Solving XOR and Pole-balancing problems using a multi-population NEAT

by
William Michael Lawrence

A thesis presented for the degree of
Master of Philosophy

Centre of Computational Intelligence
Faculty of Computing, Engineering and Media
De Montfort University
UK

December 2020

Formatted for LaTeX
by MultiMarkdown

CONTENTS

Contents iii

1 Abstract 1

2 Acknowledgements 2

3 Introduction 3

4 Structure of Thesis 6

5 Literature Review 7
- 5.1 Evolution . . . 7
  - 5.1.1 Cooperative Coevolution . . . 8
  - 5.1.2 Representation . . . 8
- 5.2 Artificial Neural Networks . . . 9
  - 5.2.1 Catastrophic forgetting . . . 10
- 5.3 Neuroevolution . . . 10
  - 5.3.1 NEAT . . . 11
  - 5.3.2 Modular Neuroevolution . . . 13
  - 5.3.3 Neuro Evolution in Computer Games . . . 13
  - 5.3.4 Deep Learning . . . 14
  - 5.3.5 Evolving Deep Neural Networks . . . 15
- 5.4 Multi-population . . . 15
  - 5.4.1 Multistage Evolution . . . 16
- 5.5 Fitness . . . 16

6 Proposed Work 18
- 6.1 Contribution to Knowledge . . . 19

7 Proposed System 21
- 7.1 NEAT Python . . . 21
- 7.2 Pole balancing problem . . . 21
- 7.3 The XOR problem . . . 22
- 7.4 XOR and Polebalancing set up . . . 22
- 7.5 System stages . . . 24

8 Results 26
- 8.1 XOR problem . . . 26

8.2 Double Pole Balancing 29

9 Discussion 31

10 Further Work 32

11 Conclusions 33

12 Appendix 34

Bibliography 35

1. ABSTRACT

This work looks at the use of multi-population utilisation for Neuroevolution in evolving controllers for the XOR and pole balancing problems. The single population is split into a number of smaller populations with a lower fitness thresholds the fittest individuals from each sub-population are then seeded into the final population with a higher fitness threshold. The results so far are inconclusive that a number of smaller populations are more efficient than a single large population. Different results were compared by using the total number of generations multiplied by the population to give an approximate metric of resources required to solve the problems using different methods. Small populations of around 5–10 individuals showed comparable results in resources required to reach desired level of fitness. There are a large number of parameters to consider when designing with Neuroevolution, multi-populations increase these factors by a considerable magnitude. It may be that further changing of parameters may yield better results. Future work using novelty search may suit multi-population applications better allowing a greater coverage of the search space.

2. ACKNOWLEDGEMENTS

I would like to thank my first and second supervisors Dr. Mario Gongora and Prof. Shengxiang Yang for their support and advice. I would also like to thank Dr. Chigozirim Justice Uzor for his help with the Python programming language. Finally I would like to thank my family for their support and understanding during my studies.

3. INTRODUCTION

Evolution has demonstrated how great innovation has happened through natural selection over many life cycles. Evolutionary computing has looked to mimic this cycle in a much accelerated time frame although simple simulations can be run on a normal PC in a minute or two, research labs often run more complex evolutionary simulations on server farms for a day or more. A population of solutions survive by being the most successful at solving a particular problem then producing offspring which may be more effective at solving the particular problem.

Neural networks are a simplified model of neurons in the brain. Typically they are made up of a number of layers artificial neurones connected together. The network is trained on a particular problem, this calibrates weights between the neuron connections. Once the network has been trained it can used on a task. As well as being able to interpolate it can also to some extents extrapolate to solve particular problems.

Both Neural Networks and evolutionary computing have strengths and weaknesses. Evolutionary Computing is good at finding innovative solutions to problems such as developing an antenna for space flight [25] While neural networks are traditionally very strong with pattern classification and recognition. Evolutionary computing may get stuck in local maxima unable to pull itself away from what looks like quite a good solution . We all forget things over time particularly if we do not require or recall the information for a while, how-

ever simple neural networks have traditionally suffered from what is known as catastrophic forgetting. Once trained for a particular task if they are then retrained for a different task, they completely forget the first task. There are many parameters which can be changed for a neural network. Although much work has been done to investigate and optimise these parameters, it is not always straight forward to predict a changes effect on a network.

It was therefore suggested that evolution may be able to help optimise neural networks - thus Neuroevolution was born. A very successful example of Neuroevolution is Neuroevolution Augmenting Topologies (NEAT)s. One of the first applications was as game AI in a game called Nero. [32] rNEAT soldiers could be trained then demonstrated their training in battle. The game showed how the AI was able to evolve various human like activities such as taking cover and appearing to shoot before hiding again. The AI players were also able to develop team behaviour. RNEATs where shown to be very effective at evolving solutions for a wide variety of applications.

Assistant Professor Jeff Clune and the Evolving Artificial Intelligence Laboratory at the University of Wyoming has further taken this work to a new level. They looked at how most things in nature showed modularity and hierarchy. They hypothesised that the reason for this was the cost of connections. Traditional neural networks do not have a cost for connections, therefore the network connections become messy and interconnected unlike in nature where many organisms including the brain show modularity and hierarchy. Once a cost for connections was made the networks were much simpler and modular. They also showed an elegant structure. Although the work is still in its early

stages when they retrained the network on a different problem they started to see some residual remembering of previous solutions [10].

Neuroevolution aims to maintain diversity of the population by often using speciation [36]. Speciation groups similar gene structures together and restricts mating between different gene structures. This preserves diversity in the population and allows niche populations time to develop rather than allowing the population be dominated by an initially strong solution. By combining speciation with multi-populations this work investigates if a further search space can be preserved to create a greater diversity. Speciation is discussed further in the literature review. In the natural world physical barriers such as seas, temperature and altitude have supported a diversity of animals adapted for their particular environment. This work investigates whether similar affects can be created using a multi-population solution.

Multi-population methods have proven to be efficient at solving a wide variety of real world problems and have been used in fields such as mathematics and engineering and areas such as scheduling, path planning, control and estimation. [14]

This work investigates using multi-population techniques with Neuroevolution. Although the Neuroevolution uses speciation to reduce premature convergence, multi-population provides a new dimension to maintain the diversity of solutions. This work discusses whether a multi-population Neuroevolution implementation can solve XOR and pole-balancing benchmark tests more efficiently than a single population. The NEAT implementation of Neuroevolution is used. The software implementation of the NEAT is first ran as a

standard single population implementation. The code is then amended to run as a multi-popuation NEAT. All other parameters except population size are kept the same. The control implementation of the NEAT is compared with the Multi-population implementation by comparing the total individual population cycles to give an indication of which method is more efficient at solving the above benchmark problems. This comparison is discussed in further detail in later sections.

4. STRUCTURE OF THESIS

This work first looks at a review of the relating technologies and a brief overview of Artificial Neural Networks which are central to Neuroevolution. A broader discussion on types of evolution then follows with more information specifically on Neuroevolution and NEAT the particular type of Neuroevolution used in this work. Multi-population is then discussed and how it has been used successfully in a range of evolutionary and other computational intelligence methods. The proposed work of how multi-population can be successfully used in Neuroevolution is then explored. Information is then shared regarding the technical implementation of a multi-population NEAT. The results of implementation are shown against benchmarks problems XOR and Pole-balancing problems and further detail given regarding these benchmarks. The results are then given from various testing runs using these benchmarks. The work concludes with a discussion of the results and conclusion of the findings with some recommendations for further work.

5. LITERATURE REVIEW

Neuroevolution is the combining of Neural networks with evolution, in this section an overview of these important areas to the discipline is given. This is followed by a review of Neuroevolution and the development of this area and sub genres. There is then a discussion on the breadth of use of multi-population as it has been used in various disciplines of machine learning. The section concludes with a review of the fitness function as an important aspect of population learning.

5.1 Evolution

Neuroevolution is built on a history of evolutionary problem solving which has had some remarkable successes, below outlines some of the many developments in this broad field.

Evolutionary algorithms are a simplified copy of Darwinian evolution. A problem is posed. The solutions are codified into genotypes and phenotypes. A fitness function is derived. What does a slightly good solution look like, what does a better solution look like what would the best solution look like. Once we can create a graded fitness function for our problem we are well on the way to solving it with a Evolutionary algorithm.

One of the first and simplest evolutionary methods is the Genetic algorithm. A problem is coded as above and then the fitness function is applied. Those individuals with the highest fitness get the best chance to reproduce those

with the lowest fitness are least likely to get much of their DNA into the next generation. Genes are crossed over between individuals and offspring are created. Genes Mutate to bring in variety. This is a random process which sometimes brings about an increase in fitness but other time produces a reduction in fitness. All Genetic algorithms use a process of increasing probability for reproduction for fitter individuals and an increased probability of death for the least fit individuals. This is carried out on a random basis so the population as a whole gets progressively fitter towards some objective.

Fascinating work has been done in the area of Genetic Programming. John Koza is a pioneer in this area and has done a huge amount of work to progress the field [12]. Genetic Programming has produced many inventions and has led to machines holding patents for a number of inventions in use today. Genetic Programming has also been used for the study of evolution in such projects as Avida [20] and Tierra. What struck me about much of this work is when the solution are animated how much they are like bacteria or insects.

Thomas Ray created an artificial life simulator to model evolution [23] He argued that we only had one example of evolution to base our understanding of how evolution worked which is here on earth. He speculated that evolution could be very different for other lifeforms. By synthesising digital organisms, Thomas was able to demonstrate how evolution worked in an alternative setting from studying evolution of natural life on earth. The great advantage of this evolution was it could be studied over many thousands of generations instead of the relatively stationary snapshot of evolution which can be seen in biological evolution.

5.1.1 Cooperative Coevolution

In game playing competitive coevolution has been used to train AI players. However cooperative coevolution has been used less but can produce modularised solution

5.1.2 Representation

The representation of an Genetic Algorithm is made up of a genotype which is the Genetic representation and the phenotype which is the implementation of the solution. The simplest encoding is a direct representation which maps 1 to 1 genotype to phenotype. However in nature this is not how chromosomes are encoded. Indirect encoding is used which can reduce the number of genotypes required for a given phenotype. This can reduce complexity and increase success rate of the Genetic Algorithm. In a comparative study of direct and indirect coding for school timetabling direct encoding were shown to have a 60% - 67% chance of success, while indirect encoded solutions were shown to have a 93% - 100% chance of success [22]

5.2 Artificial Neural Networks

An essential element of any NeuroEvolution system is the Neural Network. An Artificial Neural Network (ANN) has been defined as

"... a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state

response to external inputs."

In "Neural Network Primer: Part I" by Maureen Caudill, AI Expert, Feb. 1989

ANNs are loosely based neuronal structure of the mammalian cerebral cortex. They are normally made up of inter connected layers of artificial neurons. They have an input layer, hidden layers and an output layer. Rather than being programmed they are trained to solve particular problems. Each neuron sums weighted inputs to produce an output if the sum is above a certain threshold. The output then feeds into the next layer of the network. Training data with solutions is used and the weights adjusted automatically to minimise the error. Once a network has been trained on a particular problem it is then able to provide solutions to unseen problems.

Neural networks provide distinct advantages in areas where problems are so complex that it is difficult to model them with traditional computing methods. Pattern recognition is one such example. Speech and handwriting recognition is difficult for traditional computing methods as everyones handwriting and speech is very slightly different for the same words. Taking a step back and looking at the whole pattern and maybe making a little guess is something which is intuitively learnt by the human brain and can be modleled with a neural network but it very difficult to capture in a logical strategy using the current state of logic computing. [9]

By combining Neural Networks with Evolutionary Algorithms, Neuroevolution enables evolution to support with the difficult task of weight and topology setting while evolution gives a suitable vehicle to support with storing weights

from one generation to the next, which can easily be lost with training of Neural Networks which can lead to catastrophic forgetting.

5.2.1 Catastrophic forgetting

Catastrophic forgetting has been a long time problems of neural networks which has hampered their usablity and reuseablity for new tasks. Essentially a standard neural network will forget most of its learning on a task when a new task is learned. This means that a neural network cannot be used as a general solver or even complete a series of different tasks. This has limited applications to a single problem domain per network.

Recently there has been a number of breakthroughs in this area. One approach has been to slow down learning of nodes important for previous tasks by using elastic weight consolidation. [11]. The amount a weight can change will depend on how important it is to a task. This approach has been used successfully with sequential tasks on Atari games. An alternative approach in neuromodulation has been to emulate diffusion used in the brain. The experiment used was a foraging task. Extra information on the season was inferred through diffusion and this lead to different behaviours being recalled for different seasons as required [38]

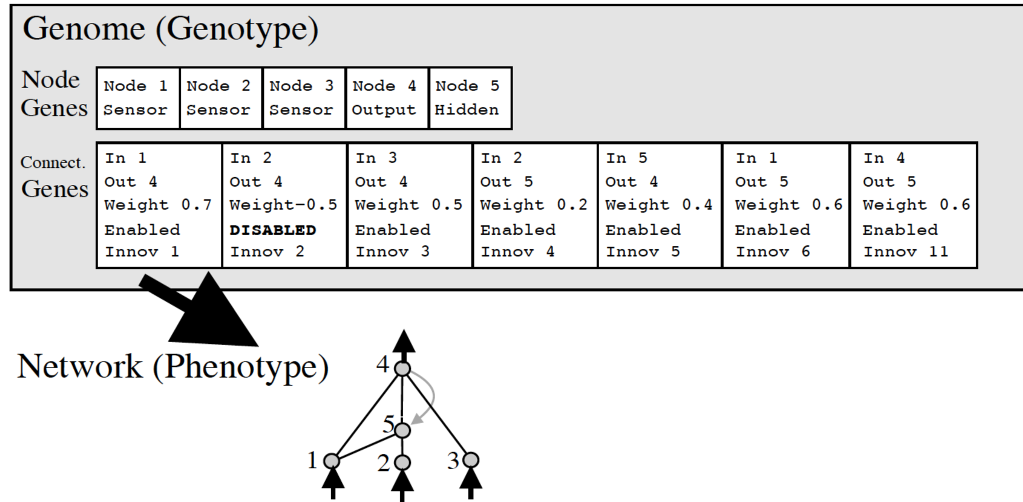
5.3 Neuroevolution

A major inspiration for the investigation of Neuroevolution is the evolution of brains in nature. Neuroevolution uses evolutionary algorithms in the design

and training of Neural Networks. There are so many interacting changing variables that it can be difficult to optimise a neural network. Also it can be difficult and time consuming to train deep neural networks. By using evolution some or all of these parameters can be optimised. Some algorithms only evolve neural weights while others evolve weight and structure. A distinction can also be made between those which evolve weights and structure together and those which evolve them separately (through memetic algorithms). In direct encoding schemes the genotype maps directly to the phenotype while in indirect encoding schemes the scheme encodes how the neural network should be generated. Neuroevolution of Augmenting Topologies (NEAT) directly encodes the genotype it tracks genes to allow crossover between different topologies and creates innovation by specification. This protects phenotypes with more complex structures from being broken by crossover with other This work has been further expanded by using indirect encoding HyperNEAT. Most recently Spectrum-diverse Unified Neuroevolution Architecture (SUNA) uses a diversity preserving strategy to reduce the convergence of solutions, which is a typical problem with evolutionary methods. Many Machine learning algorithms perform qualitatively differently and far better at greater scale. This has been shown to be the case with Neuroevolution. Neuroevolution is now shown to be a competitive alternative to gradient based methods for Neural Network training [30]

5.3.1 NEAT

The NEAT Evolutionary Algorithm generates and evolves Neural Networks using a Genetic Algorithm it's unique contribution to neuroevolution is that it tracks genes with historical markings. This allows crossover between similar topologies but protects innovation by speciation. NEAT was developed by Ken Stanley in 2002 [29] During the 1990s a number of Topology and Weight Evolving Artificial Neural Networks (TWEANNs) were developed. The encoding either uses direct encoding or indirect encoding. Direct encoding specifies every connection in the genotype in the phenotype. Indirect encoding usually specifies rules for encoding so one genotype can specify a number of phenotypes. Indirect encoding allows for more complex evolution, however their methods of encoding can lead to more complex issues of bias due to the particular method of indirect encoding used. NEATs use direct encoding for this reason. NEATs have node genes and connection genes. The genetic encoding scheme is design to allow corresponding sense to be easily lined up when two genomes crossover during mating [31] shown below in an extract from the original paper [29]



The original set up of NEATs is a network with no hidden nodes, the evolving structure then adds nodes as required to keep the structure as simple as possible.

Graph encoding is often a better representation of a network and it is argued it allows for better crossover. To reduce destructive effects of crossover changing the structure of the genotype too radically genes are tracked with a global innovation number. Every new gene created is given a new number. It it then possible to know how to line up genes so incremental changes are made instead of huge often destructive changes which would completely change how the system operates.

Speciation is used to support variety being preserved in the population. Species are determined by calculating the number of disjoint and and excess genes between a pair of genomes. From this the value of separation is calculated. The species are grouped by distance from each other. If a genome does not fit into an existing group, a new group is formed. Explicit fitness sharing

within the species stops any group from getting too large and dominating the group. Some preliminary research shows that NEATs perform much worse without speciation [19].
The NEAT method was validated on polebalancing tasks where it it performed 25 time faster that Cellular Encoding and 5 times faster than ESP [31]

Unlike other TWEANNs NEATs start out with zero hidden nodes and they are only added as required. This minimises the search space and makes NEATs more efficient. RtNeat evolves in real time and were used for training soldiers in the game Nero [32] CPPN NEAT create increasingly complex Compositional pattern-producing networks CPPN NEAT not only performs the optimising function of evolution but also the complexity function, allowing solutions to become more complex as required. As they can add further complexity to an already optimised structure this gains the advantage of speed. The complexification can also help the NEAT escape local minimal by changing the search landscape itself by forming a new structure. Alabation studies have proved that a NEAT requires historical markings, speciation and incremental growth from minimal structure to fully reap the above benefits of efficient evolution of network structure. [31]

5.3.2 Modular Neuroevolution

Modular Evolution has been successfully used for multi-legged locomotion. The modular approach produced animals with symmetrical gait, while conventional controllers tend not to deal with well with the variety of terrain required and non-modular solutions usually produce an irregular gait more

similar to that seen in crippled animals. The modular design lead to better fitness and travelling further [37] In computer game agent design the modular approach is used to break the agent control problem into sub problems to increase convergence rates and efficacy of evolution. [24]

5.3.3 Neuro Evolution in Computer Games

Neuroevolution has been used widely in games [26]. The main areas it has been used is state action selection, direct action selection, modelling player experience, content generation and strategy selection. The domains and applications are a wide array but generally neuroevolution has been used because it has produced record-beating performance in many areas. Neuroevolution also has broad applicablility, it can be used for supervised, unsupervised and reinforcement tasks. All neuroevolution requires is some sort of numeric evaluation of the current trial. Neuroevolution has also proved to be scalable and maintain diversity using a variety of methods such as niching, multiobjective methods and novelty search. When topology of the network is evolved the system may be able to support open ended evolution. Neuroevolution is also being used to generate a new generation of games which would be difficult if not impossible to develop using conventional computing methods. In these games neuroevolution is used to develop new types of weapon or create varieties of virtual flowers. These games can bring unique individuality to the game however this can bring it's own problems. Debugging and quality assurance can be difficult to achieve using current methods when the games code is essentially a black box. It may require new (evolving?) tools to test and debug

neuroevolving software. Neuroevolution is most often used to play a game or control a NPC in the game. Neuroevolution has also been successfully used for procedural Content Generation (PCG). Neuroevolution has also been used to predict experience or preference of players. The earliest use of neuroevolution was in state evaluation for deciding the value of playing certain moves. Fogel designed an architecture which taught itself to above master level by playing against itself [7]. Some games are too complex to be able to predict future moves and outcomes even with powerful computers. Games like Civilisation have just too many possible options to be able or the forward model is stochastic. Neural Networks are still able to make a decision from learnt behaviour of similar situations before. Some games one must choose a longer term strategy, Whiteson et. Al. Used NEAT hybridised with other forms of reinforcement was very effective at stategy selection [39].

5.3.4 Deep Learning

Deep learning has been successfully employed to create expert Atari games players by DeepMind. Using a convolutional Neural Network with a variant of Q learning and just the raw visual data from the screen; the strategy was able to out perform neuroevolution (Beam Rider, Breakout, Enduro, Pong and Seaquest) and human experts on 3 games (Breakout, Enduro and Pong) [18] Deep learning has also been used to demonstrate multiagent co-operative and competitive learning with the Atari game of Pong. [34]

5.3.5 Evolving Deep Neural Networks

DeepNEAT shows that the evolutionary approach can be used to optimize Neural Networks [17]. The work shows that it is possible to construct more complex deep learning structures than can be done by hand. The topology, components, and hyperparameters of the architecture can all be optimized simultaneously.

A variation coDeepNEAT has two populations of modules and blueprints which are evolved separately. During evolution the modules and blueprints are combined together. CoDeepNEAT can evolve repetitive modular structures efficiently.

Random mutation often breaks functionality on deep neural networks due the millions of weights it is required to change. It is impossible to ascertain whether some of the mutation has been beneficial and if so which parts of the mutations should be made to yield the best results. By calculating the gradient of the change which can be made according to the sensitivity of the networks output to that weight, safe mutations can be made to the network [13]

5.4 Multi-population

Multi-population has been successfully used in a variety of algorithms such as Genetic Algorithms [8] Differential evolution [40] and genetic programming (GP) [15]. A comprehensive survey of multi-population methods asserts that multi-population methods can be applied to practically any optimization problem domain [14].

Multi-population techniques have proved effective

1. They can maintain diversity as different populations can be in different search spaces
2. they can allow simultaneous searching of different search areas
3. It is possible to easily embed different algorithms into different populations [14].

Both fixed and variable numbers of subpopulations have been found to be effective. If there are too many subpopulations it is wasteful of resources. However too few are not found to be effective [14] Many systems use subpopulations to solve different parts of a task such as [35] a three population architecture was used to solve non-stationary optimisation tasks, one population supplied historic estimates, while others were used in the estimation process. Swarm and slave architectures have also been successful implemented [1]

5.4.1 Multistage Evolution

Multistage evolution has been shown to maintain diversity and prevent premature convergence [4] A form of multistage evolution has been used for timetabling problems. Superior results were obtained by splitting the problem into sub problems and using separate Genetic Algorithms to solve them [3]

5.5 Fitness

The traditional fitness function usually evaluated the fittest individual as the gold standard in any generation and guides evolution in that direction. This single pointed strategy has been very successful in the early days of artificial evolution for optimising but has a number of drawbacks as the tasks have become more complicated. Crowding of similar individuals leading to premature convergence. The single pointed approach weeds out diversity, particularly if it does not show promise in the early stages of its development. Strategies tend to look for immediate pay off which can lead to solutions getting caught in local minima.

In the last few years a number of groups have started looking at alternatives to this single pointed fitness function. Pic breeder [27] developed at the University of Florida, is an online picture breeding utility which enabled users to breed a picture from other pictures. They made a remarkable discovery that people discovered amazing pictures by selecting something interesting and the results were rarely what was predicted. They further found that actually trying to create something by directly evolving for it was almost impossible. Stanley stated that there were a series of stepping stones which almost never led where you would predict. He further proved this by trying to evolve one of the discovered images by objective driven search. He found that an image which had been discovered after 76 iterations could not be developed by 30,000 cycles of development. Stanley concluded that the results for this study has implications for the study of human development beyond computer

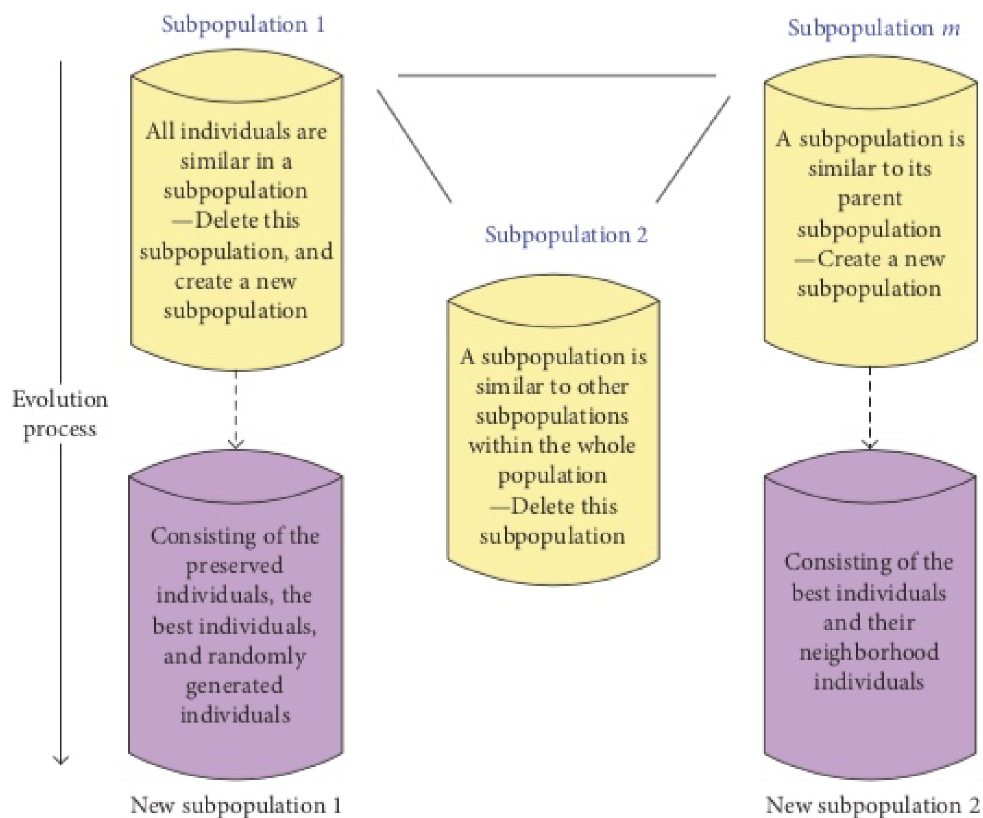
science, and organisations would need to review their objective based search if they were truly looking for innovation. From this was developed the idea of Novelty Search. If an strategy was rewarded for finding something new within the search space instead of always looking for minute increments in fitness towards a set goal, many useful diverse solutions can be found for a problem.

Curiosity search [33] worked in a similar way to Novelty search except instead of comparing behaviours with others in the population it rewarded individuals for gaining new skills and abilities compared with what that individual had previously achieved. It was argued that while Novelty search produced specialists, Curiosity search produced generalists.

Google's Deep Mind used curiosity and reward to solve the notoriously difficult Atari game Montezuma's Revenge. Early attempts at Cornell University to get DeepMind to complete the game resulted in the AI giving up easily and scoring no points but after teaching the AI a sense of reward for curiosity, it began to perform and became 'interested' in seeing what was in the next room [16].

6. PROPOSED WORK

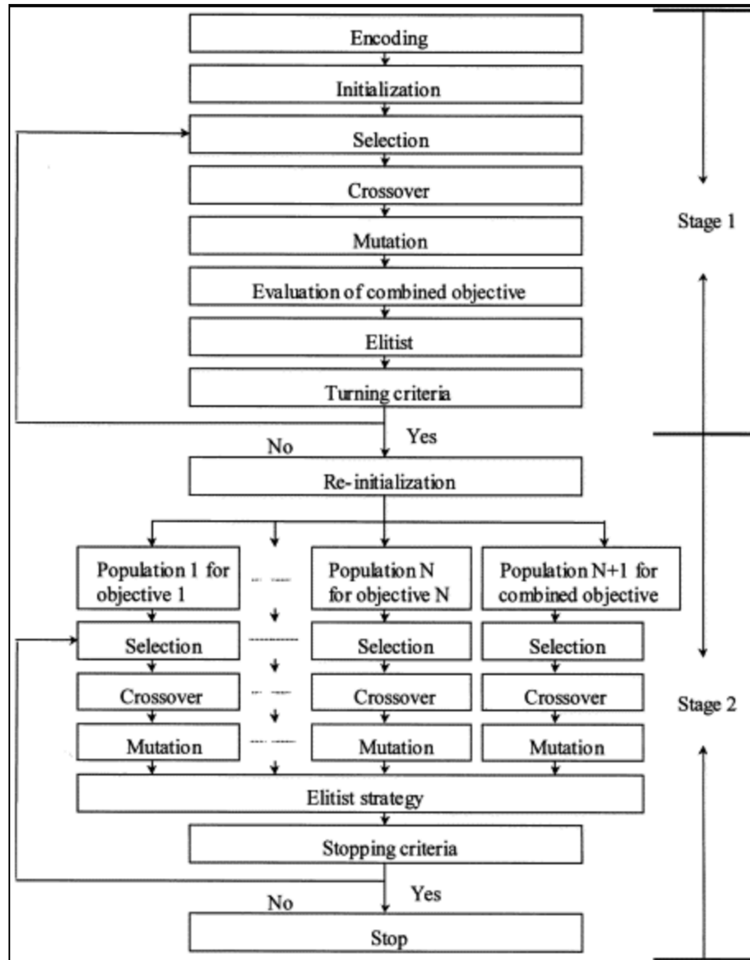
Many different sub-population approaches have been successful. EAs have shown superior results having dynamically managed sub populations which are created and deleted during the evolution process [42]



Multi-populations have been successfully used in Differential evolution by using and ensemble of mutation operators and a larger reward population for the most successful sub-population. [40]

A multi-agent population genetic algorithms have been shown to show su-

perior results than single populations when used for multi objective problems. [5] The MOGA uses a first generation to seed subsequent populations. This research uses this same successful formula for seeding population for NEAT populations.



Small sub populations feed a final population. Lower fitness thresholds are used on the first populations then a higher fitness threshold is used on the final population. A NEAT implementation in Python Is used to test using multi-populations compared with standard populations with a simple XOR

problem and a Pole balancing problem.

6.1 Contribution to Knowledge

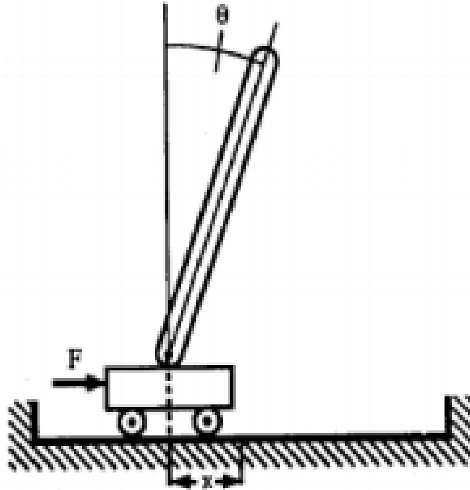
This work looks as the advantages of combining speciation with multiple populations. It uses and survival of the fittest metric to develop populations of certain sizes. Multi-populations have been used successfully for solving multi objective problems [5] and may be able to solve a wider diversity of problems. The interplay between speciation and multi-populations could open up new areas of research.

7. PROPOSED SYSTEM

7.1 NEAT Python

NEAT-Python was written by CodeReclaimers as a universal Neat implementation [28]. The code has been successfully used by CodeReclaimers to solve the XOR, pole balancing and many other problems and simulations. The code has been amended to run multiple populations and the performance compared with the original Neat implementation. The aim is to show to show that the multiple populations can produce better results per cycle of computational power used.

7.2 Pole balancing problem



The Pole balancing problem (also known as the pole cart and inverted pendulum) is a control problem which has been used as a common pseudo benchmark problem for testing AI controllers. The object of the controller is to balance the pole for a set length of time. The system requires a closed-loop feedback controller which applies a positive or negative force to the cart. The feedback signal is the angle at which the pole is to to the vertical. The target for the controller is to keep the pole balanced at vertical for as long as possible. [2]

Historically two issues have been found with the equations for the Pole balancing problem. An error has been found in the way the friction force is calculated and the direction of gravity which has been incorrectly stated as minus. [6] The example in the NEAT Python example implementation in this work uses the corrected equations. The incorrect equations still describe a dynamic system so work based on these systems is not invalid, however it may

not be comparable with results gained from the corrected equations. However this is also a general issue within the AI community that there is a lack of standardisation of benchmark tests so different studies cannot be compared.

7.3 The XOR problem

The exclusive or (XOR) is a classification problem which is often used for testing Neural Networks. The network aims to produce the outputs A XOR B for the inputs A, B in the following truth table

Table 1: XOR Problem

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Due to the activation function exact 0 and 1 cannot be generated so an approximation is use usually a tolerance of 0.1 is allowed [21] The XOR problem is a linear un-division operation, which cannot be solved by a single-layer perceptron. They can be solved by multilayer perceptron. Multilayer neural networks can always sieve the XOR problem but at cost of complication of the system. The functional perceptron and quadratic functional perceptron were also able to solve the XOR problem. [41]

7.4 XOR and Polebalancing set up

The XOR and pole balancing problem uses a feedforward network with an elitist strategy. The default sigmoid activation function is used. Sum node aggregation is used to calculate the total at each node. Simple networks with 2 inputs and 1 output are used for the XOR problem. Stagnation operation is also used so that if evolution stops developing the population can be reseeded and restarted.

7.5 System stages

A similar set up is used for testing both problems a number of small populations feed into a final population. The speciation of the NEAT stops individual solutions converging too quickly. A overview of the main stages and cycles of the neuroevolution implementation is shown below.

For all sub populations:

- Encoding
- Randomly initialise population
- Selection
- Crossover
- Mutation
- Fitness Evaluation
- Elitest Strategy
- Run until intermediate fitness is achieved

Final population:

- Seed best individuals from sub populations
- Encoding
- Randomly initialise population

- Selection
- Crossover
- Mutation
- Fitness Evaluation
- Elitest Strategy
- Run until final fitness target achieved

8. RESULTS

8.1 XOR problem

This is a control group of single population Neuroevolution strategy however all other parameters are kept the same as for the multi-population the column for the final population size and initial fitness are blank as there is only one population and one fitness criteria. A population of 150 was run 3 times and a population of 10 was run 3 times. The total population size multiplied by the number of generations give the total individual cycles which is the metric used to compare the single population results with the multi-population results. The fitness score here is the raw genotype fitness and is calculated by doubling the total values required. So with the XOR the total value max value of the fitness is 4.

Table 2: XOR Single Population

Initial pop. size	Final pop. size	to-tal	Initial fitness	Final Fit-ness	Inital Generation complete	Genera-tion complete	Total Individual Cycles
150	-	150	-	3.9	23	23	3450
150	-	150	-	3.9	22	22	3300
150	-	150	-	3.9	85	85	12750
10	-	10	-	3.9	382	382	3820
10	-	10	-	3.9	1072	1072	10720
10	-	10	-	3.9	365	365	3560

The above table shows the total individual cycles varied significantly for each population, however were quite similar when comparing the two population

size.

The following shows 10 populations of 10 individual combined into a final population of 10 individuals. The sub populations needed to reach a fitness of 2.9 then the final population needed to reach a fitness of 3.9, matching the control population.

XOR - Small multi-population

Table 3: XOR - Small multi-population 2.9 Fitness

Initial pop. size	Final pop. size	total	Initial fitness	Final Fitness	Initial Generation complete	Generation complete	Total Generations	Total Individual Cycles
10	10	110	2.9	3.9	3887, 19, 942, 44 41,385, 106,185 623,150,	108	6098	60980
10	10	110	2.9	3.9	567, 884, 2061, 119, 208, 250,510, 117,80, 246	279	1942	53210

The initial results here show that the sub-population strategy were generally took more total individual cycles than the control groups.

The following shows 10 populations of 10 individuals combined into a final population of 10 individuals. The sub-population runs were repeated with with a higher initial fitness but the same target fitness. This effectively meant the sub-populations were doing more “work”

Table 4: XOR - Small multi-population 3.2 Fitness

Initial pop. size	Final pop. size	total	Initial fitness	Final Fitness	Initial Generation complete	Generation complete	Total Generations	Total Individual Cycles
10	10	110	3.2	3.9	61, 204, 54, 272,498,140, 395,111, 30, 54	272	2091	20910
10	10	110	3.2	3.9	234, 53, 31, 68, 404, 158, 146, 24, 129, 1072	244	2391	23910
10	10	110	3.2	3.9	465, 37, 246, 51, 190, 714, 2240, 29, 678, 366	5399	12750	53990

The above results show lower total population cycles, so it would seem that raising the initial fitness target made the system more efficient, however all results were still higher than the control group.

Table 5: Tiny multi-population

Initial pop. size	Final pop. size	total	Initial fitness	Final Fitness	Initial Generation complete	Generation complete	Total Individuals Cycles
5	2	12	3.2	3.8	1109,223	2607	11874
5	2	12	3.2	3.8	1561,1041	6232	25474

Results from the tiny sub-populations appear to further narrow the gap between multi-populations and the control population

8.2 Double Pole Balancing

Again the first results are the control group showing a single population with the total individuals included for comparison. As the problem is more complex the original example used a larger population. The fitness score is again the raw fitness score and shows the time in seconds the system was able to balance the pole for.

Table 6: Original

Initial pop. size	Final pop. size	total	Initial fitness	Final Fitness	Initial Generation complete	Final Generation complete	Total Generations	Total Individuals Cycles
250	-	250	-	60	-	57	57	14250
250	-	250	-	60	-	13	13	3250
250	-	250	-	60	-	78	78	19500

The following shows 10 populations of 10 individuals each combined into a final population of 10 individuals.

Table 7: Population 10

Initial pop. size	Final pop. size	total	Initial fitness	Final Fitness	Initial Generation complete	Generation complete	Total Generations	Total Individual Cycles
10	10	110	10	60	731, 121, 1678, 416, 92, 999, 873, 373, 283	2221	5566	55660
10	10	110	10	60	1528, 836, 73, 369, 836, 516, 975, 568, 1157	0	68580	685800
10	10	110	10	60	782, 810, 1574, 1967, 1826, 1965, 162, 839, 1705,	2937	14567	145670

These results show that having lots of small sub populations made the system much less efficient greatly increasing the number of total individual cycles to complete.

The following results are for two initial populations of 10 individuals combined together with a final population of 2 individuals.

Table 8: Double Pole Balancing

Initial pop. size	Final pop. size	total	Initial fitness	Final Fitness	Initial Generation complete	Final Generation complete	Total Generations	Total Individuals Cycles
10	2	20	10	60	263, 957	0	1220	12200
10	2	20	10	60	1411, 720.	16	2147	21326
10	2	20	10	60	1043, 457	2674	4174	19174

Following from the results of the XOR experiment much smaller populations were used and these results were much closer in total individual cycles than the control group. We can see these final results are not so different from the original single population.

9. DISCUSSION

Results were calculated using a population per generation metric. That is if there was a population of 100 and it took 100 generations to solve the problem this would be 10,000 population generations. The calculation is not completely accurate as a population of 5 running for 1000 generations probably does not use exactly the same computational power as a population of 1000 running for 5 generations, however it gives an approximate measure to compare the computational requirements of different algorithms. It can be seen that for this particular problem it does not appear that splitting the population into sub-populations has yet yielded any particular computing advantage especially when one considers that one larger population is probably more computationally efficient than two smaller ones. The simpler XOR problem was more efficiently solved with a single population, however the more complex double pole-balancing problem, multi-populations were shown to be nearly as affective. It did appear that smaller subpopulations did appear more efficient than one larger ones. Also for these problems it seemed that having fewer sub populations where more efficient. It could be that with further experimentation and optimisation of the multi-population algorithm it may be more efficient than a single population solution. It may be that the problem is too trivial to warrant such an approach. It could also be that the initial and subsequent fitnesses need to be adjusted. Another possibility is that other settings such as the elitist policy needs to be review particularly in light of using smaller

generations. Novelty search is another approach that could benefit from sub-populations as they may widen the search space.

10. FURTHER WORK

Further runs would provide a better average for determining the exact performance difference between methods, especially as the range of results for the same set up can be quite wide.

A problem with multi-populations recording performance by totalling all the population cycles is that if there is a non-performant sub population it can skew the performance of all the populations as it can run and run without finding a fit individual for many generations By stopping each population after a number of iterations we could produce survival of the fittest populations. Allowing whole search areas could be quickly evaluated before moving on.

As it appeared that multi-populations performed better with more complex problems, it would be interested to test against a multi-objective.

Novelty search is currently preferred over Curiosity search for reasons of practicality as it seemed difficult to evaluate the population of agents and identify what behaviours they have learnt. This seems an interesting area of research if one can identify distinct behaviours. It seems that it may require evolved tools to measure fitness as it is difficult to work within these evolving structures and understand how to interact with them, as stated many times before the operation of Neural Networks is likened to a black box, but maybe it is time to shine more light into this. This work paves the way for further research into continuously evolving machines, small evolution. The population could be evolved and tested in different ways. One idea that was looked at

was to measure the fitness of an individual by their absence.

11. CONCLUSIONS

This work has investigated the use of multi-populations for the NEAT algorithm. Multi-populations have been successfully used in variety of algorithms such as such as Genetic Algorithms [8] Differential evolution [40] and genetic programming (GP) [15]. Multi-population NEATs have been investigated with a number of smaller feeder populations seeding into a final population. It has been found that two or three small populations appear to be more efficient that more larger populations. The multi-population NEAT has been tested against a control single population using the XOR and Pole-balancing benchmark problems.

Multi-populations have generally been more successful with multi-objective problems, it would appear that the overhead of multi-populations cannot be more efficient for simpler problems.

While the current experiments do not show advantage to using a multi-population solution, it may be that a more complex problem or different set up may yield better results. Novelty search or other types of multi-population my create better results. By stopping whole populations after a number of generations we might be able to rule out search spaces with little promise of producing valuable results before wasting much processing on them.

12. APPENDIX

The following code shows how the NEAT code was changed to create multi-populations. The config file was also duplicated and amended to alter the population size and the fitness threshold. To see the full original NEAT code visit https://github.com/CodeReclaimers/neat-python

```
def run():
    winners = {}
    for x in range(0, 9):
        # Load the config file, which is assumed to live in
        # the same directory as this script.
        local_dir = os.path.dirname(__file__)
        config_path = os.path.join(local_dir, 'config-feedforward')
        config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                             neat.DefaultSpeciesSet, neat.DefaultStagnation,
                             config_path)
        pop = neat.Population(config)
        stats = neat.StatisticsReporter()
        pop.add_reporter(stats)
        pop.add_reporter(neat.StdOutReporter(True))
        pe = neat.ParallelEvaluator(4, eval_genome)
        winner = pop.run(pe.evaluate,1000)
        # Save the winner.
        with open('winner-feedforward', 'wb') as f:
            pickle.dump(winner, f)
        print(winner)
        winners.update({winner.key: winner})
    print('\nFinal Run')
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config-feedforward-final')
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
                         config_path)
    pop = neat.Population(config, winners)
    stats = neat.StatisticsReporter()
    pop.add_reporter(stats)
    pop.add_reporter(neat.StdOutReporter(True))
    pe = neat.ParallelEvaluator(4, eval_genome)
    winner = pop.run(pe.evaluate)
    # Save the winner.
    with open('winner-feedforward', 'wb') as f:
        pickle.dump(winner, f)
    print(winner)
    visualize.plot_stats(stats, ylog=True, view=True, filename="feedforward-
fitness.svg")
    visualize.plot_species(stats, view=True, filename="feedforward-speciation.svg")
    node_names = {-1: 'x', -2: 'dx', -3: 'theta', -4: 'dtheta', 0: 'control'}
    visualize.draw_net(config, winner, True, node_names=node_names)
    visualize.draw_net(config, winner, view=True, node_names=node_names,
                       filename="winner-feedforward.gv")
    visualize.draw_net(config, winner, view=True, node_names=node_names,
                       filename="winner-feedforward-enabled.gv", show_disabled=False)
    visualize.draw_net(config, winner, view=True, node_names=node_names,
                       filename="winner-feedforward-enabled-pruned.gv",
show_disabled=False, prune_unused=True)
if __name__ == '__main__':
    run()
```

BIBLIOGRAPHY

[1] Bi Li 0001, Tusheng Lin, Liang Liao, and Ce Fan. Genetic algorithm based on multipopulation competitive coevolution. *IEEE Congress on Evolutionary Computation*, 2008. 16

[2] J Brownlee and J Brownlee. The pole balancing problem: a benchmark control theory problem. *undefined*, pages 1–12, September 2020. 21

[3] E K Burke and J P Newall. A Multistage Evolutionary Algorithm for the Timetable Problem. pages 1–12, March 1999. 16

[4] Haradhan Chel, Deepak Mylavarapu, and Deepak Sharma. A novel multistage genetic algorithm approach for solving Sudoku puzzle. In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pages 808–813. IEEE. 16

[5] Jeffery K Cochran, Shwu-Min Horng, and John W Fowler. A multi-population genetic algorithm to solve multi-objective scheduling problems for parallel machines. *Comput. Oper. Res.*, 2003. 18, 19

[6] Razvan V Florian. Correct equations for the dynamics of the cart-pole system. pages 1–6, February 2007. 22

[7] D B Fogel, T J Hays, S L Hahn, and J Quon. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 92(12):1947–1954. 14

[8] Xiaoxia Gao, Hongxing Yang, LinLu, and Prentice Koo. Journal of Wind Engineeringand Industrial Aerodynamics. *Jnl. of Wind Engineering and Industrial Aerodynamics*, 139(C):89–99, April 2015. 15, 33

[9] Ian Goodfellow and Yoshua Bengio. Deep Learning. 9

[10] Jean-Baptiste Mouret Jeff Clune Kai Olav Ellefsen. $2015_E llefsenMouretClune_N euralModularity_P LoS. \quad pages1 - -24, 2015.4$

[11] James Kirkpatrick, Razvan Pascanu, Neil C Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *CoRR*, 2016. 10

[12] J R Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. 1990. 7

[13] Joel Lehman, Jay Chen, Jeff Clune, and Kenneth O Stanley. Safe Mutations for Deep and Recurrent Neural Networks through Output Gradients. *CoRR*, cs.NE, 2017. 15

[14] Haiping Ma, Shigen Shen, Mei Yu, Zhile Yang, Minrui Fei, and Huiyu Zhou. Swarm and Evolutionary Computation. *Swarm and Evolutionary Computation*, 44:365–387, February 2019. 4, 15

[15] Goran Mauša and Tihana Galinac Grbac. Applied Soft Computing. *Applied Soft Computing Journal*, 55:331–351, June 2017. 15, 33

[16] Chris Merriman. Google's DeepMind learns curiosity and reward to complete Montezuma's Revenge | TheINQUIRER. 17

[17] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving Deep Neural Networks. *CoRR*, 2017. 15

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv.org*, 2013. 14

[19] Timothy Nodine. Speciation in NEAT. pages 1–9, May 2010. 12

[20] Charles Ofria and Claus O Wilke. Avida: a software platform for research in computational evolutionary biology. *Artificial Life*, 10(2), April 2004. 8

[21] Abdallh Otair. Solving Xor Problem Using An Optical Backpropagation Neural Networks. pages 1–5, December 2006. 22

[22] Rushil Raghavjee Nelishia Pillay. A Comparative Study of Genetic Algorithms Using a Direct and Indirect Representation in Solving the South African School Timetabling Problem. pages 1–9, January 2014. 8

[23] Thomas S Ray. Evolution, ecology and optimization of digital organisms. *Santa Fe*, 1992. 8

[24] John Reeder, Roberto Miguez, Jessica Sparks, Michael Georgiopoulos, and Georgios C Anagnostopoulos. Interactively evolved modular neural networks for game agent control. *CIG*, 2008. 13

[25] Rick Riolo, Trent McConaghy, and Ekaterina Vladislavleva. Genetic Programming Theory and Practice VIII, 1st edition. *Genetic Programming Theory and Practice VIII, 1st edition*, October 2010. 3

[26] Sebastian Risi and Julian Togelius. Neuroevolution in Games: State of the Art and Open Challenges. *arXiv.org*, pages 1–19, 2014. 13

[27] Jimmy Secretan and Nicholas Beato. Picbreeder: evolving pictures collaboratively online. *CHI*, pages 1759–1768, June 2011. 16

[28] Kenneth O Stanley. Welcome to NEAT-Python's documentation! — NEAT-Python 0.92 documentation. 21

[29] Kenneth O Stanley. Evolving Neural Networks through Augmenting Topologies. pages 1–30, May 2002. 11

[30] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, pages 1–12, January 2019. 11

[31] Kenneth O Stanley and Risto Miikkulainen. Efficient Reinforcement Learning Through Evolving Neural Network Topologies. *GECCO ()*, 2002. 11, 12, 13

[32] K.O. Stanley, B D Bryant, I Karpov, and R. Miikkulainen. Real-time evolution of neural networks in the nero video game. *AAAI*, 2006. 3, 13

[33] Christopher Stanton and Jeff Clune. Curiosity Search: Producing Generalists by Encouraging Individuals to Continually Explore and Acquire Skills throughout Their Lifetime. *PLOS ONE*, 11(9):e0162235, 2016. 17

[34] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent Cooperation and Competition with Deep Reinforcement Learning. *arXiv.org*, pages 1–12, 2015. 14

[35] Krzysztof Trojanowski and Sławomir T Wierzchoń. Studying Properties of Multipopulation Heuristic Approach to Non-Stationary Optimisation Tasks. pages 23–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. 16

[36] Leonardo Trujillo, Gustavo Olague, Evelyne Lutton, Francisco Fernández de Vega, León Dozal, and Eddie Clemente. Speciation in Behavioral Space for Evolutionary Robotics. *Journal of Intelligent & Robotic Systems*, 64(3-4):323–351, January 2011. 4

[37] Vinod K Valsalam and Risto Miikkulainen. Modular neuroevolution for multilegged locomotion. *GECCO ()*, 2008. 13

[38] Roby Velez and Jeff Clune. Diffusion-based neuromodulation can eliminate catastrophic forgetting in simple neural networks. *CoRR*, 2017. 10

[39] S Whiteson, M E Taylor, and P Stone. Empirical studies in action selection with reinforcement learning. *Adaptive Behavior*, 2007. 14

[40] Guohua Wu, Rammohan Mallipeddi, P N Suganthan, Rui Wang, and Huangke Chen. Information Sciences. *INFORMATION SCIENCES*, 329(C):329–345, February 2016. 15, 18, 33

[41] Zhao Yanling, eng Bimin, and Wang Zhanrong. Analysis and Study of Perceptron to Solve XOR Problem. pages 1–6, February 2004. 22

[42] Yadong Yu, Haiping Ma, Mei Yu, Sengang Ye, and Xiaolei Chen. Multipopulation Management in Evolutionary Algorithms and Application to Complex Warehouse Scheduling Problems. *Complexity*, 2018(4):1–14, 2018. 18