Effects of Neural Network Evolution

Kurtis McAlpine

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2020

Effects of Neural Network Evolution

Submitted by: Kurtis McAlpine

Abstract

This dissertation developed a system to explore the behavioural effects of evolving neural network structures from applying different mutation rates. Additionally how the speciation of similar network structures can affect the performance of controlling an agent in a competitive domain.

This disseration concludes by evaluating the objectivess that were explored then expanding on future directions for the work to progress.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor Dr Rob Wortham for his continued support, providing me with guidance and insight into a range of topics throughout the past year.

Chapter 1

Introduction

The future of machine learning is continuously increasing the automation of systems to improve efficiency in a domain, specific areas would include autonomous agents or vehicles that need to safely manoeuvre around an environment, where such domains have a lot of uncontrollable variables. This leads to questions, such as; Can an autonomous system continuously adapt to a changing environment or What happens when two distinct autonomous systems try to adapt to each others behaviour?

Typical systems are seen in game environments where neural networks are used to control an agent, with researchers creating game artificial intelligence that can outperform human levels. DeepMind's AlphaZero has mastered Chess by playing against using reinforcement learning (Silver et al., 2018). In Chess, there are a fixed amount of pieces per opponent, each piece has a predefined moveset that does not change throughout the game, such parameters are trivial for a neural network to learn, whats difficult is trying to solve the game of Chess.

As a domain becomes more complex the parameterisation of the behaviours grows larger, making it increasingly difficult for developers to build a fixed topology networks, since they would likely require an in depth understanding of such a high dimensional space. Advancements have been researched widely in order to provide a solution to this problem, neuroevolution is the field concerned with implementing a genetic algorithm to optimise the search for a solution in a reinforcement learning domain. The benefits of neuroevolution were found to have effective results evolving both the weights and topology of neural networks, leading to the area of Topology and Weight Evolving Neural Networks (TWEANNs) (Rojas, 1996*a*), a subset of neuroevolution algorithms primarily focused on the optimisation of network structure. Although there is a breadth of research detailing the effectiveness of neuroevolution at producing efficient solutions to a given problem, there is a gap in the literature with respect to how the evolutionary behaviour of a neural network changes, specifically, when these networks are placed in a shared domain with stochastic properties.

This project attempts to explore the effects of evolving neural network topologies with different rates of mutation in a competitive environment when put up against the traditional fixed network topology structure.

1.1 Research Aims and Objectives

This project aims to explore the different techniques and methods of producing optimal neural networks in a reinforcement learning domain. There was minimal literature which combined the implementation of distinct neural network topologies and evolutionary approaches in a competitive environment. Therefore, the objective of this project is to investigate and assess the effects of evolving network topologies when they compete against a fixed topology counterpart. With the implementation of a genetic algorithm and evolving topologies, the following research questions were proposed in order to meet the primary aim:

RQ1: Does the mutation rate affect the evolutionary behaviour of the underlying network?

RQ2: Does the implementation of speciated networks provide better performance than that of a sufficiently sized fixed topology network when put head-to-head in a shared domain?

1.1.1 Research Method

This project used the development of a piece of software in order to perform experiments required to answer the above research questions. By using the foundations discussed throughtout the literature and technology review in the next chapter, this helped the development to remain current with popular methods and approaches being used today, as well as provide potential for future work.

1.2 Contributions

This project will aim to provide the following contributions:

- A review of the implementation of neural networks and genetically inspired evolutionary algorithms.
- A web based system that allows for the construction and observation of different neural network topologies for controlling agents in a competitive environment.

1.3 Project Outline

The structure of this dissertation is divided into 5 chapters as follows:

Chapter 1 Introduces the work outlined in the project, following with the aims and objectives, which defines the research questions of the project.

Chapter 2 Explores the literature and technology that encouraged the development of the project, areas including: Neural Network Structures and Implementations, Reinforcement Learning Techniques and Genetic Algorithms.

Chapter 3 Discusses the implementation of the system and experiments with hypotheses that will allow the project aims to be met.

Chapter 4 Analysis of the results obtained from the experiments with evaluation of the hypotheses.

Chapter 5 Final discussions answering the research questions and concluding the project.

Chapter 2

Literature and Technology Review

This chapter introduces the current research that is fundamental to understanding neural network structures. Following that, the review of a genetically inspired algorithm for the optimisation of neural networks. Finally, the discussion of neuroevolution, a methodology to evolve the structural properties along with the weights of a neural network.

2.1 Artificial Neural Networks

An artificial neural network is a computational model where its structure is biologically influenced by the network of neurons in the human brain (Jain, Mao and Mohiuddin, 1996). Although the architectures used today differ from their biological inspiration, which has led to the development of different types of neural networks, the core mechanics still remain very similar. The individual neurons of the network are connected to at least one other neuron, where each connection is weighted, allowing the model to evaluate the importance of the connections in the network (Svozil, Kvasnicka and Pospíchal, 1997). Inputs to the neurons are propagated through the network connections via successive activation functions, approximating a final output value.

Warren McCulloch and Walter Pitts first conceptualized the model of a neuron (Kurenkovi, 2015), where such neurons would produce a fixed signal if it received enough stimuli to excite it to a value greater than the activation threshold. In contrast, the neuron would not produce a signal while the activation threshold was not reached. Therefore, this type of neuron allows for binary activation, where the use of a step function would be most appropriate, as such fired signals can only be a constant value.

2.1.1 Perceptron

Psychologist Frank Rosenblatt invented the first, feed-forward, single layered network called a *perceptron* that would work based on the McCulloch-Pitt's model (Rosenblatt, 1958). Equation 2.1 shows the implementation of Rosenblatt's perceptron, where the output y is given by applying the activation function f to the weighted sum of input values. It takes n number of connections, where w_i and x_i are the respective weights and inputs for each connection, a bias value b is then added, which allows to translate the decision boundary away from the origin.

$$y = f(b + \sum_{i=1}^{n} x_i w_i) \tag{2.1}$$

The applied activation function f is the step function, given in equation 3.1

$$f(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2.2}$$

Perceptron Learning
The perceptron learning algorithm deals with finding optimal weights in order to compute a linear separation. The network learns to produce a desired output by iteratively adapting network parameters based on what it has previously computed (Rojas, 1996*b*). The algorithm is simply implemented in three steps:

1. Random weight initialization for each connection and set $t = 0$ where t is the iteration step
2. Evaluate output y from an input vector $(x_1, ..., x_n)$
3. Update the weights using equation 2.3

$$w_i(t+1) = w_i(t) + r(d - y(t))x_i \tag{2.3}$$

   where r is the learning rate and d is the desired output for the given input.

A study (Minsky and Papert, 1969) was conducted to identify the perceptrons ability to learn linear decision boundaries for boolean logic functions. They discovered a single perceptron was not capable of learning linearly inseparable functions like XOR, but combining multiple layers of perceptrons would make it possible to learn such patterns.

2.2 Deep Learning

Deep learning is a machine learning technique influenced by what we understand about how the human brain functions. As the possible levels of computation has vastly increased in recent years, its made it possible to build much larger models of networks that can be trained with large amounts of data (Goodfellow, Bengio and Courville, 2016). A survey (Schmidhuber, 2015) summarising the history and advancements in deep learning identified that network topologies containing more hidden layers showed higher levels of performance when solving complex problems. This project will implement a multi-layer perceptron as the basis for the network topology, in order to teach agents the behaviour required to perform actions in an environment.

2.2.1 Multi-layered Perceptron

A multi-layered perceptron is a feed-forward network built with one or more hidden layers of neurons. It works by propagating inputs throughout the individual layers using some non-linear activation function, allowing the network to solve complex non-linear problems (Zhao Yanling et al., 2002).

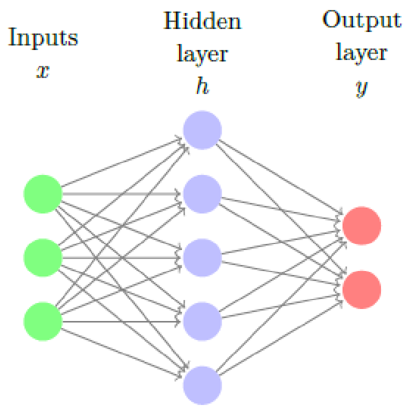


Figure 2.1: Multi-layered perceptron with a single hidden layer (François-Lavet et al., 2018)

2.3 Activation Functions in Deep Learning

An activation function can either be linear, like the previously discussed step function or nonlinear, choosing which type of function to use depends on the problem domain. When building a multi-layered networks to solve complex problems, each of the individual layers will typically use the same non-linear activation function (Bishop, 2006).

2.3.1 Summary of Non-linear Activation Functions

This section highlights three common types of non-linear activation functions that are widely used (Nwankpa et al., 2018) in deep learning.

Sigmoid function
Sigmoid also known as logistic or squashing function (Turian et al., 2009) is typically used in feedforward neural networks. The output of the Sigmoid function is given by equation 2.4

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

The function takes real values as inputs and generates a probabilistic output. Optimal results are found in logistic regression and binary classification problems where network topology is shallow (Neal, 1992).

Hyperbolic Tangent (tanh)
The *tanh* function is a smoother variant of the Sigmoid (LeCun et al., 2015), which is scaled and translated about a zero-centred origin. The output values are in the range of -1 to 1, given by equation 2.5.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.5}$$

The close relationship between Sigmoid and *tanh* can be seen by $tanh(x) = 2\sigma(2x) - 1$ (Goodfellow et al., 2016), however, *tanh* is the preferred choice of function when working with multi-layered networks, as it has shown to provide a more efficient learning performance (Olgac and Karlik, 2011).

Rectified Linear Unit (ReLU)
ReLU is currently the most widely-used activation function (Nair and Hinton, 2010) in deep learning. ReLU has an effective learning rate, outperforming both Sigmoid and *tanh* functions in a deep learning environment (Zeiler et al., 2013).

$$f(x) = max(0, x) = \begin{cases} x_i, & \text{if } x_i \geq 0 \\ 0, & \text{if } x_i < 0 \end{cases} \tag{2.6}$$

A study (Foerster et al., 2016) utilized ReLU with deep learning, which trained agents on how to act and behave with other agents in an environment. During the learning phase, their implementation also integrated backpropagation to optimize the network, which they later discussed was "uniquely powerful" within the domain of teaching agents communicative behaviour between each other.

2.4 Learning Algorithms

Supervised learning is an area of learning algorithms that typically requires prerequisite knowledge of the problem domain, such that, given an input, a desired output is known. The algorithms learn by finding correct mappings of inputs to outputs, so that unseen data can be evaluated (Roessingh et al., 2017). In the process of learning, the algorithms will typically use some cost function to determine the accuracy of the predicted output, a popular choice is the *mean squared error* (Nielsen, 2018):

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{2.7}$$

Where w and b are the respective weights and biases for the network, n is the size of the training data x, $y(x)$ is the target output for a single given x input and a is a vector of actual output of the network for all x inputs. The closer the value of $C(w, b)$ is to 0, the more accurate the network outputs are for the given inputs. Therefore, optimal weights and biases are found using *gradient descent* when $C(w, b) \approx 0$ (Nielsen, 2018).

Gradient descent
Simply put, gradient descent is an algorithm used to optimize the training of neural networks. It works by iteratively updating the network weights with respect to the negative gradient of the cost function, until the gradient converges to a global minimum, which is considered a solution.

2.4.1 Backpropagation

Backpropagation is one of the most used supervised learning algorithms for training neural networks. It works by propagating inputs throughout each of the network layers, computing the cost function, then backpropagating this value to adjust the network weights. The minimum value of the cost function is then considered a solution to the search problem, which is found by evaluating the gradient of the cost function after each iteration (Rojas, 1996*b*).

The backpropagation algorithm
The implementation of the algorithm is highlighted as:

1. **Initialize**: Randomize starting weights for network.

2. **Feedforward**: Propagate inputs until the output is calculated.
3. **Compute output error**: Compare expected output with computed output.
4. **Backpropagation**: Compute the error for each previous layer, adjusting the weights accordingly.
5. **Output**: Evaluate the gradient of the cost function.
6. **Termination**: Repeat from step 2 until cost function converges to a minimum.

A study (Flower et al., 2006) was undertaken to train an autonomous agent using neural networks with backpropagation to navigate around a virtual environment. Their implementation of the network used a single hidden layer, consisting of 5 neurons, which they discussed was sufficient to train the agent to follow roads and avoid obstacles. Although, they indicated their network performance was susceptible to the given training data. For example, the agent may have chosen to turn left to avoid an obstacle, when in reality the better action was to turn right, this may have been caused by the training data containing more left turns, making the network biased to turn left. However, they went on to conclude that when trained with an unbiased distribution of data, the agent showed realistic behaviour when navigating around complex environments consisting of dynamic variables, something that is difficult to implement with a rule-based system. Additionally, they discussed building additional neural networks to control the individual components of the agent, such as steering and acceleration, suggesting it could produce more realistic outcomes.

2.4.2 Reinforcement Learning

Reinforcement learning is a technique influenced by behavioural psychology (Sutton, 1984) which uses prior experience it has gained in order to perform a sequence of actions in an environment. Unlike backpropagation, which typically requires knowledge of the domain on how an agent should behave, reinforcement learning uses evaluative feedback in the form of some cumulative reward depending on how well the agent is performing in the environment. This allows the agent to optimize its future choice of action from what it has previously learnt (Sutton and Barto, 1998).

In recent years, reinforcement learning has been used in combination with deep learning to solve challenging areas that possess extremely large state spaces. Such works include learning to play games of perfect information at the level of human professionals, like Go (Silver et al., 2016), which has a state space of 10^{170} (Maddison et al., 2014) possible positions, compared to the 10^{43} state space of Chess (Chinchalkar, 1996). DeepMind's *AlphaZero* has mastered the game of Go, Chess and Shogi, employing reinforcement learning techniques to train a neural network by playing against itself, thus learning from experience which moves would be more advantageous in the future (Silver et al., 2018).

There are four key elements when implementing reinforcement learning: a *policy*, a *reward function*, a *value function* and a *model*. According to (Silver et al., 2018), the policy is the most important element, as it solely determines the behaviour of an agent.

Policy
Given a state in an environment, the policy is what defines which actions are to be performed by the agent. In order to obtain an optimal policy, it is important to balance exploring new actions with exploiting what was previously learnt. Too much exploration may never lead to a desired state, while too much exploitation would lead to suboptimal policies (Shani et al., 2018).

Reward function
The reward function uses evaluative feedback to determine if an action performed was good or bad for some state. The given reward after each action is used to update the policy, where low valued rewards may force the policy to explore another action for that state as it aims to maximize the long-term reward (Sutton and Barto, 1998).

Value function
Unlike the reward function, which utilizes short-term reward feedback, the value function is concerned with the rewards that an agent can accumulate from future states given its current state. This may indicate the policy to favour what appears to be a less rewarding intermediate state knowing that future states are higher valued (Sutton and Barto, 1998).

Model
The model is an optional element depending on the choice of reinforcement learning algorithm. It is a planning method that can indicate how an environment might behave in future states without actually employing the actions to reach such states (Sutton and Barto, 1998).

2.5 Genetic Algorithms

Genetic algorithms are adaptive population based algorithms (a type of evolutionary algorithm) that can be implemented to assist in solving and optimising complex problems (Rahnamayan and Wang, 2009). The design of genetic algorithms by computer scientist, Holland, was heavily influenced by natural selection and the biological processes involved in the genetics of living organisms (Holland, 1992*b*). Holland identified that his work on adaptive and reproductive plans could be used in applied fields of science, allowing to represent complex behaviour with a population of computer systems (Bäck, 1996).

Potential solutions to a search problem are known as *chromosomes*, which are encoded as finite strings of alphabets (*genes*). The values of the individual genes are called *alleles* (Goldberg, 1989). A common form of genetic encoding is binary, where each chromosome is represented by a string of bits (Fogel, 1995).

2.5.1 Structure of the Genetic Algorithm

Initialization
The process of generating an initial population, this is achieved by randomly selecting a set of solutions with some encoding from the problem domain.

Fitness evaluation
Each individual in the population is evaluated by an *objective function*, which provides a way to determine how well they are performing within the problem domain. To measure an individual's relative fitness, a transform is applied to the objective function, this is known as the *fitness function* (De Jong, 1975). Choosing the individuals that maximise the value of the objective function is what allows for the simulation of evolution (Kreinovich, 2001), this follows Darwinian's *"survival of the fittest"* principle, where the better performing individuals adapt to their environment (Zhong et al., 2005).

Selection
The process of selecting individuals for recombination utilizes the fitness function to determine the better performing individuals in the current population. Thus, the higher its performance values, the higher chance it has of producing offspring (Blickle and Thiele, 1996). However, selecting only the fittest individuals does not guarantee an optimal solution, and unfit individuals should not be prematurely removed from the population as they may lead to better solutions in later generations (Shukla et al., 2015). Scheme Theorem (Holland, 1992*a*) or *roulette selection* is a method that assigns each individual a probability to be selected, where such probability is directly proportional to their fitness. This allows for unfit individuals to remain existent, but not

dominant with respect to producing offspring.

Recombination
Also known as crossover, this is the production of new offspring with a combination of two parents genes to repopulate the domain. Various ways of achieving crossover exist, including that of **k-point crossover** and **uniform crossover**, both being popular choices (Sastry, Goldberg and Kendall, 2005). Choosing which method of crossover to use is typically dependent on the problem domain as well as the type of encoding used for the chromosomes (Umbarkar and Sheth, 2015). The parent genes are randomly selected from the current population pool and recombined by the crossover probability p_c (Goldberg, 1989).

- **K-point crossover**
  K-point crossover produces new offspring by a combination of two parents genes at random crossover points. The number of crossover points (k) is randomly selected (Spears and Jong, 1995). The example below shows the production of two offspring using k-point crossover, where k = 3.

Parent 1:	1 | 0 1 | 0 0 | 0
Parent 2:	0 | 0 0 | 1 1 | 1
Offspring 1:	1 | 0 0 | 1 1 | 1
Offspring 2:	0 | 0 1 | 0 0 | 0

Figure 2.2: An example of k-point crossover.

- **Uniform crossover**
  Uniform crossover uses a swapping probability to randomly swap alleles in the parents genes to produce new offspring. The crossover probability p_c typically has a value of 0.5 (Spears and Jong, 1995). Below is an example of the produced offspring from two given parents, using uniform crossover.

Parent 1:	1 0 1 0 1 0
Parent 2:	0 0 0 1 1 1
Offspring 1:	1 0 0 1 1 1
Offspring 2:	0 0 1 0 1 0

Figure 2.3: An example of uniform crossover.

Mutation

Using recombination allows for the opportunity to produce better chromosomes if the parents genes are dissimilar. However, if the parents alleles are identical, then the recombination process is redundant, and if the entire population shares the same alleles, the gene will remain unchanged indefinitely throughout the successive generations. For diversity to exist in the population, a mutation operator can be applied, allowing for the problem space to be fully explored (Hassanat et al., 2018). A typical mutation method known as bit-flip mutation, where the bits in a binary string are inverted with probability p_m, known as the mutation probability (Sastry et al., 2005). Holland discusses that using a small mutation rate ensures the gene does not greatly deviate from its ancestor (Bäck, 1996).

Replacement
The newly generated offspring from the crossover and mutation operators replaces the existing population and starts a new generation in the evolutionary cycle. Choosing parents with high fitness values should allow for the overall average fitness to gradually increase, as their offspring will be among the fittest individuals in the population (Sastry et al., 2005).

Termination condition
If the termination condition is not met, continue searching the domain space by creating new generations of populations.

2.6 Exploration and Exploitation

The behaviour of the genetic algorithm is determined by the relationship between exploiting what currently works and exploring the search space to find something that works better (Herrera and Lozano, 1996). This means that for the genetic algorithm to be successful in searching for a solution, a balanced ratio between exploration and exploitation should be maintained. Effectively searching the domain for solutions is known to be more efficient with a diverse population (Michalewicz, 1996), however, such diversity in the population is not necessarily created due to a balanced relationship between exploration and exploitation (Liu et al., 2013).

It is important to understand how the operators of a genetic algorithm contribute to exploration and exploitation. A study (Eiben and Schippers, 1998) identified a general belief that recombination and mutation would allow the algorithm to explore the problem space, whereas selection would exploit the fittest individuals of the population. However, Eiben and Schippers would go on to argue that more research is required to understand how such operators can affect the algorithm's performance.

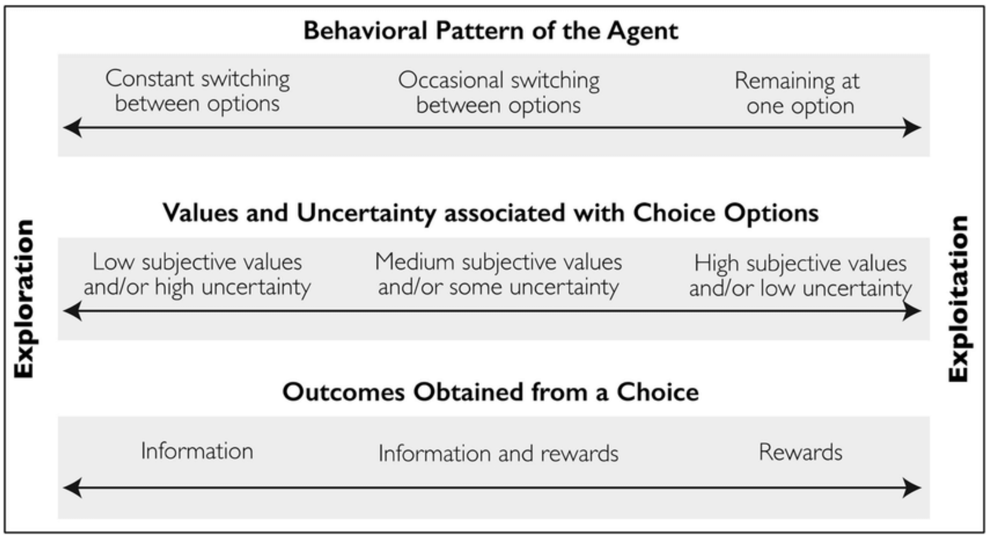

Behavioral Pattern of the Agent
Constant switching between options
Occasional switching between options
Remaining at one option
Values and Uncertainty associated with Choice Options
Low subjective values and/or high uncertainty
Medium subjective values and/or some uncertainty
High subjective values and/or low uncertainty
Outcomes Obtained from a Choice
Information
Information and rewards
Rewards
Exploration
Exploitation

Figure 2.4: Exploration - Exploitation continuum (Mehlhorn et al., 2015)

2.6.1 Genetic Algorithms to Assist Deep Learning

The use of genetic algorithms has successfully been implemented to assist in the training of neural networks (Schaffer et al., 1992). In section 2.4, it was discussed how gradient-based algorithms, such as backpropagation are used to train neural networks. In deep learning, the use of backpropagation can be replaced by or used in conjunction with genetic algorithms, this has successfully shown to be an improvement in the learning performance and overall agent behaviour (David and Greental, 2014).

Schaffer et al., (1992) suggests an advantage of applying genetic algorithms to optimize the weights in a neural network is avoiding a local minima. This becomes possible as the algorithm can globally search the weight space.

In addition to using genetic algorithms to find optimal network weights, they can also be utilized to evolve the network topology for solving reinforcement problems. (Lehman and Miikkulainen, 2013*a*). Implementation of this evolutionary strategy has shown to match the performance of typical reinforcement learning techniques (Hausknecht et al., 2014).

2.6.2 Limitations with Genetic Algorithms

Regardless of genetic encoding and implementation of operators, a study (Schwab, 2004) discusses that the amount of generations required to evolve a possible solution is too high, resulting in a time consuming process. This is typically the case when the genetic algorithm suffers from premature convergence, where the population becomes trapped in a local optima, therefore limiting the explorative ability of the algorithm due to lack of population diversity (Chen et al., 2008).

2.7 Neuroevolution

Neuroevolution is typically applied to reinforcement learning environments and used in conjunction with genetic algorithms to optimise the learning process, examples of common applications include evolutionary robotics and artificial life (Lehman and Miikkulainen, 2013*b*). Using properties of genetic algorithms, neuroevolution takes potential solutions for a given problem and evaluates them according to their fitness values, it then determines how to handle each individual solution, which includes the mutation and evolution of the topological structure of networks for the successive generations of populations.

Application of neuroevolution is best suited to problems where topological exploration and exploitation can optimise the topologies of networks for non-linear search problems. Linear problems would typically implement a local optimisation method, such as backpropagation with gradient descent, therefore, likely to become stuck in a local minima (Kelemen et al., 2008). Optimisation techniques with the ability to globally search for solutions in non-linear problems are known as Topology and Weight Evolving Artificial Neural Networks.

2.7.1 Topology and Weight Evolving Artificial Neural Networks

Topology and Weight Evolving Artificial Neural Networks or TWEANNs are a subset of neuroevolution algorithms with focus on evolving the parameters and the topology of the neural network in order to find an efficient solution. Traditional neural network implementations require manual configuration for the structure, such as providing the fixed number of hidden node connections between the input and output nodes. The main issue with this is it becomes extremely impractical as the complexity of the problem increases, which directly has an impact of the complexity of the neural network. (Rojas, 1996*a*).

2.7.2 The GNARL Algorithm

GeNaralized Acquisition of Recurrent Links or GNARL is the construction of a recurrent networks through the use of neuroevolution. The number of input and output nodes are initially provided, which are immutable throughout the process of evolution. Instead, hidden nodes are inserted or *'gnarled'* into the networks, constructing unique network topologies that can help optimise the search for an optimal solution. GNARL allows for an optional bias node, with a constant input value, where all other non-input nodes apply the sigmoid activation function. Mutations of networks in GNARL are handled by taking the top 50% performing networks as designated parents for the successive generation and applying differing severities of parametric or structural mutations. Where parametric mutations alter the weights of the connections and structural mutations alters the number of hidden nodes and connections in the network (Angeline, Saunders and Pollack, 1994). Since GNARL does not apply crossover during its evolutionary process, it avoids a common problem in neuroevolution known as the *competing conventions problem* (Stanley and Miikkulainen, 2002).

2.7.3 Competing Conventions Problem

Competing conventions or the *Permutations problem* (Radcliffe, 1993) is when neural networks with the same functional structure are encoded differently. As seen in 2.5, a neural network with a hidden layer of 3 nodes [A, B, C], can have its hidden layer be represented in 3! = 6 distinctive ways. Since TWEANNs do not restrict the type of topological innovations, this becomes a problem in algorithms that attempt to implement crossover, where matching genes may not correctly align, producing damaged offspring. The proposed method of *Neuroevolution of Augmenting Topologies* (Stanley and Miikkulainen, 2002) can produce viable offspring through biologically inspired crossover.

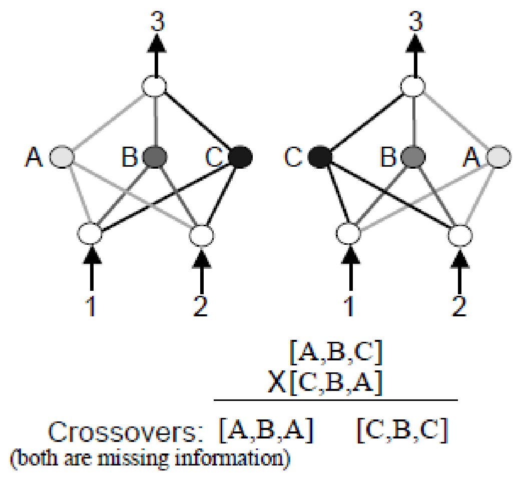


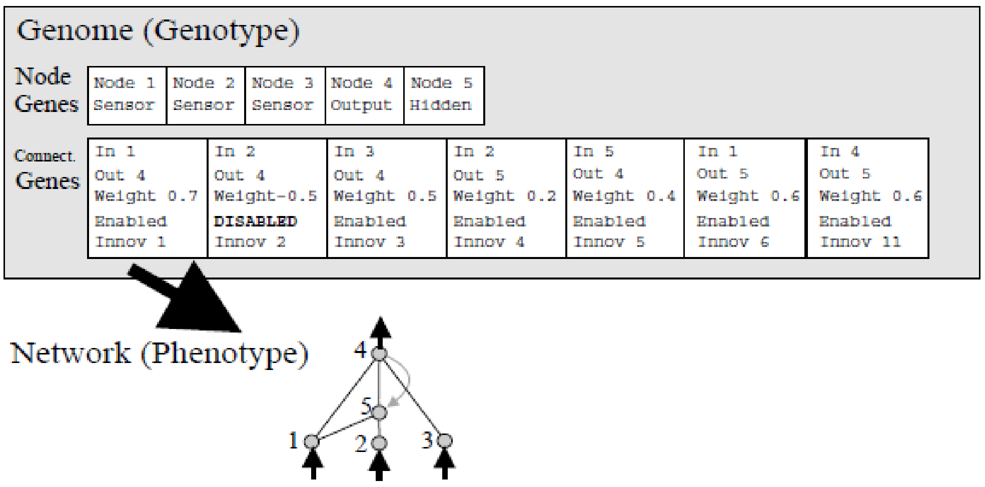
Figure 2.5: An example of the competing conventions problem, where two networks have the same functional structure, but represented differently, such that crossover will result in offspring with a loss of information.

2.8 Neuroevolution of Augmenting Topologies

Neuroevolution of Augmenting Topologies (NEAT) is a genetic algorithm that addresses competing conventions in neuroevolution whilst effectively evolving the topology of networks. It achieves this by tracking matching genes through historical markings and speciation of similar networks to protect new innovations in the population. This section will summarise the core components of Stanley and Miikkulainen's NEAT.

2.8.1 Genetic Encoding

As seen in 2.6, network genomes are linear representations of connections with its two corresponding nodes in the topology, where nodes are set to be either input, output or hidden nodes. The connections encapsulate the incoming and outgoing node with a connection weight, a flag bit indicating whether such connection is enabled or disabled and an *innovation number*, which allows NEAT identify matching genes in different genomes.



Node Genes				
Node 1 Sensor	Node 2 Sensor	Node 3 Sensor	Node 4 Output	Node 5 Hidden

Connect. Genes						
In 1	In 2	In 3	In 2	In 5	In 1	In 4
Out 4	Out 4	Out 4	Out 5	Out 4	Out 5	Out 5
Weight 0.7	Weight-0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6	Weight 0.6
Enabled	**DISABLED**	Enabled	Enabled	Enabled	Enabled	Enabled
Innov 1	Innov 2	Innov 3	Innov 4	Innov 5	Innov 6	Innov 11

Figure 2.6: Genetic encoding of a genome and its topological representation (Stanley and Miikkulainen, 2002).

2.8.2 Historical Markings of Genes

Given a population of diverse genomes, it is important to identify the origin of genes, such that matching genes can be correctly aligned in order to avoid the competing conventions problem when performing crossover. New connections added to the topology of a network, both through initialization and mutation operations are assigned a unique *innovation number*, therefore, any two genomes sharing the same genes will also share the same innovation number for these genes. Any genes that are not matching are considered either *disjoint* or *excess*, depending on where they appear in the genetic encoding as seen in 2.7. Identifying the disjoint and excess genes between genomes in NEAT allows for the possibility to speciate similar topologies.

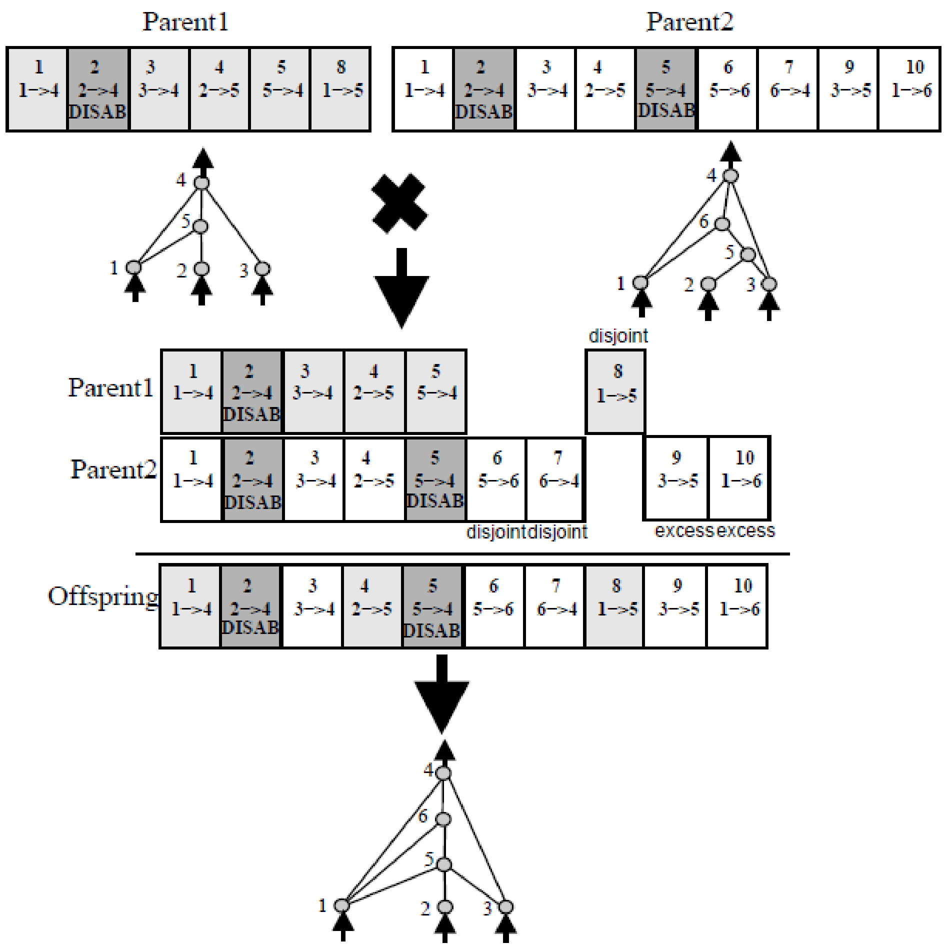


Figure 2.7: Aligning the genomes of two different network topologies by using the innovation numbers of genes, allowing to perform crossover to create offspring. Genes that exist in both parent genomes are inherited randomly where disjoint and excess genes are inherited only from the fittest performing parent genome. Any inherited genes has a set probability of being disabled given it is disabled in any parent (Stanley and Miikkulainen, 2002).

2.8.3 Network Mutation

Aside from the typical weight optimization, there are two main mutation operations in NEAT, both of which can augment the topologies to create diverse populations (2.8). These are:

1. **Add Connection** - Creates a new connection between two given nodes.
2. **Insert Node** - Given a connection, a node is inserted by disabling this connection, then adding two new connections, one from the original incoming node to the new node and another from the new node to the original outgoing node.

Although this type of topological mutation can explore a vast dimensionality of solution space, the process of adding new structure to a network creates a non-linearity to the current solution, causing fitness degradation. The result of this makes it difficult for new innovations to survive in the population, as they may be outperformed by other networks before having a chance to optimize. This leads to the idea of speciation of networks to protect them from being disregarded too early in the search process.

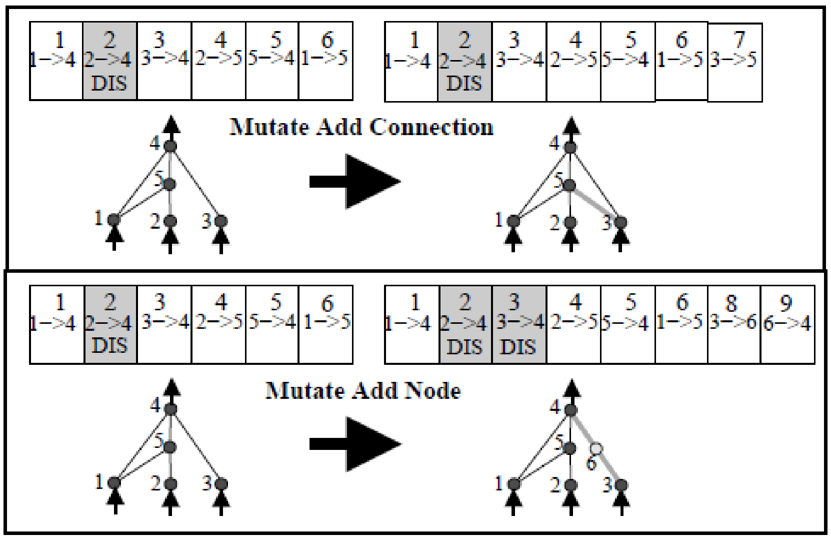


Figure 2.8: Example of NEAT topology mutations with the corresponding genetic encodings. The first mutation creates a new connection between nodes 3 and 5 and is given the innovation number of 7. The second mutation inserts a new node (6) on the connection between nodes 3 and 4. The existing connection with innovation 3 becomes disabled and two new connections with innovations 8 and 9 respectively are added to the genetic encoding (Stanley and Miikkulainen, 2002).

2.8.4 Protecting Innovation through Speciation

Dividing the population into groups of similar topological structure is called speciation, allowing genomes to compete within their individual niches. Utilizing the historical markings in genomes through the innovation numbers, genomes that differ from each other over a given threshold are considered incompatible and are assigned a new species, whereas those within the threshold are considered the same species. Equation 2.8 is used to calculate the compatibility distance between any two genomes.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{2.8}$$

The excess E and disjoint D genes are identified through aligning the encodings, where N is the total number of genes in the largest genome. Genes that are not excess or disjoint are matching genes and $\overline{W}$ is the average weight difference of such genes. The coefficients c_1, c_2 and c_3 are constant values that determine the importance of each term. Distance δ is calculated and compared against a compatibility threshold δ_t, which allows to speciate the population.

2.8.5 Fitness Sharing

Genomes of the same species will explicitly share fitness within their niche, essentially, the fitness values become normalized by the population size of their respective species. This encourages speciation throughout the evolutionary process as a single species is unlikely to dominate the population the larger it becomes. Equation 2.9 is used to calculate the new fitness f'_i for each network i. The sharing function sh evaluates to a 1 if δ is greater than δ_t else 0, therefore $\Sigma_{j=1}^{n} sh(\delta(i,j))$ is equal to the size of the species that network i belongs to.

$$f'_i = \frac{f_i}{\Sigma_{j=1}^{n} sh(\delta(i,j))} \tag{2.9}$$

2.9 Summary

This project seeks to observe the evolutionary behaviours of neural networks. The literature and technology review explored key research into areas of neural network implementations, reinforcement learning, genetically inspired algorithms and the structural optimization of networks through neuroevolution techniques.

Chapter 3

Implementation

This chapter introduces the implementation of the system and the experiment carried out. Over the course of the development process several challenges and issues arose that would negatively impact the behaviour of the algorithms being implemented. Several adaptations had to be made in order to fit within the time constraints of the project, but, whilst still maintaining the desired property of evolving neural networks. As the project became increasingly complex, a more effective data handler was implemented that would automatically retrieve and locally store on a per generational basis.

All final implementations were built with JavaScript, as it can simply be run on most modern browsers without the need for any additional packages to be downloaded and installed, such that it allowed for portability and ease of testing on multiple devices.

3.1 Environment Implementation

The environment was a simple 2d square that allowed neural networks to be mapped to an agent entity in order to achieve a goal. The goal was for either agent to achieve a score of 10 before the other, being awarded a win. Rules were set to not allow the simulation to run indefinitely, due to a network not having learned an optimal strategy. The size of the environment was restricted to a 100px by 100px square, which housed 2 agents to be controlled by neural networks and 2 obtainable resources, good food and bad food 3.1. Each agent started with an energy level of 300, where each action would move them 1 pixel in the selected direction and consume 1 energy. Obtaining the good food resource replenished 100 energy to that agent as well as rewarding it with 1 score, where bad food deducted 100 energy and deducted 1 score, if an agents energy reached 0, it died, else, if it's score was 10, it won. Once the environment was inactive, the evolutionary algorithm discussed below produced the next generation of networks and mapped them to the agents. The environments continued this loop until the termination condition was met.

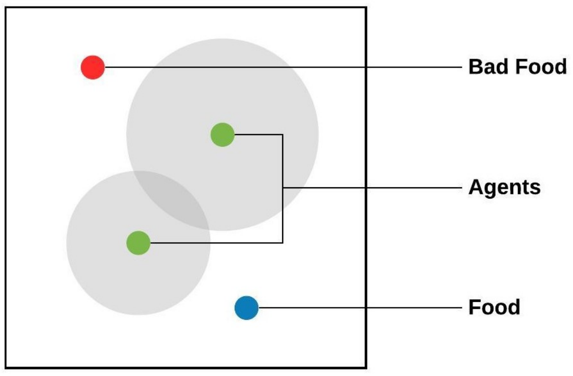


Figure 3.1: A prototype design of a single environment, showing the agents and food that reside within it.

Using the prototype design in 3.1 and wanting to keep the development specific to web browser technologies, it was decided a HTML canvas would be used to render the environments. With use of a simple JavaScript script, the number of required canvas environments would be automatically generated based on a given population size, saving needless time of having to manually build them all individually.

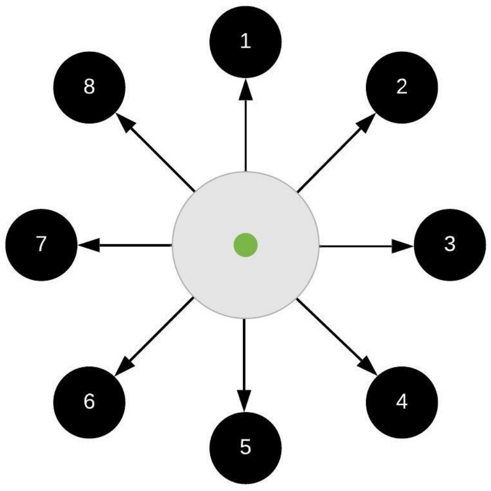


Figure 3.2: The 8 directional movement options for an agent, each representing a single output node in a neural network.

3.2 Agent to Network Mapping

Using a one to one mapping of agent and neural network 3.3 the directional movement of an agent was easily given by a network with 8 output nodes. Therefore, the node with the largest output after evaluating the inputs determined the movement direction 3.2. In order for the agent to effectively interact in the environment, its neural network needed to have a sufficient number of input nodes, given the environment entities below, it was decided that 8 input nodes would be enough:

- **Competitor**
- **Good food**
- **Bad food**

Each entity had a distance and an *angle* (calculated using the $atan2$ function native to the JavaScript $Math$ library) from the agent in question, therefore, this gave 6 input nodes registering values for the 3 entities positional activity, where the final 2 inputs are for the agents own x and y positions relative to the environment.

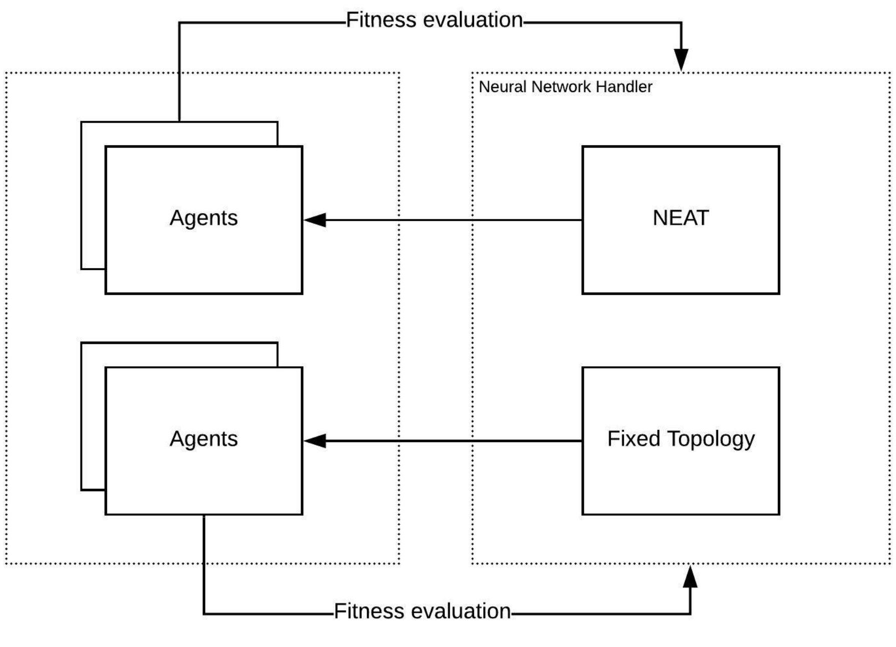


Figure 3.3: Mapping of NEAT and fixed topology networks to distinct populations of agents.

3.3 Neural Networks

The decision to use a feed-forward neural network design means they can simply be implemented using a graph data structure. A graph consists of nodes and edges that can easily represent the neurons and connections respectively of a neural network.

For any given feed-forward network in a non-linear domain, there are 3 types of nodes that must be considered for correct implementation. Therefore, the graph data structure must support nodes that have a property identifying them as one of the following types:

1. **Input** nodes are immutable that pass incoming values into the network structure evaluation.
2. **Hidden** nodes are immutable in fixed topology networks.
3. **Output** nodes are immutable and hold the evaluation of inputs fed into the network.

Nodes in the data structure must also have properties that allow them to identify a list of incoming node connections, which would be an empty list if such node is of type input and a list of outgoing nodes.

A list of connections is maintained which must have properties indicating the incoming and outgoing nodes and the respective weight between such connections.

3.3.1 Configuration Object

The configuration object is used only for the initialization of the neural networks, it holds values indicating the number of specified input nodes, output nodes and hidden nodes. Throughout this project, a config value of 0 for hidden nodes must be treated as NEAT specific and handle topological innovations, whereas any value greater than 0, the network is considered as fixed topology and can only have its connection weights optimised.

3.4 Evolutionary Algorithm Design

The algorithm designed for both NEAT and fixed topology populations will follow the structure of the genetic algorithm discussed in section 2.5.1. Since this algorithm will need to work for both types of populations, modifications to the core phases are implemented in order to accomodate for the NEAT specific genomes, mainly, the diverse topological innovations that do not follow a strict layered structure.

3.4.1 Population Initialization

The initialization of populations for NEAT 3.4 and fixed 3.5 topology networks uses a simple $INIT()$ function 1 that when passed a configuration object will generate a number of specified networks based on the globally set variable $PopulationSize$. A value of 8 hidden nodes is provided for the configuration object for fixed topologies, which should be sufficient for the network to learn a strategy that can move an agent around the environment based on the similarities in the study discussed in 2.4.1. The weights of the connections for both types of topologies were uniformly initialised with a random value between -1.0 and 1.0, the decision for this was if connections were initialised with weight values of 0.0, there would be many redundant generations with no network behaviour at the start of the evolutionary cycle.

Algorithm 1 Population Initialization

Input: C (Network configuration object)
Output: $P_1 \ldots P_N$ (Population of network genomes for the given config)

```
1: function INIT($C$)
2:     $N \leftarrow PopulationSize$
3:     for $k \leftarrow 1$ to $N$ do
4:         $P_k \leftarrow NeuralNetwork(C)$
5:     return $P[]$
```

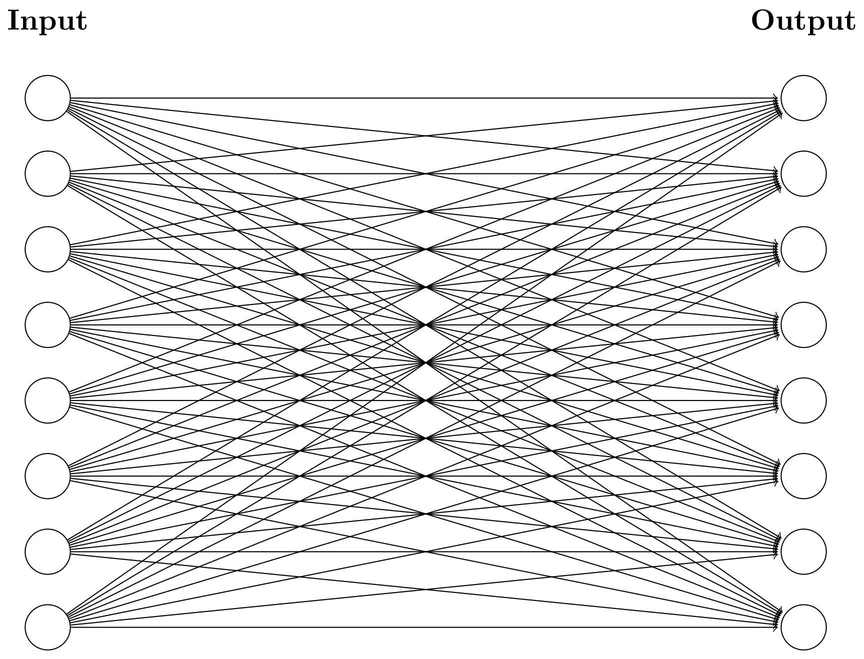


Figure 3.4: Initialization network for the population of NEAT networks, with a config file specifying 8 input nodes, 0 hidden nodes and 8 output nodes. All input nodes are directly connected to the output nodes.

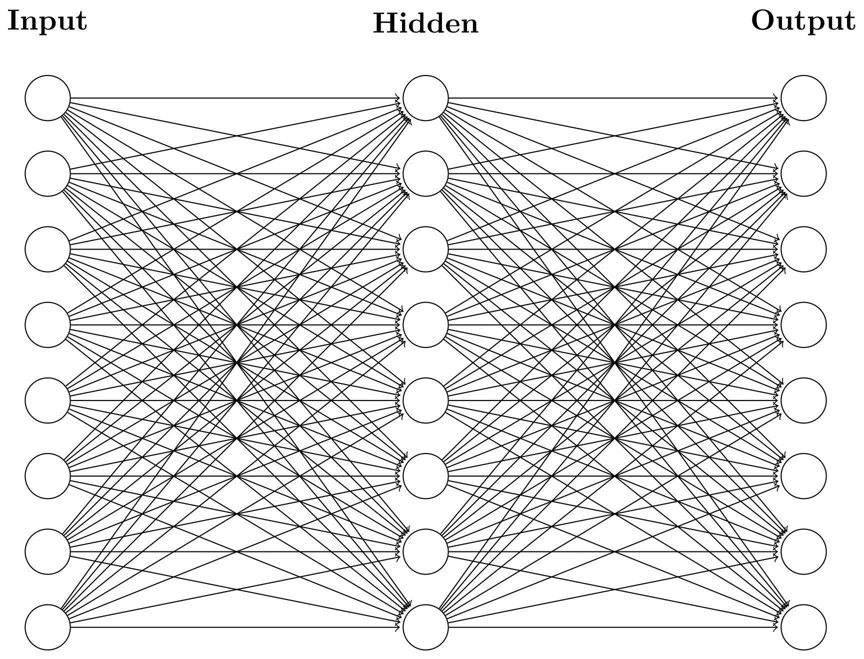


Figure 3.5: Initialization network for the population of fixed topology networks, with a config file specifying 8 input nodes, 8 hidden nodes and 8 output nodes.

3.4.2 Fitness Evaluation

A genomes fitness should reflect the performance of the agent it is controlling in the environment setting. Using a typical reinforcement learning approach, it is possible to determine how good or bad the agent is performing. Given the entities that reside within the environment, there are numerous ways to identify the greater success of an agent. However, since roulette selection 2.5.1 was the selection method of choice, an agent's fitness can not go below 0. Therefore, the implementation of the fitness function used to calculate an individual's fitness score (F_i) used within the system therefore was as follows:

$$F_i = \begin{cases} 150x - 150m - n, & \text{if } 150x - 150m - n > 0 \\ 0, & \text{if } 150x - 150m - n \leq 0 \end{cases} \tag{3.1}$$

Where x is the total number of good food resources obtained, n is the number of network outputs that moved the agent further away from the good food resource, and m is the total number of bad food resources obtained.

3.4.3 Crossover

The implementation of the crossover function was of type uniform **??**, using only a single function meant it had to handle both NEAT and fixed topologies. For NEAT specific topologies, genome alignment for matching genes was done with the use of the innovation number **??**, which was stored as a global key-value pair throughout, where key was the innovation number and the pair was a hashed gene value. In fixed topologies, each pair of connections in two different topologies was simply verified as a matching gene by comparing the incoming and outgoing node indexes.

3.4.4 Mutation

In comparison to Stanley and Miikkulainen's implementation of NEAT, where mutation operators include the possibility of disabling connections between nodes, the implementation developed in this project did not disable connections, but rather removed them completely from the genome. One of the benefits of doing this was to improve the computational efficiency of the topologies, as such topologies were no longer being constructed with redundant disabled connections, using their genome encoding. Algorithm 2 shows the implementation of the switch case when selecting one of the operators seen in 3.2, where 3.1 shows the single mutation possible for fixed topologies which evolves the weights of the connections.

Algorithm 2 Network Mutation

```
Input: $G$ (Network genome)
Output: $G'$ (Mutated network genome)

1: function MUTATE($G$)
2:     $R \leftarrow RandomMutation$
3:     switch $R$ do
4:         case 0 : $G.InsertNode$
5:         case 1 : $G.RemoveNode$
6:         case 2 : $G.AddConnection$
7:         case 3 : $G.RemoveConnection$
8:         case 4 : $G.AdjustWeight$
9:     return $G'$
```

Before any gene mutation was added to the genome representation of a network, it was compared against the global key-value pair object of existing genes and used the key as an innovation number if there was a matching value. If no value matches were found, the gene was added as a new entry to the object. Some specific types of mutations were not allowed, for example,

Table 3.1: Fixed Topology Mutation Operators

Fixed Topology Mutation Operators		
Operation	**Description**	**Value**
Adjust weight	Adjusts the weight of a connection by a value randomly selected in a given range.	(-1.0) - (1.0)

Table 3.2: Mutation types and values that can innovate the NEAT network topologies.

NEAT Mutation Operators		
Operation	**Description**	**Value**
Insert Node	Inserts a node into a randomly selected connection.	N/A
Remove Node	Removes a randomly selected node.	N/A
Add Connection	Creates a new connection between 2 randomly selected nodes.	N/A
Remove Connection	Removes a randomly selected connection.	N/A
Adjust Weight	Adjusts the weight of a connection by a value randomly selected in a given range.	(-1.0) - (1.0)

since input and output nodes are immutable 3.3, they can not be selected for removal from the structure. Also, output nodes can not have any outgoing connections.

3.4.5 Termination Condition

Due to the time constraints on the project and the intended aims outlined in 1.1, an exhaustive search for an optimal solution was not feasible, therefore, a termination condition was not implemented, but rather, a controlled number of 1000 generations were explored.

3.4.6 Speciation (NEAT topologies only)

Speciation was used in order to restrict the crossover function to similar genomes, therefore, allowing fitness sharing to scale an individual's score for the protection of innovation. Table 3.3 shows the values used when measuring how different two NEAT genomes were. At the end of each generation, algorithm 3 divided the population into its distinct species by comparing against a genome representation that defined a species already discovered, else they were assigned a new species if one did not match. For this to be possible, an object was created that stored a random genome representation for all species discovered across all generations. This also helps if a species were to go extinct, its genome would still exist in the object, such that, any future topologies within the distance threshold would not be considered a new species.

Algorithm 3 Genome Speciation

Input: G (Network genome)
Output: G' (Network genomes with assigned species)

```
1: function Speciate($G$)
2:     while $s$ in $Species$ do
3:         if $\delta < \delta_t$ then return $G.Species = s$
4:     return $G.species$ = new $Species$
```

Table 3.3: Parameter values for the speciation of NEAT networks using the distance measure equation 2.8 A value of 3.0 for the weight coefficient c_3 was chosen because identical topologies can have very different output behaviour based on the connections weights. Therefore, network topologies may look very similar, but they can in fact be a different species.

NEAT Mutation Operators		
Parameter	**Description**	**Value**
Excess nodes coefficient c_1	Weight of excess gene numbers.	2.0
Disjoint nodes coefficient c_2	Weight of disjoint gene numbers.	2.0
Average weight differences coefficient c_3	Weight of average weight difference.	3.0
Distance threshold d_t	Threshold that determines if two genomes are of the same species.	4.0

3.5 Experiment Implementation

3.5.1 Hypotheses

From the research questions declared in section 1.1 the following hypotheses were derived and will be checked meet in order to the aims of the project:

H1: Networks evolved using different mutation rates will share the same behavioural effects.

H2: Speciated networks will perform better than sufficiently sized fixed networks when competing in the same domain.

A single experiment structure was designed that allowed for the observation of the effects of neural network evolution and was repeated 3 times. To keep as many of the variables discussed throughout 3.4 controlled, the only variable that changed for each experiment was the mutation rate as this had the most significance on exploring the solution space, the values used for the mutation rate were 0.1, 0.2 and 0.5. Keeping the structure of the experiment consistent meant it provided a fairer way to observe the effects each mutation rate had in order to evaluate the hypotheses **H1** and **H2**.

3.5.2 Experiment Structure

The experiment integrated two populations of agents within the environment discussed here **??**, one population of agents were controlled by NEAT speciated networks, where the other population of agents were controlled by a fixed topology network. The experiment ran for 1000 generations, where the goal for each generation was for the agents to score higher than the opposing agent, where a score of 10 was the maximum possible.

During the experiment, there were no restrictions to how many mutations could occur. Since the crossover and individual mutation operator probabilities are controlled to 0.7 and 0.2 respectively for speciated networks, the probability of a specific mutation occuring is dependent on its mutation rate. Therefore, any offspring produced had to following probability of being mutated:

Mutation rate of 0.1: $0.1 \cdot 0.7 \cdot 0.2 = 0.014$

Mutation rate of 0.2: $0.2 \cdot 0.7 \cdot 0.2 = 0.028$

Mutation rate of 0.5: $0.5 \cdot 0.7 \cdot 0.2 = 0.07$

3.5.3 Data Handling

From running the experiment, data was logged using the following methods:

1. Firefox Web Console.
2. Custom built script that would automatically gather data and download as Comma-separated values (.CSV) files.

This allowed for the following metrics to be analysed with respect to the effects of neural network evolution:

- Fitness Performance
- Genome Speciation

- Network Topology
- Network Outputs
- Computational Performance
- Performance in Achieving Environment Goal

3.6 Implementation Issues

During the implementation phase, several issues arose that would have had an impact on the experiment to be carried out, these included:

Cyclic Networks
Since the network structure followed a feed-forward pattern, it was important that no recurrent connections would appear or sequences of connections that resulted in cyclic paths. To overcome this, the node data type was assigned an index property, resulting in no connection allowed from a node with an index greater than the index it is going to. This was implemented as a required check before processing the mutation, if the check would have resulted in a cyclic path, two more nodes are randomly selected and will be checked again, repeating until the connection is verified to maintain a feed-forward structure.

Fitness Evaluation
During early implementation, the fitness evaluation rewarded networks that moved agents towards the good food as well as reducing when moving away. What this led to was a *jittering* movement which created an exploit in maximising fitness scores, whilst completely ignoring the main objective - to obtain a good food resource. The simple fix for this was to no longer provide a reward when moving towards the goal.

3.7 Implementation Summary

The chapter has discussed the implementation details of the system developed in order to meet the project aims. JavaScript was chosen as the language of choice due to its portability and ease of access of multiple devices. Details of the environment and agents were discussed, which followed with the design of the neural network structures and evolutionary algorithm. Finally, defining the hypotheses from which the experiment could be implemented.

Chapter 4

Results and Discussion

The results from the experiments outlined in 3.5.2 are presented in this chapter, beginning with the technical setup for which the experiment was conducted. Followed by the evaluation of the hypotheses defined in section 3.5.1.

4.1 Technical Setup

The experiment was carried out using the following hardware and software:

- AMD Ryzen 7 3700X CPU with 32 GB RAM
- EVGA GeForce GTX 1080 Ti
- Windows 10 Pro
- Firefox Version 75.0
- Code written in JavaScript

4.2 H1: Evolutionary Effects

The hyptohesis H1 predicted that *Networks evolved using different mutation rates will share the same behavioural effects.* To evaluate this hypothesis, the following results on network behaviour are analysed.

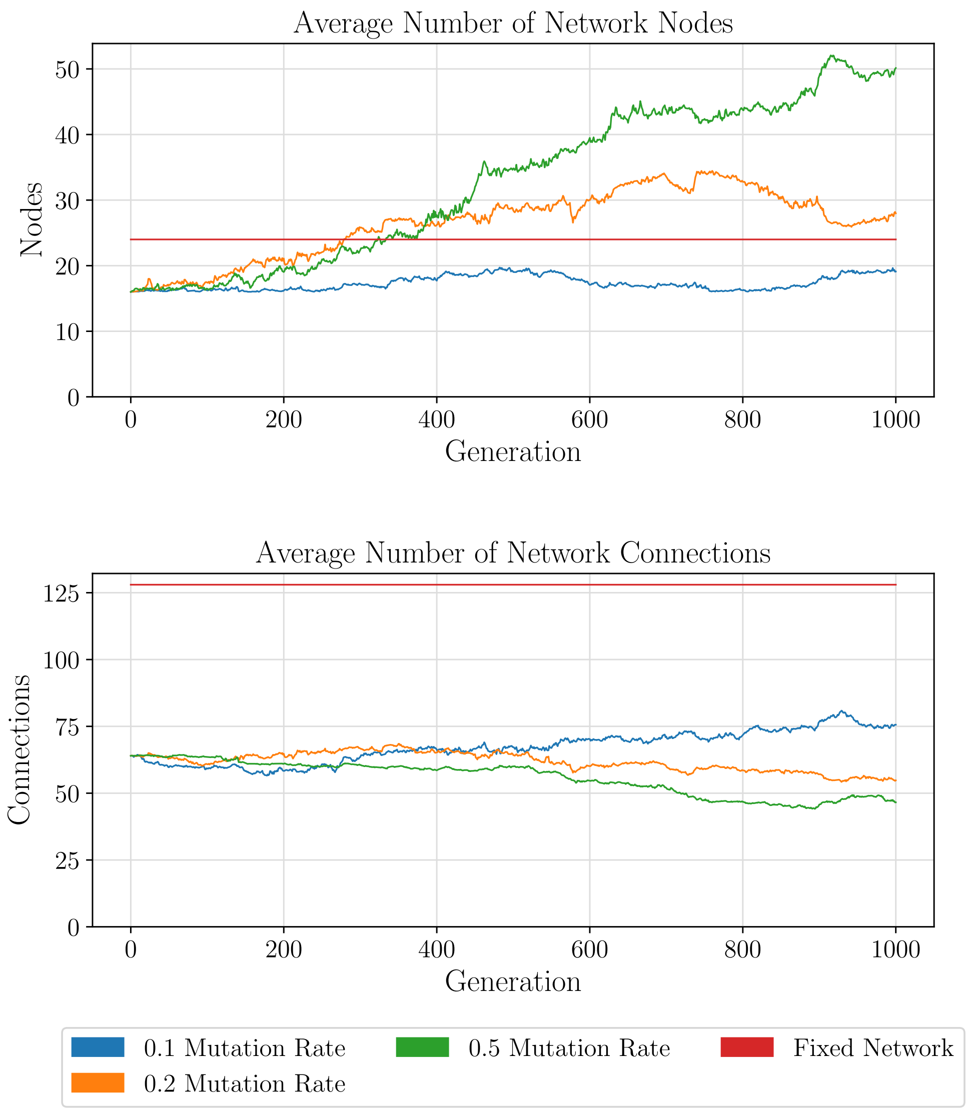


Figure 4.1: Evolutionary transformations of network topology under different mutation rates per generation. Mutation rate of 0.1 showed an overall increase in both network connections and network nodes. Mutation rate of 0.2 had an increase in network nodes, but a decrease in network connections. Where a mutation rate of 0.5 resulted in a significant increase in network nodes and the least amount of connections.

4.2.1 Network Topology

What can first be identified from 4.1 is there is no identical behaviour between how the topologies are mutated or evolve into the structure of the fixed network. Perhaps you could argue that a mutation rate of 0.1 was showing indication of a convergence, however, a much more exhaustive search would be required as 1000 generations was not enough. Interestingly, both the higher mutation rates of 0.2 and 0.5 could explore the solution space at an increased rate preferred to have network topologies with much lower network connections, from their respective initialisation structure and the connection count in the fixed network topology. As expected, each mutation rate had an increase in network nodes, which is essential in order to evaluate any non-linearity in the domain.

4.2.2 Network Computations

Looking at 4.2 and 4.3, there are similarities in network activity. The number of computations for a mutation rate of 0.1 in both types of topologies follows a pattern of low variance with an initial gradual decline. This would indicate that both topologies did not have enough time in order to develop a satisfactory strategy for the requirements on the environment, therefore, the agents they were controlling would have continuously kept dying by not being able to complete the objective, aside from this, speciated networks required around 25,000 computations whereas the fixed topology required 250,000 on average, this is greater by a factor of 10 before any plausible strategy had been found.

Following, a mutation rate of 0.2 (4.3) for fixed networks kept the same pattern as it did when using a mutation rate of 0.1, low variance and no increase in computations that would indicate any success of controlling an agent. However, in 4.2, the variance began to increase for a mutation rate of 0.2, as does the consistency of the number of computations being executed. This shows the evolutionary effects with a mutation rate of 0.2 acting on speciated networks was able to induce some structural innovations, such that the network behaviour was superior at controlling an agent to some extent as opposed to the fixed network. Similarly, there can be seen a factor of roughly 10 times as many compuations required by the fixed networks.

Finally, a mutation rate of 0.5 (4.2, 4.3) shows to have the greatest increase in variance of the number of computations required, in fact, it is seen to be higher in the fixed networks than it is in the speciated networks. With a mutation rate of 0.5, these topologies can now explore the solution space at a faster rate, resulting in them discovering a strategy more quickly. This improved the agents survivability in the environment, leading to the overall increase in number of computations. What is interesting to see, is even with the diverse topological differences 4.1 (0.5), the rate of computational requirement increased simultaneously between both types of networks. However, this only continued until generation 800, where this pattern then diverged, with speciated networks becoming their most computational demanding, yet still less than 5 times that of the fixed networks. What this divergence indicates is that the speciated networks have discovered a much more optimal strategy that the fixed networks cannot compete with, due to the nature of the environment, the good food is a limited resource, therefore it's impossible for both topologies to be most successful at any given time.

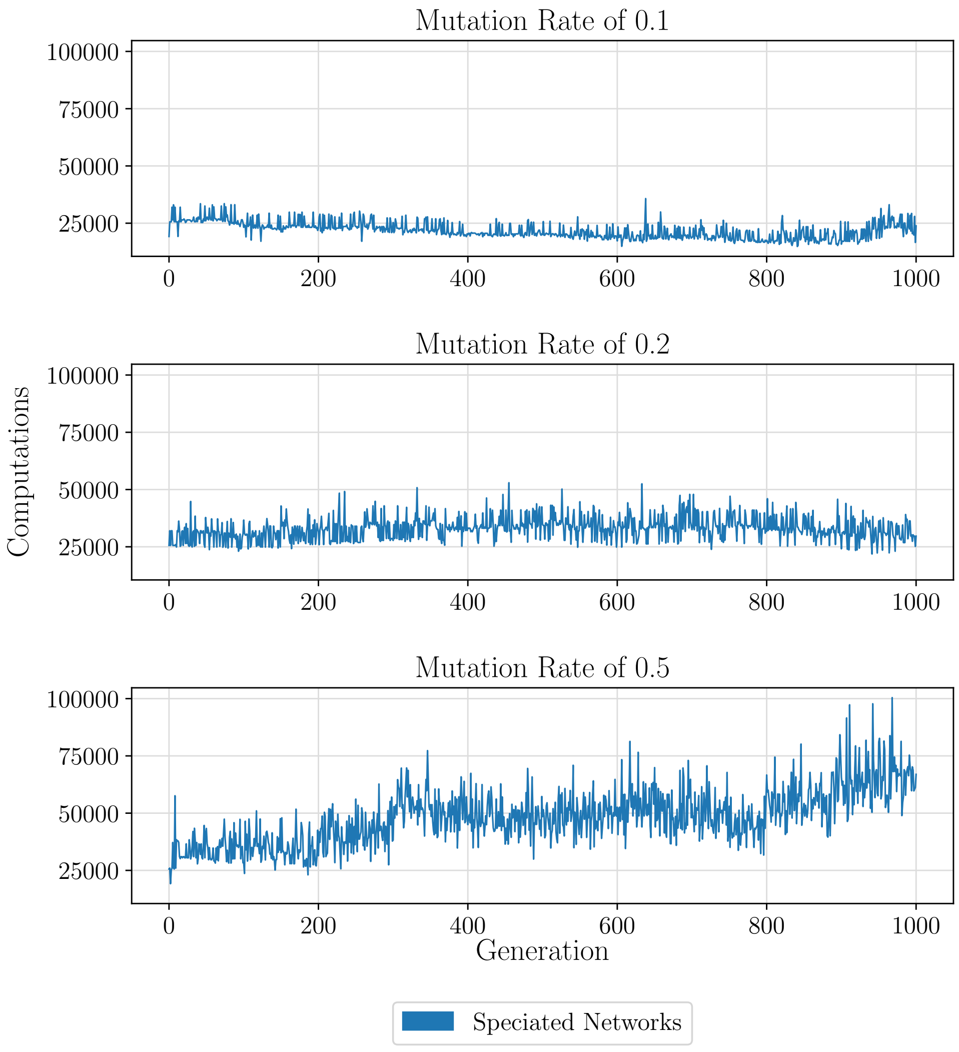


Figure 4.2: Average number of computations executed for speciated networks per generation. Mutation rate of 0.1 had the least variance in computations and showed a gradual decline until generation 1000. Mutation rate of 0.2 had a slight increase in variance, where Mutation Rate of 0.5 had the highest variance including the largest overall increase in computations.

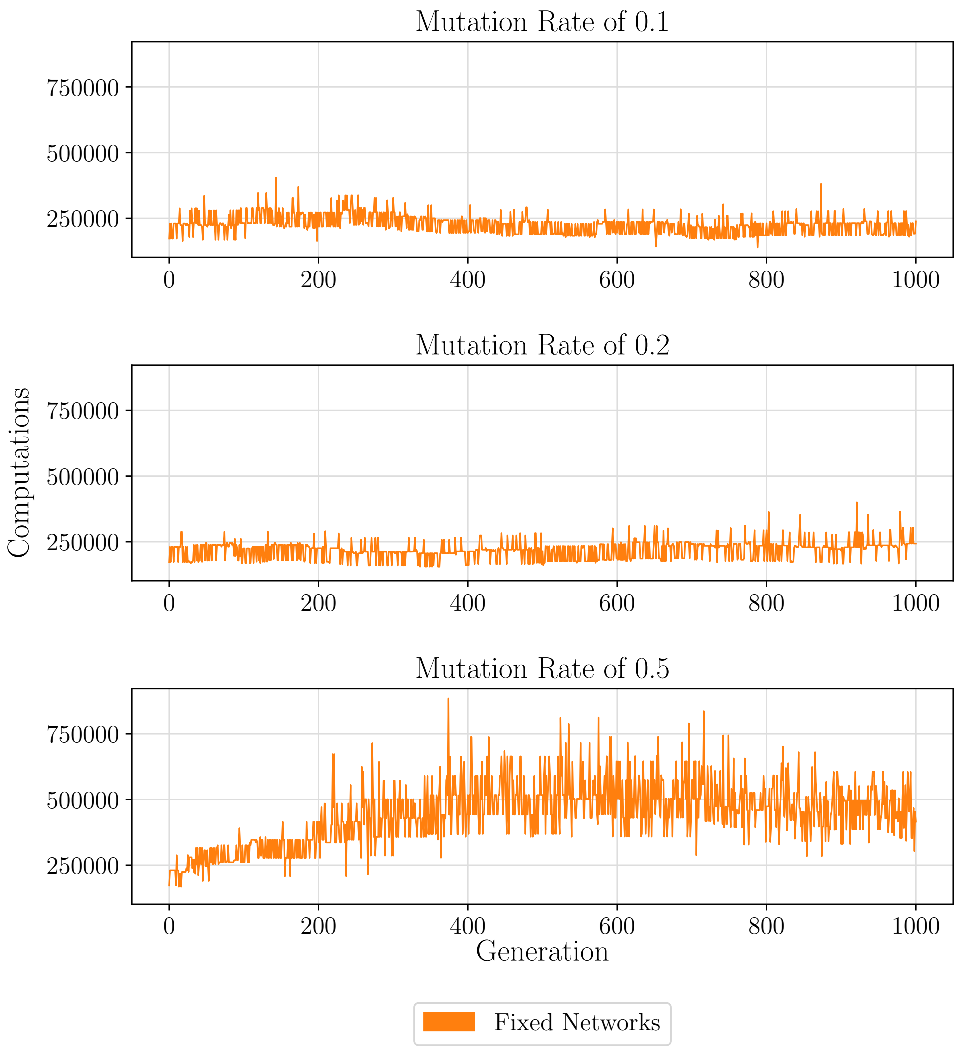


Figure 4.3: Average number of computations executed for fixed networks per generation. Mutation rate of 0.1 had the least variance in computations. Mutation rate of 0.2 consistently performed less computations than 0.1, until generation 600, where the variance increased. Mutation Rate of 0.5 had the highest variance including the largest increase in computations, until generation 1000, where the solution begins to optimise with lesser variance.

4.2.3 Network Output

Measuring the network output is primarily concerned with the frequency of which an output is selected and if there exist any similarities between the two distinct topologies being tested. Firstly, 4.4 are the network frequency outputs with a mutation rate of 0.1, what can be seen is that the fixed networks had a much more diverse output selection, this is due to them being initialized with a single hidden layer of 8 nodes to begin with, being able to evaluate non-linearity from the outset. Whereas the output frequency for speciated networks is very localised, mainly to outputs (2) and (6) as a result of lack of topological innovation. From this, it can be said that there are no similarities between the output frequency of speciated and fixed topology networks with a mutation rate of 0.1.

The output frequency in 4.5 can be observed to show a much similar behaviour between the networks. With a mutation rate of 0.2, the fixed networks evolved a more constrained output selection to just three outputs, likewise with the speciated networks. Looking closer at outputs (2) and (6), its clear to see how both topologies are evolving simultaneously, even with outputs (5) and (8), they may have distinct use between either topology, yet, it is clear to see the similarities in frequency, one decreases as does the other. However, even though the frequency of these outputs are extremely similar, it is already shown that speciated networks under a mutation rate of 0.2 are many factors more efficient than fixed networks when computing these outputs.

Finally, 4.6 shows that a mutation rate of 0.5 had an increased output selection range for speciated networks, when compared to rates of 0.2 and 0.1. Similarly, the favourable outputs (2), (5) and (8) were shared by both types of topologies. The divergent behaviour seen in 4.2 with a mutation rate of 0.5 can also be seen here, where the output frequency increased for speciated networks and decreased for fixed networks as they tend towards 1000 generations.

Looking at the output frequency for each individual mutation rate, there is no clear indication of a preferred selection of outputs for both types of networks, even though the rules and settings of the environment remained consistent throughout.

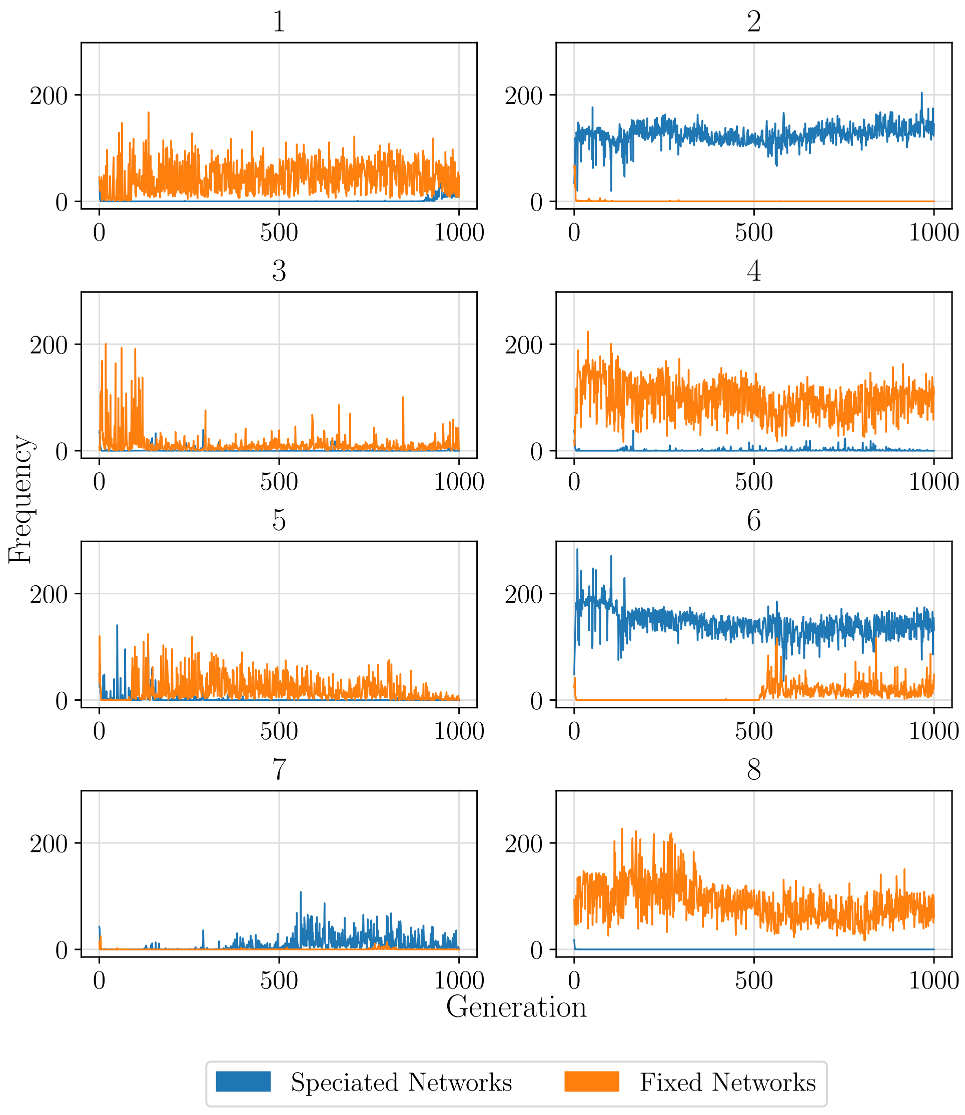


Figure 4.4: Frequency of the 8 output nodes between speciated networks and fixed networks per generation with a mutation rate of 0.1.

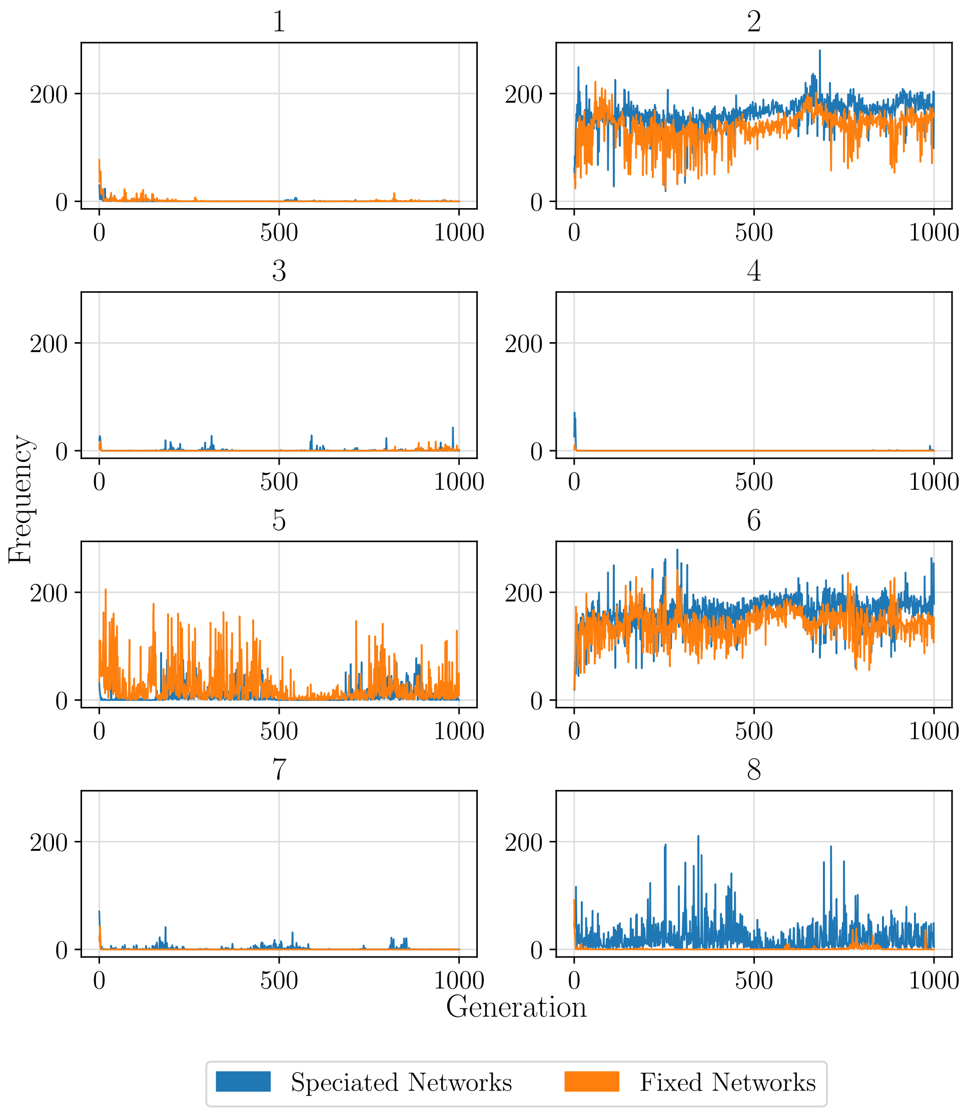


Figure 4.5: Frequency of the 8 output nodes between speciated networks and fixed networks per generation with a mutation rate of 0.2. There can be seen similarities in network behaviour, where (1), (3), (4) and (7) are essentially disregarded for both network types. Outputs (2) and (6) show a pattern in frequency during the evolution process, where both increase and decrease together.

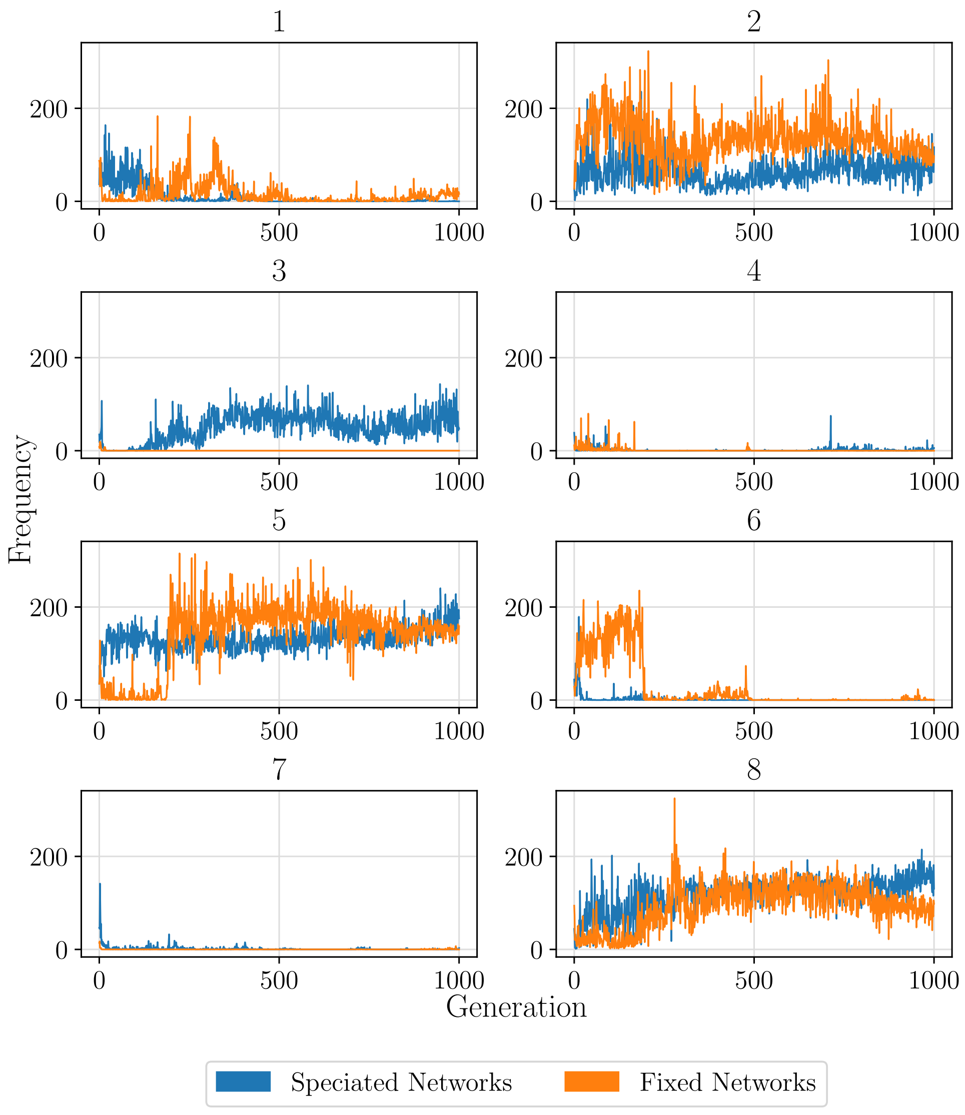


Figure 4.6: Frequency of the 8 output nodes between speciated networks and fixed networks per generation with a mutation rate of 0.5. Outputs (2), (5) and (8) have the highest frequency and (4), (6) and (7) are essentially disregarded by both.

4.2.4 Observable Effects of Evolution

Optimisation through Crossover

As shown above, a mutation rate of 0.1 had an overall increase in both network nodes and connections. An observation to take from this is that the parents being selected for crossover had minimal excess and disjoint nodes, meaning the majority of the genes were matching. Therefore, the search for a better solution of offspring appears to happen through the optimisation of the network weights from the uniform crossover of genes, essentially, the pre-existing genes in the population pool were being exploited rather than new ones being explored. Also, since there is minimal coexisting species in 4.7 (Mutation Rate of 0.1), any new topological innovations were unlikely to survive due to fitness sharing, unless it had an immediate positive impact on fitness performance or no impact at all.

Large Genome Exploration

As seen in, 4.1 a mutation rate of 0.5 had a significant increase in node mutations being added into the network topology, yet, an overall decrease in network connections. This resulted in network genomes having a lot of redundant nodes genes without any connection genes forming a connected path. Because of how the crossover implementation works for speciated networks, by passing on all excess and disjoint genes from the fittest parent, it can be seen that these larger genomes are producing the fittest individuals. Yet, from 2.8.1, the network topology is constructed using the genome encoding, but it is only the functional components of this structure that are used for evaluations during execution, so why might it be that these large genomes with redundant information dominate the population? One observation would be that these genomes maintain a large solution space at any given time, meaning connections can easily be added to these redundant nodes, creating a new path and providing a wider range of innovations without the prior need of node insertions - since they already exist in the genome encoding. The effects of this would also contribute to the higher variances seen in the other metrics observed above.

The results and observations above show that different mutation rates acting on the network topologies for 1000 generations produced distinctive behavioural effects with respect to the evolution of topology. However, output frequency between speciated and fixed networks showed similarities.

4.3 H2: Speciation Performance

The hypothesis H2 predicted that speciated networks will perform better than sufficiently sized fixed networks when competing in the same domain. To check this, the speciation of networks was recorded, along with the fitness performances of both speciated networks and fixed networks. The performance with respect to the environment goal is also compared.

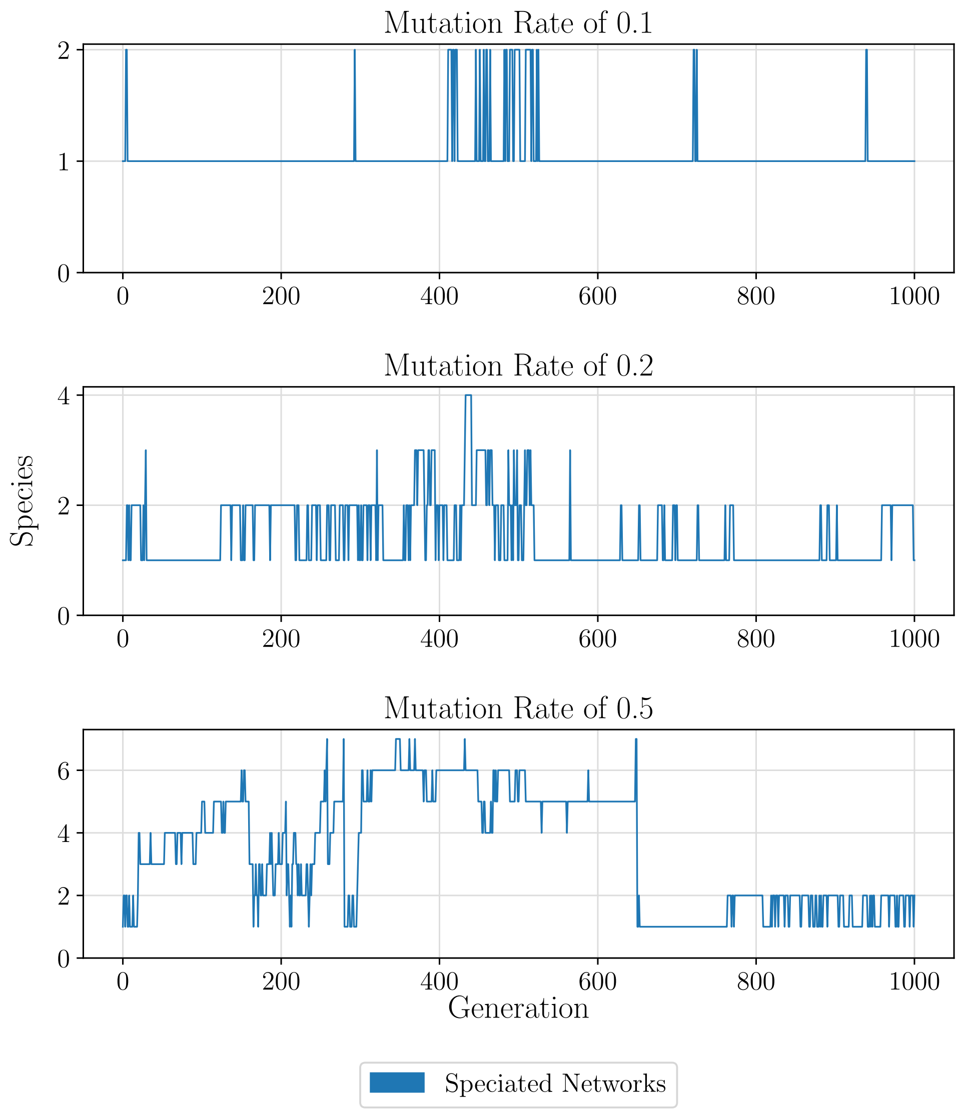


Figure 4.7: Number of distinct species that the population of 100 networks was divided into per generation. With a mutation rate of 0.1, the population was mainly dominated by a single species. Mutation rate of 0.2 showed increased rates of speciation, with a maximum of 4 species existing at a given time. Mutation rate of 0.5 shows the highest levels of speciation, with a maximum of 7 species existing at a given time.

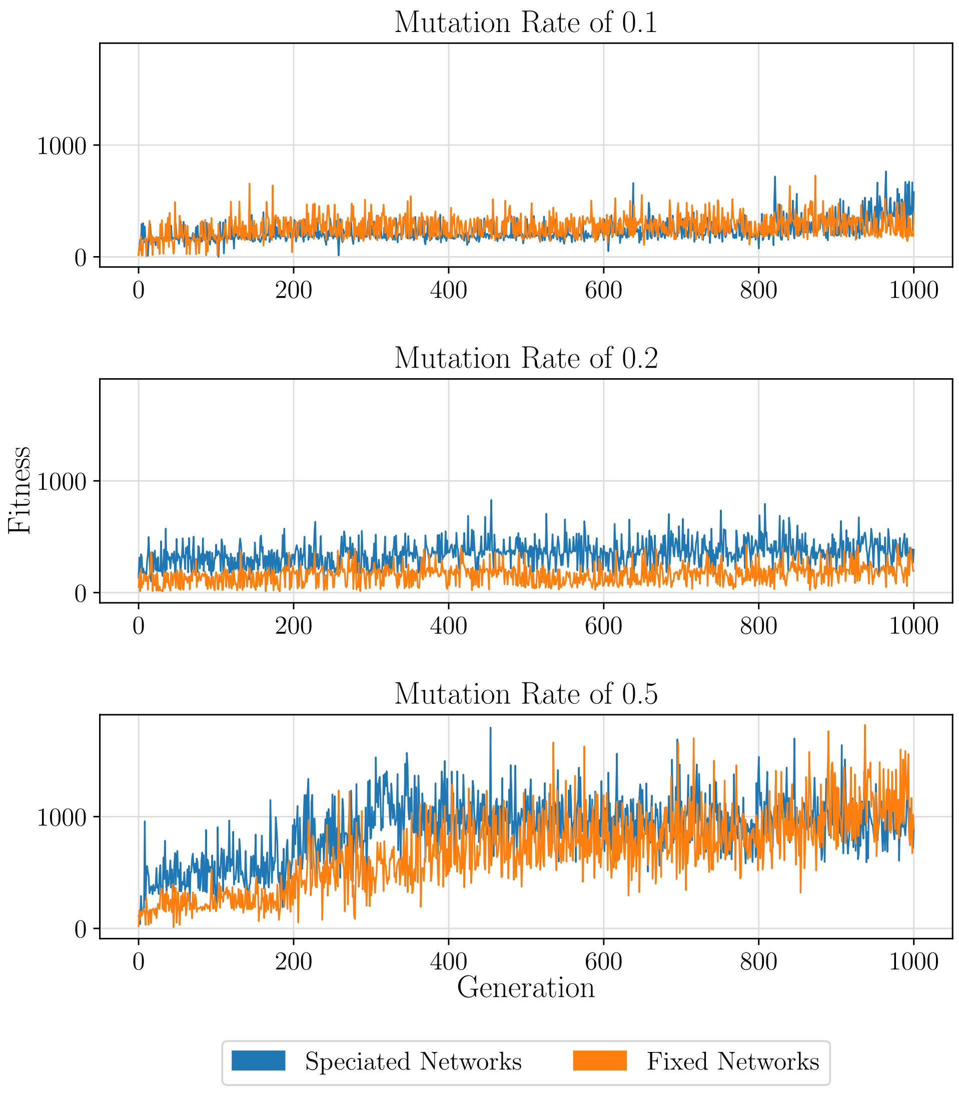


Figure 4.8: Fitness performance of the best species against the best fixed topology network per generation. Mutation rate of 0.1 shows consistent levels of performance between both networks, until generation 1000, where speciated networks show a sudden increased performance. Mutation rate of 0.2 shows speciated networks maintained better performance levels throughout all 1000 generations.

4.3.1 Speciation and Performance

Looking at 4.7, it is clear to see how the different mutation rates have an impact of the number of species discovered as well as the number of species existing in a single population at any given time. Overall it is seen that most speciation occurs during the first 700 generations for each mutation rate. Though, a mutation rate of 0.5 is shown to clearly maintain a more diversified search space through multiple species coexisting, each providing their own niche solution.

When comparing these levels of speciation to the fitness performance in 4.8, for a mutation rate of 0.1, it can be seen that there is no observable impact of speciation on fitness performance. For a mutation rate of 0.2, there is an initial increase in speciation as well as fitness performance, however, speciation levels decrease, including the amount of coexisting species, but the fitness performance remains somewhat consistent above that of the fixed topology. Finally, for a mutation of 0.5, there is an increased rate of speciation as with fitness performance, however, between generations 400 - 650, was the highest period of species coexisting, where fitness performance showed an overall decline.

After 1000 generations mutation rates of 0.1 and 0.2 for speciated networks finished with slightly better performance levels than fixed networks, contrast to mutation rate of 0.5 where fixed networks finished with similar performance. This may be an indication that higher counts of coexisting species has a detrimental effect on network evolution, but more data would be required to verify. However, it is most likely the case of fixed networks learning a more optimal strategy, since similar patterns can be seen in 4.9, where the frequency of each outcome for a mutation rate of 0.5 converges to 33.3%

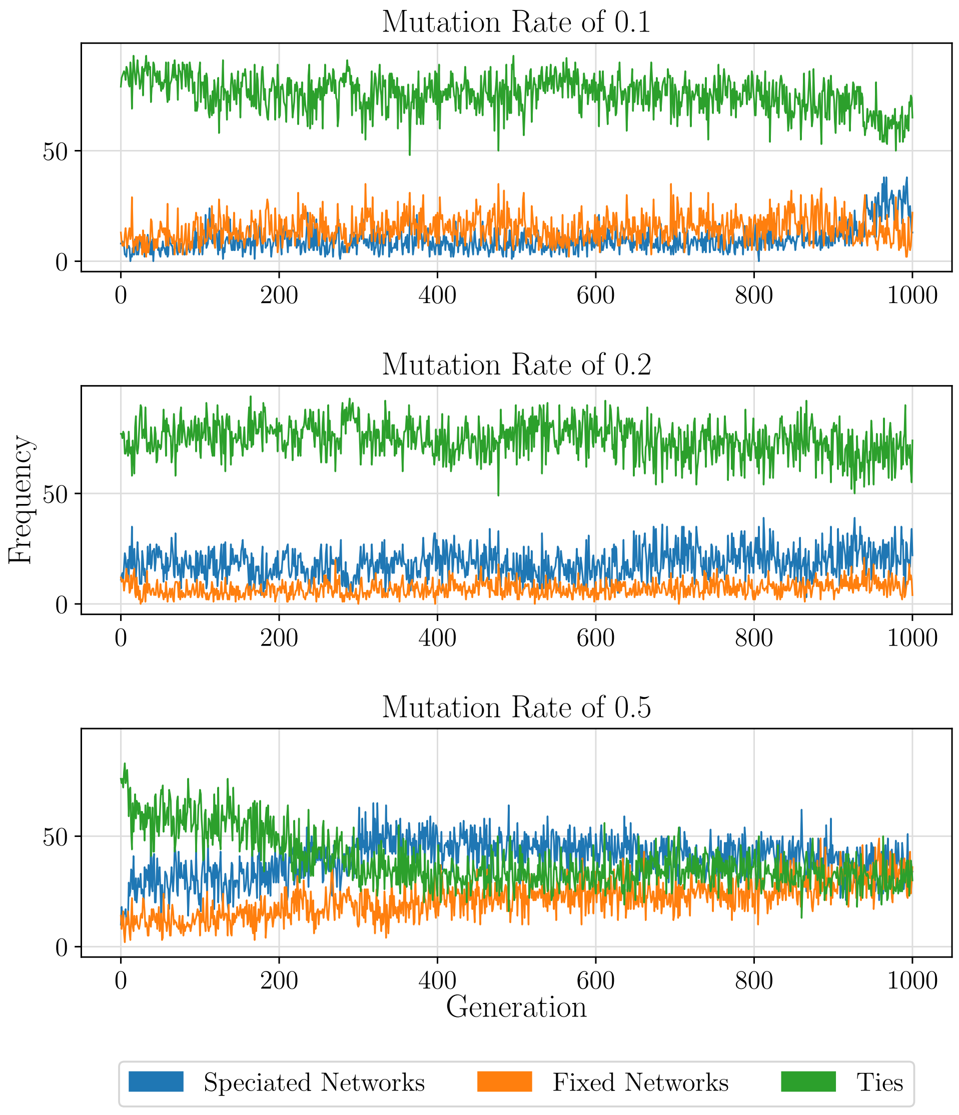


Figure 4.9: The frequency of winning environments for speciated networks versus fixed networks when controlling an agent in a competitive environment. A mutation rate of 0.1 shows fixed networks were more successful than speciated networks until towards the end of the evolutionary process, where speciated networks became more successful, yet overall, more than 50% environments ended as a tie. Similarly for a mutation rate of 0.2, more than 50% ended as a tie, however, speciated networks were consistently more successful over the 1000 generations. Mutation rate of 0.5 shows speciated networks were very successful, achieving a win rate over 50%, until fixed networks evolved an equally optimal strategy, with all results converging.

4.4 Observation of the Speciation of Networks

The observation made below is shown to have a potential impact on the rate of speciation, especially in higher mutation rates.

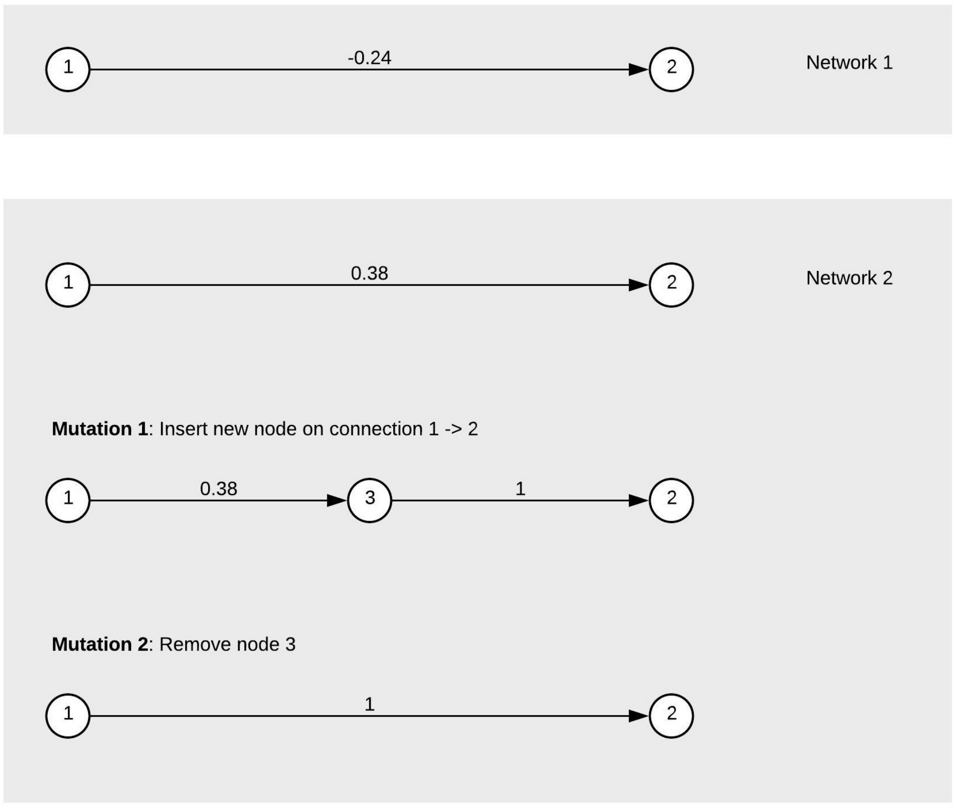

1
-0.24
2
Network 1
1
0.38
2
Network 2
Mutation 1: Insert new node on connection 1 -> 2
1
0.38
3
1
2
Mutation 2: Remove node 3
1
1
2

Figure 4.10: Mutation issue that may lead to a bias when calculating the average weight differences for speciation of a network.

4.4.1 Weight Bias

From these results, there is an observable property of the mutation operation that inserts a node, which may result in a bias in the speciation of a network. From the implementation, it is stated that all networks are initialized with a uniform weight distribution with values between -1 and 1. 4.10 shows two networks sharing a matching gene, with input (1) and output (2) nodes and weights -0.24 and 0.38 respectively. A sequence of node specific mutations results in network 2 with a weight of 1, the weight difference is now to be more distinctive than it originally was - without any weight specific mutations. Since input and output nodes are immutable node (3) would have had priority for removal. This may be a plausible reason for the increased speciation in the early generations of higher mutation rates seen in 4.7 and why there is minimal speciation in the later generations. However, further testing would be required to confirm.

From the above discussions and observations, it is not conclusive to say that speciation of networks has a positive impact on the performance compared to fixed networks.

4.5 Results Summary

This chapter has presented and analysed the results gathered from the experiment that was implemented. Although the number of generations explored was limited due to time constraints, it was still possible to evaluate the hypotheses regarding the effects of neural network evolution.

Hypotheses 1 was checked by evaluating the network topologies under different mutation rates showed differing methods of optimisation, a mutation rate of 0.1 favoured the increased rates of crossover to pass on more optimised genes, whereas a mutation rate of 0.5 produced large network genomes, maintaining a large solution space to be explored. Followed by identifying similarities in the frequency of network outputs, specifically in mutation rates of 0.2 and 0.5, as pairings of output frequency appeared to coevolve.

Finally hypotheses 2 checked the effects of speciation with respect to fitness performance, but no direct indication of relatedness was found.

Chapter 5

Conclusion

This chapter begins by answering the research questions laid out in 1.1 with use of the results previously presented. This leads to addressing the limitations of the implementation and discussing possible areas of future work. Finishing with the concluding remarks of the project.

5.1 Research Questions

The project aims introduced the following research questions to be answered:

5.1.1 Research Question 1

Does the mutation rate affect the evolutionary behaviour of the underlying network?

From the results discussed in the last chapter, it was seen that varying rates of mutation did in fact have distinctive evolutionary behaviours with respect to topology for speciated networks. A mutation rate of 0.1 was observed to optimise through crossover of matching genes, whereas a mutation rate of 0.5 developed large genomes of redundant information thus maintaining a greater solution space by passing it on from generation to generation. Comparing speciated networks to fixed networks showed higher mutation rates had similar output behaviour, to the extent where individual output frequencies would change at the same rate, essentially coevolving. Even though 1000 generations has shown some plausible results, a more thorough search would allow the possibility to see if these behaviours are maintained.

5.1.2 Research Question 2

Does the implementation of speciated networks provide better performance than that of a sufficiently sized fixed topology network when put head-to-head in a shared domain?

To answer this, a competitive environment was created that allowed two agents to be controlled by distinct topologies, one by speciated networks and the other by a fixed network. From the previous results, it was found there was no indication that the use of speciating network topologies in a competitive domain can find better performing solutions. In fact, in some cases where there was a high number of coexisting species, it could be seen that overall fitness performances declined in comparison to the fitness performance of a fixed topology. However, it is not conclusive, as the observation of weight bias from mutations could impact the levels of speciation when network topologies were small. Again, this issue may have a much more conclusive answer if an exhaustive search were able to be applied.

5.2 Limitations

5.2.1 Neural Network Analysis

It became difficult to measure and analyse the resulting networks during the evolution process. The initial structures of a neural network were shown in the implementation, but as the network evolves it's impossible to predict how many nodes and connections will appear. Even when using such a small population size of 100, it would have been an extremely arduous process to measure each individual network. Therefore, it was not feasible to analyse clearer representations of the evolving topologies during the generations explored and was limited to comparisons using an overview of the relative genome sizes.

5.2.2 Performance

Although JavaScript allowed for portability for testing and running in multiple locations, it did not offer the best performance compared to some other better suited languages, such as C++, which would have allowed for much more optimal memory management when dealing with the large data structures that were encountered with this project. Although using such appropriate languages do have their own overheads, mainly that of compilation and build time would not appear to have been an issue due to the final size of the codebase.

5.3 Future Work

Below are directions of future work that could be implemented based on the outcomes of this project.

5.3.1 Network Structure

The network structure use was a simple feed-forward structure, this resulted in its own limitations, specifically that of cyclic paths. A better structure may be of a recurrent network, allowing for connections in both directions. A recurrent structure would also be of a benefit in an environment where memory may be required to discover an optimal solution.

5.3.2 Weight Initialization

Throughout this project, all connection weights during the initialization of the network structure had a uniform distribution of random values ranging between (-1.0) and (1.0), as previously stated this was to avoid any redundant generations of inactivity in the early phases of evolution and fit within the limitations of the project schedule. However, this may have induced biased behaviour from the outset, a fairer method would initialize all weights with an equal value, though, this would require many more generations to be explored before sufficient strategies would begin to evolve. This would also be beneficial to the weight bias observed in the discussion.

5.4 Conclusion

This dissertation set out to identify the effects of neural network evolution when being mutated at different rates. This was achieved by the development of a system that allowed for the implementation of various neural network structures to be evolved through the use of a genetically inspired algorithm. The work in this dissertation was motivated by the area of neuroevolution, outlined in section 1

Reviewing the work of neuroevolution and the advancements through NEAT inspired topologies, the main body of this dissertation was dedicated to the design and implementation of a system that would allow the research questions in section 1.1 to be answered. Although the results were not conclusive it was shown that using a sufficiently sized fixed network can to a certain extent share the evolutionary effects as speciating networks when operating in the same domain, but a deeper evaluation is still required.

However, this project resulted in the successful implementation of a web-based system capable of the construction and observation of the behaviours of distinct neural network topologies in a competitive environment.

Chapter 6

Appendix

6.1 Ethics Checklist

UNIVERSITY OF BATH

Department of Computer Science

12-Point Ethics Checklist for UG and MSc Projects

Kurtis McAlpine

Student

Effects of Neural Network Evolution

Academic Year or Project Title

Dr Rob Wortham

Supervisor

This form must be attached to the dissertation as an appendix.

Does your project involve people for the collection of data other than you and your supervisor(s)? YES / [✓ NO]

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Have you prepared a briefing script for volunteers?* YES / NO

   Briefing means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* YES / NO

   All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Is there any intentional deception of the participants?* YES / NO

Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* YES / NO

The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. *Will participants voluntarily give informed consent?* YES / NO

Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete.

6. *Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)?* YES / NO

Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.

7. *Are you offering any incentive to the participants?* YES / NO

The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

8. *Are you in a position of authority or influence over any of your participants?* YES / NO

A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. *Are any of your participants under the age of 16?* YES / NO

Parental consent is required for participants under the age of 16.

10. *Do any of your participants have an impairment that will limit Their understanding or communication?* YES / NO

Additional consent is required for participants with impairments.

11. *Will the participants be informed of your contact details?* YES / NO

All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.

12. *Do you have a data management plan for all recorded data?* YES / NO

All participant data (hard copy and soft copy) should be stored securely, and in anonymous form, on university servers (not the cloud). If the study is part of a larger study, there should be a data management plan.

Bibliography

Angeline, P. J., Saunders, G. M. and Pollack, J. B. (1994), 'An evolutionary algorithm that constructs recurrent neural networks', *Trans. Neur. Netw.* **5**(1), 54–65.
URL: *https://doi.org/10.1109/72.265960*

Bäck, T. (1996), *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, Inc., New York, NY, USA.

Bishop, C. (2006), *Pattern Recognition and Machine Learning*, Springer-Verlag New York Inc.
URL: *https://www.springer.com/gp/book/9780387310732*

Blickle, T. and Thiele, L. (1996), 'A comparison of selection schemes used in evolutionary algorithms', *Evol. Comput.* **4**(4), 361–394.
URL: *http://dx.doi.org/10.1162/evco.1996.4.4.361*

Chen, Y. et al. (2008), Solving deceptive problems using a genetic algorithm with reserve selection, pp. 884 – 889.

Chinchalkar, S. (1996), 'An upper bound for the number of reachable positions', *ICGA Journal* **19**, 181–183.

David, O. E. and Greental, I. (2014), Genetic algorithms for evolving deep neural networks, *in* 'Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation', GECCO Comp '14, ACM, New York, NY, USA, pp. 1451–1452.
URL: *http://doi.acm.org/10.1145/2598394.2602287*

De Jong, K. A. (1975), An Analysis of the Behavior of a Class of Genetic Adaptive Systems., PhD thesis, Ann Arbor, MI, USA. AAI7609381.

Eiben, A. E. and Schippers, C. A. (1998), 'On evolutionary exploration and exploitation', *Fundam. Inf.* **35**(1-4), 35–50.
URL: *http://dl.acm.org/citation.cfm?id=297119.297124*

Flower, D. et al. (2006), Simulated visual perception-based control for autonomous mobile agents., Vol. 2006, pp. 725–730.

Foerster, J. N. et al. (2016), 'Learning to communicate with deep multi-agent reinforcement learning'.

Fogel, D. B. (1995), *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, Piscataway, NJ, USA.

François-Lavet, V. et al. (2018), 'An introduction to deep reinforcement learning', *Foundations and Trends® in Machine Learning* **11**(3-4), 219–354.
URL: *http://dx.doi.org/10.1561/2200000071*

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Goodfellow, I., Bengio, Y. and Courville, A. (2016), *Deep Learning*, MIT Press. `http://www.deeplearningbook.org`.

Hassanat, A. et al. (2018), 'An improved genetic algorithm with a new initialization mechanism based on regression techniques', *Information* **9**, 167.

Hausknecht, M. et al. (2014), 'A neuroevolution approach to general atari game playing', *Computational Intelligence and AI in Games, IEEE Transactions on* **6**, 355–366.

Herrera, F. and Lozano, M. (1996), 'Adaptation of genetic algorithm parameters based on fuzzy logic controllers'.

Holland, J. H. (1992*a*), *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, MIT Press, Cambridge, MA, USA.

Holland, J. H. (1992*b*), 'Genetic algorithms', *Scientific American* **267**(1), 66–73.
URL: *http://www.jstor.org/stable/24939139*

Jain, A. K., Mao, J. and Mohiuddin, K. M. (1996), 'Artificial neural networks: A tutorial', *IEEE Computer* **29**, 31–44.

Kelemen et al. (2008), *Computational Intelligence in Medical Informatics*, 1st edn, Springer Publishing Company, Incorporated.

Kreinovich, V. (2001), 'Genetic algorithms: What fitness scaling is optimal?', **24**.

Kurenkovi, A. (2015), 'A 'brief' history of neural nets and deep learning'.
URL: *https://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/l*

LeCun, Y. et al. (2015), 'Deep learning', *Nature* **521**, 436–44.

Lehman, J. and Miikkulainen, R. (2013*a*), 'Neuroevolution', *Scholarpedia* **8**(6), 30977. revision #137053.

Lehman, J. and Miikkulainen, R. (2013*b*), 'Neuroevolution', *Scholarpedia* **8**(6), 30977. revision #137053.

Liu, S.-H. et al. (2013), 'Exploration and exploitation in evolutionary algorithms: A survey', *ACM Comput. Surv.* **45**(3), 35:1–35:33.
URL: *http://doi.acm.org/10.1145/2480741.2480752*

Maddison, C. J. et al. (2014), 'Move evaluation in go using deep convolutional neural networks'.

Mehlhorn, K. et al. (2015), 'Unpacking the exploration–exploitation tradeoff: A synthesis of human and animal literatures.', *Decision* **2**.

Michalewicz, Z. (1996), *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*, Springer-Verlag, Berlin, Heidelberg.

Minsky, M. and Papert, S. (1969), Perceptrons - an introduction to computational geometry.

Nair, V. and Hinton, G. E. (2010), Rectified linear units improve restricted boltzmann machines, *in* 'Proceedings of the 27th International Conference on International Conference on Machine Learning', ICML'10, Omnipress, USA, pp. 807–814.
URL: *http://dl.acm.org/citation.cfm?id=3104322.3104425*

Neal, R. M. (1992), 'Connectionist learning of belief networks', *Artif. Intell.* **56**(1), 71–113.
URL: *http://dx.doi.org/10.1016/0004-3702(92)90065-6*

Nielsen, M. A. (2018), 'Neural networks and deep learning'.
URL: *http://neuralnetworksanddeeplearning.com/*

Nwankpa, C. et al. (2018), 'Activation functions: Comparison of trends in practice and research for deep learning'.

Olgac, A. and Karlik, B. (2011), 'Performance analysis of various activation functions in generalized mlp architectures of neural networks', *International Journal of Artificial Intelligence And Expert Systems* **1**, 111–122.

Radcliffe (1993), 'Genetic set recombination and its application to neural network topology optimisation', *NEURAL COMPUTING AND APPLICATIONS* **1**, 67–90.

Rahnamayan, S. and Wang, G. (2009), Center-based sampling for population-based algorithms, pp. 933–938.

Roessingh, J. J. et al. (2017), Machine learning techniques for autonomous agents in military simulations — multum in parvo, *in* '2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)', pp. 3445–3450.

Rojas, R. (1996*a*), *Neural Networks: A Systematic Introduction*, Springer-Verlag, Berlin, Heidelberg.

Rojas, R. (1996*b*), *Neural Networks: A Systematic Introduction*, Springer-Verlag, Berlin, Heidelberg.

Rosenblatt, F. (1958), 'The perceptron: A probabilistic model for information storage and organization in the brain', *Psychological Review* pp. 65–386.

Sastry, K., Goldberg, D. and Kendall, G. (2005), *Genetic Algorithms*, Springer US., Boston, MA, USA.

Schaffer, J. D. et al. (1992), Combinations of genetic algorithms and neural networks: a survey of the state of the art, *in* '[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks', pp. 1–37.

Schmidhuber, J. (2015), 'Deep learning in neural networks: An overview', *Neural Networks* **61**, 85–117.
URL: *http://dx.doi.org/10.1016/j.neunet.2014.09.003*

Schwab, B. (2004), *Ai Game Engine Programming (Game Development Series)*, Charles River Media, Inc., Rockland, MA, USA.

Shani, L. et al. (2018), 'Exploration conscious reinforcement learning revisited'.

Shukla, A. et al. (2015), Comparative review of selection techniques in genetic algorithm, *in* '2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)', pp. 515–519.

Silver, D. et al. (2016), 'Mastering the game of go with deep neural networks and tree search', *Nature* **529**, 484–489.

Silver, D. et al. (2018), 'A general reinforcement learning algorithm that masters chess, shogi, and go through self-play', *Science* **362**(6419), 1140–1144.
URL: *https://science.sciencemag.org/content/362/6419/1140*

Spears, W. and Jong, K. A. D. (1995), On the virtues of parameterized uniform crossover.

Stanley, K. and Miikkulainen, R. (2002), 'Evolving neural networks through augmenting topologies', *Evolutionary Computation* **10**(2), 99–127.

Sutton, R. S. (1984), Temporal Credit Assignment in Reinforcement Learning, PhD thesis. AAI8410337.

Sutton, R. S. and Barto, A. G. (1998), *Reinforcement Learning: An Introduction*, MIT Press.
URL: *http://www.cs.ualberta.ca/ sutton/book/the-book.html*

Svozil, D., Kvasnicka, V. and Pospíchal, J. (1997), Introduction to multi-layer feed-forward neural networks.

Turian, J. et al. (2009), Quadratic features and deep architectures for chunking., pp. 245–248.

Umbarkar, D. A. and Sheth, P. (2015), 'Crossover operators in genetic algorithms: A review', *ICTACT Journal on Soft Computing (Volume: 6 , Issue: 1)* **6**.

Zeiler, M. D. et al. (2013), On rectified linear units for speech processing, *in* '2013 IEEE International Conference on Acoustics, Speech and Signal Processing', pp. 3517–3521.

Zhao Yanling et al. (2002), Analysis and study of perceptron to solve xor problem, *in* 'The 2nd International Workshop on Autonomous Decentralized System, 2002.', pp. 168–173.

Zhong, J. et al. (2005), Comparison of performance between different selection strategies on simple genetic algorithms., Vol. 2, pp. 1115–1121.