses quicker than the higher code size IEs. (2) would also be sensible to assume because one would imagine that as we increase the code size, we increase the size of solution space available to search within. Again however, the VAE derived IEs with

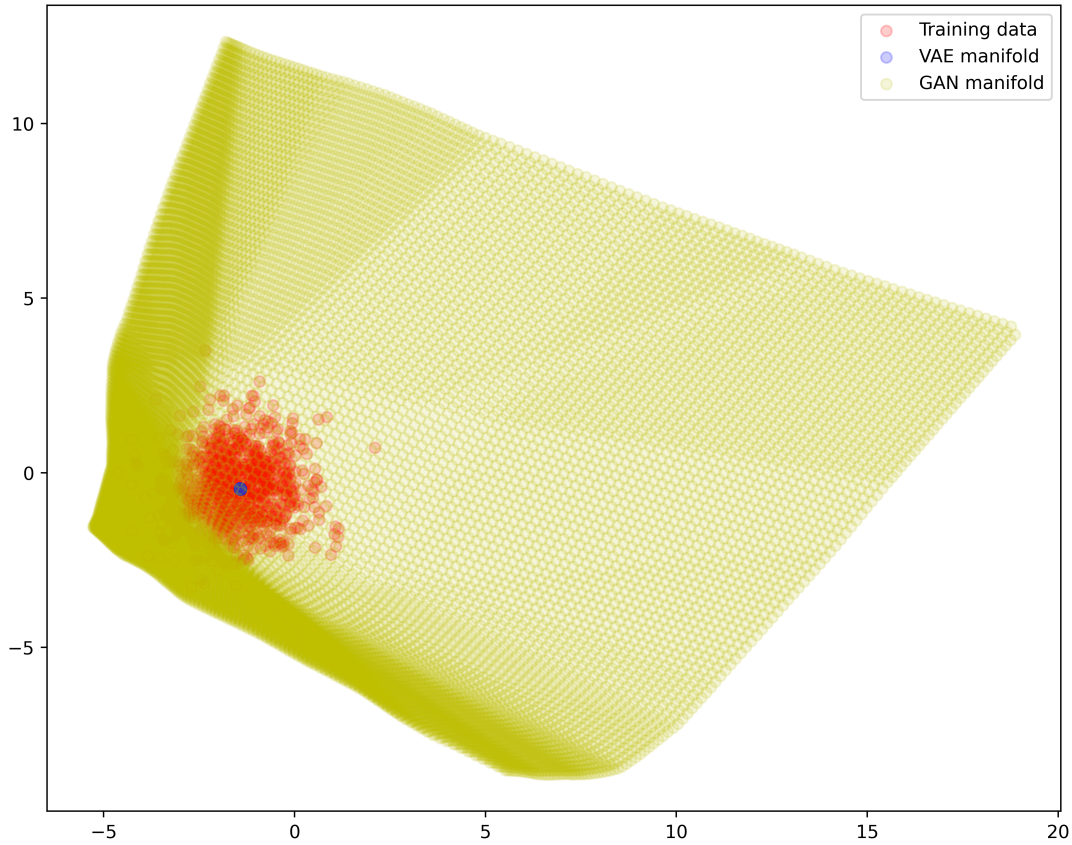


FIGURE 4.10: PCA results for Frozen Lake neural controller weight space mapping the training data (red) and enumerations over the GAN (yellow) and VAE (blue) latent spaces to a 2 dimensional space for visualisation. The latent space enumerations were performed between -3 and 3 with a step size of 0.05.

a code size of 2 and 3 are shown in Figure B.2b to plateau at a much lower fitness than the IE with a code size of 1.

For each of the three different types of IE, 5 IEs are trained and for each of those 5 IEs 5 evolutionary runs are performed. This is quite a small sample size, so it may be the case that more reliable results could be obtained with larger sample sizes, this may in turn result in evidence that leads to slightly different hypotheses. Furthermore, the IEs used in these experiments all contain 64 hidden nodes, so the results could vary as the architecture of the IE varies.

We do not observe that one type of generative model is dominant across all three domains tested. We can however say that the AE is never the *best* generative model to use to derive an IE, either the VAE or the GAN perform better (dependant on the domain).

Finally, we sought more information regarding the poor performance of the VAE in Frozen Lake (Figure 4.6). To this end, we used principal component analysis (PCA) to reduce the dimensionality of the training data for FL, and enumerations over the latent spaces of both the high performing GAN and the low performing VAE, Figure 4.10 shows the results. It is shown that the generator of the GAN produces network weights that spread over the entirety of the training data, whereas the decoder of the VAE produces weights in a very small section of the space and does not cover the training data at all. It is thus not surprising that no solutions are produced by the VAE derived IE because all values in its latent space are mapped to

a part of the weight space that has not been shown to produce any solutions on the source domains.

A number of pathologies have been shown to exist within the training procedure of VAEs, one of them being a poor approximation of the data distribution, which is what we observe in Figure 4.10. In [154], this is attributed to the training objective of the VAE which consists of both minimising the reconstruction loss and the encouragement that the latent variable, z, be distributed according to a unit gaussian distribution. It is shown in [154] that one of the conditions under which this degeneration occurs is when the posterior over z, modelled by the encoder, is difficult to approximate as a gaussian distribution for many of the training points. This could be the reason we see such a poor approximation of the training data by the VAE as shown in Figure 4.10. Potential remedies suggested by [154] are using more complex distributions over the latent variables of the VAE, or increasing the size of the latent space. However, results shown in Figure B.1 suggest that a code size of 3 is still not high enough to resolve this issue. It is however interesting to note that this degeneration was not an issue for the VAE in CMC or BW suggesting that the training data in those instances was easier to model.

One may be tempted to assume that the issues with the VAE here are reminiscent of the generalisation issues attribute to autoencoders and VAEs described in Section 3.4 when the code size is less than that of the search space dimensionality. However, those generalisation issues mainly affect interpolative and extrapolative performance, whereas here the VAE is not even able to model its training data, as evidenced by Figure 4.10.

4.6 Future Work

In future work, we would like to further test the interpolation capabilities of the IE techniques used in this chapter on RL control tasks. It would be interesting to determine whether the performance gains are maintained as interpolation gets more severe (i.e. when the target domain parameters, τ, are further away from the source domain parameters). We would also like to test the extrapolation capabilities, where we would consider how well IEs perform on target domains that have parameters, τ, way *beyond* those of the source domains. Of course, we expect interpolation and extrapolation performance to differ depending on the particular domain and fitness function, however, calculating an average over commonly used benchmarks might inform us as to when and where these techniques begin to degrade.

Other extensions include a more extensive hyperparameters sweep where code sizes above 2 are considered and compared to smaller code alternatives. Other hyperparameters such as hidden layer size and activation functions used in the decoder or generator are bound to affect performance. It would be useful to correlate hyperparameters to performance on particular fitness functions sets. Also, increasing the size of the hidden layers would in turn test performance on networks with a larger number of weights than those used in this chapters' experiments - this would demonstrate the scaling capabilities of these learnt IEs.

It would also be informative to determine how many training points are need in order to train a well performing IE. If optimisation on the source domains is expensive, which would be a fair assumption on many tasks, only a small number of training points might be able to be generated using a direct encoding; determining whether the subsequently trained IE would interpolate, extrapolate, and generate

well performing solutions under these constraints is of great importance. We performed some initial experimentation that reduced the training set size, this did not decrease performance suggesting promising results.

In this work we only used 3 types of generative model, however, there are many other different types and variations within each type that could be explored. We hypothesise that a denoising autoencoder would have a similar regularising effect on the decoder, much like the decoder of the VAE, which could result in better performance. We also believe using conditional GANs (as in [57] and [58]) would result in a much more precise search manifold outputted by the generator, this could in turn increase search speed. These are just two of many different types of generative model that could be explored.

It would be exciting to extend these techniques beyond toy benchmark problems to real world control problems that are currently typically solved by RL. This could be applied to any problem in which the environment changes slightly but not so much that a completely different controller is needed. Examples include multi-agent interactions in which an adversary or teammate change their policy, control of robotic systems in a changing environment, financial trading strategies where external environmental factors affect how well a policy performs, or the control of industrial processes such as the control of a chemical reaction under some particular temperature.

Despite the fact that extending these techniques to larger control networks would be interesting, it is unclear whether the speed advantages of an EA with an generative model derived IE would be greater than using gradient-based techniques. Gradient descent is hypothesised to work well in larger networks because the probability of finding poor local minimum decreases quickly as network size increases [21]. EAs advantages would most likely be realised on smaller networks where local minimum are more prevalent and techniques such as crossover and dynamic mutation rates could help this to be overcome. Further advantages on small networks are likely to come from the techniques introduced in this chapter, with evidence of our generative model derived IEs assisting in overcoming local minima illustrated in Section 4.4.2 and Figure 4.7. An important part of future work would therefore be to determine the network sizes for which our techniques produce advantages over gradient-based technqiues.

4.7 Conclusion

In summary, we have demonstrated the ability of three generative models, namely autoencoders, VAEs and GANs, to produce indirect encodings that successfully evolve neural controller solutions for unseen target domains in transfer learning control tasks. We have also highlighted certain settings in which these IEs significantly outperform two baseline techniques with respect to speed of search. A comparison of the three different models has been carried out and analyses have been performed with respect to the poor performance of the VAE in the Frozen Lake environment. Finally, future work suggests ways in which certain failure modes could be prevented and how to demonstrate the viability of these techniques on larger real world problems.

Chapter 5

Evolving Navigational Strategies Using GRUs in NEAT

Smaller robotic platforms such as Micro-Aerial Vehicles (MAVs) have the potential to carry out tasks in indoor environments that are often too dangerous for a human to perform. Some of these tasks include: search-and-rescue after natural disasters, radioactivity monitoring, and the surveillance of safety critical infrastructure. The size and manoeuvrability of these systems allows them to access areas that would be inaccessible for larger robots. In order to perform useful tasks in these domains, the robotic systems need to be equipped with a number of specific algorithms such as: maximal coverage, collision avoidance, and navigation.

Autonomous navigation is an important task to optimise for many robotic systems operating autonomously in an unknown environment. Simultaneous Localisation and Mapping (SLAM) is the process whereby an agent constructs a map of the environment whilst also localising itself within it. Once a map of the environment has been generated, the agent can perform path planning through the map to navigate to a desired goal. The success of this navigation is dependent on the success of SLAM.

Algorithms that perform SLAM as a subroutine are plentiful and can adequately deal with static, structured and limited size environments [128]. This has been demonstrated by its success on a number of different robotic platforms including BigDog [152], Unmanned-Aerial Vehicles (UAVs) [3, 126], helicopters [142] and even autonomous vehicles [12]. Despite these successes, SLAM is still a relatively computationally expensive process which can be problematic for smaller robotic platforms with limited computational power such as Micro-Aerial Vehicles (MAV).

An alternative suite of algorithms known as *bug algorithms* aim to navigate through an environment without building an explicit representation (map) of the area, this is a lot less computationally expensive and memory intensive. Bug algorithms are designed to work on agents that have limited sensor capabilities and are typically reactive in nature, responding to local objects such as walls when the agent comes into contact with them. Bug algorithms typically operate under the assumption that an agent has a predefined goal position that it is trying to get to. Often, the agent knows the relative position (or the azimuth angle, or bearing) of the goal but is not aware of the overall structure of the environment. A number of these algorithms have been developed for simulation and simple robotic applications, however there is not one individual algorithm that performs dominantly over all environments; it is often the case that each algorithm has its own subset of environments that it performs well in.

The fact that bug algorithms are hand designed raises the question as to whether there *exists* more effective control policies for navigation that have not yet been conceived of, which are not as computationally expensive as SLAM. The recent surge of interest in using machine learning methods for optimising agent controllers has led

to success in a wide variety of domains with many breakthroughs coming from the areas of reinforcement learning (RL) and neuroevolution (NE). Both of these techniques automatically discover control policies for agents situated in an environment, however the method of optimisation differs considerably. Both methods have been shown to outperform humans at a number of different tasks with one of the most widely used benchmarks being the Atari domain [95, 104, 138]. This suggests that human level skill or policies of an equivalent skill are by no means always a global optima in the solution space and it is often the case that an undiscovered, better performing policy exists waiting to be located.

The focus of this chapter is to automatically discover control policies for navigation that outperform hand designed algorithms, such as bug algorithms, whilst, at the same time, do not have the same vast computational and memory overheads of algorithms such as SLAM. We use I-Bug [140] as the baseline bug algorithm due to its lack of reliance on a global coordinate system (unlike most other bug algorithms), and only requiring sensors similar to those already implemented on real robots (explained more thoroughly in Section 5.1.1).

For the automatic discovery of control policies, we again turn to evolutionary algorithms for assistance, in particular, the neuroevolutionary algorithm Neuroevolution of Augmenting Topologies (NEAT) [137] described in Section 2.4.1. NEAT evolves the architecture of a network as well as the weights, often resulting in much smaller networks than those in which the architecture has been hand designed. This can significantly reduce the computational power and memory load, which is especially advantageous for smaller robotic systems.

Due to the fact that many environments that we test on consist of corridors with dead ends, we hypothesised that the inclusion of long term memory cells into NEAT would result in greater performance. To this end, we extended NEAT into NEAT-GRU, which has the ability to mutate gated recurrent unit (GRU) cells [19] into the networks, thereby enhancing its memorisation capabilities. We also further tested the capabilities of NEAT-GRU on a more complicated task, one in which the bearing angle to the target is not provided to the agent. It is often the case that robots do not have sensors with the ability to measure the bearing to a target, only those that measure distance [13]. This task requires the agent to accumulate long term distance to the target readings, as well as monitoring in which direction the agent is travelling.

In this chapter, we aim to answer the following research questions:

1. Can NEAT repeatedly generate controllers that outperform a specific bug algorithm know as I-Bug, thereby demonstrating that hand designed bug algorithms can be sub optimal?

2. Does the inclusion of long term memory units, specifically GRUs, into NEAT lead to increased navigational performances?

3. Can controllers be evolved for a much harder navigational task in which the bearing to the target is not provided as input to the controller?

Contributions

1. We empirically demonstrate that NEAT can evolve controllers that outperform I-Bug, thereby proving that I-Bug is not a globally optimal policy on a set of randomly generated test environments.

2. We introduce NEAT-GRU, an extension to the neuroevolutionary technique NEAT. Within NEAT-GRU, GRUs can be mutated into NEAT networks just like hidden nodes and their parameters are optimised via mutation operators.
3. We show empirically that NEAT-GRU can also produce control policies that outperform I-Bug on a large set of randomly generated test environments.
4. We provide evidence which suggests NEAT-GRU is superior than NEAT at producing solutions for these navigation domains thereby inferring that long term memory units provide additional assistance to the control networks.
5. We show that NEAT-GRU can produce solutions that solve a much harder navigation task in which bearing information about the target is *not* provided to the controller. NEAT was unable to produce any solutions for this task, suggesting the need for more complex memory structures when evolving control policies for robotic systems that do not have access to bearing information.
6. We design a fitness function for both the bearing and no bearing environments that encourages NEAT and NEAT-GRU solutions to be *repeatedly* found.

5.1 Related Work

The most simple maze navigational algorithm is wall following. By continuously aligning to the wall on either the left or right of the agent, an exit is guaranteed to be found as long as the maze has no disjoint walls or loops, this is known as a simply connected maze. Apart from the long run-time of this algorithm, it is also unsuitable for indoor navigation as indoor environments are often not simply connected - they contain loops or disjoint sections which can cause wall following to get stuck in an infinite loop.

5.1.1 Bug Algorithms

Bug algorithms do not require simply connected environments and can deal with unknown obstacles or arbitrary shapes. Lumelsky and Stepanov [82] pioneered these algorithms by introducing three: Com, Bug1 and Bug2. The simplest of the three, Com (Figure 5.1), is carried out as follows:

1. Move along a straight line towards the target until one of the following occurs:
   (a) The target is reached.
   (b) An obstacle is met. Follow the obstacle boundary in the prespecified local direction (i.e. left). Go to step 2.
2. Leave the obstacle boundary at a point z if the agent can move along a direct line towards the target. Go to step 1.

Although Com works successfully in a number of environments, there are a number of cases in which the agent can get stuck in an infinite loop. Bug1 and Bug2 are more complex versions of Com which aim to overcome some of its issues such as infinite loops and long path lengths.

A number of other bug algorithms have been developed [59, 60, 81, 83, 86, 124, 125, 153] each with their own advantages and disadvantages. Most note worthy for the application of these algorithms to robotics is I-Bug [140]. In I-Bug, the target is

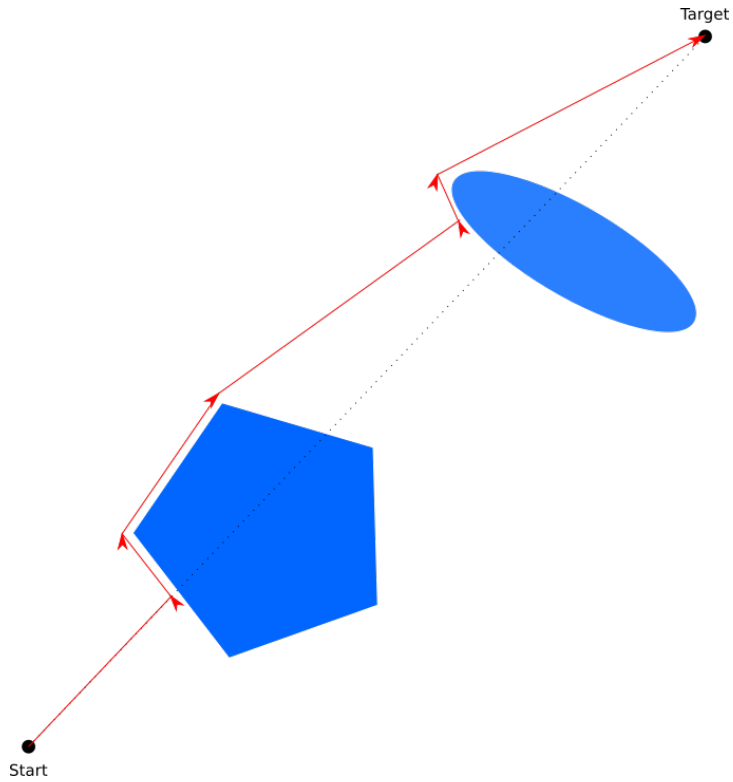


FIGURE 5.1: The 'Com' bug algorithm. The agent moves along a straight line towards the target until an obstacle is met, it will then follow the obstacle until it can continue on its path to the target.

assumed to have a wireless intensity beacon which continuously provides the distance to the target. It is also assumed that the agent can detect when it is horizontally aligned with the target and has some form of short range proximity sensors that can detect walls. These constraints are most appropriate for use on MAVs due to the availability of similar physical sensors that can determine these values. Previous work has demonstrated collision avoidance on MAVs via use of Ultra-Wideband Frequency chips which are able to communicate intra-drone distances to one another [13]. There is also no requirement for I-Bug to have a global coordinate system, whereas this is a requirement for most other bug algorithms but can often be problematic to implement in real robotic scenarios.

In the following description of I-Bug, 'intensity' refers to the intensity of a wireless signal with respect to the target, it is assumed to be a maximum of 1 when the agent is at the target. I-Bug consists of 3 possible actions or movement primitives:

- u_{fwd} : The robot goes straight forward in the direction it is facing, stopping only if: 1) it contacts an obstacle, 2) hits the target, 3) detects a local maximum of intensity along its line of motion.
- u_{ori} : The robot rotates counterclockwise, stopping only when it is aligned with the target.
- u_{fol} : The robot travels around an obstacle boundary counterclockwise, maintaining contact to its left at all times, stopping only when it reaches a local maximum intensity.

With this combination of sensors and actions, I-Bug is able to carry out Algorithm 1. In this algorithm $h_i(x)$ refers to the intensity of the wireless signal at the

Algorithm 1: I-Bug

```
while not at target do
    i_L ← h_i(x);
    Apply u_ori;
    Apply u_fwd;
    if h_i(x) = 1 then
        at_target ← true;
        terminate;
    end
    if i_L ≠ h_i(x) then
        i_H ← h_i(x);
    end
    do
        u_fol;
    while h_i(x) ≤ i_H;
end
```

agents' current state, x. This algorithm stores two values throughout its execution: i_L and i_H. The intensity i_H is the intensity recorded when the agent contacts an obstacle following the termination of u_{fwd} and i_L is the value recorded just prior to the execution of u_{fwd}. Conceptually, they are the intensities at which the agent hits and leaves objects. i_L is used to determine whether the agent has moved following the u_{fwd} action and i_H is used to determine when to terminate wall following and continue to move towards the target. Wall following will only terminate if a local maximum of intensity is reached (this is part of the movement primitive) and then conditioned on whether the current intensity is greater than i_H - in other words, it will only stop wall following when it is approaching the target and has now begun to move away from it (local maximum) then conditioned on whether it is closer to the target than it was when it first contacted the obstacle.

Further analysis of the I-Bug algorithm and other bug algorithms highlights that certain variables - in the case of I-Bug: i_H - must be stored over long time periods at certain times. Therefore it is very possible that controllers optimised to carry out policies of similar or greater performance than I-Bug would require mechanisms that support the ability to store information over long time periods. This was one of the main motivations for extending NEAT to include GRUs.

5.1.2 Evolutionary Techniques

A comparative study of generalised maze solvers explored the performance of controllers evolved via NEAT using objective based search, novelty search (NS) and a hybrid of the two [127]. NS is an evolutionary algorithm that searches the space of behaviours rather than trying to optimise an explicit objective. It does so by assigning a behaviour characteristic (BC) to each individual and then gives a reward based upon how different the BC of the individual is with respect to an archive of previously novel individuals. This encourages exploration into novel behavioural spaces which can help mitigate deception - a problem whereby explicit objectives do not illuminate a path to the global optima. The aim of [127] was to evolve simulated robot controllers to solve unseen mazes as oppose to learning a policy to solve the same maze. It was found that both NS and the hybrid approach solved significantly

more mazes than the objective based approach and were able to generalise to larger and more difficult mazes. The advantages of NS over objective search are echoed in the earlier work on NS [76–78] and also in quality diversity algorithms (a hybrid of NS and objective based methods) [105] in which maze domains are a key benchmark task.

A common benchmark task used to test a neuroevolutionary algorithms' memory capability is the T-Maze domain. This domain has a number of forms: one being a discrete state and action space and the other being of a more continuous nature. There also exists the double T-Maze domain that contains more reward locations than the original maze. The main requirement for success in all these tasks is the ability to remember the location of a large reward over a significant number of time steps. It is often the case that standard recurrent connections struggle in this domain as they are unable to deal with long term dependencies.

A discrete version of the T-maze and double T-maze domains were solved by using neuromodulated plasticity whereby the synaptic weights of normal neural network connections were modified online via hebbian rules and modulatory neurons [132]. It was shown that the networks that were evolved using fixed connections performed significantly worse on both domains whereas networks that were evolved with modulatory neurons outperformed both fixed weight and hebbian architectures in the double T-maze domain achieving a maximum score. Other work [113, 116] shows the advantages of NS in evolving similar neuromodulatory networks on the discrete version of the T-maze domain and [113] argues the reason is that evolving memory is highly deceptive and proposes a behavioural diversity technique similar to NS that achieves similar performance gains. An alternative way to encourage individual neurons to exhibit specific memory functions in the T-maze domain is by evolving networks and neurons in two separate populations, which helps due to neurons of different sub functions being prevented from mating [43].

Indirect encoded versions of NEAT have also been tested on these T-maze domains via the Adaptive HyperNEAT [115] and Adaptive ES-HyperNEAT [112] algorithms. The main idea behind the original HyperNEAT algorithm [135] is that the weights of the synaptic connections are determined by querying a Compositional Pattern Producing Network (CPPN) with the 2 dimensional coordinates of the connection being queried. Given that the connection strength is a function of its position, the CPPN is topologically aware and also has the ability to produce repeating motifs or symmetries similar to the human brain. Extending HyperNEAT to include connections that are modified online [112, 115] generates networks that can solve the continuous T-maze domain by instilling them with memory.

The Evolvable Neural Turing Machine (ENTM) is an algorithmically simpler version of the original Neural Turing Machine (NTM) in that it can be trained using evolutionary operators and is not required to be differentiable [48, 84, 85]. It has been shown in [48] that an ENTM can be trained to solve the continuous T-maze domain and the continuous double T-maze domain - a task that had not yet been solved by any other algorithm so far. Also worth of note is [62] in which a GRU with a memory block is introduced. In this architecture, each GRU has an associated memory block which it can explicitly read and write to - similar to the ENTM - with the idea being that the memory block is 'shielded' from irrelevant information. Even though this work does not technically conduct experiments with an agent situated in a T-maze, one of the experiments carried out is a sequence recall task which is analogous to the discrete T-maze experiment. It is shown that this new GRU memory block architecture significantly outperforms previous NEAT architectures with long term memory cells.

Another interesting solution to the discrete versions of the T-maze and double T-maze tasks is by evolution of Continuous Time Recurrent Neural Networks (CTRNNs) [9]. The state of the neurons in a CTRNN are described by a set of differential equations with respect to time. The idea being that the weights of the synapses are kept constant throughout the run but the internal network dynamics facilitates long term memory via its dynamic neuron potentials.

Although not benchmarked on the T-maze domains, minimal criteria evolutionary techniques are effective in solving other maze domains [10, 79]. These techniques work by allowing all solutions that meet some minimal criteria into the reproductive gene pool, which helps to greatly improve the diversity of the population. Interestingly, in [10], minimal criteria methods are used to coevolve both the agents and mazes, leading to an increasingly complex pool of environments as well as solutions to these environments.

5.1.3 Reinforcement Learning

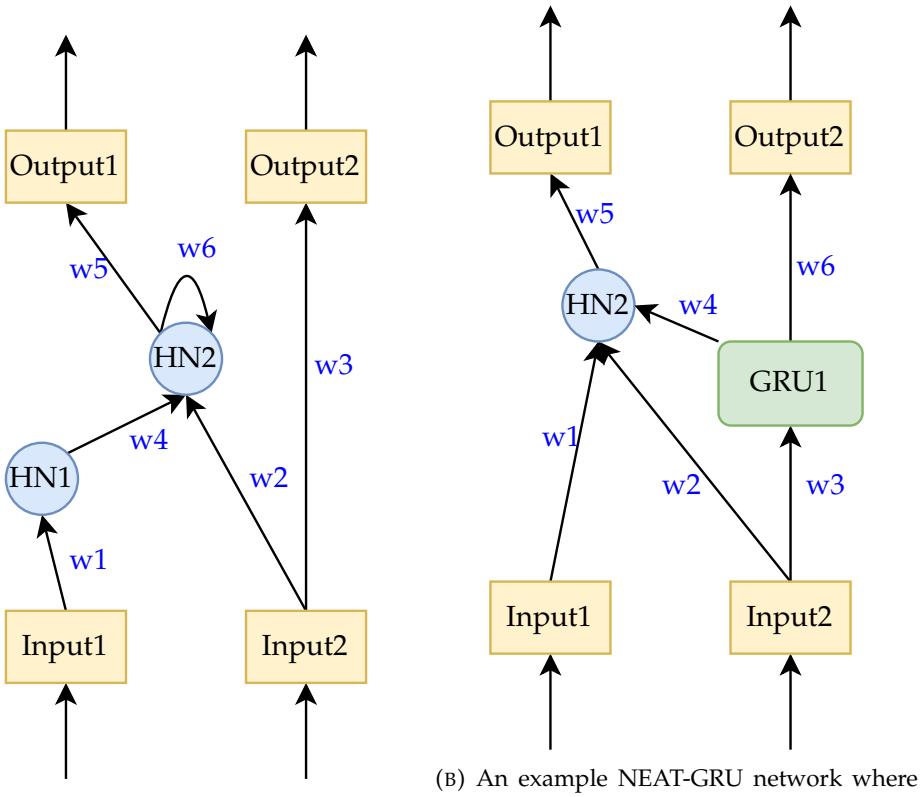
Autonomous maze solving has been extensively explored within the RL community with the most recent work being able to learn navigation policies from raw visual input [55, 61, 71, 92, 94, 99, 139, 141, 155]. Many of these algorithms can learn to navigate in complicated mazes with the assistance of Long Short Term Memory units (LSTM) [4, 55, 92, 94] which allows the agents to store relevant information for a long period of time. The advantages of using LSTMs over feedforward or pure recurrent architectures in navigation domains has been repeatedly shown [4, 92, 94]. Despite there being a small number of examples of RL algorithms (mainly policy gradient variants) approaching navigation using a continuous action space [102], most of the current approaches use a discrete action space.

5.1.4 Novelty of our Work

T-maze domains are very effective at evaluating the abilities of evolved solutions at keeping track of long term dependencies, however they are not aimed at evolving generalised maze solving agents. Agents or robots operating in a lifelike domain will come across many types of navigational challenges that were not part of their training environment. We are more interested in agents that can evolve generalised behaviours and to this end, training on a T-maze domain would not instill our agents with the behavioural requirements of interest to us. However, the previous work on the T-Maze domains does highlight to us the importance of using memory components capable of retaining long term information in maze domains and motivated the use of specific long term memory units in our own work.

The work most similar to ours and the only work that addresses generalised maze solving is [127]. However, the comparison is based upon one metric: the number of mazes solved. In our work, we additionally consider the distance it takes each agent to solve the maze, we believe this to be an equally important metric to consider. Long term memory components are not considered in [127] whereas we augment our version of NEAT to include GRUs. Furthermore, the techniques studied in [127] are not compared against hand designed navigational algorithms whereas we benchmark against I-Bug.

Our domain has a fully continuous state and action space, whereas the action space in the continuous T-maze domain in [48] consists of 3 discrete action choices which arguably makes for easier control compared to our continuous wheel speed range.



(A) An example NEAT network with a recurrent connection on the second hidden node.

(B) An example NEAT-GRU network where a GRU cell has been mutated in between the second input and the second output.

FIGURE 5.2: A comparison of an example NEAT and NEAT-GRU network. Inputs and outputs to the network are shown as yellow rectangles. Hidden nodes and GRU nodes are highlighted in blue and green respectively. Blue text illustrates network weights on connections.

Finally, in a lot of the work on NS and in the work on generalised maze solving [127], the agent has four pie-slice sensors that informs the agent of the direction towards the goal. Similarly, in some of the T-maze domains the agent is given some signal at the beginning of the run that informs it of the direction in which to turn in order to find the reward. We evaluate our NEAT-GRU on a task in which the agent does not have access to any information regarding the direction of the goal and therefore has to determine this based on distance information alone.

5.2 NEAT-GRU

In this work we modify NEAT to include Gated Recurrent Units (GRUs) in a new system called NEAT-GRU. GRUs were chosen as oppose to LSTMs due to a similar performance in other domains despite having to optimise a smaller number of parameters [22]. NEAT-GRU is similar to the NEAT-LSTM introduced in [107], however there are a number of significant differences.

The implementation of NEAT-LSTM in [107] performs an unsupervised training procedure on the LSTM blocks before the main evolutionary process begins. It does this in order to maximise the entropy between adjacent neurons in a layer such that they each memorise different features of the domain. This enables the freezing of the parameters inside the LSTM blocks during the subsequent evolutionary process.

This unsupervised procedure is time consuming and requires additional data sampling which can be difficult to obtain depending on the domain. One could bypass the unsupervised training procedure of [107] altogether and use LSTMs instead of GRUs, however, as shown in subsequent work in Section 5.6, using GRUs in NEAT results in slightly faster convergence and better generalisation ability.

The GRU cells in our NEAT-GRU are inserted into the networks in the same way as any other hidden nodes, they are mutated in with some probability. Figure 5.2 shows a comparison between NEAT and NEAT-GRU where a GRU cell has been mutated into the network. NEAT-GRU has all the same capabilities as NEAT and is therefore still able to mutate in hidden nodes with or without recurrent connections. The parameters or weights of the GRUs are modified in the same way as the other NEAT weights, via a perturbation of the original weight where the perturbation is drawn from a uniform distribution between the negative and positive of a mutation power variable. Furthermore, just as in NEAT, there is occasionally a severe mutation in which the perturbation value completely replaces the weight as oppose to just being appended to it. For simplicity, crossover is not used in NEAT-GRU.

5.3 Experimental Setup

In the following section we describe the setup of 3 experiments:

1. I-Bug on the generalised maze solving task with bearing.
2. Evolution on the generalised maze solving task with bearing.
3. Evolution on the no bearing task with a simplified environment.

I-Bug is not tested on the no bearing experiment due to the fact that it cannot operate without a bearing sensor.

The robotic simulator ARGoS [103] is used in this work for both the baseline I-Bug experiments and for the training and testing of the evolved solutions. ARGoS has been designed to be as accurate of a representation of the real world as possible providing detailed models of robots commonly used in research labs. In this work we use the provided Foot-Bot model [30] due to the fact that it is equipped with 24 local laser sensors, a range sensor, and a bearing sensor - these are the same sensors that are required for the I-Bug algorithm.

5.3.1 I-Bug

I-Bug was evaluated on a simulated Foot-Bot on 209 randomly generated test environments with a constant size of 14m x 14m. The same 209 environments were used as the test set throughout all the experiments described in Section 5.3. These environments were the same test set that was originally used in [89]: a comparative survey of bug algorithms that similarly uses ARGoS and the same Foot-Bot model as our work. The maze generation algorithm used in [89] leads to a wide variety of environments that closely resemble real life indoor office or living environments with corridor and room like structures. An example of one of these test environments is shown in Figure 5.3.

The following simulation configuration is the same as in [89]. For the I-Bug evaluations the positions of the 24 proximity sensors on the Foot-Bot were modified in the following way. 20 of the sensors were positioned in a wedge at the front of the robot to simulate a depth sensor or stereo camera for obstacle detection. There are

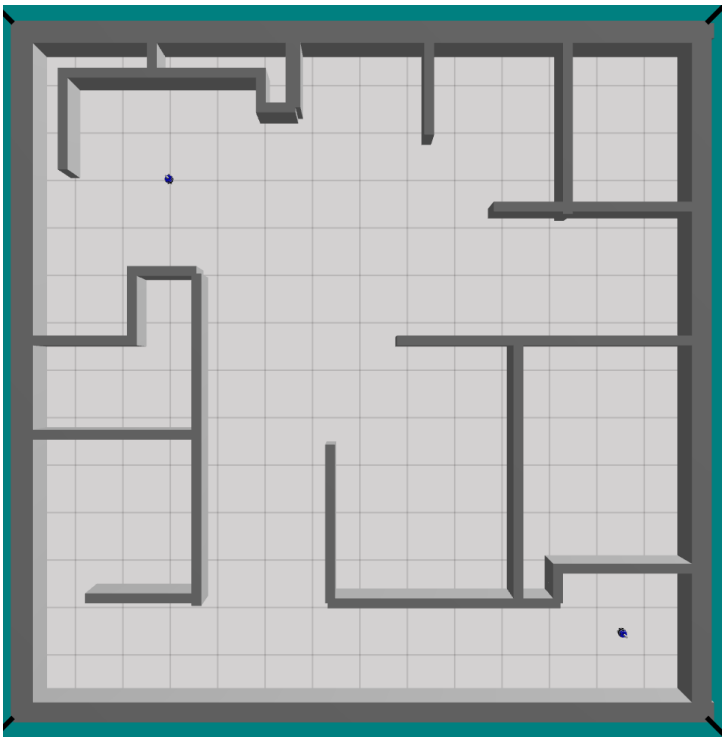


FIGURE 5.3: An example of one of the randomly generated environments used in the test set. The environment has a number of deceptive rooms and corridor structures. One of the robots is the target and is motionless throughout the run whereas the other robot contains the navigation algorithm and aims to find the other robot.

also 2 sensors at 90 degrees to the front of the robot and 1 sensor directly behind the robot. The size of the sensors was also increased to two meters. This was modified due to the fact that a hand designed wall following behaviour was more robust with a higher density of sensors at the front of the robot. A wall following behaviour based upon this sensor structure was used as the u_{fol} motion primitive described in Section 5.1.1. The rest of the I-Bug algorithm was implemented exactly as in Algorithm 1.

The performance of I-Bug was then tested on the 209 test environments. The agent had 300 simulated seconds to navigate through each environment - this time limit seemed sufficient to allow an agent to navigate to a number of dead ends, realise its mistake, and subsequently proceed down an alternative route to the target. The success percentage of I-Bug was recorded as the number of mazes in which the target was found as a percentage of the total number of mazes in the test set. Furthermore, the agents' trajectory lengths were recorded and normalised by divided through by the A* path length, which represents the shortest possible path through the maze. This A* length is calculated using a grid connectivity graph approach over a grid of size 140 × 140 representing the environment. This grid resolution results in a sufficiently accurate path length.

5.3.2 Evolutionary Setup

Bearing Experiments

In order to encourage a generalised maze solving behaviour the environments used in training are randomly generated according to the same parameters that generated

the test set. All genomes in the population were evaluated on the same set of 10 mazes, however this set of 10 mazes was randomly generated at the beginning of each new generation. The evolutionary process was ran for 1000 generations with a population size of 150. Each agent had 300 simulated seconds to find the target. Every 25 generations, the 3 best genomes from the current generation, the best 2 genomes from previous generation and the best genome from two generations ago were tested on the test set and their performance was recorded. This genome test procedure was chosen based upon the idea that the highest performing genomes on the training set will most likely (but not always) have the highest performance on the test set. Also, genomes from previous generations were chosen due to the fact that the procedurally generated training set of the current generation might have been a set of environments that, by chance, does not give an accurate representation of the test set. Limited computational resources meant that we could not test every genome produced by the evolutionary process. 20 runs were performed using both NEAT and NEAT-GRU.

Given that the evolutionary process could produce wall following behaviour within the first few generations meant that we could dispose of the dense sensor wedge and 2m long proximity sensors used in the I-Bug experiments. Instead of 24 proximity sensors, only 12 were used and they are situated equally spaced around the robot. Also, the size of the sensors was reduced to 0.2m. Using less sensors results in smaller networks in the first generation by reducing the number of network inputs. The inputs to the networks were the 12 proximity sensors, 1 range sensor giving the distance to the target and 2 bearing sensors: the clockwise and counter-clockwise relative bearing. The network outputs were the speeds of the left and right wheels of the Foot-Bot.

A fitness function was designed to maximise two different metrics: the number of mazes solved and the trajectory per A* length. Initially a weighted sum of both values was used however the evolutionary process only generated policies that maximised one of the two metrics despite any configuration of weightings assigned to each metric. The agents would either evolve a wall following behaviour (maximising the number of mazes solved but doing so with a relatively large path length) or evolve a greedy behaviour in which it headed straight for the target (reducing path length but not arriving at many targets, often due to the agent getting stuck in rooms).

However, it was noticed that a very small number of policies in the first generation were capable of finding at least one of the 10 targets of the 10 randomly generated training mazes by sheer chance. This observation resulted in a sparser fitness function, f_1, which we subsequently used for all the bearing experiments:

$$f_1 = \begin{cases} \frac{1}{l^{0.5}}, & \text{if maze solved,} \\ 0, & \text{otherwise,} \end{cases} \tag{5.1}$$

where l is the trajectory length per A* length taken to find the target.

This fitness function resulted in the best trade off between the two metrics due to the fact that the agent must *first* find the target, and only once it has, is rewarded with the inverse of the normalised length of the path taken to get there. This prevented the incentivisation of behaviours that headed straight towards the target regardless of whether or not the target was arrived at (resulting in a short path length) because these behaviours would now get a score of 0. An exponent of 0.5 was chosen in order to 'flatten' the function slightly such that a genome was not rewarded too greatly for doing well in one particular environment.

If the agent crashed into a wall the fitness score was divided by 10 in order to deter the agents from crashing, resulting in f_2:

$$f_2 = \begin{cases} \frac{f_1}{10}, & \text{if crash,} \\ f_1, & \text{otherwise.} \end{cases} \tag{5.2}$$

The median of the f_2 scores for each of the 10 randomly generated training environments was used as the final fitness value for each individual genome.

Initial experiments revealed that tuning the default NEAT hyperparameters resulted in better performance. Too high of an add node mutation rate resulted in significant network bloat, we therefore kept these values low: 0.005 and 0.003 for the hidden node and GRU node add mutation rates respectively. Furthermore, we increased the survival threshold from the default values of 0.2-0.4 to 0.55 in order to assist performance by reducing selection pressure. The weight mutation power was set to 1.5 for both normal weights and the GRU node weights which was slightly less than the default values of 1.8-2.5. In order to provide a fair comparison, crossover for the NEAT experiments was turned off to match the lack of crossover used in NEAT-GRU. A full list of hyperparameters used for the bearing experiments for both the NEAT and NEAT-GRU experiments can be found in tables C.1 and C.2 respectively in Appendix C.1.

No Bearing Experiments

A much harder task was designed to further test the cognitive abilities of NEAT-GRU. For this task, all sensors are removed from the Foot-Bot apart from the distance sensor. This sensor configuration removes the ability of the agent to know its relative orientation towards the target, therefore locating the target must be done by accumulating distance measurements and performing significant cognition in order to ascertain the direction in which to travel. This sensor configuration commonly occurs in robotics where a distance sensor - such as an Ultra-Wideband Frequency chip - provides only distance information with respect to other chips, as in the work of [13].

Given that this task is much more difficult, the environment used to train and test the evolved solutions was simplified. An environment of size 10m × 10m was used that contained no obstacles apart from a wall surrounding its perimeter. This environmental configuration was complex enough to demonstrate a considerable performance difference between NEAT and NEAT-GRU. All of the proximity sensors were disabled for this task due to the fact that the agents could learn a perimeter wall following behaviour that led them close to the goal without having to learn a more complex policy.

The agent began in one corner of the 10m × 10m environment and the target in the opposing corner. Each agent was evaluated on the same environment 5 times however the starting orientation of the agent is different for each evaluation. This is to prevent overfitting to a set orientation in which an agent 'memorises' a particular path from a particular starting position to the target. Each agent is given 80 simulated seconds to find the target, this is far less than the bearing sensor experiments due to the fact that the environment is smaller and there are no obstacles to navigate around. Like the bearing experiments the network outputs are the left and right wheel speeds of the Foot-Bot whereas the only network input in these experiments is the target distance.

The fitness function f_1 (Equation 5.1) was not successful in this experiment due to the fact that none of the genomes in the first generation could locate the target. This resulted in a fitness of 0 for every individual in the population, thereby providing no training signal for the evolutionary algorithm. In order to provide a significant reward gradient, f_3 was used:

$$f_3 = (L - d)^3 \tag{5.3}$$

where L is the diagonal length of the arena (the maximum distance the agent can be from the target) and d is the final distance between the agent and the target at the end of the run. It was found that cubing the fitness lead to a larger rate of success by encouraging a greater reward difference between individuals. Like the bearing experiments, if an agent crashed, the fitness was divided by 10, resulting in f_4:

$$f_4 = \begin{cases} \frac{f_3}{10}, & \text{if crash.} \\ f_3, & \text{otherwise.} \end{cases} \tag{5.4}$$

As before, the final fitness for each individual was taken to be the median of its f_4 values from the 5 training environments.

NEAT and NEAT-GRU were ran 10 times each for 5000 generations per run and with a population size of 150. A slightly higher mutation rate for adding nodes and GRU nodes was used compared to the bearing experiments: 0.006 and 0.006 respectively. Furthermore, a weight mutation power of 0.5 was used in order to reduce significant weight changes within the GRUs. As in the bearing experiments, crossover is disabled for both NEAT and NEAT-GRU. A full list of the hyperparameters used for the NEAT and NEAT-GRU experiments are shown in tables C.3 and C.4 in Appendix C.1.

The code for all the experiments described in Section 5.3.2 is available at `https://github.com/jamesbut/neat_gru_bug`.

5.4 Results

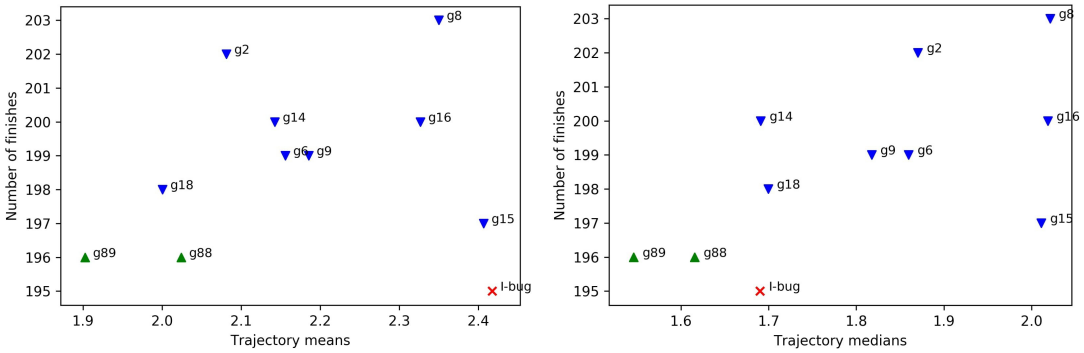
5.4.1 I-Bug

On the 209 test environments I-Bug achieved a success percentage of 93.3% meaning that it managed to find the target in the maze 195/209 times. The mean and median of the trajectory lengths per A* lengths over all the environments were 2.4174 and 1.6900 respectively. I-Bug was not tested on the no bearing experiments because without the relative angle towards the target I-Bug will not work.

5.4.2 Evolutionary Results

Bearing Experiments

Out of 20 evolutionary runs for NEAT-GRU, 10 of the runs produced genomes (out of those evaluated on the test set, highlighted in Section 5.3.2) that outperformed I-Bug in maze success percentage and in the mean of the trajectory length per A* length values over all the test environments. 5 of the aforementioned 10 runs contained more than one genome that outperformed I-Bug. However, it was often the case that they were similar in behaviour to previous genomes in the same run so it is not worth noting the *total* number of genomes found that outperformed I-Bug; the number of



(A) All of the solutions outperform I-Bug in terms of the number of finishes and the trajectory means per A* mean metrics.

(B) I-Bug has a relatively low trajectory median compared to the evolved solutions, with only G88 and G89 outperforming it in all 3 metrics.

FIGURE 5.4: Scatter charts showing the performance metrics for I-Bug and for the 10 genomes produced by NEAT-GRU that outperformed I-Bug. A smaller trajectory length is more desirable. The 2 solutions that outperformed I-Bug in all 3 metrics are highlighted in green and the 8 solutions that outperformed I-Bug in only 2 metrics are highlighted in blue.

	2 metric winners	3 metric winners
NEAT	3/20	0/20
NEAT-GRU	**10/20**	**2/20**

TABLE 5.1: A table highlighting the number of evolutionary runs out of 20 in which at least one genome outperformed I-Bug on the 209 test environments for both NEAT and NEAT-GRU.

runs in which evolution could produce at least one outperforming solution is more significant.

Out of these 20 runs, 2 of the runs produced genomes that outperformed I-Bug in all 3 metrics: maze success percentage and the mean and median of the trajectory length per A* length values over all the environments. Despite 2/20 being a small number of successful number of runs with respect to all of three metrics, the fitness score during training is optimised according to the mean of the trajectory length per A* length as oppose to the median.

Figure 5.4 illustrates the performance metrics of the 10 genomes that beat I-Bug in at least 2 metrics. Some of the solutions produced were significantly better than I-Bug ($p < 0.0001$ based upon trajectory lengths per A* length). For example, one solution named 'G89' had a success rate of 196/209 (93.7%), a trajectory length per a star length mean of 1.9024 and a median of 1.5459. This trajectory length mean is significantly less than the equivalent I-Bug value meaning that the agent was able to get to the target in a much shorter path length despite being able to find more targets than I-Bug. There also exist solutions that have a larger success rate at the expense of having longer path lengths. For example 'G8' had a success rate of 203/209 (97.1%), a trajectory mean of 2.3499 and a median of 2.0212. It therefore locates the target on a much larger number of occasions than I-Bug, it does so with a mean trajectory length per A* length less than I-Bug, however, it does not outperform I-Bug on the normalised trajectory length median. A video showing the behaviour of these genomes is available at `https://youtu.be/8EqyeuX_lR0`

Out of the 20 evolutionary runs for NEAT, 3 of the runs produced genomes that

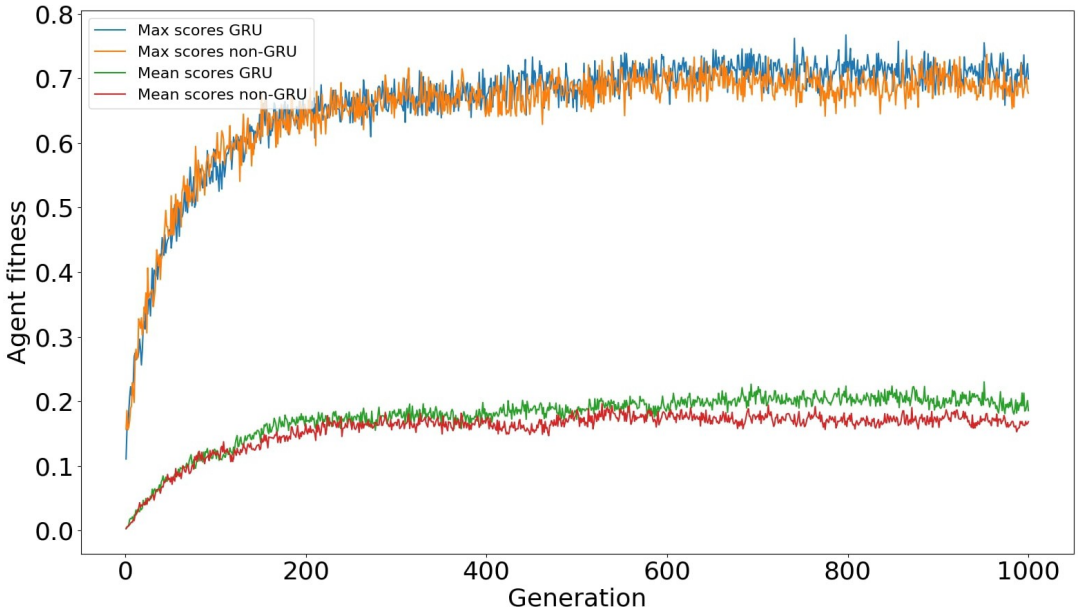


FIGURE 5.5: A graph showing the average population fitness and the max population fitness during training for both GRU and non-GRU versions of the bearing experiment. It shows the slight fitness increase attributed to the inclusion of GRUs. The results are averaged over 20 runs.

outperformed I-Bug at maze success percentage and in the mean of the trajectory length per A* length values over all the environments. 0 of the winning solutions beat I-Bug in all 3 metrics. These results are highlighted in table 5.1. Furthermore, Figure 5.5 highlights the average and maximum fitness of the whole population at each generation during training for both NEAT and NEAT-GRU. It highlights the slight advantage in performance offered by GRUs.

No Bearing Experiments

Out of the 10 evolutionary runs for NEAT-GRU, all 10 produced a solution capable of solving the task from all 5 orientations. In contrast, out of the 10 runs using NEAT, 0 of them produced solutions that could solve the task in all 5 orientations. Figure 5.6 shows the maximum fitness so far for the population during training for the no bearing task. It shows the dramatic increase in performance due to the inclusion of GRUs into the NEAT networks.

5.5 Discussion

The results in section 5.4.2 clearly highlight that I-Bug is not a globally optimal solution in the domain of generalised maze solving. It also shows that evolution and more specifically, NEAT and NEAT-GRU, have the ability to produce better control policies than those that have been previously hand designed. This continues to add to the previous body of work that demonstrates the ability of machine learning techniques to outperform human designed algorithms. Even though the solutions evolved in this work are an improvement on previous bug algorithms, there is no guarantee - and is highly unlikely - that this work has found globally optimal solutions either. It would be interesting to apply novelty search techniques to

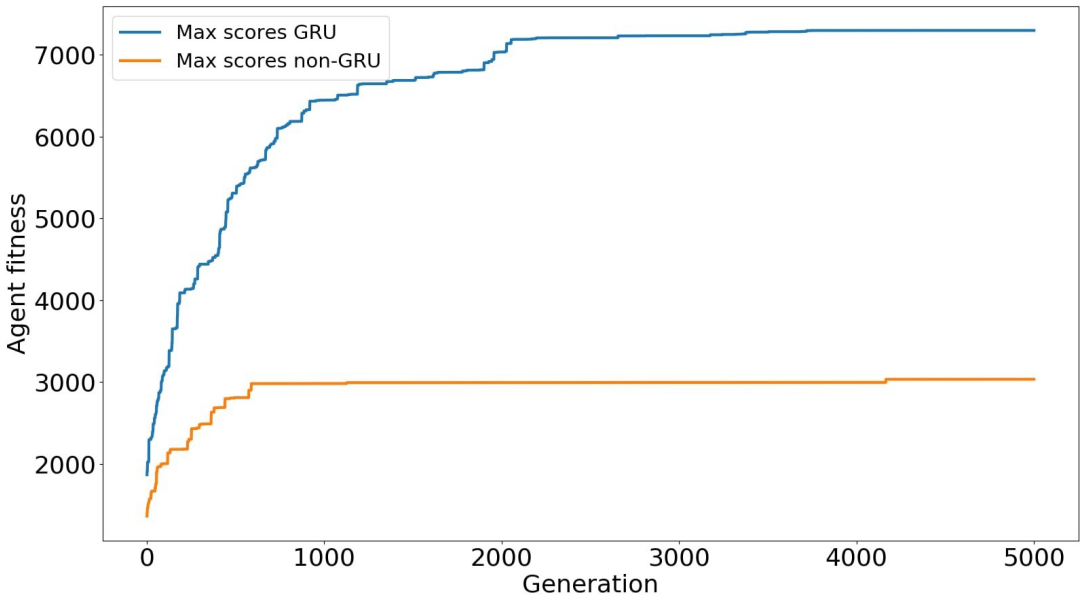


FIGURE 5.6: A graph showing the maximum fitness so far of the population during training for both GRU and non-GRU versions of the non-bearing experiment. It shows a dramatic fitness increase attributed to the inclusion of GRUs and how the non-GRU version plateaus at a score of 3000. The results were averaged over 10 runs.

the algorithms used in this work in order to attempt to illuminate additional high performing policies.

This work does highlight that the inclusion of long term memory units into NEAT leads to a better performance in the bearing experiments and is seemingly *essential* for completion of the no bearing experiments. This reinforces the growing body of evidence [4, 62, 92, 94] suggesting the inclusion of long term memory units into control networks improves performance in maze like environments. Surprisingly, however, NEAT without GRUs was able to outperform I-Bug despite networks of this type being unable to store information over long periods. It was also observed during the experiments that networks without *any* hidden units performed comparatively well, with one genome achieving a success rate of 189/209 (90.4%) and a mean path length per A* path length of 2.1. This suggests that the maze following environments used to test I-Bug and other bug algorithms, and likely many real indoor environments, do not actually require much cognition to solve.

The results of the no bearing tasks suggests the high relative difficulty of this task in relation to the bearing version. With only the distance to the target at each point in time, the agent does not know which way to turn in order to approach it unless it builds some form of target location model by accumulating distances through time. Furthermore, the agent additionally must be aware of its own actions in order to build this model because it needs to know how the distances change as a function of these actions. NEAT theoretically has the ability to do this via recurrent connections that can originate at output nodes however the inclusion of GRU nodes into this greatly helps performance by allowing appropriate information to accumulate. It is also impressive that this ability can be learned with a continuous state and action space.

The results from the no bearing tasks are important due to the fact that certain robotic systems do not have access to relative bearing information, such as [13]. A relative bearing sensor might even be available but it could greatly increase the

weight of a robotic system. This can be particularly troublesome in the case of MAVs where the payload cannot exceed some low threshold value. The results highlighted here show that it is indeed possible to learn a neural network control policy without a bearing sensor but the success is highly dependent on the presence of long term memory units.

5.6 Subsequent Work

A student at KU Leuven, Ruben Broekx, subsequently completed a masters project [14] extending the work presented in Chapter 5. The causal factors leading to the significant performance increases for NEAT-GRU in the no bearing environment are explored in greater detail. It also carries out a more extensive comparison study between NEAT-GRU and other related algorithmic baselines in order to ascertain the specific factors attributing to the ability of NEAT-GRU to successfully navigate without bearing information.

In [14] the performance of NEAT without recurrent units, NEAT with simple recurrent units, NEAT-LSTM, and NEAT-GRU are compared on the no bearing environment. As expected NEAT without recurrent units performs very poorly on this task, NEAT with recurrence performs better than its equivalent without recurrence, however, both NEAT-GRU and NEAT-LSTM achieve the highest performance, echoing the results in Section 5.4.2. The performance of NEAT-GRU and NEAT-LSTM are shown to be very similar (no statistical significance). However, it is demonstrated that NEAT-LSTM does have slightly slower convergence due to the greater number of parameters requiring optimisation. It is also shown that NEAT-GRU performs slightly better in generalisation experiments compared to NEAT-LSTM. Interestingly, a finer grained look at the internals of the NEAT-GRU solutions revealed the superfluousness of the reset gate of the GRU in this particular domain. An empirical study was carried out showing statistically comparable performance between NEAT-GRU and NEAT-GRU with a GRU cell modified such that it does not contain a reset gate [14].

5.7 Future Work

In future, we would like to extend the work on the no bearing experiments by incorporating obstacles back into larger environments and reintroducing proximity sensors. However, given the difficulty of learning to navigate without bearing information, curriculum learning techniques [5] may be required in order to solve more complex environments with obstacles. We would also like to apply NEAT-GRU to real robotic navigation tasks, such as with MAVs where the only available sensor information would be the target distance and small proximity sensors. Given a swarm of MAVs equipped with these sensors, they could learn to coordinate with respect to each other and navigate to other members of the swarm in complicated indoor environments.

It would also be interesting to analyse how one could optimise GRUs via evolutionary operators in a more efficient manner. Rawal and Miikkulainen [107] began to explore this when they introduced NEAT-LSTM by optimising a secondary objective that encourages a larger amount of information to be stored in the LSTM unit. In future work, this info-max objective could be included as part of the NEAT-GRU loss function and compared against not using this secondary loss function. One

could also explore whether meaningful crossover procedures exist that are beneficial to GRUs undergoing evolutionary optimisation. Equally, more efficient mutation strategies could exist for NEAT-GRUs. Previous work [80] has explored 'safe mutations' for neuroevolution in maze environments, it would be exciting to determine whether these techniques result in improvements to NEAT-GRU performances in both the bearing and no bearing environments.

As we have seen in Section 5.6 some parameters or modules within GRU cells can be shown to be superfluous in particular domains. This realisation was arrived at via manual investigation, however, we could easily apply the indirect encoding techniques from chapters 3 and 4 to *automatically* infer information of this sort about the parameter space. The cross pollination of indirect encoding techniques and long term memory cells is explored in more detail in Section 6.3.

5.8 Conclusion

We investigated the ability of evolution, specifically NEAT, to evolve control policies for generalised navigational environments. We proposed NEAT-GRU which is a modification of NEAT that has the ability to mutate GRUs into the NEAT networks. We compared the solutions produced by NEAT and NEAT-GRU to a particular type of bug algorithm called I-Bug which is particularly suited for use in real robotic domains. We showed that the I-Bug algorithm is not a globally optimal solution and that both NEAT and NEAT-GRU can evolve solutions that outperform it repeatedly. We introduced a harder domain in which *only* the distance to the target of the maze is given as sensor input to the network and demonstrated that NEAT-GRU can successfully evolve solutions to this task, whereas NEAT fails in every run. This demonstrates that significant cognition is required to solve this task and that long term memory units greatly assist in these types of tasks found in real robotic domains. We designed two fitness functions that encourages solutions to be found *repeatedly* on both the bearing and no bearing environments: Equations 5.2 and 5.4 respectively.

Chapter 6

Conclusion

In this chapter, we revisit the research questions highlighted in Chapter 1 and illustrate how the results of this thesis answers them. We then embed our research into the wider context and explain what avenues it opens up. We also highlight the limitations of our work and suggest potential future work.

6.1 Research Questions & Contributions

RQ1: Can we use generative models to construct indirect encodings for evolutionary algorithms to perform transfer learning for the optimisation of continuous functions?

We first set out to explore whether generative models can be used to derive IEs that would aid with transfer learning for the optimisation of continuous functions. Although previous works [40, 57, 58] had tested whether IEs derived from generative models could evolve solutions on unseen problems, the focus had not been on transfer learning. As a result, no previous work had performed quantifiable interpolative or extrapolative analyses with respect to the source and target problems. In Chapter 3, we sought more information regarding how three different types of generative model derived IEs performed when the optimisation of the target functions required interpolation and extrapolation of knowledge via the IE. We experimented on a two dimensional continuous optimisation problem where the extent of interpolation and extrapolation was clear.

We showed that for certain target problems, all three types of IE outperformed the DE baselines in terms of speed of search and accuracy of the solutions found. Therefore, we have shown that, in certain circumstances, using IEs derived from generative models results in clear advantages over alternative techniques in transfer learning for continuous optimisation. However, it was shown that the performance of the IE with respect to the DE baselines was highly dependant on the location of the target problem maxima. When the target problem maxima was very close to the starting point of the DE, the DE would locate a solution faster. As the distance between the target problem maxima and the starting point of the DE grew, the search speed advantages of the three types of IE were quickly realised.

It is exciting that we have shown that transfer learning can be performed this way on continuous functions as many modern problems involve the optimisation of a set of continuous values. Examples include, the optimisation of neural networks [44], engineering design variables [87, 122], control policies [95], and non-linear regression tasks [73]; illustrating the techniques applicability to the optimisation of continuous functions unlocks their use on many real world problems.

RQ2: Which generative model type produces an indirect encoding that improves search performance the most on continuous optimisation problems, and under what conditions?

A clear limitation of all previous works [18, 40, 57, 58, 96, 106] was that none had conducted comparative analyses between the different types of generative model used to derive IEs. It was therefore not clear what the advantages and disadvantages of each generative model were. Nor was it clear whether limitations observed by one IE type were indicative of an issue with that particular type, or caused by the particular task being considered. As a result, in Chapter 3 we compared the performance of autoencoders, VAEs, and GANs at producing IEs for the optimisation of simple continuous functions.

We showed that in certain instances GANs can mode collapse, which had the effect of producing an IE that had terrible performance. Autoencoders and VAEs did not suffer from mode collapse, although they did model the underlying distribution of source solutions of a particular family of functions poorly when the code size was less than the search space dimensionality. As a result, the autoencoder and VAE derived IEs of code size 1 performed very poorly on the extrapolative experiments compared to the GAN derived IE of code size 1, which performed the best out of all the techniques examined. We also showed that all 3 generative model derived IE types struggled to locate a solution with as high of a fitness as those found by the DE alternatives (although not by much). However, these solution accuracy issues seemed to affect the GAN derived IE to a larger degree than the other IE types. We hypothesised that that was due to the manifold folding that is more extreme in the GAN derived IEs, as a result we suggested some regularisation techniques that may alleviate this.

With regards to the question of which model type produced the best IE for some continuous optimisation problems, the answer is: it depends. It depends on whether the pernicious issues affecting each type such as mode collapse or poor generality are present. If the dimensionality of the latent code size is chosen to be less than that of the search space, a GAN is the clear winner when extrapolating to new problems. If the latent code size is the same as that of the search space, the VAE seems to outperform in extrapolative experiments. Finally, to achieve the search speed advantages of the IEs without losing accuracy of the solution found, we suggest switching back to a DE once the fitness of the best solution found begins to plateau.

Knowledge of the pathologies and pitfalls of certain generative model derived IE types can assist when choosing which algorithm to apply to future problems. It will also help to debug future issues, and the remedies suggested in Section 3.4 may help a practitioner find a solution. It is also useful to know which type of generative model derived IE is more robust to larger extrapolations; if one knows beforehand that more extreme extrapolation is required, the results in this thesis suggest that it is more sensible to use a GAN.

RQ3: Can we use generative models to construct indirect encodings for evolutionary algorithms to perform transfer learning in reinforcement learning control tasks?

We have for the first time applied the techniques explored in Chapter 3 (generative model derived IEs) to perform transfer learning in RL control tasks. We did this by showing that optimising target functions from the same family as the source functions (as in Chapter 3) is akin to performing transfer learning in RL control tasks.

This allowed us to transfer the techniques used in Chapter 3 to RL tasks with minimal modifications. We demonstrated that there was always at least one IE type that beat the baselines in terms of search speed and fitness of the solution found for three OpenAI Gym environments.

A current bottleneck in deep RL is long training times [120, 138]. By demonstrating a way to perform transfer learning to reduce search times on RL control tasks, we offer an alternative way to reduce long training times. Given that RL is used in a wide variety of areas, such as nuclear fusion [29], robotics [66], optimising cooling systems [151], and algorithm creation [33], the techniques introduced here could have a large impact on the time taken to solve many important problems.

RQ4: Does the inclusion of gated recurrent units into NEAT result in improved evolutionary search performance on generalised maze solving tasks?

Autonomous navigation is an important subroutine in a number of different tasks, such as search-and-rescue, surveillance, and radioactivity monitoring. Often these tasks are dangerous or time consuming for humans to perform, thereby creating a demand for robotic systems that can perform these tasks instead. Despite the ability of algorithms like SLAM to create explicit map representations over which path planning can be performed, these algorithms are very computationally expensive. Larger robots are often unsuitable for tasks such as search-and-rescue and surveillance where smaller, more agile robots, such as MAVs would be more successful. However, MAVs only have a small payload and therefore cannot carry computational devices capable of performing SLAM in real time. Furthermore, even if they could carry these devices, the power required to serve such intensive computation would drain valuable battery life needed to power flight. Due to this, there is a demand for navigational algorithms that are much less computationally expensive and have a smaller memory footprint than SLAM.

Alternative navigational algorithms with a reduced computation and memory footprint exist, such as Bug Algorithms. Inspired by works that use machine learning to improve upon hand designed algorithms for agent control policies [95, 104, 138], we explored whether EAs can produce computationally light navigational policies that outperform Bug Algorithms. NEAT is an appropriate choice of EA because it optimises the weights *and* the architecture of the neural network controller, often resulting in much smaller networks than those where the architecture is hand designed. These smaller networks require less memory and floating point operations to run. However, navigation can often require the memorisation of information over long term horizons so that robots do not reexplore already discovered paths. Although NEAT can remember information via its recurrent connections, these often struggle to remember information over long term horizons [53].

In Chapter 5, we explored whether the inclusion of GRUs into NEAT improved performance on generalised maze solving tasks, which can be viewed as a proxy for navigation. GRUs do have the ability to remember information over long time horizons, much like LSTMs, however they consist of less parameters than an LSTM, resulting in reduced memory and computational requirements.

Firstly, we showed that NEAT outperformed hand designed navigational algorithms. However, we were particularly interested in whether a modified version of NEAT with long term memory cells affected performance. We designed and implemented a novel algorithm, namely NEAT-GRU, that has the ability to mutate GRU

cells into NEAT networks. For generalised maze solving, NEAT-GRU improved performance over baselines and the original NEAT algorithm. We also designed a fitness function which resulted in the locating of high fitness solutions repeatedly and reliably.

RQ5: Does the inclusion of gated recurrent units into NEAT result in improved evolutionary search performance on a much harder navigational task whereby bearing information is not available?

It is often the case that robots have sensors that can provide distance information to a target but not information regarding its relative angle [13]. Therefore, it is important to have computationally cheap navigational algorithms that do not require explicit information with respect to the bearing. In Chapter 5, we explained that previous hand designed algorithms do not have that ability, and demonstrated that NEAT cannot evolve a controller that can locate a target under these conditions. To this end, we applied our newly created NEAT-GRU to this much harder navigational task. This task required the utilisation of previous distance to the goal readings in order to ascertain which direction the agent to steer towards. As a result, it was shown the NEAT-GRU greatly outperformed NEAT in this task due to its ability to remember information over much longer time periods.

6.2 Wider Impact

The overall aim of this thesis was to explore techniques to improve the search speed of evolutionary algorithms. The hope was that if we could do this, we could provide more credence to the argument that EAs should be taken seriously for continuous optimisation problems, especially for the optimisation of neural networks. Furthermore, a higher search speed reduces the amount of computational resources, and in turn the cost, needed to run EAs. Given that EAs are used in many real world applications, their improvement will have a significant impact.

In this thesis, we have improved the search speed of EAs in two distinct ways. We showed that one can use generative models to produce IEs that greatly improve search speed when performing transfer learning in both continuous function optimisation and RL control tasks. We also showed that one can greatly improve evolutionary search speed and unlock new fitness plateaus in certain navigational tasks by adapting a previous algorithm, NEAT, to include GRU cells. The evidence for EAs as a competitor to gradient based techniques for optimising neural networks has been growing [123, 138], and the present work contributes to this growing body of evidence.

Using generative models to create IEs, as we have done in Chapters 3 and 4, is actually a technique that can be combined with other black box optimisation techniques apart from EAs. The technique essentially transforms the search space prior to the main optimisation process, which does not actually dictate the algorithm used for subsequent search in the latent space. Other black box optimisation techniques such as Particle Swarm Optimisation or Simulated Annealing could be used instead of an EA. Additionally, gradient based techniques could perform the main optimisation process as long as gradients through the IE could be calculated.

Furthermore, in this thesis we highlighted the fact that transfer learning can be viewed as a particular type of optimisation procedure. The procedure consists of reusing knowledge gained from optimising source problems to optimise target problems from the same function 'family'. We showed how this can be applied to RL

control tasks. However, given that many modern machine learning techniques can be viewed as instances of continuous optimisation, the techniques considered here can be applied to a wide variety of areas.

In this thesis, we only considered IEs of the form $f : \mathbb{R}^n \to \mathbb{R}^m$. However, one can also use the techniques presented here to create IEs of the form $f : \mathbb{R}^n \to \mathbb{Z}^m$ or $f : \mathbb{R}^n \to \{0,1\}^m$. A post processing step mapping the real numbered output of the IE neural network to $\mathbb{Z}^m$ or $\{0,1\}^m$, respectively, must be performed, although, this would consist of a trivial rounding operation. In this way, the ideas in this thesis could be opened up to the field of *combinatorial* optimisation as well, such as routing, supply chain optimisation and time tabling.

We additionally posit that the work in Chapter 5 unlocks a way to perform computationally inexpensive navigation on robots with limited sensor capabilities. This in turn enables small robotic systems such as MAVs to perform important tasks such as search-and-rescue, surveillance and monitoring.

6.3 Limitations & Future Work

One of the main limitations throughout all of the experiments in this thesis is the relatively small number of evolutionary runs that were performed. For example, in Chapters 3 and 4, each experiment was run five times. This small number of runs can lead to empirical mean fitnesses that are quite far from the true mean fitnesses. If they are too far apart, it may even give the impression that one algorithm outperforms another with respect to the mean fitnesses, when in fact the opposite is true. In the experiments performed in this thesis, when an algorithm has outperformed other algorithms, it has done by a large margin. Therefore, we expect algorithm ranking issues to not have been a problem in this thesis.

There are a number of ways to combat the small number of evolutionary runs. Additional computing power is the first obvious choice. EAs scale well with the number of CPU cores, so providing more CPU cores can provide near linear speed ups. Recently, software frameworks that run EAs on the GPU have been created, one of which is called EvoTorch [143]. Not only are the EA algorithms carried out on the GPU, but vectorized OpenAI Gym environments are supported, which can also be carried out on the GPU. Future experiments would benefit from using these new EA frameworks so they can produce more runs and gather more accurate results.

This thesis aimed primarily to show a proof-of-concept with respect to the power of generative models for producing IEs for transfer learning. We have shown a number of control problems and a small set of continuous optimisation problems for which generative model derived IEs greatly outperform baselines. However, the problems tested on are only a small handful and may not be indicative of the performance of these techniques on different types of problems. Additionally, due to computational limitations, the search space dimensionality of the problems considered in this thesis never exceeded 100. In the future, it would be interesting to test whether using IEs derived from generative models scale to larger neural networks such as those with convolutional architectures or small transformers, however, we do foresee scaling issues such as those hypothesised in Section 4.5. More efficient software such as EvoTorch could assist with performance issues regarding larger search spaces.

One of the main arguments of this thesis is that by increasing the speed of EAs using our introduced techniques we make EAs more competitive with respect to gradient-based methods. A limitation of this thesis is that our techniques are never

compared directly against gradient-based techniques to give quantifiable evidence of this claim; it would be informative to perform this comparison in future work. Performing this direct comparison would give us information regarding whether our technique is preferable over gradient-based techniques, and, under which circumstances. If the results of those comparisons find situations under which our techniques outperform gradient-based techniques, this could have a disruptive impact on how certain optimisation problems are solved.

One unoptimised hyperparameter throughout all of the experiments in this thesis has been the population size used in the EAs. The population size can have a large effect on the optimisation time of the EA and the fitness of the solution found [56, 117]. Given that one of the main metrics used to compare the techniques in this thesis was the optimisation time, a more fair comparative analysis should appropriately set the population size of each technique to realise its full potential. Without an optimised population size it is unclear whether one technique underperforms compared to the others due to an unoptimised population size or because of the fundamental limitations of the algorithm. It is also observed in [56] that as the number of dimensions of a problem grows a larger population size is required to achieve the same number of successful runs in which a solution is found in a set number of generations. In our work the optimisation speed in the lower dimensional latent space of the generative models was compared against the higher dimensional space of the direct encodings using the *same* population size. The results of [56] suggest that this comparison unfairly disadvantages the direct encoding by not increasing the population size to account for the larger number of optimisable parameters. Computational resource limitations prevented the optimisation of population sizes in this thesis, however, important future work would be to determine whether the conclusions of this thesis still hold when population sizes are appropriately set.

One interesting avenue of future work that would somewhat link the work in all 3 chapters of this thesis would be to use generative models to produce IEs for long term memory cells, such as GRUs. Despite a number of attempts to improve the architecture of long term memory cells [108], and large studies being carried out that attempted to determine which architecture performed the best on a wide variety of tasks [47], the GRU and LSTM architectures still outperform many alternatives. Nevertheless, it may be that case that within these architectures the weights of optimised cells are similar to the weights of other optimised cells. In other words, it may be that for well performing GRUs, on a number of different tasks, the weight values exist on a lower dimensional manifold. One could use a generative model to construct this manifold, and subsequently use the generative part as an IE in evolutionary search (as was done in Chapters 3 and 4). This would result in faster search due to a smaller number of parameters to optimise, and due to a more focused area of search.

Only three types of generative model were used to construct IEs in this thesis but there are more that could be used. Variations of those used in this thesis exist such as denoising autoencoders [146], Wasserstein GANs [2], and conditional VAEs [131] and GANs [93]. We hypothesis that a denoising autoencoder would have a similar regularisation effect on the manifold created by the IE as the VAE did. As we have seen, a more regularised manifold results in slightly faster search speeds. A Wasserstein GAN could decrease the frequency of mode collapse, which resulted in the GAN derived IE performing very poorly in Section 3.3.1. An IE could also perform a more focused search by conditioning on the domain parameters, τ, as in Chapter 4, or function parameters, such as α and β, in Chapter 3. This conditioning

could be achieved by using either a conditional VAE or a conditional GAN. Conditioned search would have the effect of reducing the search space to only the areas containing solutions to one particular function, as opposed to areas containing solutions to the entire family of functions. Generative models that are quite different in nature to the ones considered in this thesis could also be considered, such as diffusion models [130]. Diffusion models have recently produced impressive results with text-to-image models, namely Stable Diffusion [1]. These impressive results may also be able to be realised in the context of creating IEs.

6.4 Final Remarks

Overall, this work is the first to show that autoencoders, variational autoencoders (VAE), and generative adversarial networks (GAN) can be used for performing efficient transfer learning within RL tasks. It does so by narrowing the entire neural network controller weight space to a much smaller space containing the correlated optima of a family of RL tasks. This work is also the first work to perform a comparative analysis of these generative model derived IE types on both RL tasks and more general mathematical function optimisation tasks. Additionally, we proposed a novel algorithm capable of improving performance in a number of navigational tasks as compared to current techniques. In doing so, we add to the growing body of evidence that EAs are a competitive technique for real valued optimisation and demonstrate their applicability at creating controllers for RL control tasks and robotic navigational tasks.

Appendix A

Additional Mathematical Optimisation Results

A.1 Bivariate quadratic linear results

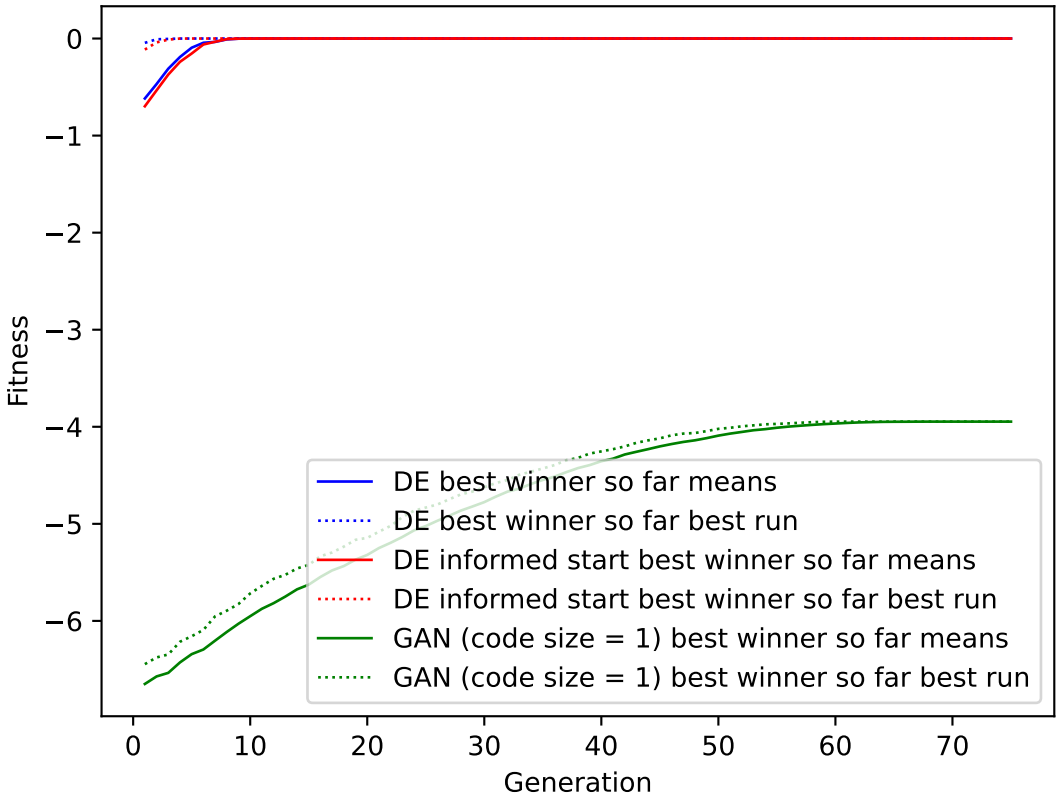


FIGURE A.1: 5 evolutionary runs for a DE, a DE with informed start, and a GAN derived IE with a code size of 1 on the target problem, $(\alpha = 2, \beta = 2)$. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

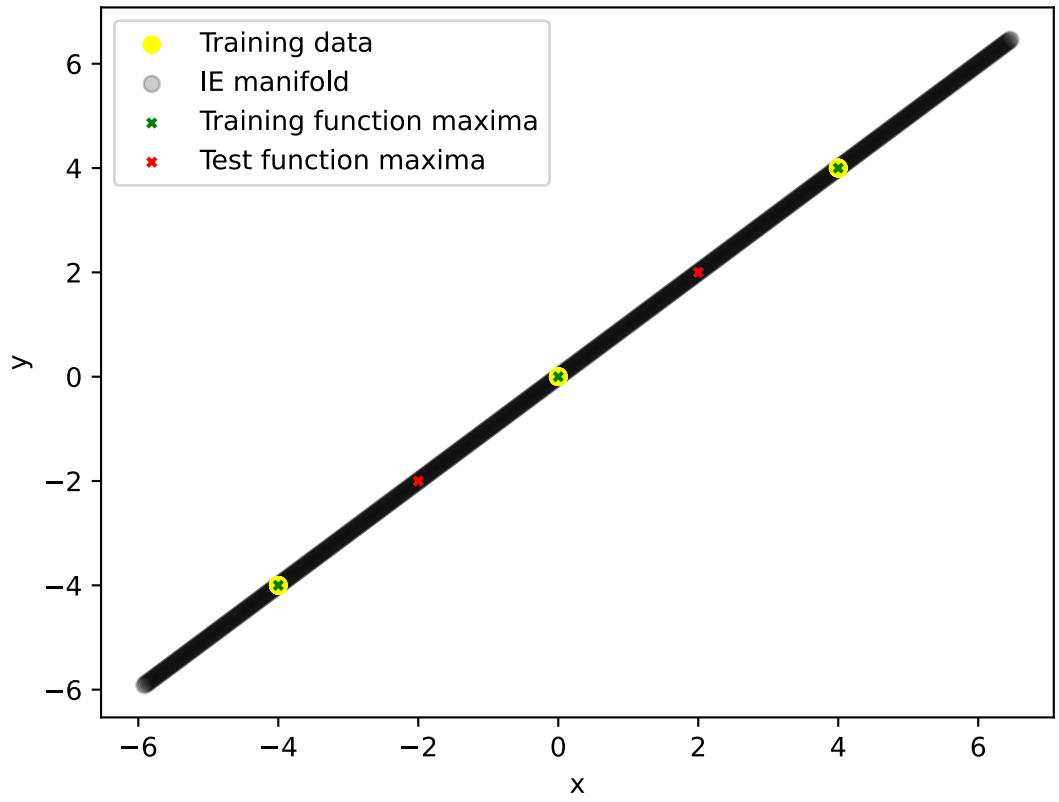


FIGURE A.2: Enumerated manifold of the best performing VAE derived IE of code size 1. Enumeration occurs over the [-3, 3] latent space at increments of 0.01. Maxima for the source and target problems are shown as green and red crosses respectively. Training data for the generative models is shown in yellow.

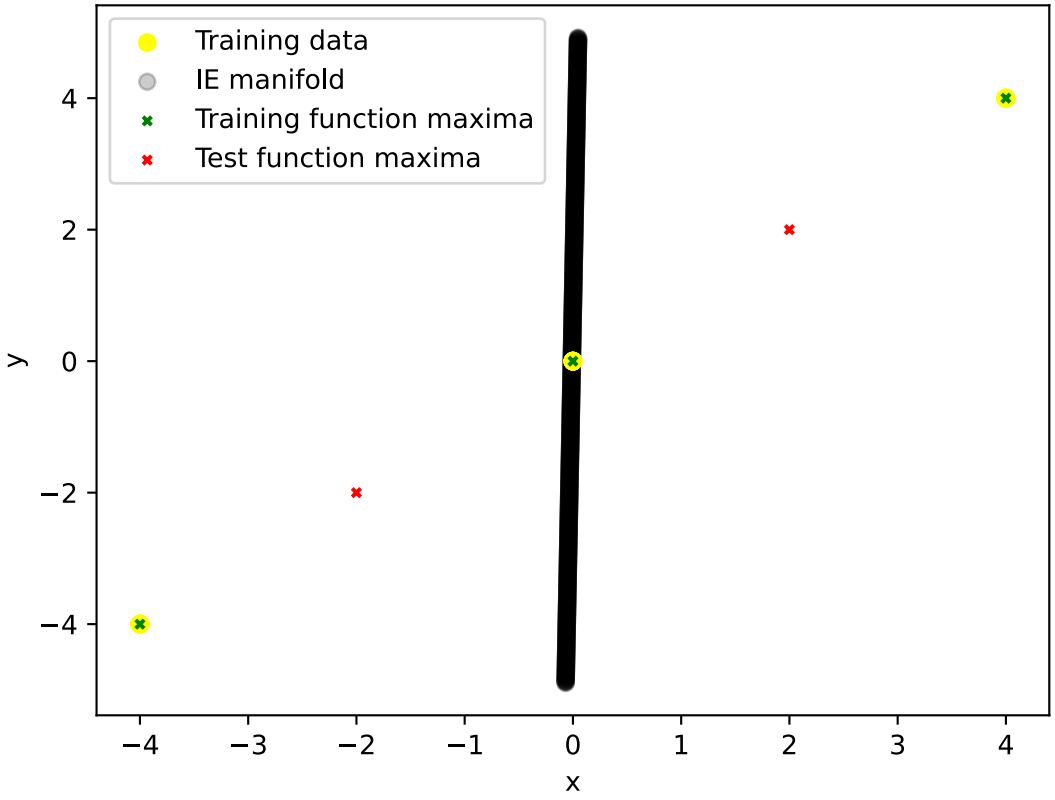


FIGURE A.3: Enumerated manifold of the best performing GAN derived IE of code size 1. Enumeration occurs over the [-30, 30] latent space at increments of 0.05.

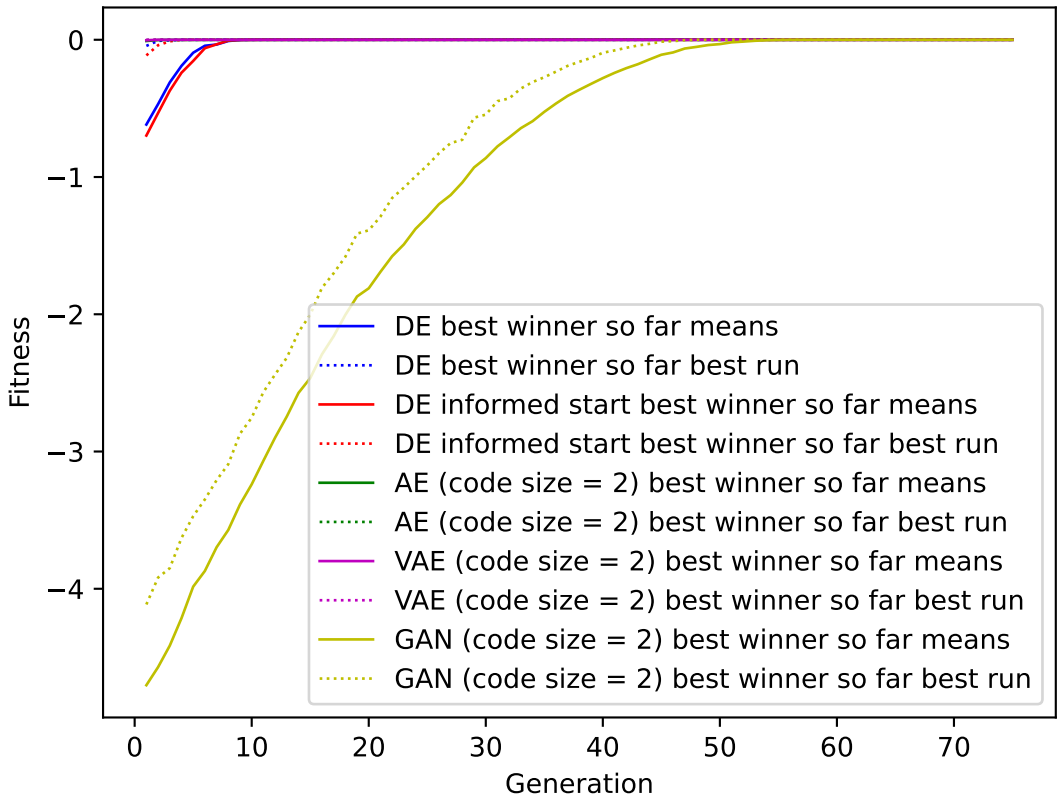


FIGURE A.4: 5 evolutionary runs for a DE, a DE with informed start, and all 3 IE types with a code size of 2 on the target problem, $(\alpha = 2, \beta = 2)$. The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

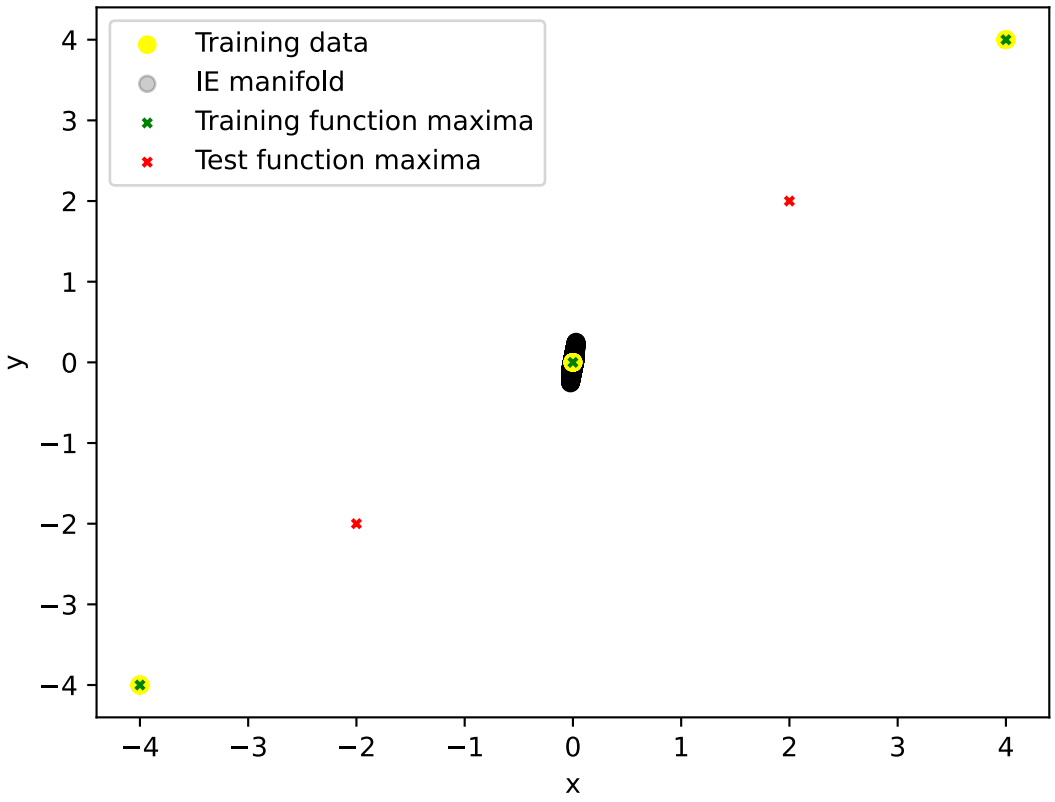


FIGURE A.5: Enumerated manifold of the best performing GAN derived IE of code size 2. Enumeration occurs over the [-3, 3] latent space at increments of 0.05. Mode collapse has occured over the training points situated at (0, 0).

A.2 Bivariate quadratic non-linear results

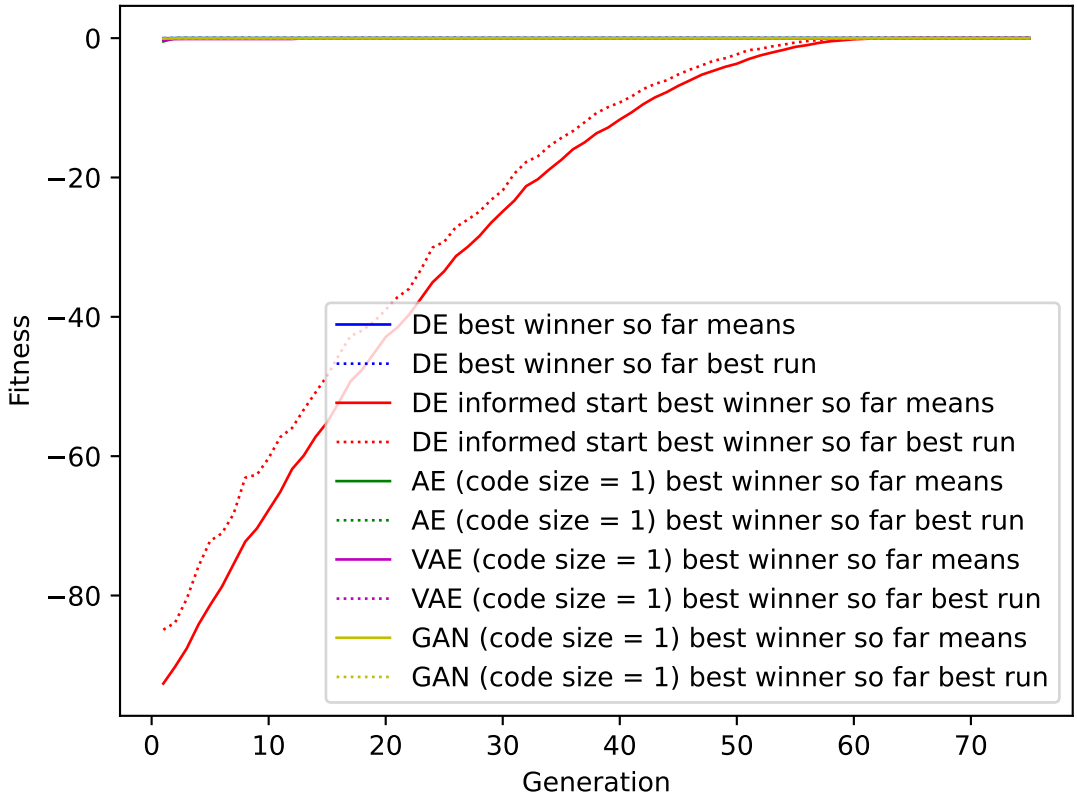


FIGURE A.6: 5 evolutionary runs for a DE, a DE with informed start, and all 3 IEs with a code size of 1 on the target problem, ($\alpha = 0.5, \beta = 0.25$). The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

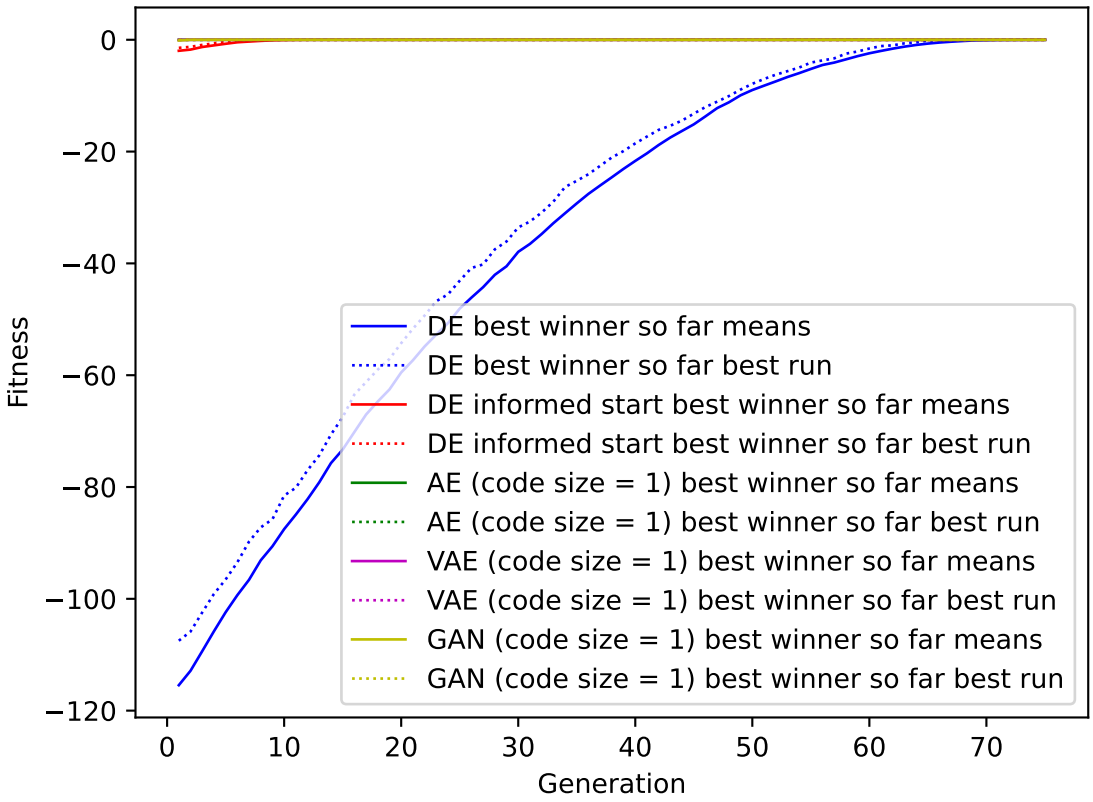


FIGURE A.7: 5 evolutionary runs for a DE, a DE with informed start, and each of the 3 IEs with a code size of 1 on the target problem, ($\alpha = -3.5, \beta = 12.25$). The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

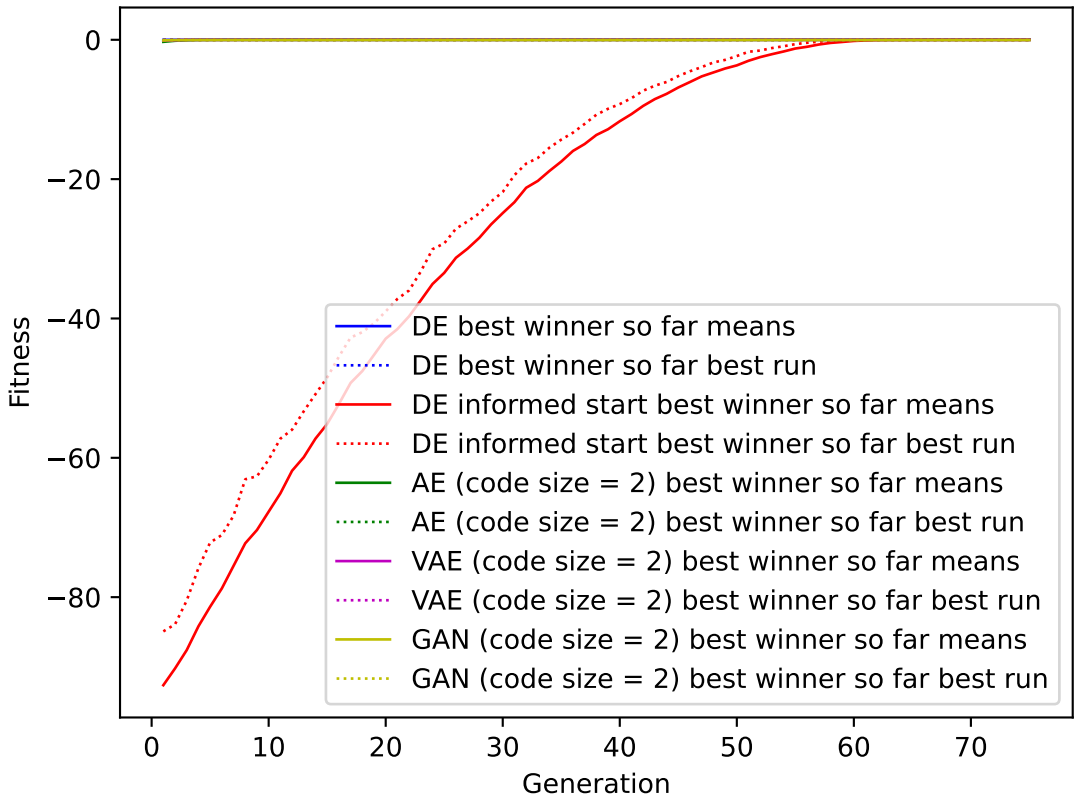


FIGURE A.8: 5 evolutionary runs for a DE, a DE with informed start, and all 3 IEs with a code size of 2 on the target problem, ($\alpha = 0.5, \beta = 0.25$). The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.

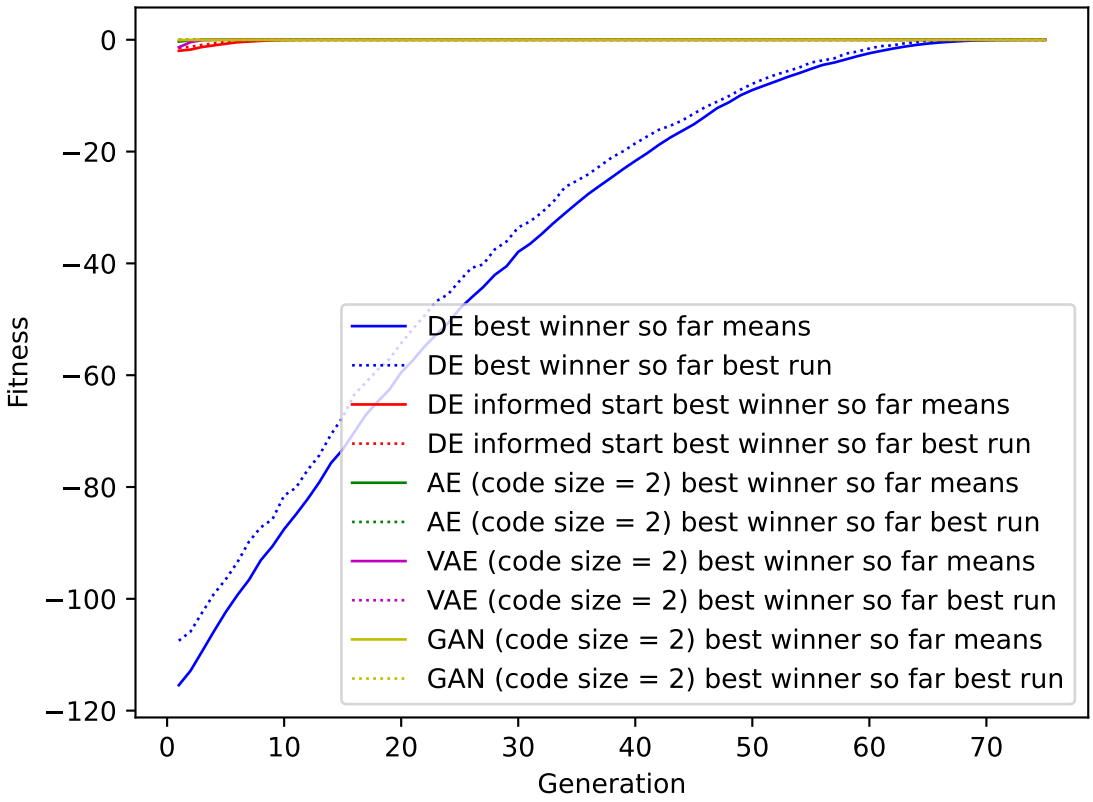
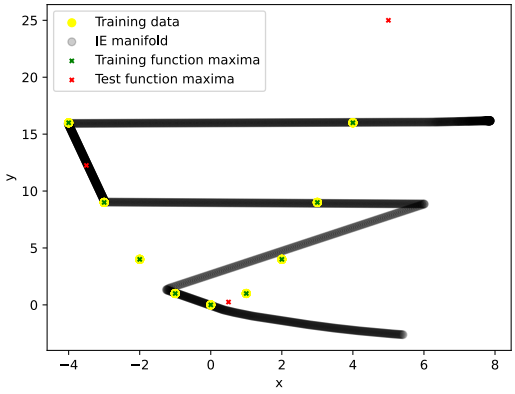
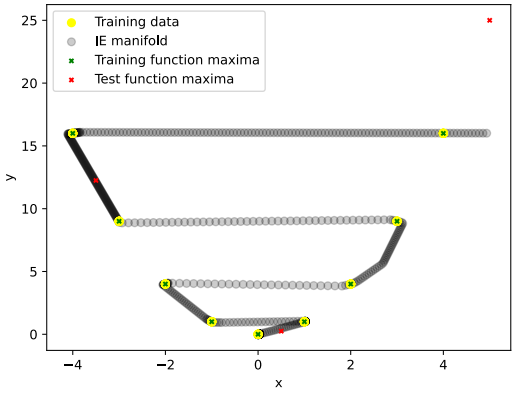


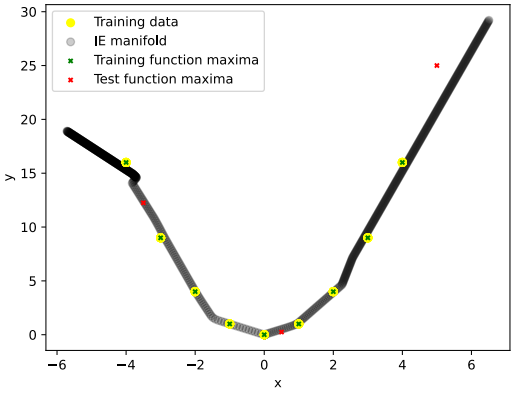
FIGURE A.9: 5 evolutionary runs for a DE, a DE with informed start, and each of the 3 IEs with a code size of 2 on the target problem, ($\alpha = -3.5, \beta = 12.25$). The fitnesses plotted are those of the best winner so far in the evolutionary run. The solid lines are the mean values of the 5 runs, whereas the dotted lines are the fitness values of the highest performing run.



(A) Autoencoder, enumeration range = [-0.2, 1.2] in increments of 0.001.

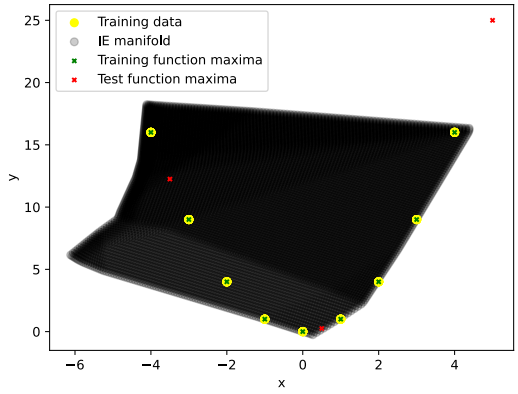


(B) VAE, enumeration range = [-3, 3] in increments of 0.01.

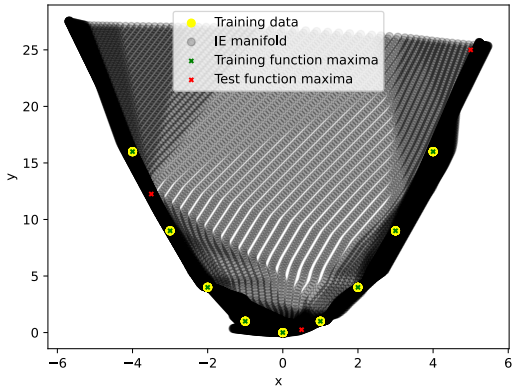


(C) GAN, enumeration range = [-3.5, 3.5] in increments of 0.01.

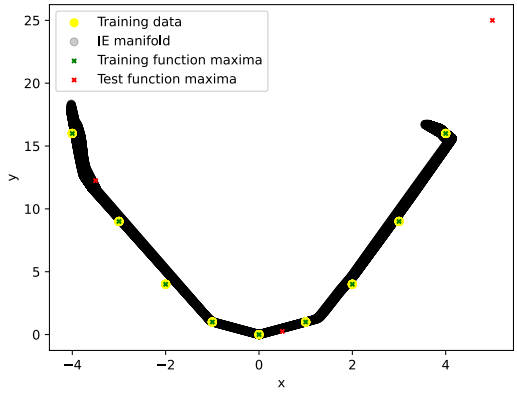
FIGURE A.10: Enumerated manifolds over the latent space of all 3 code size 1 IEs on the target problem, $(\alpha = -3.5, \beta = 12.25)$. Maxima for the source and target problems are shown as green and red crosses respectively. Training data for the generative models is shown in yellow.



(A) Autoencoder, enumeration range = [0.0, 1.0] in increments of 0.01.



(B) VAE, enumeration range = [-3, 3] in increments of 0.03.



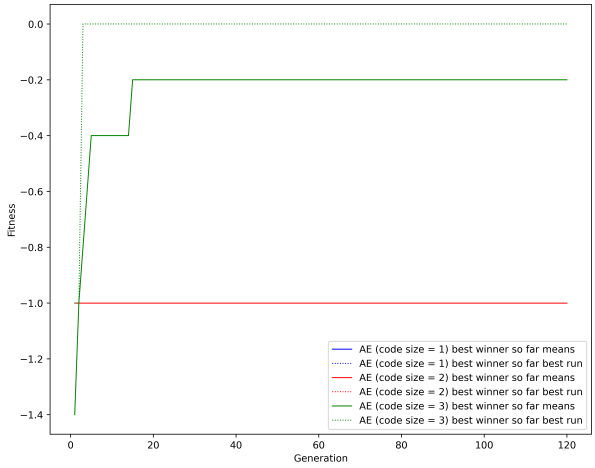
(C) GAN, enumeration range = [-3.0, 3.0] in increments of 0.03.

FIGURE A.11: Enumerated manifolds over the latent space of all 3 code size 2 IEs on the target problem, $(\alpha = -3.5, \beta = 12.25)$. Maxima for the source and target problems are shown as green and red crosses respectively. Training data for the generative models is shown in yellow.

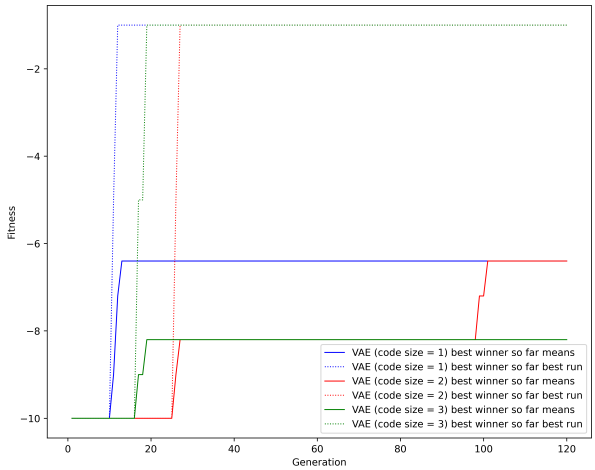
Appendix B

Additional Control Domain Results

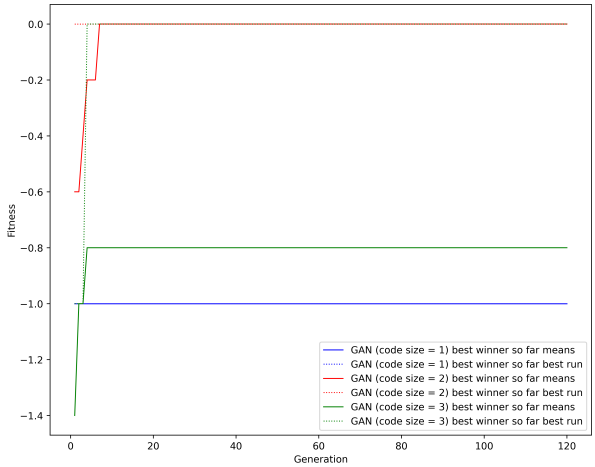
B.1 Frozen Lake results



(A) Autoencoder.



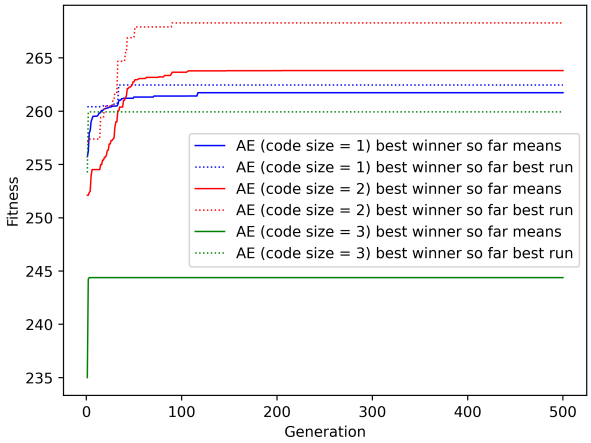
(B) VAE.



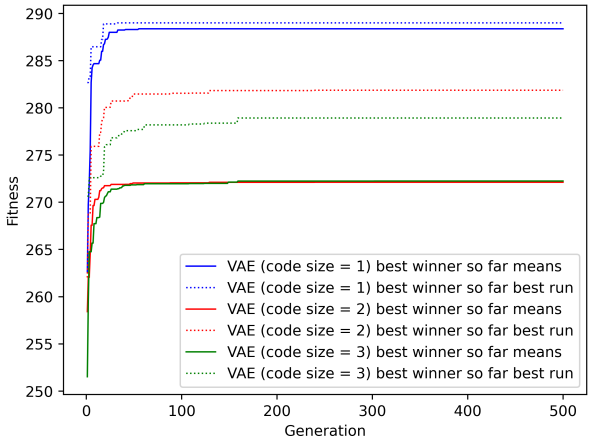
(C) GAN.

FIGURE B.1: Comparison of IE performance between code sizes 1, 2, and 3 on FL with test goal position (1,3). The fitnesses plotted are those of the best winner so far, this is the best solution found so far during the evolutionary run. The solid lines are the mean fitnesses over 5 evolutionary runs and the dotted line is the best run according to the final generation fitness.

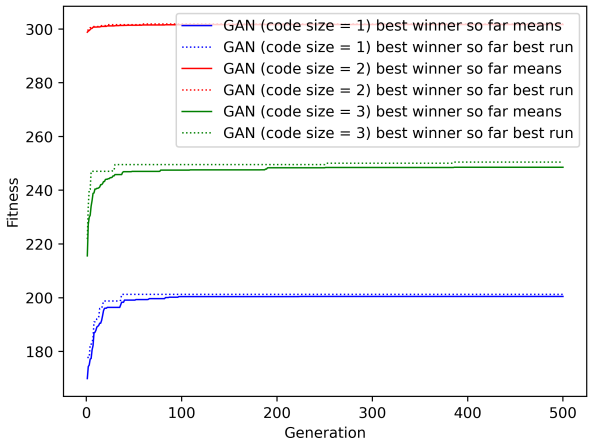
B.2 Bipedal Walker results



(A) Autoencoder.



(B) VAE.



(C) GAN.

FIGURE B.2: Comparison of IE performance between code sizes 1, 2 and 3 on BW with test knee speed 5. The fitnesses plotted are those of the best winner so far, this is the best solution found so far during the evolutionary run. The solid lines are the mean fitnesses over 5 evolutionary runs and the dotted line is the best run according to the final generation fitness.

Appendix C

Hyperparameters

C.1 NEAT-GRU experiment hyperparameters

The following four tables describe the NEAT hyperparameter settings used in the evolutionary experiments in Section 5.3.2. There is a table for both NEAT and NEAT-GRU on both the bearing and non-bearing environments.

trait_param_mut_prob	0.5
trait_mutation_power	1.0
linktrait_mut_sig	1.0
nodetrait_mut_sig	0.5
weight_mut_power	1.5
gru_weight_mut_power	1.5
recur_prob	0.05
disjoint_coeff	1.0
excess_coeff	2.0
mutdiff_coeff	0.0
gru_compat_coeff	0.0
compat_thresh	3.0
age_significance	1.0
survival_thresh	0.55
mutate_only_prob	1.0
mutate_random_trait_prob	0.1
mutate_link_trait_prob	0.1
mutate_node_trait_prob	0.1
mutate_link_weights_prob	0.8
mutate_gene_rate_prob	0.1
mutate_gru_link_weights_prob	0.5
mutate_gru_gene_rate_prob	0.1
mutate_toggle_enable_prob	0.01
mutate_gene_reenable_prob	0.01
mutate_add_node_prob	0.005
mutate_gru_add_node_prob	0.0
mutate_add_link_prob	0.04
interspecies_mate_rate	0.001
mate_multipoint_prob	0.6
mate_multipoint_avg_prob	0.4
mate_singlepoint_prob	0.0
mate_only_prob	0.2
recur_only_prob	0.2
pop_size	150
dropoff_age	25
newlink_tries	20
print_every	199
babies_stolen	0
num_runs	1
num_gens	1000
num_trials	10

TABLE C.1: A table of NEAT hyperparameters used for the NEAT (no GRU) bearing experiments.

trait_param_mut_prob	0.5
trait_mutation_power	1.0
linktrait_mut_sig	1.0
nodetrait_mut_sig	0.5
weight_mut_power	1.5
gru_weight_mut_power	1.5
recur_prob	0.05
disjoint_coeff	1.0
excess_coeff	2.0
mutdiff_coeff	0.0
gru_compat_coeff	0.0
compat_thresh	3.0
age_significance	1.0
survival_thresh	0.55
mutate_only_prob	1.0
mutate_random_trait_prob	0.1
mutate_link_trait_prob	0.1
mutate_node_trait_prob	0.1
mutate_link_weights_prob	0.8
mutate_gene_rate_prob	0.1
mutate_gru_link_weights_prob	0.5
mutate_gru_gene_rate_prob	0.1
mutate_toggle_enable_prob	0.01
mutate_gene_reenable_prob	0.01
mutate_add_node_prob	0.005
mutate_gru_add_node_prob	0.003
mutate_add_link_prob	0.04
interspecies_mate_rate	0.001
mate_multipoint_prob	0.6
mate_multipoint_avg_prob	0.4
mate_singlepoint_prob	0.0
mate_only_prob	0.2
recur_only_prob	0.2
pop_size	150
dropoff_age	25
newlink_tries	20
print_every	199
babies_stolen	0
num_runs	1
num_gens	1000
num_trials	10

TABLE C.2: A table of NEAT hyperparameters used for the NEAT-GRU bearing experiments. Note the non-zero probability of mutate_gru_add_node_prob.

trait_param_mut_prob	0.5
trait_mutation_power	1.0
linktrait_mut_sig	1.0
nodetrait_mut_sig	0.5
weight_mut_power	0.5
gru_weight_mut_power	0.0
recur_prob	0.05
disjoint_coeff	1.25
excess_coeff	1.25
mutdiff_coeff	0.0
gru_compat_coeff	0.0
compat_thresh	3.0
age_significance	1.0
survival_thresh	0.4
mutate_only_prob	1.0
mutate_random_trait_prob	0.1
mutate_link_trait_prob	0.1
mutate_node_trait_prob	0.1
mutate_link_weights_prob	0.7
mutate_gene_rate_prob	0.5
mutate_gru_link_weights_prob	0.0
mutate_gru_gene_rate_prob	0.0
mutate_toggle_enable_prob	0.01
mutate_gene_reenable_prob	0.01
mutate_add_node_prob	0.006
mutate_gru_add_node_prob	0.0
mutate_add_link_prob	0.04
interspecies_mate_rate	0.001
mate_multipoint_prob	0.6
mate_multipoint_avg_prob	0.4
mate_singlepoint_prob	0.0
mate_only_prob	0.2
recur_only_prob	0.2
pop_size	150
dropoff_age	25
newlink_tries	20
print_every	199
babies_stolen	0
num_runs	1
num_gens	5000
num_trials	5

TABLE C.3: A table of NEAT hyperparameters used for the NEAT (no GRU) **no** bearing experiments.

trait_param_mut_prob	0.5
trait_mutation_power	1.0
linktrait_mut_sig	1.0
nodetrait_mut_sig	0.5
weight_mut_power	0.5
gru_weight_mut_power	0.5
recur_prob	0.05
disjoint_coeff	1.25
excess_coeff	1.25
mutdiff_coeff	0.0
gru_compat_coeff	0.0
compat_thresh	3.0
age_significance	1.0
survival_thresh	0.4
mutate_only_prob	1.0
mutate_random_trait_prob	0.1
mutate_link_trait_prob	0.1
mutate_node_trait_prob	0.1
mutate_link_weights_prob	0.7
mutate_gene_rate_prob	0.5
mutate_gru_link_weights_prob	0.5
mutate_gru_gene_rate_prob	0.3
mutate_toggle_enable_prob	0.01
mutate_gene_reenable_prob	0.01
mutate_add_node_prob	0.006
mutate_gru_add_node_prob	0.006
mutate_add_link_prob	0.04
interspecies_mate_rate	0.001
mate_multipoint_prob	0.6
mate_multipoint_avg_prob	0.4
mate_singlepoint_prob	0.0
mate_only_prob	0.2
recur_only_prob	0.2
pop_size	150
dropoff_age	25
newlink_tries	20
print_every	199
babies_stolen	0
num_runs	1
num_gens	5000
num_trials	5

TABLE C.4: A table of NEAT hyperparameters used for the NEAT-GRU **no** bearing experiments.

Bibliography

[1] Stability AI. *Stable Diffusion*. 2022. URL: https://github.com/CompVis/stable-diffusion (visited on 10/29/2022).

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein Generative Adversarial Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 214–223. URL: https://proceedings.mlr.press/v70/arjovsky17a.html.

[3] A Bachrach et al. "RANGE - robust autonomous navigation in GPS-denied environments". In: *2010 IEEE International Conference on Robotics and Automation*. May 2010, pp. 1096–1097.

[4] Bram Bakker. "Reinforcement Learning with Long Short-term Memory". In: *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*. MIT Press, 2001, pp. 1475–1482.

[5] Yoshua Bengio et al. "Curriculum Learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: Association for Computing Machinery, 2009, 41–48. ISBN: 9781605585161. DOI: 10.1145/1553374.1553380. URL: https://doi.org/10.1145/1553374.1553380.

[6] Peter J Bentley et al. "COIL: Constrained Optimization in Learned Latent Space - Learning Representations for Valid Solutions". In: *CoRR* abs/2202.02163 (2022). DOI: 10.48550/arXiv.2202.02163.

[7] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategies–a comprehensive introduction". In: *Natural computing* 1.1 (2002), pp. 3–52.

[8] Garrett Bingham, William Macke, and Risto Miikkulainen. "Evolutionary Optimization of Deep Learning Activation Functions". In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. GECCO '20. Cancún, Mexico: Association for Computing Machinery, 2020, 289–296. ISBN: 9781450371285. DOI: 10.1145/3377930.3389841. URL: https://doi.org/10.1145/3377930.3389841.

[9] Jesper Blynel and Dario Floreano. "Exploring the T-Maze: Evolving Learning-Like Robot Behaviors Using CTRNNs". In: *Applications of Evolutionary Computing*. Springer Berlin Heidelberg, 2003, pp. 593–604. ISBN: 978-3-540-36605-8.

[10] Jonathan C. Brant and Kenneth O. Stanley. "Minimal criterion coevolution: a new approach to open-ended search". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2017*. 2017, pp. 67–74.

[11] Hans J Bremermann et al. "Optimization through evolution and recombination". In: *Self-organizing systems* 93 (1962), p. 106.

[12] G Bresson et al. "Simultaneous Localization and Mapping: A Survey of Current Trends in Autonomous Driving". In: *IEEE Transactions on Intelligent Vehicles* 2 (3 Sept. 2017), pp. 194–220.

[13] Bastian Broecker, Karl Tuyls, and James Butterworth. "Distance-based multi-robot coordination on pocket drones". In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2018.

[14] Ruben Broekx. "An Empirical Investigation of Using Gated Recurrent Units in Evolved Robotic Controllers". MSc Thesis. Katholieke Universiteit Leuven, 2020.

[15] James Butterworth, Rahul Savani, and Karl Tuyls. "Evolving Indoor Navigational Strategies Using Gated Recurrent Units in NEAT". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '19. Prague, Czech Republic: Association for Computing Machinery, 2019, 111–112. ISBN: 9781450367486. DOI: 10.1145/3319619.3321995. URL: https://doi.org/10.1145/3319619.3321995.

[16] James Butterworth, Rahul Savani, and Karl Tuyls. "Generative Models over Neural Controllers for Transfer Learning". In: *Parallel Problem Solving from Nature – PPSN XVII*. Ed. by Günter Rudolph et al. Cham: Springer International Publishing, 2022, pp. 400–413. ISBN: 978-3-031-14714-2.

[17] James Butterworth et al. "Evolving Coverage Behaviours For MAVs Using NEAT". In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '18. Stockholm, Sweden: International Foundation for Autonomous Agents and Multiagent Systems, 2018, 1886–1888.

[18] Oscar Chang et al. "Agent Embeddings: A Latent Representation for Pole-Balancing Networks". In: *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '19. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2019, 656–664. ISBN: 9781450363099. URL: https://dl.acm.org/doi/10.5555/3306127.3331753.

[19] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* (2014). URL: http://arxiv.org/abs/1406.1078.

[20] Kyunghyun Cho et al. "On the Properties of Neural Machine Translation: Encoder-Decoder Approaches". In: *CoRR* abs/1409.1259 (2014). URL: http://arxiv.org/abs/1409.1259.

[21] Anna Choromanska et al. "The Loss Surfaces of Multilayer Networks". In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Guy Lebanon and S. V. N. Vishwanathan. Vol. 38. Proceedings of Machine Learning Research. San Diego, California, USA: PMLR, 2015, pp. 192–204. URL: https://proceedings.mlr.press/v38/choromanska15.html.

[22] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* (2014). URL: http://arxiv.org/abs/1412.3555.

[23] Dan Ciresan, Ueli Meier, and Jurgen Schmidhuber. "Multi-column Deep Neural Networks for Image Classification". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. June 2012, pp. 3642–3649. DOI: 10.1109/CVPR.2012.6248110.

[24] Jeff Clune et al. "On the Performance of Indirect Encoding Across the Continuum of Regularity". In: *IEEE Transactions on Evolutionary Computation* 15.3 (2011), pp. 346–367. DOI: 10.1109/TEVC.2010.2104157.

[25] Charles Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. Ed. by John Murray. 1859.

[26] Charles Darwin and Alfred Wallace. "On the Tendency of Species to form Varieties; and on the Perpetuation of Varieties and Species by Natural Means of Selection". In: *Zoological Journal of the Linnean Society* 3.9 (July 1858), pp. 45–62. ISSN: 0024-4082. DOI: 10.1111/j.1096-3642.1858.tb02500.x.

[27] Yann Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *NIPS* 27 (June 2014).

[28] Kenneth De Jong. *Evolutionary Computation – A Unified Approach*. Jan. 2006. ISBN: 978-0-262-04194-2.

[29] Jonas Degrave et al. "Magnetic control of tokamak plasmas through deep reinforcement learning". In: *Nature* 602 (Feb. 2022), pp. 414–419. DOI: 10.1038/s41586-021-04301-9.

[30] M Dorigo et al. "Swarmanoid: A Novel Concept for the Study of Heterogeneous Robotic Swarms". In: *IEEE Robotics Automation Magazine* 20 (4 2013), pp. 60–71.

[31] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. 2nd. Springer Publishing Company, Incorporated, 2015. ISBN: 3662448734.

[32] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*. Vol. 53. Springer, 2003.

[33] Alhussein Fawzi et al. "Discovering faster matrix multiplication algorithms with reinforcement learning". In: *Nature* 610.7930 (2022), pp. 47–53. ISSN: 1476-4687. DOI: 10.1038/s41586-022-05172-4. URL: https://doi.org/10.1038/s41586-022-05172-4.

[34] Chelsea Finn, Pieter Abbeel, and Sergey Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, 1126–1135. URL: https://dl.acm.org/doi/10.5555/3305381.3305498.

[35] Dario Floreano and Stefano Nolfi. *Evolutionary Robotics*. Cambridge: MIT Press, 2000.

[36] David B. Fogel. "An Introduction to Evolutionary Computation". In: *Evolutionary Computation: The Fossil Record*. 1998, pp. 1–28. DOI: 10.1109/9780470544600.ch1.

[37] Gary B. Fogel. *Evolutionary Programming*. Ed. by Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok. Springer Berlin Heidelberg, 2012, pp. 699–708. ISBN: 978-3-540-92910-9. DOI: 10.1007/978-3-540-92910-9_23.

[38] L. J. Fogel, A. J. Owens, and M. J. Walsh. "Artificial Intelligence through a Simulation of Evolution". In: *Biophysics and Cybernetic Systems: Proc. of the 2nd Cybernetic Sciences Symp.* Washington, DC: Spartan Books, 1965, pp. 131–155.

[39] Richard M Friedberg. "A learning machine: Part I". In: *IBM Journal of Research and Development* 2.1 (1958), pp. 2–13.

[40] Adam Gaier, Alexander Asteroth, and Jean-Baptiste Mouret. "Discovering Representations for Black-Box Optimization". In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. GECCO '20. Cancún, Mexico: Association for Computing Machinery, 2020, 103–111. ISBN: 9781450371285. DOI: 10.1145/3377930.3390221.

[41] Ross Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 580–587. DOI: 10.1109/CVPR.2014.81.

[42] David E. Goldberg. "Simple genetic algorithms and the minimal, deceptive problem". In: *Genetic algorithms and simulated annealing*. Ed. by Lawrence Davis. Research Notes in Artificial Intelligence. London: Pitman, 1987, pp. 74–88.

[43] Faustino J Gomez and Jürgen Schmidhuber. "Co-evolving Recurrent Neurons Learn Deep Memory POMDPs". In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2005, pp. 491–498.

[44] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[45] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

[46] F. Gray. *Pulse code communication*. US Patent 2,632,058. 1953. URL: http://www.google.com/patents/US2632058.

[47] Klaus Greff et al. "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2017), pp. 2222–2232. DOI: 10.1109/TNNLS.2016.2582924.

[48] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. "Evolving Neural Turing Machines for Reward-based Learning". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 2016, pp. 117–124.

[49] Nikolaus Hansen. "The CMA Evolution Strategy: A Tutorial". In: *CoRR* abs/1604.00772 (2016). arXiv: 1604.00772. URL: http://arxiv.org/abs/1604.00772.

[50] Nikolaus Hansen and Andreas Ostermeier. "Completely derandomized self-adaptation in evolution strategies". In: *Evolutionary computation* 9.2 (2001), pp. 159–195.

[51] Nikolaus Hansen et al. "Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009". In: *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*. 2010, pp. 1689–1696.

[52] William E Hart, Thomas E Kammeyer, and Richard K Belew. "The Role of Development in Genetic Algorithms". In: *Foundations of Genetic Algorithms*. Ed. by L DARRELL WHITLEY and MICHAEL D VOSE. Vol. 3. Foundations of Genetic Algorithms. Elsevier, 1995, pp. 315–332. DOI: https://doi.org/10.1016/B978-1-55860-356-1.50019-4. URL: http://www.sciencedirect.com/science/article/pii/B9781558603561500194.

[53] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[54] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[55] Max Jaderberg et al. "Reinforcement Learning with Unsupervised Auxiliary Tasks". In: *CoRR* abs/1611.0 (2016). URL: http://arxiv.org/abs/1611.05397.

[56] Thomas Jansen, Kenneth A. De Jong, and Ingo Wegener. "On the Choice of the Offspring Population Size in Evolutionary Algorithms". In: *Evolutionary Computation* 13.4 (Dec. 2005), pp. 413–440. ISSN: 1063-6560. DOI: 10.1162/106365605774666921. URL: https://doi.org/10.1162/106365605774666921.

[57] Marija Jegorova, Stéphane Doncieux, and Timothy Hospedales. "Behavioral Repertoire via Generative Adversarial Policy Networks". In: *IEEE Transactions on Cognitive and Developmental Systems* (2020), p. 1. DOI: 10.1109/TCDS.2020.3008574.

[58] Pouya Rezazadeh Kalehbasti, Michael D. Lepech, and Samarpreet Singh Pandher. "Augmenting High-Dimensional Nonlinear Optimization with Conditional GANs". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '21. Lille, France: Association for Computing Machinery, 2021, 1879–1880. ISBN: 9781450383516. DOI: 10.1145/3449726.3463675.

[59] I Kamon and E Rivlin. "Sensory-based motion planning with global proofs". In: *IEEE Transactions on Robotics and Automation* 13 (6 Dec. 1997), pp. 814–822.

[60] I Kamon, E Rivlin, and E Rimon. "A new range-sensor based globally convergent navigation algorithm for mobile robots". In: *Proceedings of IEEE International Conference on Robotics and Automation*. Vol. 1. Apr. 1996, 429–435 vol.1.

[61] Michal Kempka et al. "ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning". In: *CoRR* (2016). URL: http://arxiv.org/abs/1605.02097.

[62] Shauharda Khadka, Jen Jen Chung, and Kagan Tumer. "Evolving memory-augmented neural architecture for deep memory problems". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2017*. 2017, pp. 441–448.

[63] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations* (Dec. 2014).

[64] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: International Conference on Learning Representations, 2015.

[65] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes*. 2013. DOI: 10.48550/ARXIV.1312.6114. URL: https://arxiv.org/abs/1312.6114.

[66] Jens Kober, J. Andrew Bagnell, and Jan Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274. DOI: 10.1177/0278364913495721. eprint: https://doi.org/10.1177/0278364913495721. URL: https://doi.org/10.1177/0278364913495721.

[67] Kostas Kouvaris et al. "How evolution learns to generalise: Using the principles of learning theory to understand the evolution of developmental organisation". In: *PLOS Computational Biology* 13.4 (2017), pp. 1–20. DOI: 10.1371/journal.pcbi.1005358.

[68] Taras Kowaliw et al. "Artificial Neurogenesis: An Introduction and Selective Review". In: *Growing Adaptive Machines: Combining Development and Learning in Artificial Neural Networks*. Ed. by Taras Kowaliw, Nicolas Bredeche, and René Doursat. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 1–60. ISBN: 978-3-642-55337-0. DOI: 10.1007/978-3-642-55337-0_1. URL: https://doi.org/10.1007/978-3-642-55337-0_1.

[69] John R. Koza. "Genetic programming as a means for programming computers by means of natural selection". In: *Statistics and Computing* 4 (1994), pp. 87–112. DOI: 10.1007/BF00175355.

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[71] Tejas D. Kulkarni et al. "Deep Successor Reinforcement Learning". In: *CoRR* (2016). URL: https://arxiv.org/abs/1606.02396.

[72] Raz Lapid and Moshe Sipper. "Evolution of Activation Functions for Deep Learning-Based Image Classification". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '22. Boston, Massachusetts: Association for Computing Machinery, 2022, 2113–2121. ISBN: 9781450392686. DOI: 10.1145/3520304.3533949. URL: https://doi.org/10.1145/3520304.3533949.

[73] Robin J. Leatherbarrow. "Using linear and non-linear regression to fit biochemical data". In: *Trends in Biochemical Sciences* 15.12 (1990), pp. 455–458. ISSN: 0968-0004. DOI: https://doi.org/10.1016/0968-0004(90)90295-M. URL: https://www.sciencedirect.com/science/article/pii/096800049090295M.

[74] Yann LeCun. "Generalization and Network Design Strategies". In: *Connectionism in Perspective*. Ed. by R. Pfeifer et al. Zurich, Switzerland: Elsevier, 1989.

[75] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[76] Joel Lehman and Kenneth O Stanley. "Abandoning Objectives: Evolution Through the Search for Novelty Alone". In: *Evolutionary Computation* 19 (2 June 2011), pp. 189–223.

[77] Joel Lehman and Kenneth O Stanley. "Exploiting open-endedness to solve problems through the search for novelty". In: *Proceedings of the Eleventh International Conference on Artificial Life (Alife XI)*. MIT Press, 2008.

[78] Joel Lehman and Kenneth O Stanley. "Novelty Search and the Problem with Objectives". In: *Genetic Programming Theory and Practice IX*. Ed. by Rick Riolo, Ekaterina Vladislavleva, and Jason H Moore. Springer New York, 2011, pp. 37–56. ISBN: 978-1-4614-1770-5. DOI: 10.1007/978-1-4614-1770-5_3. URL: https://doi.org/10.1007/978-1-4614-1770-5_3.

[79] Joel Lehman and Kenneth O Stanley. "Revising the Evolutionary Computation Abstraction: Minimal Criteria Novelty Search". In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2010, pp. 103–110.

[80] Joel Lehman et al. "Safe mutations for deep and recurrent neural networks through output gradients". In: *Proceedings of the Genetic and Evolutionary Computation Conference* (2018).

[81] V Lumelsky and T Skewis. "A paradigm for incorporating vision in the robot navigation function". In: *Proceedings. 1988 IEEE International Conference on Robotics and Automation*. Apr. 1988, 734–739 vol.2.

[82] V Lumelsky and A Stepanov. "Dynamic path planning for a mobile automaton with limited information on the environment". In: *IEEE Transactions on Automatic Control* 31 (11 Nov. 1986), pp. 1058–1063.

[83] V J Lumelsky and T Skewis. "Incorporating range sensing in the robot navigation function". In: *IEEE Transactions on Systems, Man, and Cybernetics* 20 (5 Sept. 1990), pp. 1058–1069.

[84] Benno Lüders, Mikkel Schläger, and Sebastian Risi. "Continual Learning through Evolvable Neural Turing Machines". In: *Proceedings of the NIPS 2016 Workshop on Continual Learning and Deep Networks (CLDL 2016)*. 2016.

[85] Benno Lüders et al. "Continual and One-Shot Learning Through Neural Networks with Dynamic External Memory". In: *EvoApplications*. 2017.

[86] E Magid and E Rivlin. "CautiousBug: a competitive algorithm for sensory-based robot navigation". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. Sept. 2004, 2757–2762 vol.3.

[87] Steven Manos and Leon Poladian. "Optical fibre design using evolutionary strategies". In: *Engineering Computations* 21 (Sept. 2004), pp. 564–576. DOI: 10.1108/02644400410545164.

[88] Warren Mcculloch and Walter Pitts. "A Logical Calculus of Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.

[89] K.N. McGuire, G.C.H.E. de Croon, and K. Tuyls. "A comparative study of bug algorithms for robot navigation". In: *Robotics and Autonomous Systems* 121 (2019), p. 103261. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2019.103261. URL: https://www.sciencedirect.com/science/article/pii/S0921889018306687.

[90] Risto Miikkulainen et al. "Chapter 15 - Evolving Deep Neural Networks". In: *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Ed. by Robert Kozma et al. Academic Press, 2019, pp. 293–312. ISBN: 978-0-12-815480-9. DOI: https://doi.org/10.1016/B978-0-12-815480-9.00015-3. URL: https://www.sciencedirect.com/science/article/pii/B9780128154809000153.

[91] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969. ISBN: 978-0-262-63022-1.

[92] Piotr W Mirowski et al. "Learning to Navigate in Complex Environments". In: *CoRR* (2016). URL: https://arxiv.org/abs/1611.03673.

[93] Mehdi Mirza and Simon Osindero. "Conditional Generative Adversarial Nets". In: *CoRR* abs/1411.1784 (2014). arXiv: 1411.1784. URL: http://arxiv.org/abs/1411.1784.

[94] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q Weinberger. Vol. 48. PMLR, 2016, pp. 1928–1937.

[95] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015), p. 529.

[96] Matthew Andres Moreno, Wolfgang Banzhaf, and Charles Ofria. "Learning an Evolvable Genotype-Phenotype Mapping". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '18. Kyoto, Japan: Association for Computing Machinery, 2018, 983–990. ISBN: 9781450356183. DOI: 10.1145/3205455.3205597.

[97] Gregory Morse and Kenneth O. Stanley. "Simple Evolutionary Optimization Can Rival Stochastic Gradient Descent in Neural Networks". In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. GECCO '16. Denver, Colorado, USA: Association for Computing Machinery, 2016, 477–484. ISBN: 9781450342063. DOI: 10.1145/2908812.2908916. URL: https://doi.org/10.1145/2908812.2908916.

[98] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. "Evolution algorithms in combinatorial optimization". In: *Parallel Computing* 7.1 (1988), pp. 65–85. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(88)90098-1. URL: https://www.sciencedirect.com/science/article/pii/0167819188900981.

[99] Junhyuk Oh et al. "Control of Memory, Active Perception, and Action in Minecraft". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. JMLR.org, 2016, pp. 2790–2799.

[100] Lauren Parker, James Butterworth, and Shan Luo. "Fly Safe: Aerial Swarm Robotics using Force Field Particle Swarm Optimisation". In: *CoRR* abs/1907.07647 (2019). arXiv: 1907.07647. URL: http://arxiv.org/abs/1907.07647.

[101] F. Pezzella, G. Morganti, and G. Ciaschetti. "A genetic algorithm for the Flexible Job-shop Scheduling Problem". In: *Computers & Operations Research* 35.10 (2008). Part Special Issue: Search-based Software Engineering, pp. 3202–3212. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2007.02.014. URL: https://www.sciencedirect.com/science/article/pii/S0305054807000524.

[102] Mark Pfeiffer et al. "Reinforced Imitation: Sample Efficient Deep Reinforcement Learning for Map-less Navigation by Leveraging Prior Demonstrations". In: *CoRR* (2018). URL: http://arxiv.org/abs/1805.07095.

[103] C Pinciroli et al. "ARGoS: A Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems". In: *Swarm Intelligence* 6 (4 2012), pp. 271–295.

[104] Tobias Pohlen et al. "Observe and Look Further: Achieving Consistent Performance on Atari". In: *CoRR* (2018). URL: https://arxiv.org/abs/1805.11593.

[105] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. "Quality Diversity: A New Frontier for Evolutionary Computation". In: *Frontiers in Robotics and AI* 3 (2016), p. 40.

[106] Nemanja Rakicevic, Antoine Cully, and Petar Kormushev. "Policy Manifold Search: Exploring the Manifold Hypothesis for Diversity-Based Neuroevolution". In: *Proceedings of the Genetic and Evolutionary Computation Conference* (2021), pp. 901–909. DOI: 10.1145/3449639.3459320.

[107] A Rawal and R Miikkulainen. "Evolving Deep LSTM-based Memory Networks Using an Information Maximization Objective". In: *GECCO 2016*. 2016, pp. 501–508.

[108] Aditya Rawal and Risto Miikkulainen. "From Nodes to Networks: Evolving Recurrent Neural Networks". In: *CoRR* abs/1803.04439 (2018). arXiv: 1803.04439. URL: http://arxiv.org/abs/1803.04439.

[109] Esteban Real et al. "Regularized Evolution for Image Classifier Architecture Search". In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'19/IAAI'19/EAAI'19. Honolulu, Hawaii, USA: AAAI Press, 2019. ISBN: 978-1-57735-809-1. DOI: 10.1609/aaai.v33i01.33014780. URL: https://doi.org/10.1609/aaai.v33i01.33014780.

[110] Ingo Rechenberg. "Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution". PhD thesis. Technical University of Berlin, 1971.

[111] Mark Ridley. *Evolution*. John Wiley & Sons, Ltd., 2000.

[112] S Risi and K O Stanley. "A unified approach to evolving plasticity and neural geometry". In: *The 2012 International Joint Conference on Neural Networks (IJCNN)*. June 2012, pp. 1–8.

[113] Sebastian Risi, Charles E Hughes, and Kenneth O Stanley. "Evolving Plastic Neural Networks with Novelty Search". In: *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems* 18 (6 Dec. 2010), pp. 470–491.

[114] Sebastian Risi and Kenneth O. Stanley. "Deep neuroevolution of recurrent and discrete world models". In: *Proceedings of the Genetic and Evolutionary Computation Conference* (2019).

[115] Sebastian Risi and Kenneth O Stanley. "Indirectly Encoding Neural Plasticity As a Pattern of Local Rules". In: *Proceedings of the 11th International Conference on Simulation of Adaptive Behavior: From Animals to Animats*. Springer-Verlag, 2010, pp. 533–543. ISBN: 3-642-15192-2, 978-3-642-15192-7. URL: http://dl.acm.org/citation.cfm?id=1884889.1884945.

[116] Sebastian Risi et al. "How Novelty Search Escapes the Deceptive Trap of Learning to Learn". In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2009, pp. 153–160. ISBN: 978-1-60558-325-9. DOI: 10.1145/1569901.1569923.

[117] Olympia Roeva, Stefka Fidanova, and Marcin Paprzycki. "Influence of the population size on the genetic algorithm performance in case of cultivation process modelling". In: *2013 Federated Conference on Computer Science and Information Systems*. 2013, pp. 371–376.

[118] Frank Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain". In: *Psychological Review* (1958), pp. 65–386.

[119] Franz Rothlauf. "Representations for Genetic and Evolutionary Algorithms". In: *Representations for Genetic and Evolutionary Algorithms*. Vol. 104. 2006, pp. 73–96. DOI: 10.1007/978-3-642-88094-0.

[120] Marc Rothmann and Mario Porrmann. "A Survey of Domain-Specific Architectures for Reinforcement Learning". In: *IEEE Access* 10 (2022), pp. 13753–13767. DOI: 10.1109/ACCESS.2022.3146518.

[121] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986, 318–362. ISBN: 026268053X.

[122] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. "Autotuning Configurations in Distributed Systems for Performance Improvements Using Evolutionary Strategies". In: July 2008, pp. 769 –776. ISBN: 978-0-7695-3172-4. DOI: 10.1109/ICDCS.2008.11.

[123] Tim Salimans et al. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. 2017. DOI: 10.48550/ARXIV.1703.03864. URL: https://arxiv.org/abs/1703.03864.

[124] A Sankaranarayanan and M Vidyasagar. "A new path planning algorithm for moving a point object amidst unknown obstacles in a plane". In: *Proceedings., IEEE International Conference on Robotics and Automation*. May 1990, 1930–1936 vol.3.

[125] A Sankaranarayanar and M Vidyasagar. "Path planning for moving a point object amidst unknown obstacles in a plane: a new algorithm and a general theory for algorithm development". In: *29th IEEE Conference on Decision and Control*. Dec. 1990, 1111–1119 vol.2.

[126] Shaojie Shen and Vijay Kumar. "3D Indoor Exploration with a Computationally Constrained MAV". In: *Robotics: Science and Systems*. 2011.

[127] D Shorten and G Nitschke. "Evolving Generalised Maze Solvers". In: *Applications of Evolutionary Computation*. Ed. by Antonio M Mora and Giovanni Squillero. Springer International Publishing, 2015, pp. 783–794. ISBN: 978-3-319-16549-3.

[128] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 354023957X.

[129] B. Sigl, M. Golub, and V. Mornar. "Solving timetable scheduling problem using genetic algorithms". In: *Proceedings of the 25th International Conference on Information Technology Interfaces, 2003. ITI 2003.* 2003, pp. 519–524. DOI: 10.1109/ITI.2003.1225396.

[130] Jascha Sohl-Dickstein et al. "Deep Unsupervised Learning using Nonequilibrium Thermodynamics". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 2256–2265. URL: https://proceedings.mlr.press/v37/sohl-dickstein15.html.

[131] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. "Learning Structured Output Representation using Deep Conditional Generative Models". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/8d55a249e6baa5c06772297520da2051-Paper.pdf.

[132] Andrea Soltoggio et al. "Evolutionary Advantages of Neuromodulated Plasticity in Dynamic, Reward- based Scenarios". In: *ALIFE* (2008).

[133] Kenneth Stanley et al. "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1 (Jan. 2019). DOI: 10.1038/s42256-018-0006-z.

[134] Kenneth O. Stanley. "Compositional pattern producing networks: A novel abstraction of development". In: *Genetic Programming and Evolvable Machines* 8 (2007), pp. 131–162.

[135] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. "A Hypercube-based Encoding for Evolving Large-scale Neural Networks". In: *Artificial Life* 15.2 (Apr. 2009), pp. 185–212.

[136] Kenneth O Stanley and Risto Miikkulainen. "A Taxonomy for Artificial Embryogeny". In: *Artificial Life* 9.2 (2003), pp. 93–130. URL: http://nn.cs.utexas.edu/?stanley:alifej03.

[137] Kenneth O Stanley and Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies". In: *Evolutionary Computation* 10 (2 2002), pp. 99–127.

[138] Felipe Petroski Such et al. "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning". In: *CoRR* (2017). URL: https://arxiv.org/abs/1712.06567.

[139] Lei Tai and Ming Liu. "Towards Cognitive Exploration through Deep Reinforcement Learning for Mobile Robots". In: *CoRR* (2016). URL: https://arxiv.org/abs/1610.01733.

[140] K Taylor and S M LaValle. "I-Bug: An intensity-based bug algorithm". In: *2009 IEEE International Conference on Robotics and Automation*. May 2009, pp. 3981–3986.

[141] Chen Tessler et al. "A Deep Hierarchical Approach to Lifelong Learning in Minecraft". In: *CoRR* (2016). URL: https://arxiv.org/abs/1604.07255.

[142] Sebastian Thrun, Mark Diel, and Dirk Hähnel. "Scan Alignment and 3-D Surface Modeling with a Helicopter Platform". In: *Field and Service Robotics: Recent Advances in Reserch and Applications*. Ed. by Shin'ichi Yuta et al. Springer Berlin Heidelberg, 2006, pp. 287–297.

[143] Nihat Engin Toklu et al. *EvoTorch: a scalable evolutionary computation library based on PyTorch*. 2022. URL: https://github.com/nnaisense/evotorch (visited on 10/30/2022).

[144] Paul Tonelli and Jean-Baptiste Mouret. "On the Relationships between Generative Encodings, Regularity, and Learning Abilities when Evolving Plastic Artificial Neural Networks". In: *PLOS ONE* 8.11 (Nov. 2013), pp. 1–12. DOI: 10.1371/journal.pone.0079138. URL: https://doi.org/10.1371/journal.pone.0079138.

[145] Alan Mathison Turing. *Intelligent machinery*. 1948.

[146] Pascal Vincent et al. "Extracting and composing robust features with denoising autoencoders". In: Jan. 2008, pp. 1096–1103. DOI: 10.1145/1390156.1390294.

[147] Richard A Watson and Eörs Szathmáry. "How Can Evolution Learn?" In: *Trends in Ecology & Evolution* 31 (2 Feb. 2016), pp. 147–157. ISSN: 0169-5347. DOI: 10.1016/j.tree.2015.11.009.

[148] P. J. Werbos. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". PhD thesis. Harvard University, 1974.

[149] Daan Wierstra et al. "Natural Evolution Strategies". In: *Journal of Machine Learning Research* 15.27 (2014), pp. 949–980. URL: http://jmlr.org/papers/v15/wierstra14a.html.

[150] Dennis Wilson, Kalyan Veeramachaneni, and Una-May O'Reilly. "Cloud Scale Distributed Evolutionary Strategies for High Dimensional Problems". In: *Applications of Evolutionary Computation*. Ed. by Anna I. Esparcia-Alcázar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 519–528. ISBN: 978-3-642-37192-9.

[151] William Wong et al. *Optimizing Industrial HVAC Systems with Hierarchical Reinforcement Learning*. 2022. DOI: 10.48550/ARXIV.2209.08112. URL: https://arxiv.org/abs/2209.08112.

[152] D Wooden et al. "Autonomous navigation for BigDog". In: *2010 IEEE International Conference on Robotics and Automation*. May 2010, pp. 4736–4741.

[153] Qi-Lei Xu and Gong-You Tang. "Vectorization path planning for autonomous mobile agent in unknown environment". In: *Neural Computing and Applications* 23 (7 Dec. 2013), pp. 2129–2135.

[154] Y. Yacoby, W. Pan, and F. Doshi-Velez. "Failures of Variational Autoencoders and their Effects on Downstream Tasks". In: *ICML Workshop on Uncertainty in Deep Learning* 1 (2020), pp. 1–39.

[155] Yuke Zhu et al. "Target-driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning". In: *CoRR* (2016). URL: https://arxiv.org/abs/1609.05143.

[156] Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. "Transfer Learning in Deep Reinforcement Learning: A Survey". In: *CoRR* abs/2009.07888 (2020). URL: https://arxiv.org/abs/2009.07888.