KING'S College LONDON

University of London

6CCS3PRJ Final Year
Study of the feasibility of an universal gaming AI using NEAT in Tetris.

Final Project Report

Author: Zheng Yi Hu

Supervisor: Mischa Dohler

Student ID: 1753137

April 19, 2020

Abstract

The aim of the project is an implementation of the ”Neuro-Evolution of Augmented Topologies” algorithm in Unity to find an optimal gameplay model for the classic Tetris game with individual cells state given as input data. As most modern games evolve in gameplay complexity over the years, it becomes more and more difficult to generate AIs behaviors efficiently through hardcoded behaviors. In many single-player games, most of the entertainment, depending on the game genre, will come by the challenge of overcoming obstacles and enemies, and to provide a deep and compelling gameplay to the player, most of the times a basic hard-coded AI is not enough anymore. For this reason, deep-learning algorithms are used widely in the gaming industry, and people developed and are still developing more efficient algorithms optimized for game AI deep-learning. One of these algorithms is Neuro-Evolution of Augmented Topologies, in short, ”NEAT” and this thesis will show the implementation and integration of the algorithm in a Tetris game as a testing target. Raw pixel input data is used to simulate a universal AI capable of adapting to any given game without prior knowledge of the goal of the game so that heuristics for specific games are not available to the AI.

Originality Avowal

Zheng Yi Hu

April 19, 2020

Contents

1 Introduction **3**
1.1 Gaming and AI . . . 3
1.2 My innovation contribution . . . 4

2 Background **5**
2.1 Tetris AI . . . 5
2.2 Reinforcement learning Tetris AI . . . 5
2.3 Genetic algorithms . . . 6
2.4 Genetic algorithms Tetris AI . . . 6
2.5 NEAT . . . 7

3 Implementation **9**
3.1 Genome Structure . . . 9
3.2 Historical marking . . . 9
3.3 Mutation . . . 10
3.4 Speciation . . . 13
3.5 Selection . . . 14
3.6 Crossover . . . 14
3.7 Feed-Forward . . . 18
3.8 Additional features . . . 19

4 Integration **21**
4.1 Tetris Game . . . 21
4.2 Integration with AI . . . 22
4.3 Parameters setting . . . 22

5 Legal, Social, Ethical and Professional Issues **24**
5.1 The Ethics of Artificial Intelligence . . . 24

6 Results/Evaluation **26**
6.1 Test 1: Neuro-evolution vs Fully connected network with no node mutations . . 26
6.2 Test 2: limiting mutation . . . 27
6.3 Test 3: deterministic game . . . 28
6.4 Test 4: Neuro-Evolution vs NEAT . . . 29

7 Conclusion and Future Work **31**
7.1 Conclusions 31
7.2 Limitation of the project and future solutions 33
7.3 NEAT parameters tuning 33

Chapter 1

Introduction

1.1 Gaming and AI

Artificial Intelligence plays a huge role in gaming as well as many other fields in technology. One of the main applications is the behavior of enemies in games which usually need to be unpredictable and responsive for the game to be challenging and compelling to play.
Because of the wide variety of available games and genres, it is hard to define a precise formula for a good AI; this will depend on different factors and will need to satisfy different requirements depending on the scope of the game. A universal training AI would be beneficial for this area of development which, similarly to a human player, can learn and adapt to the given game without the need of specifically being designed for the given role.
When a game's complexity is low, the best approach is usually a hard-coded program that dictates the behavior based on the game current status.
An example can be a classical SNAKE game: if the goal is to survive and grow as long as possible; by using a Hamiltonian cycle controller, it is possible to guarantee the player to never die and eventually grow until the maximum length [8]. This solution though is not optimal in terms of speed in which the goal can be completed, and many heuristic approaches can be taken when problems get more and more complex. However, when a game includes more complex mechanics, it becomes hard to design an efficient heuristic model to follow. Many times it is better to utilize a learning AI that starts by not knowing anything and improve over time based on the past experiences.

This thesis will explore the feasibility of a universal AI for games implemented in Unity. The

project aims to focus on a more general approach to the problem and work towards a universal AI that can adapt to any game that would display the game on a screen and the score, instead of an optimized AI fine-tuned to play a specific game. The AI will simulate an outsider system, that can only see the pixels on the screen at the given moment (A simulation is used instead of actual screen reading functionalities mostly to contain the computation intensity).
The AI implemented in this thesis will utilize the NEAT algorithm, which is a genetic algorithm developed by Ken Stanley in 2002 [1].
TETRIS was the target game chosen for testing since it is a classic game with very basic mechanics, but complex enough to not have a precise correspondence between inputs and outputs.

1.2 My innovation contribution

The program implements a basic version of the classic TETRIS game and an AI-based on NEAT that will attempt to learn the game and achieve the highest amount of points possible by evolving a population of 200 players.
Each player will be assigned a neural network to define its behaviors, which will receive as inputs the state of each cell in the game as either occupied or free. This way, the simulation of a screen reading capability is provided to the AI such that it will be possible to easily expand the project into an actual pixel reading AI to further develop the project.
The program also implements various UI tools to better visualize the evolution process to have a more detailed insight into how the AI is performing. These include the render of each game been played by the AIs, the render of the evolving neural networks and a saving system to not lose the learning progress once the program is terminated.
This will allow the user to easily determine how feasible and beneficial it is to work towards a universal AI with pixel inputs and set comparison benchmarks for various implementations to evaluate which ones will work the best without committing to building definitive versions of them.

Chapter 2

Background

2.1 Tetris AI

Many AIs for Tetris were developed since the game is a classic piece in the gaming history, and with great results. Many of them can play the game almost flawlessly and indefinitely, and a single game played by a trained AI could virtually last forever. Examples of successful Tetris AIs are model which use genetic algorithms or reinforcement learning with a reward system that focus on maximizing cleared lines and minimize holes as an inheritance of common human player strategies. These heuristics will often simulate a human player and give a good direction for the AI to improve. Various learning algorithms were explored in the past for the implementation of Tetris AIs. The most common being reinforcement learning techniques such as Q-learning, or genetic algorithms.

2.2 Reinforcement learning Tetris AI

Reinforcement learning application in Tetris has been widely explored because of the game's nature and popularity.

The NP-Complete nature of the game [3] and the clear scoring system in function of the cleared rows makes the game a suitable target for reinforcement learning algorithms. Although it is impossible to determine whether moves are right or wrong, it is easier to determine which strategies might or might not work in the given situations, these strategies can be translated into heuristics to guide the AI towards methods that are known to work. These strategies include:

1. Minimize holes in between blocks
2. Maximize cleared lines
3. Create an I valley to fill with an I tetromino
4. Clear lines to lower the height of the columns when necessary

Although these heuristics are proven to be an effective guideline to evaluate the fitness of the AI using reinforcement learning [4] [7], the AI will not explore new strategies to solve the problem, but will rather simulate a human game-play and stick to known methodologies that are suitable for humans and might be a limitation for a computer-driven program.

A problem with reinforcement learning in this project is that it is heavily based on heuristic and a gradient to find the optimal solutions. In Tetris, such gradient doesn't exist when the game is decomposed into individual cells and it becomes really hard to determine the right "direction" for the AI to develop from the given input information.

For this reason, Genetic algorithms became also a popular subject to explore for Tetris, where the gradient is substituted by random mutations of the population and the natural concept of the survival of the fittest.

2.3 Genetic algorithms

Genetic algorithms are natural selection inspired algorithms that are designed to improve the performance over time with the population-based approach.

In each generation, each genome of the population will have a higher or lower chance of surviving to the next generation based on their fitness value that is defined by their performance.

The better performing genomes will survive and produce children, which will substitute the worst-performing genomes of the generation. This way, in each new generation, the population will evolve into better performing instances for accomplishing a given task.

The downside of genetic algorithms is that they often tend to converge into local maximum since alternative solutions that would evolve into better performances might become extinct during the selection process.

2.4 Genetic algorithms Tetris AI

The application of genetic algorithms in Tetris showed great results with the help of appropriate heuristics to determine the fitness of each genome as demonstrated in the program developed by Yiyuan in 2013 [6].

In this project a different, but similar set of strategies is used:

1. Minimize holes in between blocks
2. Maximize cleared lines
3. Minimize aggregate height
4. Minimize bumpiness

Step prediction is also used to compute the fitness of the game state before executing the action, therefore it was possible to chose which move between the available ones would lead to the best current result and take action accordingly.

This turns out to be an extremely powerful technique in AI learning, and the number of successive steps predicted will linearly improve the performance of the AI, but exponentially increase the computational time, since each successive possible step will contain another set of available moves.

In my project, however, unfortunately, this method is unavailable due to the constraints in input data, and only the current game score for each frame is available to the AI.

2.5 NEAT

NEAT, is a genetic algorithm that solves most of these problems by introducing new concepts as innovation numbers and species, that will help in the preservation of diversity in the behaviors of the AI [1].

2.5.1 Neuro-Evolution vs Classic Genetic Algorithm

NEAT behaves similarly to a regular genetic algorithm where a population of instances is generated, and in each generation, the fittest instances are selected to survive and breed to create the next generation, while the least fit instances will be eliminated.

However, Neuro-Evolution's peculiarity is that the structure of the Neural network is mutated over time rather than only its weight values.

The structure of the genomes is considered as a mutable parameter like connection weights or biases, which let the AI to generate a more diverse set of behaviors to have a wider reach for all the available strategies a player can utilize.

2.5.2 NEAT vs Neuro-Evolution

By evolving the neural network's structure, one of the problems that will arise is the *Permutations Problem* [2], where multiple genomes with the same neural network structure but with their nodes positioned in a different order, would be identified by the algorithm as two different structures.

Since each genome structure has multiple different encoding solutions, two identical genomes with different encoding can produce an offspring. When this happens, the offspring will likely be damaged and lose information instead of improving. In NEAT this problem is avoided by having a historical marking to keep track of the evolution progress of each genome. This means that each innovation made by the genome will be labeled with a unique id in a chronological order so that it will be possible to order the genes by time and ensure that duplicates of the same genomes will also have the same encoding.

2.5.3 Historical marking

Another advantage of historical marking is the possibility of comparing two genomes to define how similar they are to each other. This tool will allow the algorithm to divide the population into species of similar genomes.

In a regular genetic algorithm, when a genome finds a working strategy that is better than the previous one, all the instances of the population will eventually try to adopt a similar strategy and improve it from that position.

In NEAT, however, a wider variety of options is left available in each generation to preserve a wider variety of genomes, therefore possible behaviors. The ”variety” of genomes in NEAT is encoded as different species: two genomes that have a similar neural network structure are said to belong to the same species, while a genome that doesn't share many common traits will belong to a different species.

At the end of each generation, the least fit genomes of each specie are eliminated rather than of the entire population to avoid the extinction of potential species that might outperform the current most-fit genomes. This solution will prevent the algorithm to get stuck in a local maximum without further improving its performance.

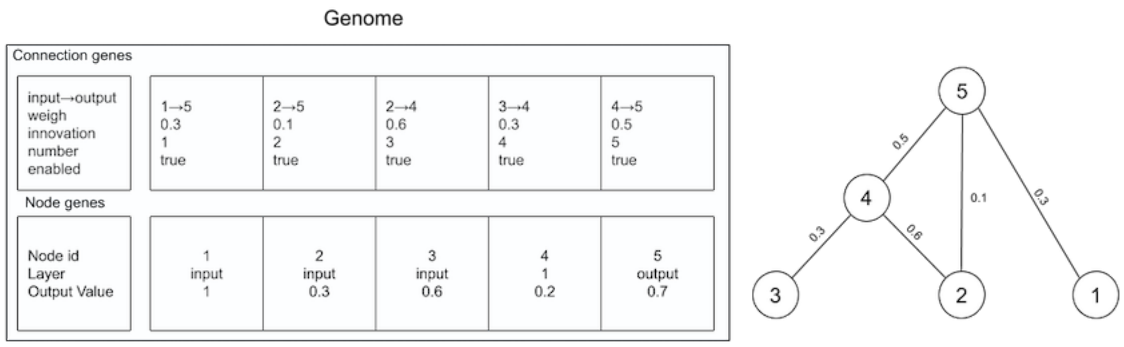
Chapter 3

Implementation

3.1 Genome Structure

The main class of the algorithm is the Genome class, it represents the neural network composed by nodes and connections (genes) that defines the outputs to be executed for some given inputs. The Genome class contains two lists of genes:

- Connection Genes
- Node Genes



Genome

Connection genes

input→output weigh innovation number enabled	1→5 0.3 1 true	2→5 0.1 2 true	2→4 0.6 3 true	3→4 0.3 4 true	4→5 0.5 5 true

Node genes

Node id Layer Output Value	1 input 1	2 input 0.3	3 input 0.6	4 1 0.2	5 output 0.7

Figure 3.1: Structure of Genes

3.2 Historical marking

Each time a new connection is created, the innovation number is increased by one. The innovation number represents the historical marking and it is used to track the evolution of the

genomes.
Two connections in different genomes will identify the same connection, which means that up to that point, the two genomes will share the same topology. This way each connection will have a unique identifier to solve the *Permutations Problem*, and it will also allow the algorithm to compare two genomes and identify the common ancestors, which is the maximum shared innovation number between the two genomes.
This will be relevant for the speciation step that will be explained in further details later in this chapter.

3.3 Mutation

In order for the genomes to evolve and explore new optimal solutions, there's a chance for each genome to mutate its structure or connection weights.
Possible mutations available are:

1. Change weights
   - Uniform perturbation
   - Randomly assign weights
2. Add node
3. Add connection
4. Enable/Disable connection

3.3.1 Change weights

This mutation changes the weight of the connection genes in a genome.

- **Uniform perturbation:**
  All the connections in a genome have their weights multiplied by a random factor.
- **Randomly assign weights:**
  A new random value is assigned to all the weights of the connections of a genome.

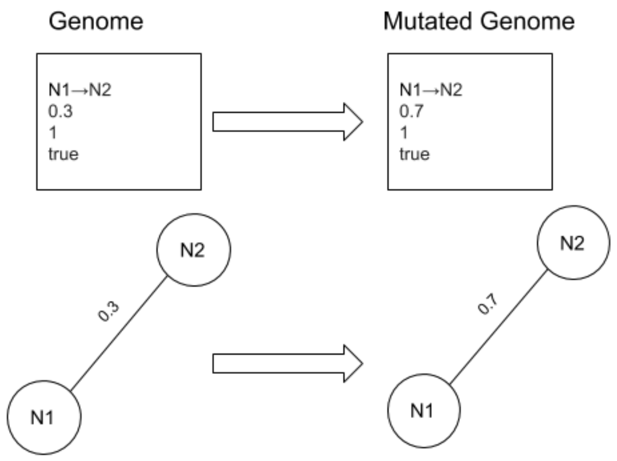


Figure 3.2: Mutate weights

3.3.2 Add node

This mutation adds a new node gene in an existing connection gene:

1. A new node gene N3 is created
2. the connection gene is disabled
3. A new connection having same weight as the old connection is created between N3 and N2
4. A new connection having weight 1 is created between N1 and N3

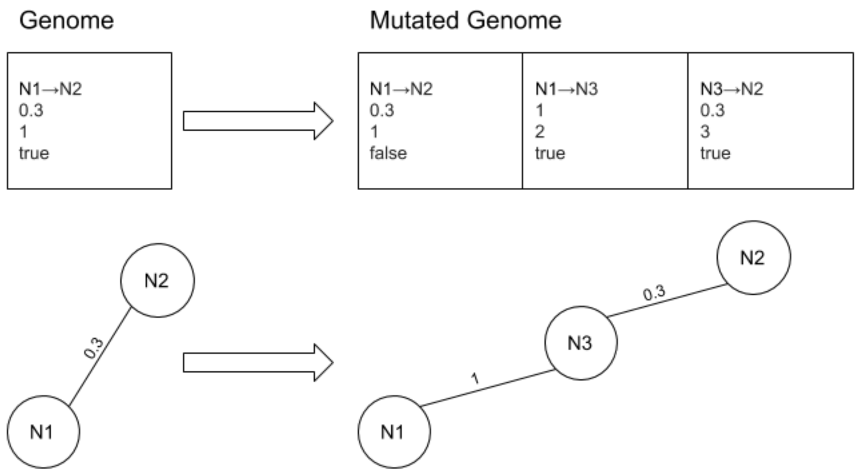


Figure 3.3: Add node mutation

3.3.3 Add connection

This mutation adds a new connection gene between two node genes.

1. Two random nodes N1 and N2 are selected where the two nodes' layers are $N1 < N2$
2. A new connection with random weight is created having N1 as input node and N2 as output node

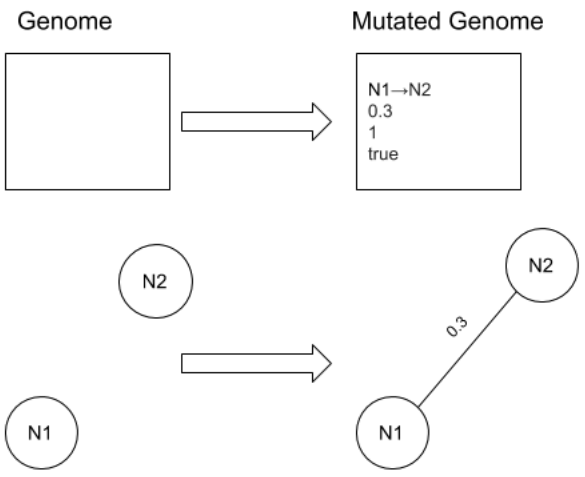


Figure 3.4: Add connection mutation

3.3.4 Enable/Disable connection

This mutation enables or disables an existing connection gene.

1. A random connection is chosen
2. if the connection was enabled, disable it. Otherwise, if the connection was disabled, enable it

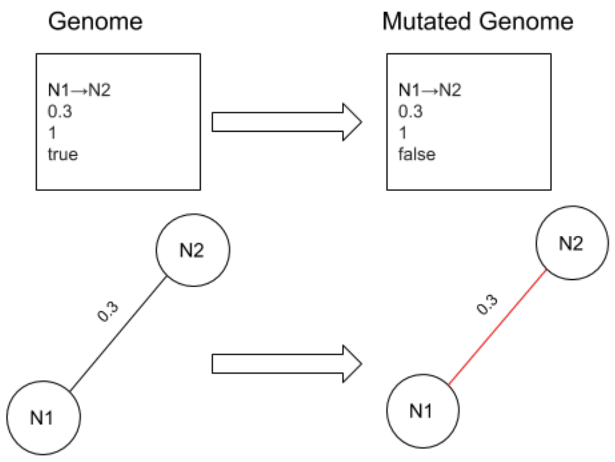


Figure 3.5: Mutation to add a new node

3.4 Speciation

Speciation is the phase where the population of genomes is divided into species based on their distance values. The evolution in different species allows the algorithm to explore different niches to have a wider reach and avoid local maximums.
To do so, the distance value δ between genomes is calculated by comparing their connection genes. Genes that don't match are called excess or disjoint genes, where excess genes are the ones that don't occur in the other genome's innovation number range, while disjoint genes are the ones that do occur in the other genome's innovation number range.

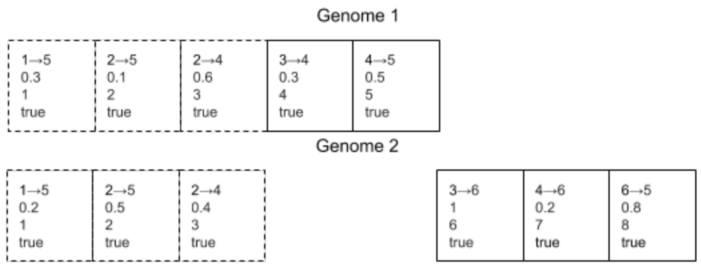


Figure 3.6: Two different genomes that share the common ancestor until innovation number 3. Genome1 contains 2 disjoint genes while Genome2 contains 3 excess genes. The average weight difference of matching genes is 0.23

The more excess and disjoint genes between two genomes, the less they share in terms of topology and common ancestor history, and the more distant they are. Different weight values in matching genes also affect the distance value between the two genomes. The distance value is calculated as follows:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot W \tag{3.1}$$

Where :

- c_1, c_2 and c_3 are constant multipliers as tuning parameters,
- E is the number of excess genes
- D is the number of disjoint genes
- W is the average weight difference of the matching genes

When two genomes are compatible $\delta < \delta_t$, they are considered to belong in the same species, δ_t being a constant threshold value.

Each species will elect a representative genome that belongs in the species, and in each generation, all genomes will be compared with the representatives of each specie to decide in which species the genome should belong in. If no species result compatible, a new species is created and the genome will be added as representative of that new species.

3.5 Selection

In each generation, every species will kill half of its population, based on genomes' fitness score. The worst performing genomes in each species are killed to be substituted by the offspring of the better performing half. This way each genome only have to compete with other genomes belonging in the same species rather than in the entire population, so that a more diverse range of behaviors is preserved over the generations.
A species-based selection will prevent the algorithm to make weaker genomes extinct before they have the chance to develop into more prominent members of the population, but at the same time will also slow down the learning process since many genomes that start to adopt losing strategies will also have a higher chance to survive.
To avoid the population being filled by poor performing genomes it's necessary to cull the species that reach a stale point. A certain amount of time is given to each species to improve, and if that limit is surpassed, the species is considered stale and therefore culled from the population.
In practice, it's necessary to find a compromise between preserving a diverse speciation and culling bad species to find a good balance for the population to improve as a whole.

3.6 Crossover

3.6.1 Offspring distribution

Once the population is halved, it is necessary to substitute the genomes with new ones. This is done through the crossover of the survival genomes. Each species will be assigned a number of available children for the species depending on the adjusted fitness of the genomes belonging in the species.
The adjusted fitness value is a normalized fitness score that takes in consideration of the number of genomes of a species; which means that given two genomes, the genome belonging to the smaller species will have a higher adjusted fitness value than the other genome.

The adjusted fitness value of each genome is calculated as:

$$f'_g = \frac{f_g}{N_g} \tag{3.2}$$

Where f_g is the fitness of the genome g and N_g is the number of genomes belonging in the same species.

This normalization also helps to prevent the fittest species to overwhelm all the other species by taking all the available offspring slots in the population.

The sum of all adjusted fitness values of the genomes belonging to a certain species is considered as the adjusted fitness value of the species:

$$f'_s = \sum_{n=1}^{N_g} f'_{gn} \tag{3.3}$$

And the sum of all adjusted fitness values of each species is considered as the total adjusted fitness:

$$f'_{tot} = \sum_{n=1}^{N_s} f'_{sn} \tag{3.4}$$

Where N_s is the number of species that exist in the population. The number of available children for each species is then calculated as:

$$n_s = \left\lfloor \frac{f'_s}{f'_{tot}} \cdot n_{tot} \right\rfloor \tag{3.5}$$

Where n_{tot} is the total number of available offspring slots, that is the number of genomes killed during the selection stage.

In conclusion, each species will be able to produce n_s children to populate the next generation.

Example:

In a population of 10 genomes there are 3 different species. Selection just occurred and 5 slots just became free to be populated by the next generation.

If the survivor genomes have the following fitness value and species,

Genome g	fitness f_g	Species S	Adjusted fitness f'_g
g_1	6	1	$\frac{6}{2} = 3$
g_2	8	2	$\frac{8}{1} = 8$
g_3	5	1	$\frac{5}{2} = 2.75$
g_4	7	3	$\frac{7}{2} = 3.5$
g_5	9	3	$\frac{9}{2} = 4.5$

The total adjusted fitness f'_{tot} will be $3+8+2.75+3.5+4.5 = 21.75$, and the offspring slots distribution will be calculated as:

Species S	Adjusted fitness of S f'_s	Number of slots allocated n_s
S_1	$3+2.75 = 5.75$	$\lfloor \frac{5.75}{21.75} \cdot 5 \rfloor = 1$
S_2	8	$\lfloor \frac{8}{21.75} \cdot 5 \rfloor = 1$
S_3	$3.5+4.5 = 9$	$\lfloor \frac{9}{21.75} \cdot 5 \rfloor = 2$

Since only 4 children are created but there are 5 available spaces, the remaining spaces will be allocated to be specie with the highest f'_s, which is S_3.

3.6.2 Offspring generation

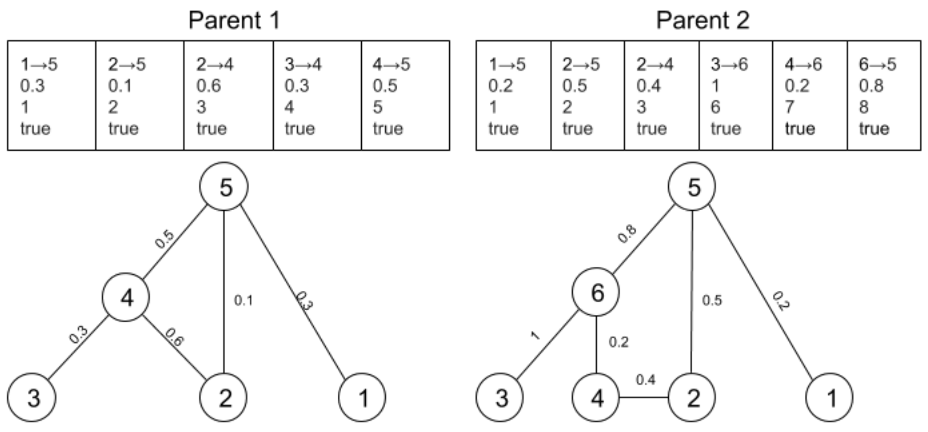


Figure 3.7: Given the same genomes as Figure 4.6 as parents for Crossover, the resulting children will be displayed in Figure 4.8

A new genome can be generated wither by cloning a parent genome or through crossover from two parent genomes.

When an offspring inherit genes from two different parents in NEAT, traditional crossover techniques will not work since the parents might not share the same topology. Also, inheriting all genomes from both parents to make sure no information will be lost between the generations will lead to an exponential and unbounded growth of the topology size. To avoid this, it is necessary to select the right genes from each parent, and a useful tool available in NEAT is the innovation number.

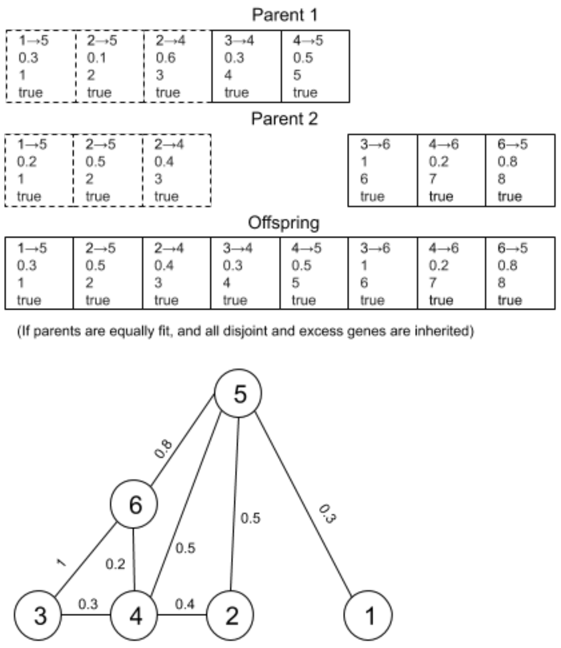
The innovation number will identify the common ancestor that two genomes inherited from; this means that all the genes that are contained in both parent genomes (matching genes) actually represent the same gene in both parents and do not need to be inherited twice in the children genome.

The innovation number of a gene only defines the structural identity, therefore two genes sharing the same innovation number might have different weight values. All the not matching genes, which are the disjoint genes and the excess genes, are only inherited from the fittest parent.

In conclusion, the child genome created by the crossover will inherit all the matching genes with weights randomly assigned from one of the two parents, and the not matching genes from the fittest parent.

In the cases where both parents share the same fitness value, not matching genes are also chosen randomly between the two parents.

The innovation numbers of the inherited genes will be maintained in the child genome so that the historical information will also be preserved.


Parent 1
Parent 2
Offspring
(If parents are equally fit, and all disjoint and excess genes are inherited)
5
6
3
4
2
1
0.8
0.5
0.5
0.3
1
0.2
0.3
0.4

Parent 1

1→5	2→5	2→4	3→4	4→5			
0.3	0.1	0.6	0.3	0.5			
1	2	3	4	5			
true	true	true	true	true			

Parent 2

1→5	2→5	2→4			3→6	4→6	6→5
0.2	0.5	0.4			1	0.2	0.8
1	2	3			6	7	8
true	true	true			true	true	true

Offspring

1→5	2→5	2→4	3→4	4→5	3→6	4→6	6→5
0.3	0.5	0.4	0.3	0.5	1	0.2	0.8
1	2	3	4	5	6	7	8
true	true	true	true	true	true	true	true

Figure 3.8: The offspring genome inherited the weights of genes 1 and 3 from parent1, and the weight of gene 2 from parent2. Assuming that both parents share the same fitness value, all disjoint and excess genes were inherited in this example

3.7 Feed-Forward

Each genome function as a traditional neural network with input, hidden and output node. Each not-input node receives as input the sum of all output values from the incoming nodes. A node N1 is defined as the incoming node for another node N2 if there is a connection gene between them that has N1 as input node and N2 as output node. All the nodes will then compute the outputs as:

$$o = \sigma\left((\sum_{n=1}^{m} w_n \cdot o_n) - w_b \cdot b) \right) \tag{3.6}$$

Where:

- σ = sigmoid function
- m = number of incoming nodes
- w = weight of the incoming connections
- a = output value of the incoming node
- b = bias

- w_b = weight of the bias connection

The sigmoid function will distribute the output value between the range of values 0 and 1 and it is defined as:

$$\sigma = \frac{e^x}{1+e^x} \tag{3.7}$$

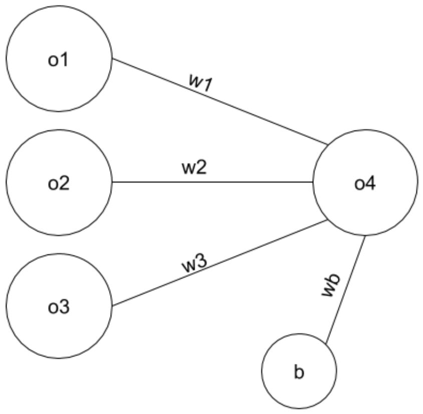


Figure 3.9: An example to feed forward with 3 input nodes (o_4,o_4,o_4) and 1 output node (o_4): $o_4 = \sigma(o_1w_1 + o_2w_2 + o_3w_3 - bw_b)$

If an output node's output value exceeds the threshold $o > \delta o$, the command assigned to the output node is executed from the game the genome is attached to.

3.8 Additional features

3.8.1 Genome Renderer

The program also features a rendering tool to visualize the game being played as well as the state of the neural networks. Only the neural network of the fittest genome from the previous generation is rendered in each generation. The current fittest genome is not rendered because it would be much less efficient to find the best genome each frame while the games are playing since the value is dynamic and keeps changing each frame.

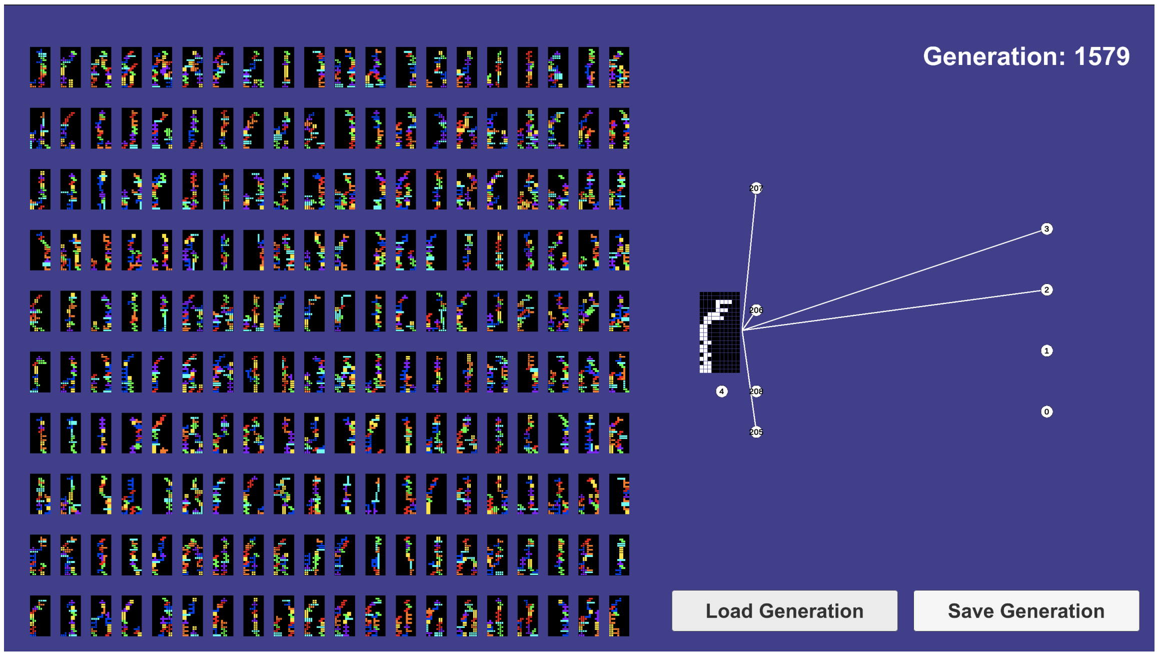


Figure 3.10: The genome renderer is on the right of the screen. Instead of visualizing each input node individually they are condensed together as a display of what the AI ”sees”.

3.8.2 Data encoder

A training session can be saved and loaded with the buttons on the bottom-right of the screen. The data of the simulation is saved in a .json file so that a training session can be paused and resumed any time. This is feature will let the user save the progress of a training session that would otherwise be lost once the program is terminated.

Chapter 4

Integration

4.1 Tetris Game

The game is created in Unity3D and implements all the basic mechanics expected from a Tetris game.

4.1.1 Mechanics

- Tetrominoes will move down every frame by one space
- Each tetromino spawns when the previous one lands
- Possible inputs are:
  - **Up** = Rotates current tetromino counter-clockwise by 90
  - **Left** = Moves current tetromino by one space to the left
  - **Right** = Moves current tetromino by one space to the right
  - **Spacebar** = Moves current tetromino down until it collides
- When a row is filled with squares, all squares in the row will be destroyed and score will increase. The squares above will also fall down to fill the gap.
- When a square touches the roof of the screen, the game is over

Each frame a game survives, 1 point is added to the game's score, and each time a line is destroyed, 200 points are added to the game's score.

4.2 Integration with AI

A binary matrix is used to keep track of all the cells that are currently occupied by tetrominoes blocks. This is necessary for the game to correctly handle the collisions and it is also used as input for the neural networks.

The size of the matrix is 10 row and 20 columns, which corresponds with the size of a Tetris that is 10 cells wide and 20 cells high.

The input nodes of the AI correspond with the cells of a single Tetris game, while the output nodes correspond with the possible actions. Therefore, an initial genome will be created with 200 input nodes and 4 output nodes.

Each genome will be created without any connection. New connections and nodes will be added to a genome only through mutations.

Genomes are initialized as empty so that the exploration only occurs in the minimal search space in a way that if a solution is found, it is more likely for it to be a minimal solution.

A population of 200 games is used in this project for testing, where to each game instantiated as GameObject is assigned a different available genome.

Each genome will then act as a brain for the game it is assigned to the state of each cell in the game is set as 1 or 0 depending on if the cell is either occupied or not by a block in the current frame; the input information is then fed into the neural network as input nodes.

The fitness of the genomes corresponds to the current score of the game they are attached to since for the scope of the project no heuristic is used to optimize the fitness value according to other parameters.

4.3 Parameters setting

In the tests conducted, the initial population is set as 200. The number was decided arbitrary based on current hardware limitations.

Each connection gene in the genomes have a 80% of mutating their weight; if this mutation occurs there is 90% chance for the weight to perturb by a random amount and 10% chance for the value to be assigned with random values. The perturbation occurs by multiplying the existing weight by a random value between -2 and 2, the result is then clamped between -1 and 1.

Mutating the weight is arguably the safest way for a genome to mutate so that the chance of the mutation can be set as high as 80%. It is also the main way the algorithm explores new solutions every generation; a lower chance of evolving would cause the algorithm to dilate the search time, therefore, slow down the learning process.

Each connection gene also have a 10% chance of being enabled or disabled based on the current connection status.

Each genome have a 3% chance of adding a new node to a random existing connection and 30% chance of adding a new connection between two random existing nodes.

The connection mutation is set to be significantly higher than the node mutation because node mutations depend on connection mutations since it can only occur on an existing connection gene.

Mutation rates are chosen to be small values to prevent the algorithm from losing the progress already achieved. It is important to find a good compromise between exploration and exploitation. Exploration means looking for new strategies to improve by mutating the neural network and exploitation means focusing on a certain structure to hopefully finding the global optima. Having a high mutation rate would allow the algorithm to search a wider range of solutions, but not converge into the optimal one. In this case, since 200 genomes are used as population size, the mutation rate can be higher, since the bigger a population is, the better it will be able to handle mutations, which might endanger the "good" genomes by overwhelming them. In other words, the more genomes are available to the algorithm, the more genomes can be allocated to explore rather than exploit without the risk of losing the progress made.

The constants for speciation are set as $c_1 = 2$, $c_2 = 2$, $c_3 = 3$ and $delta = 3$. This is because a Tetris game can be a really difficult task to solve for an AI without providing enough input data. Since in this project, only the cells' status is given, it is expected for the AI to take a long time to evolve into an optimal network, therefore, it is expecting the network to be very complex by the end of the training process. This would lead to a very large number of node genes, that added to the already existing 200 input nodes lower the value of $\frac{c_1 E}{N}$ and $\frac{c_2 D}{N}$. Having $delta$ too large or c_1 and c_1 too small would lead the population to converge into fewer species, which would kill the whole purpose of speciation.

Chapter 5

Legal, Social, Ethical and Professional Issues

5.1 The Ethics of Artificial Intelligence

The creation of an artificial intelligence inherently raises many ethical issues since machines designed to optimize the results will often contradict moral and ethical values that cannot be expressed quantitatively.

The judgment of machines can only by quantified via values and decision trees, but in an ethical aspect, this can be much more complex than that.

In the project, for example, it is clear how the elimination of under-performing genomes can be beneficial for the improvement of the whole population through reproduction via crossover, but the outcome of this simulation is not suggesting in any way that the elimination of less performing member of the world's population in any aspect will be beneficial for the population, nor will make the organism evolution process faster and more reliable.

The distinction in species in NEAT might also suggest ethnic discrimination if improperly interpreted, but it is important to remember how complex and diverse the world is and would be impossible to predict through computational simulations, therefore any outcome observable from such is not to be taken as a good indication of a real-world prediction. Genetic algorithms might be inspired by the natural evolution of organisms, but is not to be taken as an argument to justify unethical behaviors such as ethnic discrimination and massacres.

Further development in Artificial Intelligence always needs to follow strict guidelines regarding ethical or legal issues, and this project is no exception.

Even though the project is really only implementing a simple AI learning to play Tetris, this might not exclude that further development of the field will lead to the creation of intelligences that might develop a sufficiently advanced moral code to be treated as human and granted human rights.

If this visionary scenario ever becomes true, it is still unclear if these intelligences will ever be counted as people or will be governed under the same or different social rules, if they will be a threat for people's safety or economy, if either they will hold responsibility as individuals or it will be transferred to their creators or owners.

The unpredictability of the AI's behavior is also a major concern when social issues are been taken into consideration. In this project, the set of possible actions is limited to 4 in-game moves, but the project is designed to be easily adaptable to any generic game, and therefore possibly applicable to other scenarios that might not be gaming related using some sort of exploitation since it is hard for a software to identify if the program being played is an actual game or not.

This will expand the possible moves to an infinite set since it is the user's responsibility to map the output nodes of the genomes to the game commands, and malicious users might exploit the program outside of its designed field causing possible unpredictable behaviors that might even be hazardous to ethical rules or people's safety depending on the application field.

An example might be the program applied to a real-world scenario where inputs are people's personal data and as output if either hire or not a candidate as an employee to maximize the potential profit for the company. The program might suggest employing a certain set of candidates, but it would be impossible to determine if the given decision is violating the current rules concerning gender or ethnic discrimination. In fact, a neural network can become really hard to decode once evolved and it could become impossible to determine the factors that led to a given decision and such tool can be used to justify discrimination intents since the decisions made are computed by a computer that has no knowledge about discrimination issues.

Chapter 6

Results/Evaluation

When the initial population is created, the game is expected to not do anything until a connection mutation occurs, since the input nodes will not be connected to the output nodes in any way.
The different fitness values ad this stage will solely depend on the randomness of the type of tetrominoes spawned: the higher is a tetromino, the faster a game reaches the top and loses.

6.1 Test 1: Neuro-evolution vs Fully connected network with no node mutations

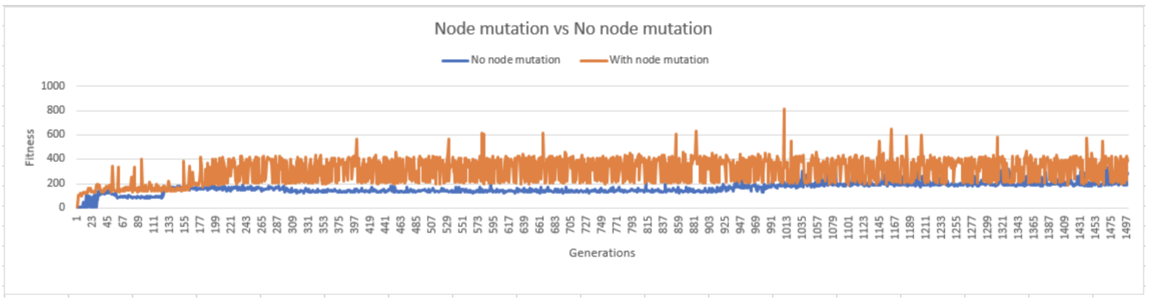


Figure 6.1: Comparison of the first 1500 generations between evolving topology and only mutating node weights

By comparing a regular fully connected neural network with no hidden layers, and a neuro-evolution network with evolving topology is noticeable from the Figure 6.1 that the learning curve of the latter is much faster.
The first steps in learning a Tetris game can be divided into:

1. Random behaviors

2. Attempt to survive as long as possible

3. Attempt to clear a line

As illustrated in the figure, the neuro-evolution algorithm managed to reach step 3 and consistently destroy lines every few generations around generation 200, while the genome with no node mutations only managed to reach that stage around generation 1000. The visible peaks in the graph indicates when a genome scored points by clearing a line. One of the reasons why the network with no hidden layers might not be optimal in this situation is the complexity and randomness of the game: having no hidden layers means that there must be a correspondence between an input and an output, but in Tetris, this correspondence doesn't exist. The output depends mostly on the relation of multiple inputs, like the current status of the field or on the relation between cells to identify which tetromino is being active. Such complex behaviors are not possible without a deeper network, and a fully connected deep network might be way too computationally intensive to be realistically feasible. Neural evolution limits the amount of computation required by developing new nodes and connections only when necessary.

Although the initial phase looked promising, a downfall of this algorithm can be noticeable from the graphs: when a line is destroyed, the next generation hardly keeps the progress and achieve the same result, which means that even though a genome develops into a promising structure, it has a chance of not performing as well in the next generation and getting discarded.
This problem might arise when the balance between exploitation and exploration is not satisfied. Each generation the best performing genomes have a relatively high chance of mutating and losing their progress since each gene in the genome has a high percentage of mutating its weights, a genome hardly remains unchanged between the generations.

6.2 Test 2: limiting mutation

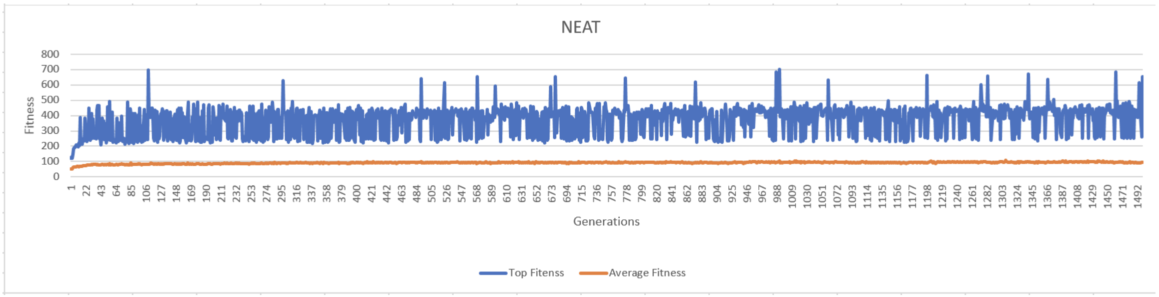


Figure 6.2: Top and Average fitness for a neural network with limited mutation

One tweak made to the algorithm to improve the exploitation is to only mutate the newly generated genomes and force a copy of the fittest genomes of each specie in a new generation. In other words,

the fittest genomes of each generation have 100% chance to generate the offspring without crossover rather than 75% chance. The survivor of the previous generation also remains unchanged since only offsprings are mutated in each generation.
This will slow down the exploration significantly depending on the total number of species, since very little genomes will be assigned to explore new solutions.
More precisely:

$$N_{mut} = N_{tot} - (\frac{N_{tot}}{2} + N_s) \tag{6.1}$$

Where on a population of size N_{tot}, $\frac{N_{tot}}{2}$ is the number of survivors from the selection stage and N_s are the fittest genomes in each species that are copied to the next generation.

This solution looks more promising than the previous one, and the transition from a stage to another highly depends on how early the randomness of the commands leads to a line destroyed.
Once that happens, the AI will be able to destroy a line more consistently over time compared to the previous implementation, which brings to another problem in comparing the different algorithms.
A reliable benchmark is not possible to achieve when so many factors are purely random, the "improved" algorithm might look better in this graph, but the consistency it finds the way to improve is not evaluated, since only one execution of the algorithm is reported.
To benchmark the various algorithm properly, a random seed is used to test each implementation, even though this method of evaluation will not be true to the scope of the project, where the aim of the AI is to solve problems in a much more generalized approach.

6.3 Test 3: deterministic game

When the game is programmed to have a deterministic sequence of tetrominoes falling down, executing the same set of commands will lead to the same outcome and fitness value. This means that by using the limited mutation implementation, it is possible to specialize the AI to play this particular sequence of tetrominoes much more efficiently since the best players in each specie will survive and have a copy of them in the population of the next generation.
This feature allows the algorithm to never lose the progress made, and once and improvement is achieved, it will be impossible for the next generation to perform worse.

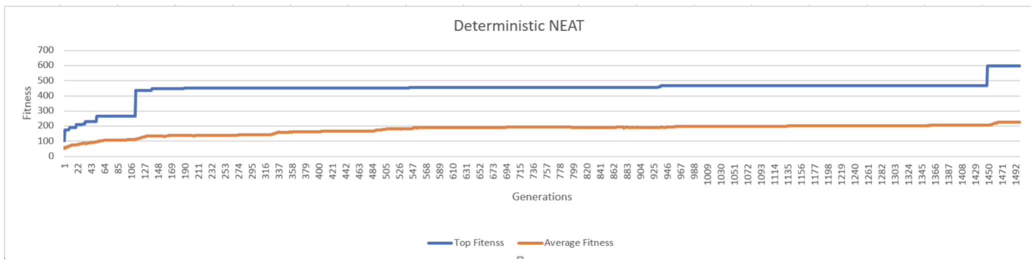


Figure 6.3: Top and Average fitness for a neural network with limited mutation and deterministic randomness

This implementation performs significantly better compared to the previous ones, and for the first time, 2 lines are destroyed in the same game over a 1500 generations training.
The population will not improve its top fitness until generation 4000, but the average fitness will slowly increase. Each training session only lasted around 6-7 hours, so further generations' data is not available. But it is safe to assume that the AI only learned how to clear those specific lines instead of learning the general strategy to clear lines and score points, which means that further improvements will take much longer, since clearing multiple lines in a single game is significantly harder than clearing a single line when the game field is still empty.

6.4 Test 4: Neuro-Evolution vs NEAT

The division into species plays a huge role in NEAT, but behind all the theoretical constructs, it is now time to visualize how much it is contributing towards the overall performance of the algorithm compared to a regular Neuro-Evolution algorithm without speciation.

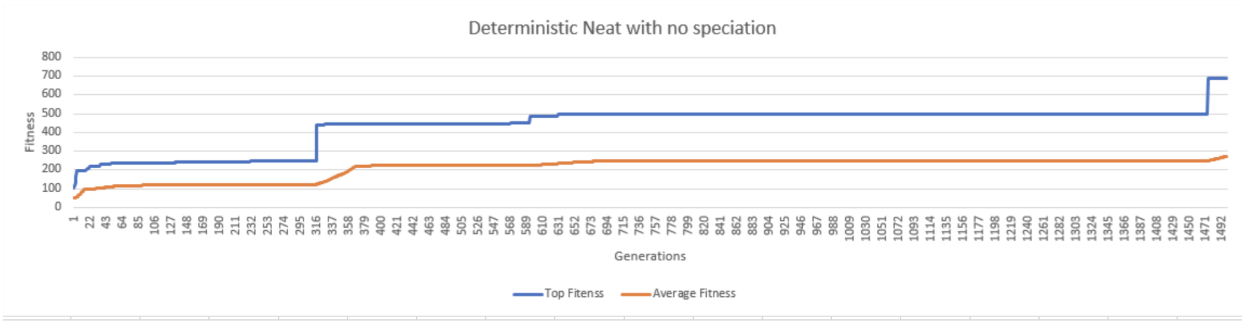


Figure 6.4: Top and Average fitness for a neural network with speciation disabled

From the graph of figure 6.4, it is noticeable how the lack of proper exploration in Neuro-evolution delayed ”step” in NEAT from generation 100 (in Figure 6.3) to generation 300. That step indicates the first time a random combination of commands happened to destroy a line in a game. In this test, both algorithms were using the same seeds for the random number generator as well as the same tuning parameters, but randomness is still a big factor in when a genome would find a way to improve. It can be seen that the destruction of the second line is found almost around the same time in both implementations.

More tests are conducted with different seeds, and in all of them NEAT was able to find improvements faster than the regular Neural-Evolution.

Speciation allowed the survival of a wider variety of behaviors which increased the searching area, therefore sped up the time to find the right commands to score a line, other than solving the local maximum problem.

Chapter 7

Conclusion and Future Work

7.1 Conclusions

By the results of the tests, it is pretty clear that the current state of the implementation is not a solution for the idea of a universal gaming AI. Test 2 showed that even an algorithm for a non-deterministic Tetris game can be quite time consuming since the algorithm ended up clearing at most 2 lines in a row during a training session of 4000+ generations.
This might be due to the complexity of the game itself since the AI has no information about the relation of the various inputs: the game cannot recognize that the current falling tetromino is ”L shaped” rather than ”Z shaped”, it only sees that a certain set of cells was activated, and the next frame another set of cells is activated, therefore NEAT would need to build a network that covers every single possible combination to truly reach a state where any state of the game will be handled optimally. That state of the evolution might even correspond to a deep fully connected network, therefore the whole strategy of evolving the network to find a minimum solution would be completely useless.

A not contained mutation is also proven to be detrimental for the efficiency of the algorithm since well-performing genomes will often mutate into worse performing ones during the transition from a generation to the next one. This way all the progress made to achieve a high performance is often lost.

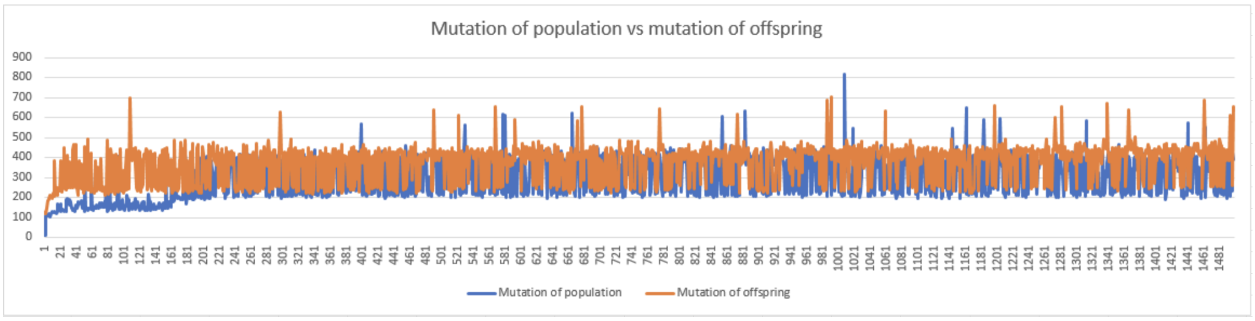


Figure 7.1: Comparison between Mutation of the whole population and mutation of offspring only

The graph in Figure 7.1 shows how by limiting mutations to only the offspring, is possible to maintain the best genomes from each generation intact and therefore have a much higher consistency in hitting peaks (clear line) and an overall higher score compared to the counterpart where mutation occurs throughout the whole population.

Test 3 showed that the predictability oh the environment is a huge factor for the learning curve of the AI. Once an obstacle is overcome by at least one player, that obstacle doesn't represent a problem anymore and at least one specie will always maintain the progress achieved. This is noticeable in Figure 6.3 where the fitness values are never lower than the previous ones, which means that given enough time, the AI may eventually find an optimal way to maximize the score with that given sequence of tetrominoes, since only improvement is allowed.
Test 4 shows the contribution of speciation for a wider exploration range.

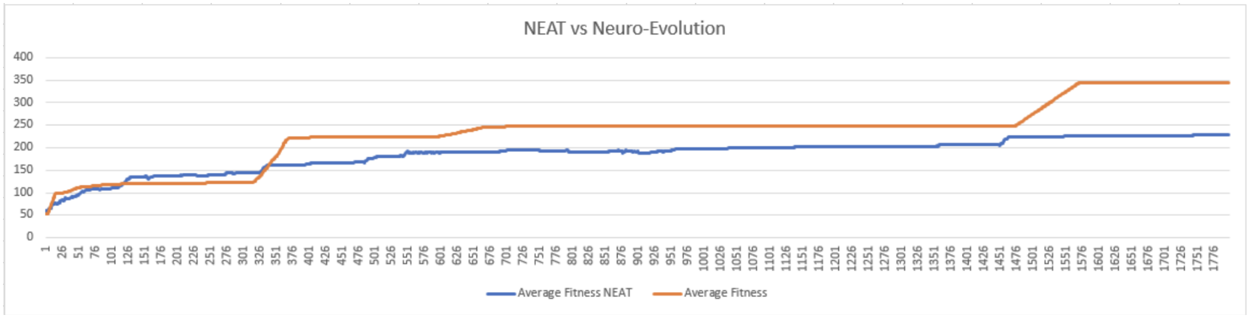


Figure 7.2: Comparison of average fitness between NEAT and Neuro-Evolution

A good way to evaluate the effectiveness of exploration is to observe the average fitness of the algorithms. If the algorithm doesn't prioritize exploration, all players tend to converge towards the top player, therefore the average fitness would be higher than the case where a more diverse variety of options that don't perform as well is also taken in consideration. In Figure 7.1 it can be noticed that NEAT shows lower average values almost across all generations, which suggests that exploration, in this case, is more effective compared to Neuro-Evolution where speciation is not utilized. The result of an effective exploration can be observed in Figure 6.3 and 6.4: The improvement NEAT was able to find at generation 114, was found only at generation 318 by Neuro-Evolution.

7.2 Limitation of the project and future solutions

Also, in this project, each input node corresponded to an entire Tetris cell rather than to a pixel in the screen, which means that only really small screen size games (10 x 20) were tested during the evaluation process, a normal-sized screen, even for classic retro games would be much larger (256 x 224 for the classical SNES resolution).

In order to solve this problem, another neural network should find similar patterns in the game field, so that shapes and colors can be recognized as parts of the same object. A Convolutional Neural Network would be a good fit for the task in a future project, given that the computational power to handle two different neural networks every frame for each individual genome would not be a limiting factor [5].

7.3 NEAT parameters tuning

To be noticed is also how tuning NEAT parameters can be a significant factor in the performance of the algorithm since for any different game a different set of parameters might turn out to be more effective than the others.

Even though the algorithm is designed to adapt and mutate according to the parameters, a wrong set of parameters can still affect the result of the learning process (A naive examples might be the case where the mutation chance is so low that the genome never generates the first connection) and the parameters can oscillate by a significant margin depending on the game itself or even on the screen size of the same game.

This means that parameters must be tuned manually according to the performance of the algorithm through human observation or heuristics, which might not be different from the specialized AIs that are also fine-tuned to play specific games in specific circumstances. In other words, designing a specialized AI for a specific game would be the same as designing a specialized set of parameters in the universal AI for a specific game.

In many circumstances, a generalized solution often comes with advantages and disadvantages compared to a more specific solution, and this is no exception. Usually, a good compromise is found between specificity and generality for the solution to be effective and flexible to be adapted in multiple situations, and the results of this thesis show that probably a universal solution for gaming AI is neither useful or effective in the current time compared to the specialized counterparts that have access to more specific data and are able to take advantage of them.

References

[1] Kenneth O. Stanley, Risto Miikkulainen. (2002). *Evolving Neural Networks through Augmenting Topologies.*

[2] Hancock, Peter. (1997). *Genetic Algorithms and permutation problems: a comparison of recombination operators for neural net structure specification.*

[3] Demaine,Hohenberger, Liben-Nowell.(2003). *Tetris is Hard, Even to Approximate*

[4] Nicholas Lundgaard,Brian McKee. (2006). *Reinforcement Learning and Neural Networks for Tetris*

[5] Matt Stevens, Sabeek Pradhan. (2016). *Learning to Perform a Tetris with Deep Reinforcement Learning.*

[6] *Tetris AI – The (Near) Perfect Bot.*
`https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/`

[7] *Coding a Tetris AI using a Genetic Algorithm.*
`https://luckytoilet.wordpress.com/2011/05/27/coding-a-tetris-ai-using-a-genetic-algorithm/`

[8] *Slitherin - Solving the Classic Game of Snake with AI.*
`https://towardsdatascience.com/d1f5a5ccd635`

Other useful links:

[9] *Tetris Game implementation.*
`https://www.youtube.com/watch?v=T5P8ohdxDjo`

[10] *NEAT introduction.*
`https://www.youtube.com/watch?v=VMQ0a4-rVxE`

[11] *Evolving Deep Neural Networks.*
`https://towardsdatascience.com/evolving-deep-neural-networks-ceb8d135d74d`