Applications of The NEAT Algorithm in Deterministic Game Environments

John Kechagias
Department of Informatics, University Of Piraeus
Supervised by Dionisios Sotiropoulos

June 19, 2024

Abstract

This paper explores the use of the NEAT algorithm combined with self-play in deterministic game environments. Our objective is to investigate how quickly models can adopt specific strategies with minimal reliance on training data and expert knowledge. We developed a game similar to the board game Catan as a platform for training models. Multilayer feedforward neural networks were used to evaluate board positions, and games were played using a minimax search strategy. In each generation, neural networks competed against the best networks from the previous generation. Poorly performing networks were eliminated, and the survivors produced offspring through crossover and mutation.

Keywords: Artificial Neural Networks · NEAT algorithm · Deterministic Game Environments · Self-play

1 Introduction

In recent years, the field of artificial intelligence has seen significant advancements in training models to perform complex tasks, particularly within game environments. One promising approach in this domain is the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [23], which evolves neural networks by adjusting both their weights and topologies. A key advantage of NEAT is its ability to avoid getting trapped in local optima, enabling the discovery of more effective strategies over time, even on complicated fitness landscapes. When combined with self-play, where agents learn by playing against themselves, NEAT shows great potential for training agents in a variety of strategic games.

This paper investigates the efficacy of the NEAT algorithm in combination with self-play to train agents in deterministic game environments. Our primary objective is to evaluate the speed and efficiency with which these models can learn specific strategies, minimizing the need for training data and usage of expert knowledge. To provide a concrete demonstration

of this methodology, we developed a game environment similar to the board game Catan. This platform allows us to systematically train models to adopt predefined strategies. By focusing on a deterministic game, we ensure that the outcomes are solely influenced by the strategies employed, thus providing a clear measure of the effectiveness of our training approach. We aim to highlight the strengths and potential limitations of integrating NEAT with self-play through a series of experiments. The results obtained from our study could help better understand the advantages that NEAT offers in such environments and lead to the development of more efficient and accessible methods for training AI agents, not only in games but also in other domains where strategic decision-making is required.

The paper is structured as follows. Firstly, we introduce the NEAT algorithm and self-play, providing some context regarding their histories, concepts, and functionalities. Then, we introduce the game environment utilized. After that, we give an overview of the training process and the gameplay algorithm used. Finally, we present the experimental results, followed by the conclusion and future work.

2 Background

This section provides an overview of the history of the NEAT and self-play algorithm.

2.1 Neuroevolution

Neuroevolution (NE) is a branch of artificial intelligence that utilizes evolutionary algorithms to produce artificial neural networks [1, 4], rules and parameters [22, 5, 6]. It is primarily applied to general game playing (GGP), robot control [19] and artificial life [16]. Drawing inspiration from Darwinism, evolutionary algorithms work by creating successive generations of neural networks mutating them and then evaluating them. After the networks have been assessed, the fittest of them are selected for reproduction and create the next generation, leading to fitter offspring in the long-run. The main advantage of Neuroevolution is that it only requires a measure of a network's performance at a task, as it searches for a behaviour instead of a value function. This has two distinct benefits. Firstly, because neuroevolution does not take into account the fitness landscape, as opposed to more generally used optimization methods like gradient descent, it is less likely to get stuck on local minima. Secondly, it does not require curated datasets of input-output pairs, as opposed to supervised learning algorithms.

In traditional NE approaches, evolving networks have a predetermined structure that is chosen before the experiment. This usually corresponds to a single hidden layer where each neuron is connected to all input and output neurons. By using a fixed layout of neurons the problem becomes a weight optimization problem where we search through the weight space for optimal values. However, it isn't always easy to select a good representation in advance. Firstly, the complexity of the structure should reflect the complexity of the problem, therefore a singular structure cannot be ideal for all problems. Secondly, although a single hidden layer is able to represent any function, it does not mean that every function can be easily represent by one. This can be partially solved by the addition of subsequent hidden layers, but this grows and complexifies the search space exponentially. Additionally, the growth of the search space exacerbates the problem of competing conventions, where different networks correspond to the same function and crossover leads to damaged offspring. This paradigm of static representations shifted in the 1990s with the emergence of topology and weight evolving artificial neural networks (TWEANNs). These new methods introduced new mutations to the evolutionary process that altered the structure of the networks; for

instance, the addition of a neuron or connection.

2.2 Neuroevolution of Augmenting Topologies

One of the most successful TWEANN algorithms is NEAT (Neuroevolution of augmenting topologies) [23]. Developed by Kenneth O. Stanley and Risto Miikkulainen in 2002, NEAT borrows a lot of concepts from genetic algorithms [12] and aims to improve upon its TWEANN predecessors. It provides meaningful crossover of different topologies through the tracking of the historical origin of genes, protects underdeveloped innovations through speciation and keeps topologies small by starting with a minimal structure and incrementally growing it.

NEAT has proven to be an efficient solution for reinforcement learning problems in various fields, including network security [25], robot control [20], and game development [9, 10]. Enhancements for the algorithm are continually being developed [3, 11]. NEAT is particularly successful when the cost function is not easily defined. The algorithm helps networks avoid local optima and develop diverse strategies by preserving innovations through speciation based on model similarity. This approach allows networks time to optimize their new structural innovations, rather than being discarded due to short-term fitness dips, resulting in species developing distinct strategies. Furthermore, NEAT only requires a measure of a network's performance, making it applicable for training models on games without relying on expert knowledge.

NEAT uses a variable-length genetic encoding where each individual (Genome) has a list of node genes and a list of connection genes:

- Node genes represent network nodes. They hold information such as the node type (input, output, hidden) as well as its innovation number.
- Connection genes represent the connection between two nodes. They hold information such as the input node, output node, connection weight, an enable bit (whether or not the gene is expressed) and an innovation number.

There are four kinds of generic mutations: two structural mutations, a weight mutation and a mutation that alters the enable bit. The structural mutations expand the network by either adding a node between an existing connection or a connection between existing nodes (see Figures 1, 2). In addition, there is the common weight mutation present in multiple other such methods where the weight of connections is either set to a random value, set to a default value or slightly altered based on a distribution. Finally, there is the mutation that alters the enable bit, thereby altering the expression of the gene.

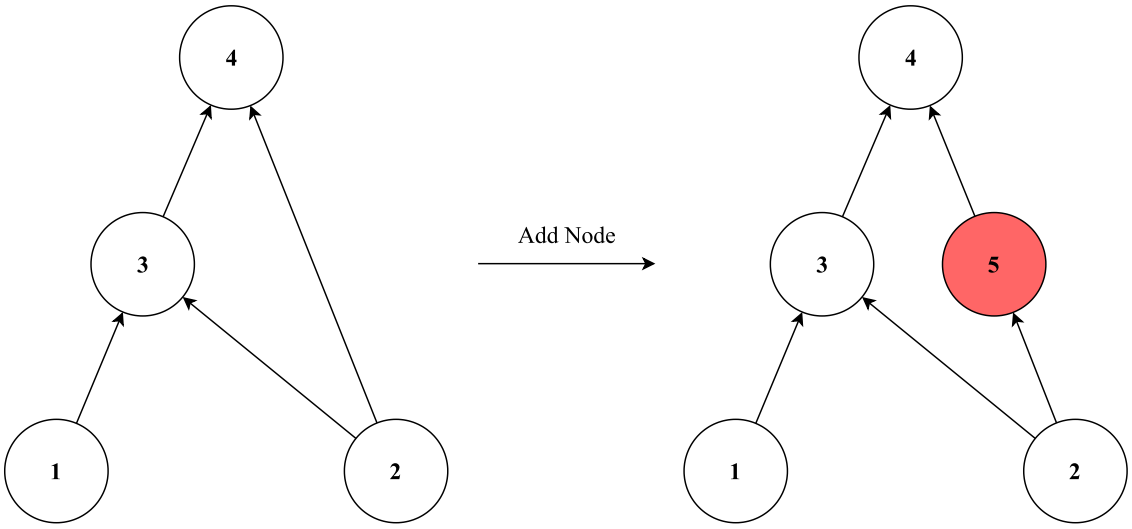

Figure 1: Add node mutation.

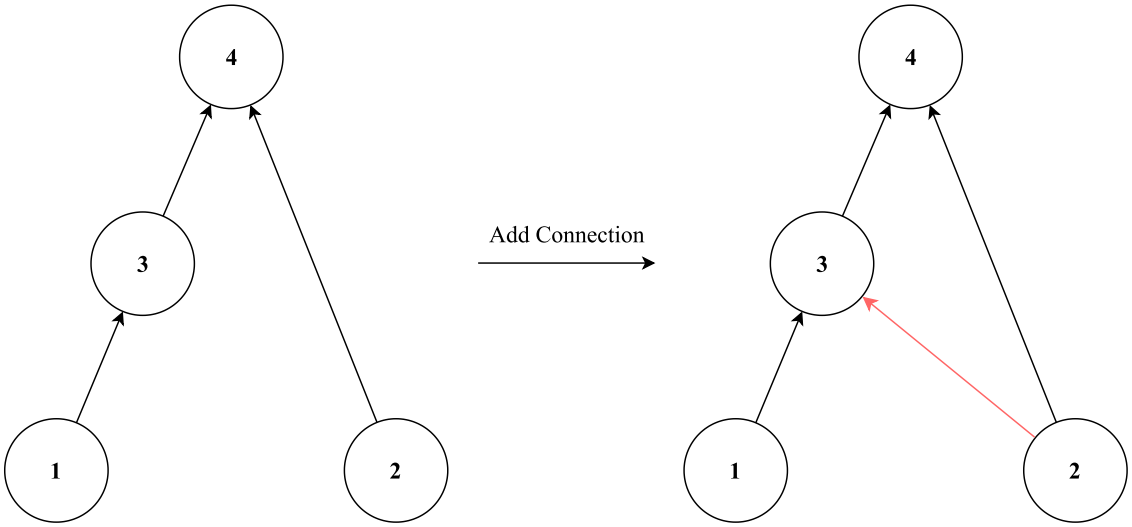

Figure 2: Add connection mutation.

2.2.1 Tracking of Genes

One problem that every genetic algorithm has to deal with the problem of competing conventions. The problem of competing conventions [18], also known as the Permutations Problem [15] arises in evolutionary algorithms when different networks correspond to the same function. During crossover, these networks are likely to yield offspring that deviate significantly from the original function, leading to a loss in innovation.

NEAT solves this by using innovation numbers that acts as identifiers for structural components. An innovation number is an extra property that is attached to genes and works like an ID. When a structural mutation occurs, it is paired with an innovation number and saved on an innovation record table. The same innovation number is used as the ID of the produced gene. New, higher numbers are generated for each additional gene. Whenever a structural mutation occurs we first check if it has already been registered on the innovation record table, if it has we make sure to use the same identifiers as previously. This allows

for the same structural component to be identified as the same across all genomes and generations.

Innovation Record	
Structural Innovation	**Innovation Number**
(1,4)	3

Innovation Record	
Structural Innovation	**Innovation Number**
(1,4)	3
(2,4)	5

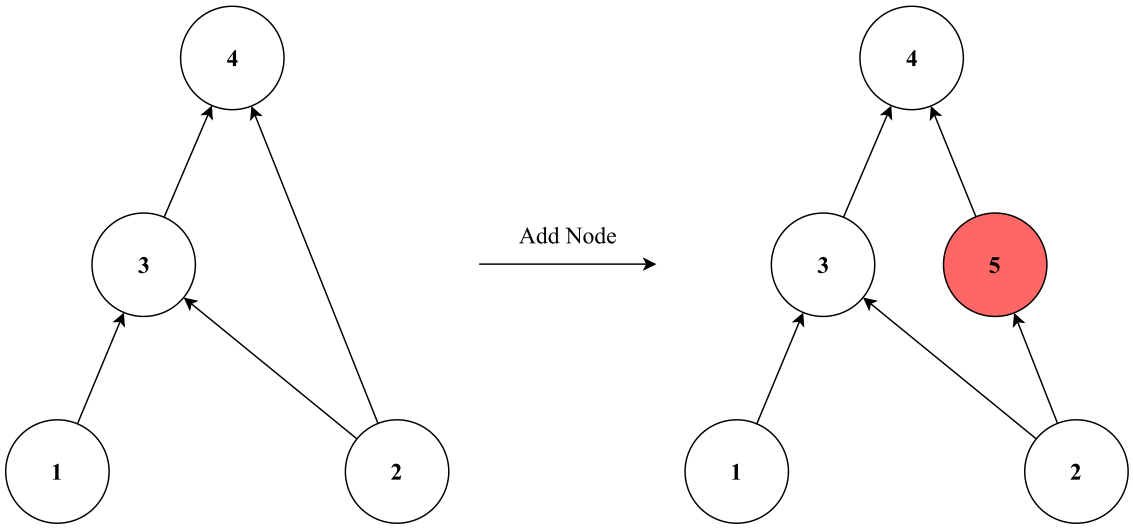

Figure 3: Structural mutation in NEAT. A node with ID 5 is added between nodes 2 and 4. This addition is recorded by adding a new entry in the innovation record, linking the mutation of the connection (2,4) with the innovation number 5. Subsequently, if the connection (2,4) is mutated in any network the produced node will be assigned ID 5. This can also be interpreted as node 5's ancestor being the connection (2,4).

The utilization of innovation numbers not only addresses the problem of competing conventions, but it also simplifies crossover, as it removes the need for expensive topological analysis. Crossing over two genomes is done by lining up their innovation number in chronological order (ascending order). Matching genes are inherited randomly, whereas nonmatching genes are are inherited from the more fit parent.

2.2.2 Speciation

To maximize the solution space covered, a diverse population of varying topologies is required. Since different topologies evolve at different rates, a mechanism is required to safeguard them from unfair competition. NEAT addresses this by introducing speciation. In each generation, genomes are separated into species based on a distance metric, allowing them to compete primarily within their own niche. The bigger the distance the more dissimilar the genomes are. The distance is computed by taking into account the number of genes in the largest genome N, excess E and disjoint D genes, as well as the overage weight differences of matching genes $\overline{W}$, including disabled genes:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \quad (2.1)$$

The hyperparameters c_1, c_2 and c_3 help fine-tune the function. Genomes whose distance fails below a given threshold can be grouped into the same species. Subsequently, genomes

are assigned to the most similar species. If a genome does fails to be assigned to a preexisting species it creates a new one. In addition, an explicit fitness sharing [7] is employed to make it hard for any species to grow large enough to take over the entire population. The method works by allotting each species a number of offspring slots that is proportional to the average fitness of its members. Within each species, individuals then compete with other for the change to reproduce. A select number of the fittest individuals reproduce via crossover, while all the other individuals are discarded. Each population is replaced in its entirety by its offspring.

2.2.3 Growing From Minimal Topology

Many predecessors of NEAT had their initial population be random topologies [8, 24]. While this approach ensures topological diversity from the start, it also introduces optimization problems, as random topologies are bound to have unnecessary genes that need time to be filtered out. NEAT adopts a different strategy. By starting with a population of topologies that only include the input and output nodes it eliminates the extra time required to weed out unnecessary genes and ensures that only necessary genes are included.

2.3 Self-Play

Self-play, a technique where agents learn by competing against variations or copies of themselves, has emerged as a powerful tool in multi-agent reinforcement learning (MARL) [17]. In this approach, agents train by interacting with copies of themselves, enabling a gradual escalation of challenge and complexity within the learning environment. Notably, one major advantage of this technique is its nonnecessity for training data, thereby eliminating the need for pre-collected datasets. A seminal contribution to this field is AlphaGo Zero [21], a model trained to play the board game Go, which achieved remarkable results against human players by utilizing a self-play learning schema. In our study, we use an approach similar to that of AlphaGo Zero. Self-play is employed to enhance our fitness function.

3 Methodology

To evaluate NEAT in a deterministic game environment, we created a game similar to the board game **Catan**. The idea is to use the NEAT algorithm in conjunction with self-play to train neural models based on specific game strategies. We monitor and evaluate their progress, and finally, conduct a live assessment where the best model faces off against a human opponent. To accomplish this, three components are required: the custom game itself, an algorithm employing a neural model for gameplay, and a system for training the neural models using NEAT and self-play.

3.1 Game Overview

The game is played on a five by five board with hexadecimal tiles (see Figures 4, 6). There are two players, denoted as **red** and **blue**. Players start on opposite corners of the board with one piece each. Each board tile can hold a maximum of ten pieces. We say that a player owns a tile when he has pieces on it. The blue player moves first and then play alternates between sides. The goal of the game is to eliminate all of the pieces of the opponent. There are two moves available to each player, **transfer** and **production**. Each board position can be represented as a five by five matrix, where each cell indicates the number of pieces on

the corresponding tile. Positive numbers denote pieces belonging to the blue player, while negative numbers denote pieces belonging to the red player.

3.1.1 Transfer Move

Players can select any number of their owned pieces from a tile and transfer them to a neighbouring tile. For this move to be valid, the following conditions must be met:

- The starting tile and the destination tile are adjacent.
- The number of selected pieces must not exceed the total number of pieces on the starting tile.
- The total number of pieces on the destination tile must not exceed the maximum allowed number of pieces on a tile.

If the destination tile is occupied by the opposing player, a fight ensues. During the fight, the number of tiles possessed by the player with the greater quantity is subtracted from both sides. For example, blue player transfers six pieces from tile $(4, 0)$ to $(3, 1)$:

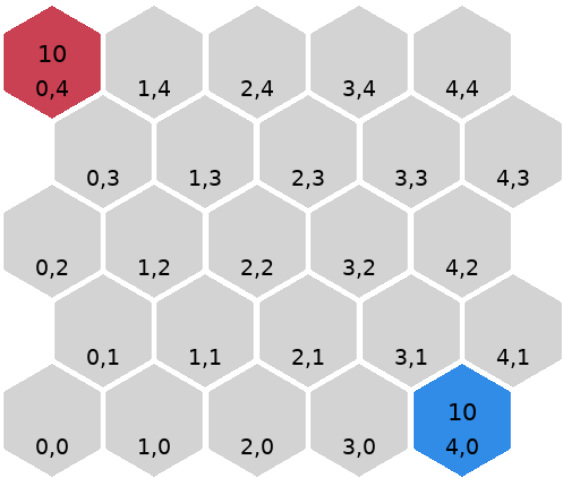

Figure 4: Before the Transfer Move.

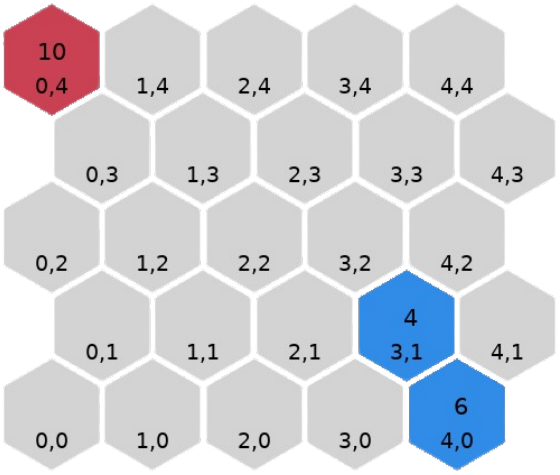

Figure 5: After the Transfer Move.

If a transfer move results is the capture of enemy pieces then it is also called a **capture move**. A transfer move is represented by a tuple consisting of the coordinates of the source tile, the coordinates of the target tile, and a number indicating the number of pieces transferred. For example, the transfer of 10 soldiers from tile $(0, 1)$ to tile $(1, 1)$ is represented as $((0, 1), (1, 1), 10)$.

3.1.2 Production Move

Players can select a tile they occupy and add one more piece to it. For this move to be valid, the number of pieces on the tile must be less than the maximum allowed number of pieces on a tile. For example, blue player produces a piece on tile $(3, 1)$:

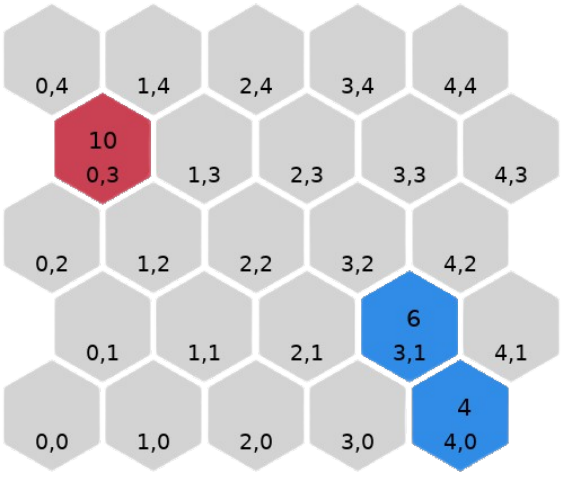

Figure 6: Before the Production Move.

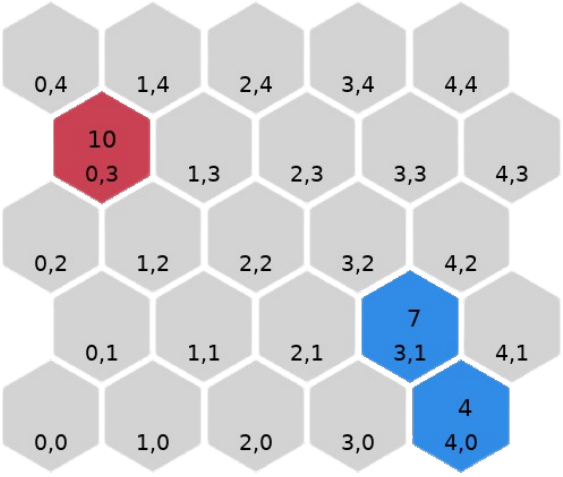

Figure 7: After the Production Move.

A production move is represented by a tuple consisting of the coordinates of the source tile and the coordinates of the target tile. For example, the production of a soldier on tile $(0, 1)$ is represented as $((0, 1), (0, 1))$.

3.2 Gameplay Algorithm

The gameplay algorithm operates by examining possible moves and countermoves up to a defined depth, utilizing a neural network to evaluate each resulting game state. These potential moves form a tree structure, allowing the use of a tree search algorithm to identify the most advantageous scenario. Since each tree node corresponds to a game state created by executing the moves leading to the move under evaluation, we can provide its matrix representation to a neural network. This network assesses the game state from the perspective of the moving player and assigns it a score. To search through the move tree, we use the principal variation search (PVS) algorithm, an enhanced version of alpha-beta pruning [2].

3.3 Move Searching

The idea is to search through all the possible moves and countermoves up to a given depth and utilize the neural network for assessing each game state. The job of the neural networks is to take the matrix form of a game state as input and output a score indicating how favourable the position is from the perspective of the player whose pieces are represented with positive numbers. Since the networks are designed to evaluate positions only from the perspective of the player with positive pieces, when we want to evaluate a position from the perspective of a player whose pieces are represented with negative numbers, we multiply the matrix by -1 to switch perspectives.

To summarize, we use principal variation search to explore moves up to the given depth. Each game state is evaluated by the neural network, and when changing player turns, we multiply the matrix by -1 to switch perspectives. After completing the search, we select the move that results in the most advantageous game state for the current player.

3.3.1 Principal Variation Search

Before explaining how the principal variation algorithm works, we have to explain the concept of the principal variation. The principal variation is the sequence of moves that players determine to be the best possible outcome given their analysis of the current position. In

other words, it is the line of play that both sides believe will lead to the most advantageous position for them.

The Principal Variation Search algorithm (PVS) [13] represents an enhancement of the alpha-beta algorithm. It relies on effective move sorting, as it is structured around the concept that moves positioned closer to the beginning of the move list are generally better and more likely to be part of the principal variation. This prioritization allows the PVS algorithm to save time by primarily exploring moves deemed to be within the principal variation. By initially probing the most promising paths, PVS can effectively prune the search tree, discarding less relevant branches [14]. Moreover, it operates under the 'fail-soft' principle, meaning that if an earlier move in the sorted list fails high (produces a cut-off with a value greater than or equal to beta), subsequent moves are searched with a reduced window around the previous result. This approach not only accelerates the search process but also enables the algorithm to focus its efforts on the most promising paths, leading to more accurate evaluations. To further improve the performance of the algorithm, multiple optimization techniques have been applied like singular extensions, late move reduction and the utilization of a transposition table.

Algorithm 1 Simplified version of the implemented principal variation search.

```
function PVS(board, α, β, depth)
    if depth ≤ 0 or there are no available moves to make then
        return EVALUATE(board)
    end if
    Let do_full_search be true
    Let best_score be 0.0
    Let available_moves be a list with all available moves
    for all m ∈ available_moves do
        MAKE_MOVE(board, m)
        if do_full_search = true then          ▷ First move is presumed to be part of the PV
            score ← -PVS(board, −β, −α, depth - 1)
        else                                   ▷ Search with a null window
            score ← -PVS(board, −α − 1, −α, depth - 1)
            if α < score < β then              ▷ If it failed high, do a full re-search
                score ← -PVS(board, −β, −α, depth - 1)
            end if
        end if
        UNDO_MOVE(board, m)
        if score > best_score then
            best_score ← score
            do_full_search ← false
        end if
        if score ≥ β then
            break                              ▷ beta cut-off
        end if
        if score > α then
            α ← score
        end if
    end for
    return best_score
end function
```

3.4 Model Training

For our purposes, a system was developed in Python utilizing the NEAT algorithm and self-play that allows for the relatively quick training and evaluation of agents. It allows users to define their own evaluation functions responsible for evaluating the performance of models based on a game and adjust the process through parameterization. To address the need for parallel training and evaluation of multiple agents, we have utilized the built-in multiprocessing module of Python and Numba, a just-in-time (JIT) Python compiler.

3.5 NEAT Implementation Overview

To keep things organized, the implementation of the NEAT algorithm[1] is separated from the implementation of the training model. The NEAT implementation is written in Python and is based on the original paper by Stanley and Miikkulainen [23]. While the core algorithm remains unchanged, we have introduced some modifications such as elite genomes and interspecies crossover.

[1] `https://github.com/JohnKechagias/neat-proc`

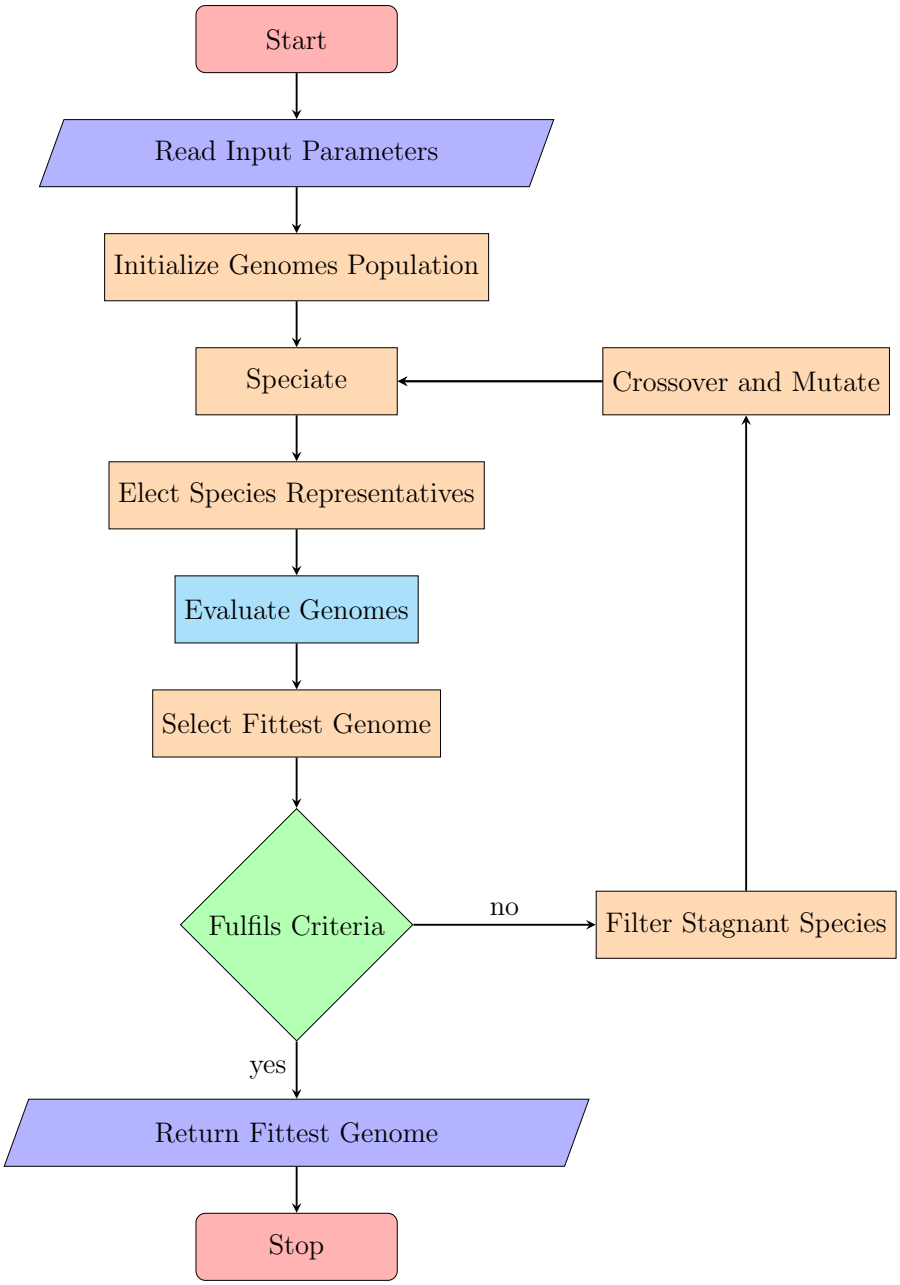

Figure 8: Flowchart of the NEAT implementation.

For our use case, we have chosen to use feedforward neural networks. Consequently, all standard mutations have been redesigned to ensure that the networks maintain their feedforward structure.

3.5.1 Genes

The gene design follows the default approach of NEAT with some adjustments. Node genes store information related to the innovation number, node type, bias, response, aggregation function, and activation function. On the other hand, connection genes store information regarding the innovation number, input node, output node, weight, enabled status, and frozen status. If a connection is frozen, its weight is not subject to mutations. The node and connection parameters are shown in Tables 2 and 3 respectively.

3.5.2 Genomes

The genomes use a direct encoding scheme and are comprised of a list of connection genes and a list of node genes. The genome parameters are shown in Table 1.

3.5.3 Species Elites

In the original NEAT implementation, every reproduction cycle involves replacing the entire population with offspring. This approach risks discarding innovations developed by the more fit individuals, since there is no guarantee that their offspring will inherit them. To avoid this, a new mechanism named **species elites** is introduced. The elites of each species are its fittest members, with the maximum number of elites for each species being specified by a hyperparameter. During the reproduction phase, the elites of each species are copied over to the new population. Consequently, the new population consists not only of offspring, but also the elites of each species. To prevent the inclusion of elites from underdeveloped species, namely those with a small population, a hyperparameter is introduced to specify the minimum size required for a species to have its elites copied over.

3.5.4 Interspecies Crossover

Although speciating the population allows structures to develop at their own pace, it can also lead to over-specialization. This occurs when a large majority of the fittest individuals of a species develop overly similar structures over time, leading to overly similar offspring. This constitutes an innovation drought for the species, jeopardizing its long-term viability and cannot be easily offset by mutations, as the mutation rate must be within a specific range for individuals to remain stable enough to retain and develop useful features. Allowing interspecies crossover can help introduce new features to species and can also lead to offspring inheriting the best of their parents' worlds. It is vital that the interspecies crossover rate remains relatively low, as larger values undermine the speciation mechanism.

3.6 Fitness Function

Given that NEAT is a genetic algorithm variant, the fitness function plays a vital role in ensuring its proper function. The fitness value measures how adept an individual is for the task. We use self-play to evaluate the genomes, pitting them against the best of the previous generation and calculating their fitness based on the game results. For the first run, because a previous generation does not exists, we randomly select genomes from the generated population to act as opponents.

Each genome competes multiple times against each opponent to determine its average performance score against that specific opponent. After each match, the performance of the model is evaluated based on the game record, which is composed of the move history and the game result. The evaluation function considers both the match result and the moves made by the model and outputs a score. Once all games are completed, we calculate the average of these scores and assign it as the fitness of the genome.

The function **evaluate_game_record** is responsible for evaluating the performance of a genome based on the game record and is what determines its desired behaviour.

3.7 Training Measurements

During training, it is essential to monitor various aspects of the population in order to get an accurate view of how the algorithm performs. Specifically, we need to track the performance

Algorithm 2 Genome evaluation logic.

Input: Genome **G**, List of opponents **O**
Parameters: The times to play against an opponent **T**
Output: Fitness of model **F**
Let **S** be an empty array ▷ An array which will hold the computed scores
for all $o \in$ **O do**
  repeat T times
    Pit the genome against the opponent
    R $\leftarrow$ play(**G**, o) ▷ The game record
    Calculate the genome score based on the game record
    s $\leftarrow$ evaluate_game_record(**R**) ▷ The score of the genome
    Add **s** to the array **S**
  end
end for
return $\frac{sum(S)}{|S|}$ ▷ The arithmetic average of the scores array

of individuals, the performance of species, and the diversity of the population. Additionally, it is essential to recognize when the process is converging on a local optimum. To accomplish this, we keep track of the following metrics for each generation:

- **Fitness of the Best Genome** The fitness of the best genome in the population indicates the highest performance achieved by any individual. This metric defines the upper bound of performance within the population for a given generation.

- **Average Genome Fitness** The average genome fitness measures the mean performance of all individuals in the population. This provides an overview of the general performance level and helps in identifying overall trends in the population fitness.

- **Adjusted Fitness of Each Species** The adjusted fitness of a species is a measure of the performance of the members of the species in relation to the best individual of the population for the generation scaled to the range 0.0 to 1.0. The closer the average performance of the members of a species is to the best recorded performance for the generation, the higher the adjusted fitness of the species. The value can be used to compare the performance of the population of a species to the population of all other species and is calculated using the arithmetic mean of the fitnesses of the genomes that belong to the species $\overline{f}$, the maximum observed fitness in all species for the generation f_{max}, and the minimum observed fitness in all species for the generation f_{min}:

$$S_{adj} = \frac{\overline{f_S} - f_{min}}{\max\{1.0, f_{max} - f_{min}\}}$$

  During tests, we avoid directly graphing the adjusted species fitnesses as the resulting graph is not easily interpreted. Instead, we use this metric to compute the average adjusted species fitness.

- **Average Adjusted Species Fitness** The average value of the adjusted species fitnesses can be used to measure the performance diversity of the population. A value close to 0.0 denotes that the performance of the best recorded member deviates significantly from the performance of the average member, whereas a value close to 1.0 denotes that all members of the population perform similarly, indicating that the algorithm is close to a local optimum.

4 Results

When playing against a human opponent the game is limited to 150 moves, after which it ends in a draw.

4.1 Experiment 1: Adopting an Aggressive Play Style

In the first experiment, we train the models to be overly aggressive and to take over the centre of the map as quickly as possible. We accomplish this by rewarding winning games, capture moves, and moves that advance pieces toward the centre. The evaluation logic is as follows:

Algorithm 3 Game record evaluation logic.

Input: Moves Record **R**, Number of pieces left **P**
Output: Score of the model **S**
Parameters: maximum score S_{max}
Ensure: $\mathbf{S} \in (0, S_{max})$
$\mathbf{S} \leftarrow 0.0$ ▷ The score of the model
if match ended in victory **then**
$\mathbf{S} \leftarrow \mathbf{S} + \frac{S_{max}}{8}$
else if match ended in defeat **then**
$\mathbf{S} \leftarrow \mathbf{S} + \min\{\mathbf{P} * 2, \frac{S_{max}}{15}\}$
else if match ended in draw **then**
$\mathbf{S} \leftarrow \mathbf{S} + \min\{\mathbf{P} * 3, \frac{S_{max}}{10}\}$
end if
$C \leftarrow (2, 2)$ ▷ The centre coordinates of the board
for all $m \in \mathbf{R}$ **do**
if m is a capture move **then**
$\mathbf{S} \leftarrow \mathbf{S} + 15$
else if m is a transfer move **then**
Compute the Euclidean distance from the centre
of the board for the old and new positions
Let $d_{\text{old}} \leftarrow d(m[0], C)$ ▷ Distance from the old position to the centre
Let $d_{\text{new}} \leftarrow d(m[1], C)$ ▷ Distance from the new position to the centre
Compute the difference between these distances
$D \leftarrow d_{\text{old}} - d_{\text{new}}$
$\mathbf{S} \leftarrow \mathbf{S} + \max\{8 * D, 0\}$
end if
end for
return S

After one hundred generations, the best model has adopted a very aggressive strategy. It rushes to the centre of the board as soon as the game starts and produces pieces whenever possible, either proactively or in response to the production moves of the opponent. The average generation time during training was 6.4 minutes. The best model has a fitness score of 76.0 and consists of 32 nodes and 28 connections.

When tested against a human opponent, the model generally adhered to an aggressive strategy regardless of the tactics employed by its opponent. However, it also employed original tactics, such as retreating when threatened. Out of 50 games played, the model defeated the human opponent 6% of the time and managed a draw 28% of the time. Its

response to player aggression is generally effective, as it often entrenches itself by staying stationary and producing pieces, or by performing a tactical retreat, or even managing a draw by rushing the player with the same amount of pieces (see Figure 9). However, its weaknesses become apparent when the opponent uses a mix of strategies, such as building forces before advancing, as it does not tweak its approach.

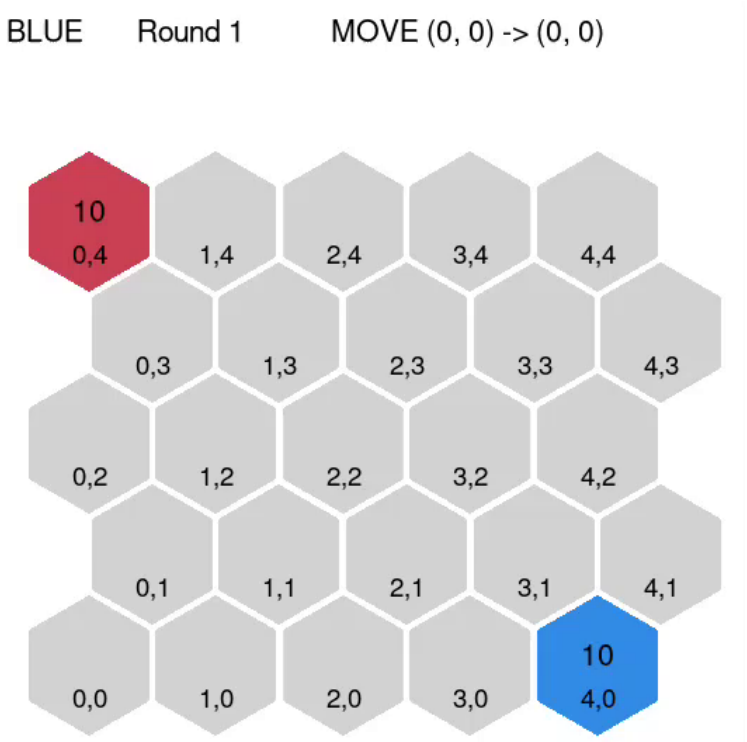

Figure 9: A match where the model, playing as red, faces a human player, playing as blue. The human player starts by trying to reach the centre of the map first, while the model responds by moving in a straight line before converging towards the centre, resulting in self-annihilation.

During the first 66 generations, the average adjusted species fitness (see Figure 10) mostly remained in the range of 0.3 to 0.5, indicating a diverse and healthy population. The initial fluctuations in the first generations are expected due to the rapid mutation of individuals. Since all individuals start with identical topologies, the fitness values naturally fluctuate as they start to differentiate before stabilizing into a trend. Moreover, the population steadily became more fit through the training process as the average fitness continued to incrementally grow, albeit in small steps (see Figure 11). Despite this, the training process began to display signs of stagnation in the later stages. From generation 66 onwards, a slight upward trend appeared in the average adjusted species fitness graph, implying potential stagnation as the training process started to converge on a local optimum. This is further supported by the fact that, in the later stages of training, matches between models commonly ended in self-annihilation as they strived to reach the centre first. This is in line with our expectations, as this is the best possible strategy according to the defined fitness function.

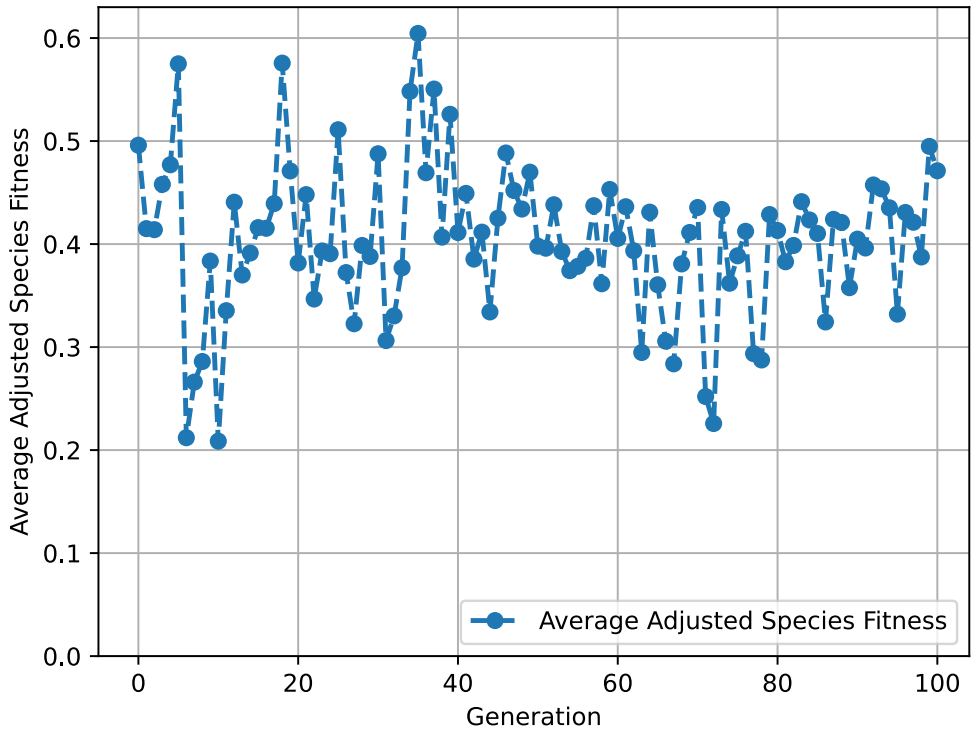

Figure 10: Average adjusted species fitness graph.

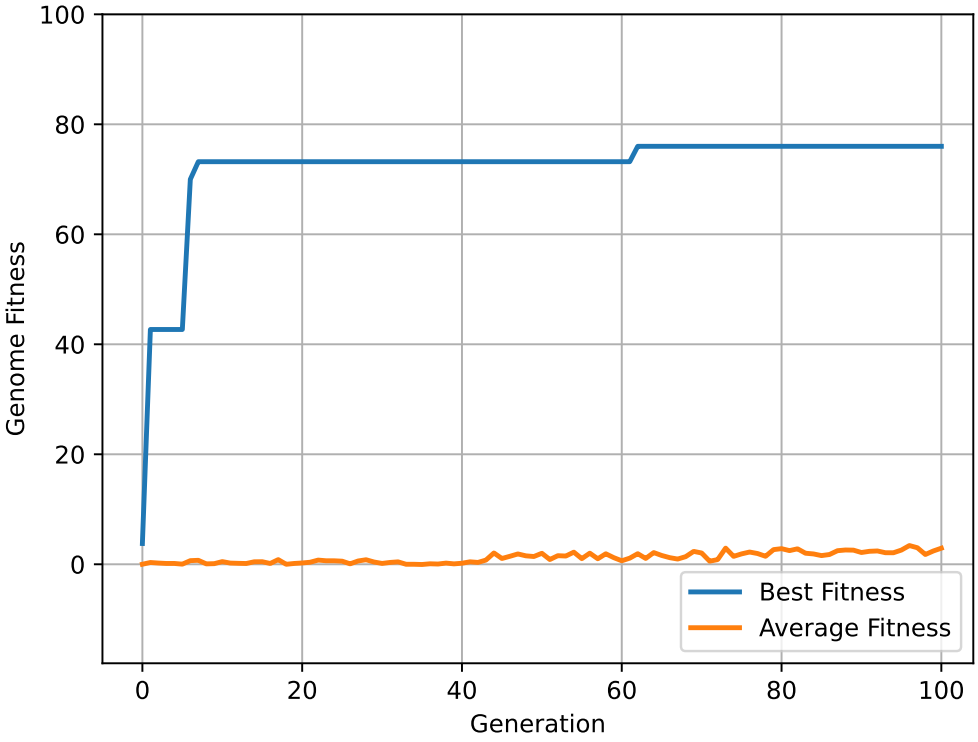

Figure 11: Genomes fitness graph.

4.2 Experiment 2: Adopting a Defensive Play Style

In the second experiment, we train the models to be overly defensive and prioritize domination of the corner of the map. We accomplish this by rewarding winning games, production moves, and moves that result in pieces getting further to the centre and closer to a corner. The evaluation logic is as follows:

Algorithm 4 Game record evaluation logic.

Input: Moves Record **R**, Number of pieces left **P**
Output: Score of the model **S**
Parameters: maximum score S_{max}
Ensure: $\mathbf{S} \in (0, S_{max})$
$\mathbf{S} \leftarrow 0.0$ ▷ The score of the model
if match ended in victory **then**
$\mathbf{S} \leftarrow \mathbf{S} + \frac{S_{max}}{8}$
else if match ended in defeat **then**
$\mathbf{S} \leftarrow \mathbf{S} + \min\{\mathbf{P} * 2, \frac{S_{max}}{15}\}$
else if match ended in draw **then**
$\mathbf{S} \leftarrow \mathbf{S} + \min\{\mathbf{P} * 3, \frac{S_{max}}{10}\}$
end if
$C \leftarrow (2, 2)$ ▷ The centre coordinates of the board
for all $m \in \mathbf{R}$ **do**
if m is a production move **then**
$\mathbf{S} \leftarrow \mathbf{S} + 10$
else if m is a transfer move **then**
Compute the Euclidean distance from the centre
of the board for the old and new positions
Let $d_{\text{old}} \leftarrow d(m[0], C)$ ▷ Distance from the old position to the centre
Let $d_{\text{new}} \leftarrow d(m[1], C)$ ▷ Distance from the new position to the centre
Compute the difference between these distances
$D \leftarrow d_{\text{new}} - d_{\text{old}}$
$\mathbf{S} \leftarrow \mathbf{S} + \max\{10 * D, 0\}$
end if
end for
return S

After one hundred generations, the best model has adopted a defensive strategy bordering on passive that focuses on amassing a large number of pieces and playing around choke points. It prioritizes the corners of the map and produces pieces whenever it is given the chance. The best model has a fitness score of 146.0 and consists of 26 nodes and 25 connections.

When tested against a human opponent, the model consistently maintained its defensive strategy regardless of the tactics of the opponent. In the 50 games played, the model never defeated the human opponent but managed to achieve a draw 68% of the time. When facing a defensive player, the match always ended in a draw since the model never made aggressive moves (see Figure 12). Against an aggressive player, the model would stay in the corners, accumulating pieces. The only winning strategy for the opponent was to move closer to the model when it wasted a move on unnecessary piece transfers and to produce pieces in response to piece production done by the model.

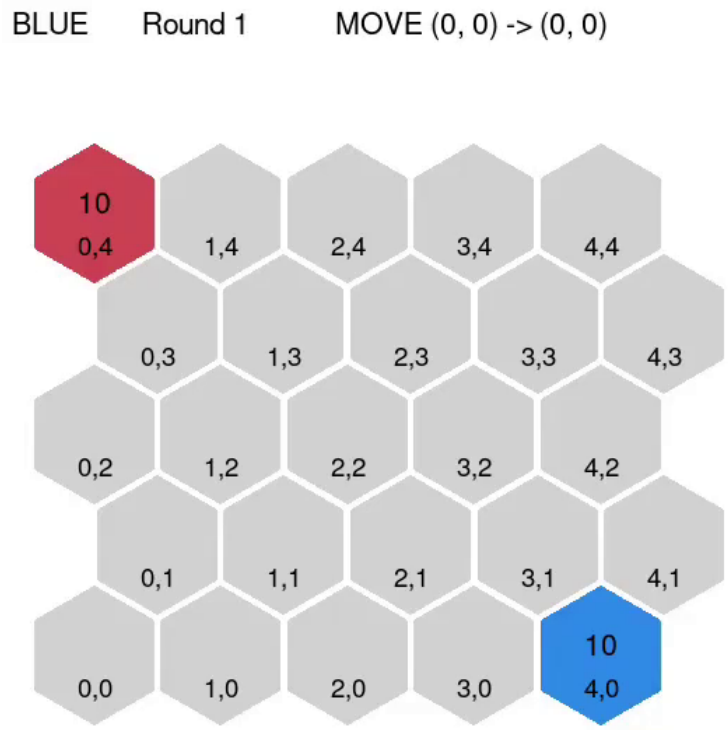

Figure 12: A match where the model, playing as red, faces a human player, playing as blue. The human opponent rushes to the top right corner of the map, while the model responds by rushing to the bottom left corner.

In contrast to the first experiment, here the average adjusted species fitness (see Figure 13) remained in the range of 0.6 to 0.75. This indicates that, although the training process did not reach a local optimum, the population was less diverse in terms of performance. Again, the initial fluctuations in the first generations are expected due to the rapid mutation of individuals. Since the average adjusted species fitness stabilized in the range of 0.6 to 0.75, it is likely that the population will stagnate in the near future. This is further supported by the fact that the best strategy according to the fitness function is a passive one, as does not incentivize interacting with the opponent apart from contesting corners. The final version of the model adheres to that strategy, meaning that there is not much room for improvement.

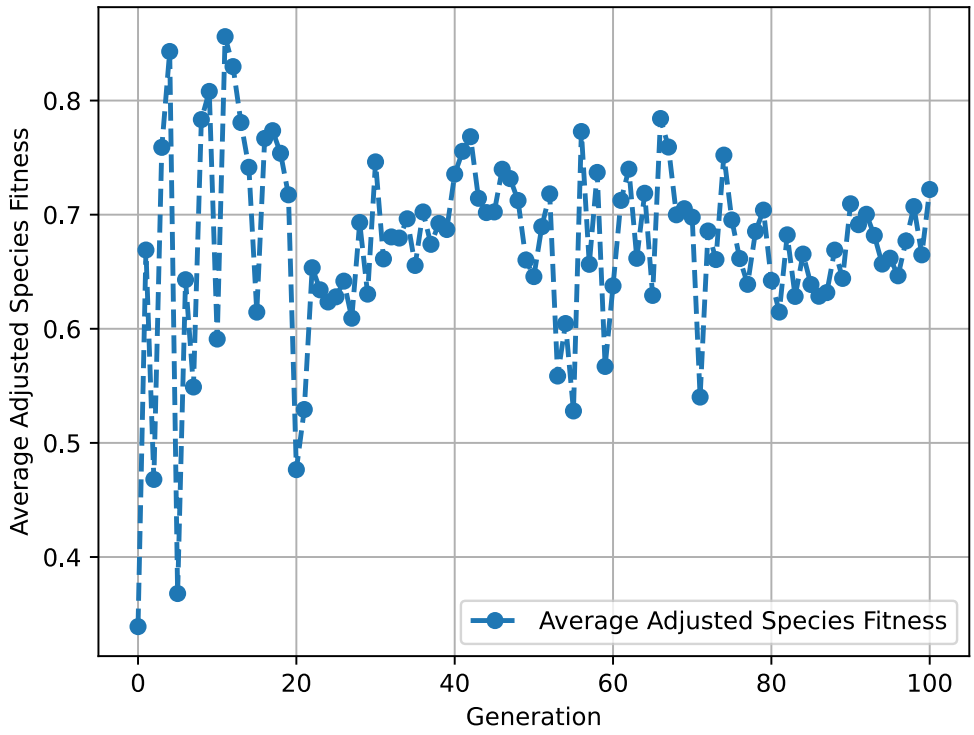

Figure 13: Average adjusted species fitness graph.

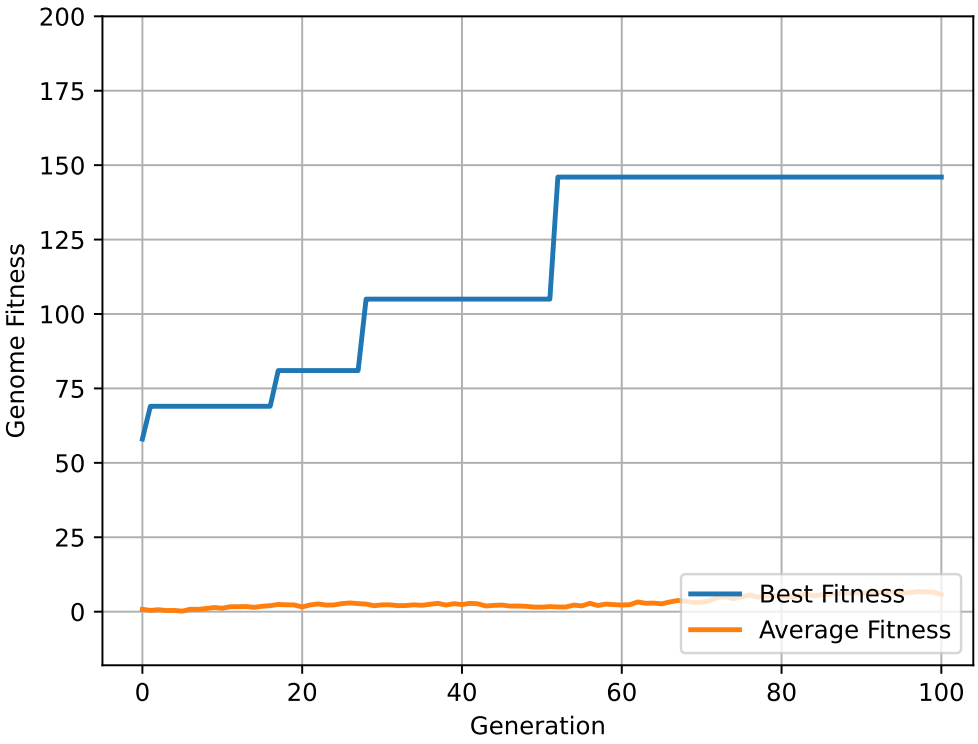

Figure 14: Genomes fitness graph.

5 Conclusion

This paper demonstrated the potential of the NeuroEvolution of Augmenting Topologies (NEAT) algorithm for training agents in deterministic game environments. We developed a training approach that integrates NEAT with self-play and conducted a series of experiments aimed at training agents to adopt specific game strategies. In all cases, the resulting models were compact and successfully learned the intended strategies.

Due to resource limitations, the number of experiments conducted was less than ideal and the game used was relatively simple. This means that the algorithm could not be showcased to its full potential. Therefore, we can address the following items for future studies:

- **Improving the gameplay algorithm:** The gameplay algorithm significantly influences how models behave. It is possible that the models are limited by the current algorithm rather than the NEAT algorithm. Enhancing the gameplay algorithm would provide a clearer understanding of the performance of the NEAT algorithm.
- **Comparing the developed algorithm with alternatives:** Running the same experiments using different techniques and comparing the results would offer a better perspective on how NEAT performs relative to other existing implementations.
- **Utilizing a recurrent neural network:** In contrast to a feedforward network, a recurrent neural network (RNN) can maintain an internal state that acts as memory. This feature could enable models to develop more sophisticated strategies by recognizing patterns over time and adapting their tactics accordingly.
- **Non-challenging game environment:** The game environment could be complexified to address the lack of strategic depth. By increasing the complexity of the game, simplistic strategies would be less effective and the fitness landscape would become harder to traverse, thus raising the development ceiling for models. This would provide a greater challenge to the models and better showcase the capabilities of the algorithm.
- **Testing of various hyperparameters:** Both the gameplay algorithm and NEAT heavily depend on hyperparameters for optimal functioning. The best values for these parameters cannot be predetermined and must be found through trial and error. Some important hyperparameters to test include:
  - Population
  - Number of Iterations
  - Activation Functions
  - Aggregation Functions
  - Mutation Rates
  - Crossover and Inter-Crossover Rates
- **Fitness functions that promote single strategies:** One of the main advantages of the NEAT algorithm is its ability to avoid getting easily stuck in local optima. Making the fitness functions more generic by removing all logic that relies on expert knowledge would eliminate any assistance given to NEAT in traversing the fitness landscape. Thus, data obtained from future tests would be more reliable and conclusive.

6 Appendix

Table 1: NEAT Parameters

Hyperparameters	**Values**
Population	200
Input Nodes	25
Output Nodes	1
Hidden Nodes	0
Feed-Forward	True
Connection Scheme	Fully Connected
Fitness Threshold	400
Reset On Extinction	False
Use Alternative Structural Mutations	True

Table 2: Node Gene Parameters

Hyperparameters	**Values**
Node Mutation Chance	1.0
Node Addition Chance	0.2
Node Deletion Chance	0.2
Activation Function	Sigmoid
Activation Function Options	[Sigmoid]
Activation Function Mutation Chance	0.0
Aggregation Function	Summation
Aggregation Function Options	[Summation]
Aggregation Function Mutation Chance	0.0
Bias Initial Mean	1.0
Bias Initial Standard Deviation	1.0
Bias Minimum Value	-30
Bias Maximum Value	30
Bias Mutation Chance	0.7
Bias Reset Chance	0.1
Bias Mutation Power	0.5
Response Initial Mean	1.0
Response Initial Standard Deviation	0.0
Response Minimum Value	-30
Response Maximum Value	30
Response Mutation Chance	0.0
Response Reset Chance	0.0
Response Mutation Power	0.0

Table 3: Connection Gene Parameters

Hyperparameters	**Values**
Connection Mutation Chance	1.0
Connection Addition Chance	0.5
Connection Deletion Chance	0.5
Connection Enabled By Default	True
Connection Enable Mutation Chance	0.01
Connection Frozen By Default	False
Connection Frozen Mutation Chance	0.0
Weight Initial Mean	0.0
Weight Initial Standard Deviation	1.0
Weight Minimum Value	-30
Weight Maximum Value	30
Weight Mutation Chance	0.8
Weight Severe Mutation Chance	0.01
Weight Reset Chance	0.1
Weight Mutation Power	0.5

Table 4: Speciation Parameters

Hyperparameters	**Values**
Compatibility Disjoint Coefficient	1.0
Compatibility Weight Coefficient	0.5
Compatibility Threshold	3.8
Maximum Stagnation	20

Table 5: Reproduction Parameters

Hyperparameters	**Values**
Crossover Rate	1.0
Interspecies Crossover Rate	0.15
Survival Rate	0.2
Elitism	2
Elitism Threshold	2
Minimum Species Size	2

References

[1] Thomas Bartz-Beielstein, Juergen Branke, Jorn Mehnen, and Olaf Mersmann. Evolutionary algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4, 05 2014.

[2] A. L. Brudno. Bounds and valuations for abridging the search of estimates. In *Problems of Cybernetics*, volume 10, pages 225–241, 1963. Translation of Russian original in Problemy Kiber-netiki 10, 141-150 (May 1963).

[3] David D'Ambrosio, Jason Gauci, and Kenneth Stanley. Hyperneat: The first five years. *Studies in Computational Intelligence*, 557:159–185, 06 2014.

[4] Dario Floreano and Claudio Mattiussi. Neuroevolution: From architectures to learning. *Evol Intell*, 1, 03 2008.

[5] David B. Fogel. *Natural Evolution*, chapter 2, pages 33–58. John Wiley & Sons, Ltd, 2005.

[6] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *CoRR*, abs/2006.05415, 2020.

[7] D.E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In editor In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49, 1987.

[8] Frédéric Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the 1st Annual Conference on Genetic Programming*, page 81–89, Cambridge, MA, USA, 1996. MIT Press.

[9] Erin Jonathan Hastings, Ratan K. Guha, and Kenneth O. Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, 2009.

[10] Daniel Hind and Carlo Harvey. Using a neat approach with curriculums for dynamic content generation in video games. *Personal and Ubiquitous Computing*, pages 1–13, 04 2024.

[11] Mikkel Angaju Rasmussen Jakob Merrild and Sebastian Risi. Hyperentm: Evolving scalable neural turing machines through hyperneat. *CoRR*, abs/1710.04748, 2017.

[12] Annu Lambora, Kunal Gupta, and Kriti Chopra. Genetic algorithm- a literature review. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 380–384, 2019.

[13] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Comput. Surv.*, 14(4):533–551, dec 1982.

[14] T. Anthony Marsland and Fred Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7:442–452, 1985.

[15] Nicholas J. Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1:67–90, 1993.

[16] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9, 10 2014.

[17] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959.

[18] J. Schaffer, Darrell Whitley, and Larry Eshelman. Cogann-92 combinations of genetic algorithms and neural networks. *Genet. Algorithms Neural Network*, pages 1 – 37, 07 1992.

[19] Ivan Sekaj, Ladislav Cíferský, and Milan Hvozdík. Neuro-evolution of mobile robot controller. *MENDEL*, 25:39–42, 06 2019.

[20] Fernando Silva, Paulo Urbano, Sancho Oliveira, and Anders Christensen. odneat: An algorithm for distributed online, onboard evolution of robot behaviours. In *ALIFE 2012: The Thirteenth International Conference on the Synthesis and Simulation of Living Systems*, pages 251–258, 07 2012.

[21] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[22] Kenneth Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1, 01 2019.

[23] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[24] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[25] Tamara Zhukabayeva, Aigul Adamova, Khu Ven-Tsen, Zhanserik Nurlan, Yerik Mardenov, and Nurdaulet Karabayev. Network attack detection using neuroevolution of augmenting topologies (neat) algorithm. *JOIV : International Journal on Informatics Visualization*, 8, 03 2024.