Will an artificial neural network teach itself the same Classic *Tetris* strategies that are used in high-level human play?

Peter N. Hebden 014hebden@farmors.gloucs.sch.uk
Farmor's School, Gloucestershire, UK

Research presented for the Extended Project Qualification

Farmor's School
United Kingdom
March, 2020

Will an artificial neural network teach itself the same Classic *Tetris* strategies that are used in high-level human play?

Peter N. Hebden

Abstract

Professional players of Classic *Tetris* (specifically – *Tetris* for the 1985 home entertainment console *NES*) employ an array of specific techniques in achieving the highest amount of points possible. The goal with this research is to attempt to produce an artificial neural network that will play Classic *Tetris*, and to examine whether the algorithm learns to use the same techniques that humans have developed over time.

Contents

List of Tables

List of Figures

1 Introduction

Artificial neural networks (ANNs) are complex algorithms that emulate biological neural networks at a basic level (Kasabov, 1996). For some tasks, designing an algorithm directly to compute them would be difficult. As one specialist puts it, "you quickly get lost in a morass of exceptions and caveats and special cases." (Nielsen, 2015). For this reason, many more complicated tasks employ the use of ANNs, as ANNs adjust the specifics of their own parameters during a process called training. The task of playing *Tetris* may seem simple for a human, but once one tries to break down the individual steps it becomes increasingly obvious that such a task would be near impossible for a human to design an algorithm to do optimally.

The environment in which the algorithm would learn would be a model of *NES Tetris* (*Tetris* for the *Nintendo Entertainment System*). One of the more obvious models with which to do this would be *NES Tetris* running on a *NES*, as there would be no need to create it and no issues with accuracy. However, an inherent flaw with this method is the cost of purchasing the necessary equipment, such as the *NES* itself, and a method with which to simulate a controller for the pin input. Although this is possible, and has been done for the technically similar *SNES* (Trinklein, 2015), there still remains the issue that this work would become less easy to repeat. Additionally, to reduce training time for the network, one would aim to have several environments running in parallel which would further increase the cost of using a real *NES* for the model.

Another possible model for the training would be the Read Only Memory (ROM) from the *NES Tetris* cartridge running on a programmable *NES* emulator such as *BizHawk* (TASVideos, 2019). This method has been used successfully before in training neural networks (Hendrickson, 2015), but runs into the issue that a person attempting to repeat this research would have to obtain a cartridge for the *NTSC* version of *NES Tetris* and also the equipment required to copy the ROM. This method would also not work as well with running many environments due to there being less room to optimise.

For these reasons, the *Tetris* model used in this work was programmed by the author, with the network's inputs directly built-in and the number of environments running only restricted by processing power. All software created for this project is open source under the *MIT License* (Hebden, 2020) in order to make reproduction straightforward. The model was created with three distinct interfaces. One interface was used for the training of the network, and another for demonstrating the result of the training. The third interface of the model is for human play for the purposes of testing accuracy. The distinction of these interfaces allows for each to be optimised, for example the training model could potentially have no visual output in order to reduce processing, whereas the demonstration model will have a full *Graphical User Interface* (GUI) designed to imitate the look of *NES Tetris*. The model will use the same calculations for gameplay, taken directly from those used in *NES Tetris*, accounting for how the *NES* processed information.

2 *Tetris* Model

Development for the model was started using *C#* in the *Unity Engine*, with a very modular approach to programming such that as much code as possible can be reused between interfaces in order to increase maintainability. Retrieving specific values for the model to use in calculations (e.g. the speed of pieces at each level, input delay from the controller for horizontal movement, scoring, etc.), was somewhat hard to obtain from a reputable source, due largely to the niche status of Classic *Tetris* and the lack of official values released by *Nintendo*. As such, all sources cited in this section come from dedicated groups of fans who have reverse engineered the workings of the game over many years. It should therefore be noted that while validity of these sources is debatable, they have successfully held up to scrutiny and been refined in cases where they have not due to the collaborative nature inherent to wikis.

All time-related values are recorded in frames, so for them to have any meaning in a system with fluctuating frame rate they must be converted into a unit of time. The frame rate, measured in frames per second (FPS), is equal to frames divided by seconds. Rearranging, we can determine the time by

$$seconds = \frac{frames}{frames \quad per \quad second} \tag{1}$$

According to editors of *TASVideos*, the website hosting the largest collection of recorded *TAS*s[1], the *NTSC*[2] version of the *NES* runs at 60.098813897441 frames per second (Mothrayas, Ilari, Cyorter, & brunovalads, 2018). For the sake of simplicity, I will refer to this value in the text as the integer 60. For example, the number of frames that each tetromino rests at each grid row before falling is 48 at level zero (Kitaru, Sval, Arcorann, & Simonlc, 2019; Mothrayas et al., 2018; Jjdb210 et al., 2018). Therefore, the time delay is just under four fifths of a second.

2.1 Movement

Tetromino rotation was implemented in accordance to the *RH Nintendo Rotation System*, which was the rotation system used for *NES Tetris* (Simonlc, 2019), and, according to the same source, tetrominoes are instantiated with their highest block at row 20. The number of frames that the falling tetromino is delayed at each grid row is given by Table 1. The next item to be implemented was horizontal movement of the falling tetromino.

Horizontal movement of the falling tetromino was more complex to implement than one might expect, as a result of a feature of *NES Tetris* known as *Delayed Auto Shift* (DAS). DAS is a behaviour commonly found in block puzzle games when the player holds the left or right key; the game will shift the falling piece, then wait, then shift it repeatedly if the key is still being held (Tepples, DIGITAL, Nicholas, Jason.tetris, & Tokihiko, 2019). The same concept can be seen in software such as word processors, where it is often the case that if

[1]A *Tool-Assisted Speedrun*, (TAS), is an attempt to complete a video game in the shortest possible time by programming exact controller inputs.

[2]NTSC, or National Television System Committee, is a standard of analog television colour.

Table 1: Frames per grid cell at each level. (Jjdb210 et al., 2018).

LEVEL	**FRAMES PER GRID CELL**
00	48
01	43
02	38
03	33
04	28
05	23
06	18
07	13
08	8
09	6
10-12	5
13-15	4
16-18	3
19-28	2
29+	1

one is to hold backspace there is a brief pause between the initial deletion and all subsequent deletions. According to editors of *Hard Drop* (Jjdb210 et al., 2018), this system is based on what is called the *DAS counter*. The DAS counter is initially zero and is incremented by one for each frame that the left or right button is held down. Whenever the counter reaches 16, the tetromino shifts and the counter is set to 10. The counter only reinitialises to zero whenever the button is released and pressed again. These findings appear to be attributed to Kitaru, and it is also noted on Tetris.wiki (Kitaru et al., 2019) that nothing is subtracted from the counter in the case of the piece being blocked, and that the counter is instantly set to 16 if the initial shift from tapping the button is blocked. Additionally, the counter is not affected during ARE[3] (Kitaru, 2010).

The line clearing mechanism works by checking if any row is full after a tetromino is locked. If any are, then the elements of that row are removed and all elements of rows with a greater index than the cleared row are moved to the below index.

The implementation of soft-drop[4] was not as straightforward as hoped, due to a misinter-

[3]Entry delay (ARE) refers to the frames between a tetromino locking in place and a new tetromino being initialised.

[4]Soft-drop is a feature of *Tetris* games describing the player's ability to hold down a button to increase

pretation of the source. The speed of soft-drop for *NES Tetris* is 1/2G, (Kitaru et al., 2019) and I initially interpreted G as being a variable correlating to the values in the right column of Table 1, standing for Gravity. This was because the webpage had no explicit information detailing the meaning of G, other than the Figure 1 table being captioned as "gravity speeds". However, upon implementing the feature like this, it became exceedingly obvious that the soft-drop was far too slow when compared with gameplay footage such as that by *YouTube* users World of Longplays (2014) and kitaru2004 (2010). At this point, I did more research and found that "gravity is expressed in unit G, where 1G = 1 cell per frame, and 0.1G = 1 cell per 10 frames" (Edo, 2010). Therefore, the rate of soft-drop for *NES Tetris* is approximately one grid row every 30th of a second.

Next, I implemented *ARE* which is otherwise known as entry delay and is the number of frames in between the locking of a landed tetromino and the instantiation of the next tetromino. The number of frames for ARE depends on the grid row the tetromino locks on and ranges from 10 to 20. When any lines are cleared, the entry delay has an additional number of frames to account for the line clearing animation that is played. However, the animation doesn't have a set number of frames each time it is run. Instead, each of the five frames of the animation happen when the total number of frames the game has had are divisible by four (Kitaru et al., 2019). From this, we can derive that the total number of additional frames is given by

$$f(x) = 20 - (x \mod 4) \tag{2}$$

where $f(x)$ is the additional frame delay and x is the total number of frames since the beginning of the program.

the rate of the piece falling. It should be noted that this is different to hard-drop, which describes the similar feature (not present in *NES Tetris*) allowing players to press a button to cause the tetromino to land instantly.

2.2 Scoring and Levelling

Table 2: Points scored for line clears at each level. (Arcorann & Simonlc, 2019).

Level	Points for 1 line	Points for 2 lines	Points for 3 lines	Points for 4 lines
n	40 * (n + 1)	100 * (n + 1)	300 * (n + 1)	1200 * (n + 1)

The final feature to implement was scoring and levelling. The score is incremented after each tetromino lands by two different means: the lines cleared, and how soft-drop was used. For lines cleared the increase in score is different depending on the amount of lines cleared, as seen in Table 2. These values are the consensus in the community due to always being accurate when tested (Arcorann & Simonlc, 2019; Tepples, Nicholas, et al., 2019). The second means of incrementing score, soft-drop, has conflicting descriptions. The scoring pages of both *Tetris Wiki* and *Tetris.wiki* state that:

> *For each piece, the game also awards the number of points equal to the number of grid spaces that the player has continuously soft dropped the piece.*
> (Tepples, Nicholas, et al., 2019)

However, a *Reddit* post by u/Poppinfreshzero raised some doubts about the accuracy of this, noting:

> *I have read in a few places that* [the score added for soft-drop] *is equal to the number of soft drops in a row when dropping a piece, but I'm only able to get a maximum of 12 points by dropping from the top. Since the board is 20 blocks deep, I would expect to get somewhere around 18-20.*
> (u/Poppinfreshzero, 2019)

I did some experimentation on *Tetris* on my own *NES* and got similar results. Additionally, I noticed that the score for soft-drop is only applied if the player holds down the soft-drop button on the frame that the tetromino locks in place. Another *Reddit* user responded to the post, stating that the scoring works in the same way as the DAS counter due to hardware limitations of the *NES* (u/x34l, 2019). After implementing these changes, I noticed no visible discrepancies between scoring on my model and scoring on the *NES*.

The method for levelling up is simple and implemented as described on *Tetris.wiki*. For most level-ups the number of line clears required is 10, but for the first level the number of required line clears is given by

$$f(x) = \min[x * 10 + 10, \max(100, x * 10 - 50)] \tag{3}$$

where $f(x)$ is the number of line clears required, x is the starting level, min(x, y) returns the smallest of x and y, and max(x, y) returns the largest of x and y. Derived from (Kitaru et al., 2019).

3 Neural Network

The process of playing a game of *Tetris* can be seen as a function with audiovisual information as input and the buttons pressed on the controller as output. Thus, to produce an artificial *Tetris*-playing agent is the same as specifying such a function. This can be achieved using a feed-forward neural network - a network consisting of neurons that are ordered into layers, with a single input layer and a single output layer (Svozil, Kvasnicka, & Pospichal, 1997; Nielsen, 2015).

3.1 Generic Feed-Forward Neural Network

Figure 1 illustrates the topology of a generic feed-forward neural network with three inputs, a hidden layer consisting of four nodes, and a single output. It represents a function which takes three arguments and returns one value. In this model, each node in the hidden layer has a value of the sum of all connected nodes in the previous layer (which, in this case, is all the nodes in the input layer) when multiplied by the weight of that connection. Hence, this will be referred to as the weighted sum. The weighted sum is then incremented by some bias which is unique for each node (Sanderson, 2017). In the figure, the weighted sum is given as Σ and the bias is given as b_n for each node n. This process is repeated to give the values for each node in the output layer, this time using the hidden layer as its input.

It has been omitted from the figure for clarity, but each node may also have an activation function which defines the output of that node based on $\Sigma + b_n$ (Haykin, 1994) (e.g. turn any input into a value between zero and one). Two common examples of such functions are threshold functions (Equation 4) and logistic functions (see Equation 5 from Verhulst (1845)). A major benefit to using a logistic function in particular, such as Equation 5, is that it introduces non-linearity which allows the network to classify more complex input (Lague, 2018). Additionally, threshold functions are often harder to train as a result of plateaus in the output (Figure 2) meaning that a small change in the network - in the wrong or right

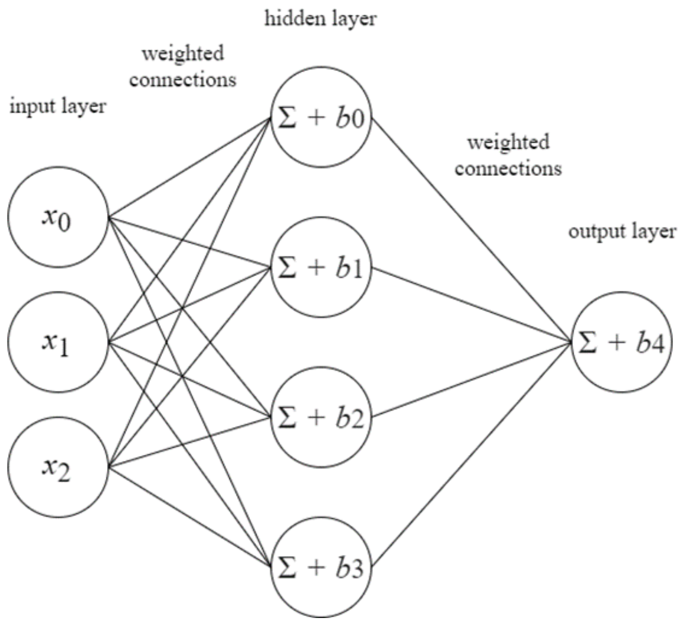
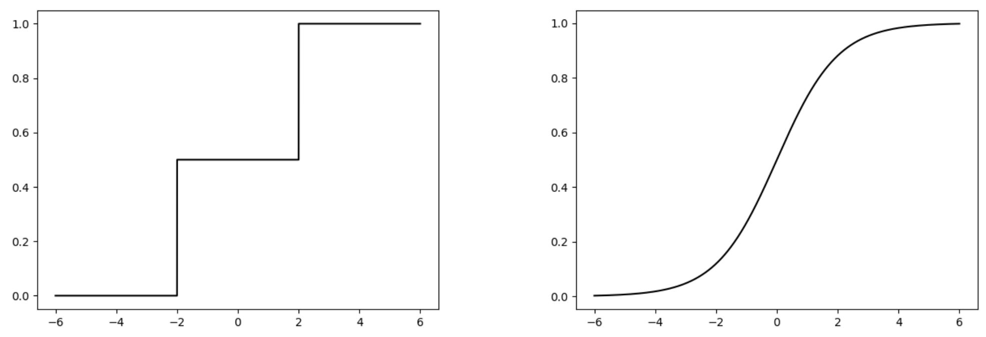


Figure 1: Topology of an example Feed-Forward Artificial Neural Network. It should be noted that the number of hidden layers and the number of nodes in each layer has been chosen arbitrarily.

direction - is significantly less likely to affect the output and will therefore not change the fitness or loss.[1] It is for this reason that a logistic function will be implemented in the network.



(a) $y = t(x)$ (b) $y = \sigma(x)$

Figure 2: Graphs a and b show the activation functions $t(x)$ (see Equation 4) and $\sigma(x)$ (see Equation 5) respectively for values of x between -6 and 6.

$$t(x) = \begin{cases} 0, & x \leq -2 \\ \frac{1}{2}, & -2 < x < 2 \\ 1, & x \geq 2 \end{cases} \tag{4}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

The network can also be represented as a mathematical function, making the way input is manipulated by the weights and biases more clear. For example, the network in Figure 1 can be written as the function of a 3×1 matrix:

[1]The *loss* is a value representing how far removed an output is from the desired output (Rosasco, Vito, Caponnetto, Piana, & Verri, 2004).

$$f\left(\begin{bmatrix}x_0\\x_1\\x_2\end{bmatrix}\right)=\sigma\left(\begin{bmatrix}w_{3,7} & w_{4,7} & w_{5,7} & w_{6,7}\end{bmatrix}\sigma\left(\begin{bmatrix}w_{0,3} & w_{1,3} & w_{2,3}\\w_{0,4} & w_{1,4} & w_{2,4}\\w_{0,5} & w_{1,5} & w_{2,5}\\w_{0,6} & w_{1,6} & w_{2,6}\end{bmatrix}\begin{bmatrix}x_0\\x_1\\x_2\end{bmatrix}+\begin{bmatrix}b_3\\b_4\\b_5\\b_6\end{bmatrix}\right)+\begin{bmatrix}b_7\end{bmatrix}\right) \tag{6}$$

where x_n is input n to the network, w_{n_1,n_2} is the weight from node n_1 to node n_2, b_n is the bias for node n, and $\sigma(x)$ is given by Equation 5. The weights and biases are not parameters to the function, they are instead constants that are determined during the training[2] of the network.

3.2 NeuroEvolution

Commonly used algorithms for training the weights and biases of a network such as *back-propagation* (Rumelhart, Hinton, Williams, et al., 1988) would not be suitable for this application due to the prerequisite of *training data*, i.e., examples of possible input accompanied by examples of what the desired output is given that input. This method works well for problems such as digit recognition (LeCun et al., 1990), however it is not known specifically what the optimal controller input is for any given frame of *Tetris*. Yiyuan Lee (2013), a Computer Science undergraduate at the National University of Singapore, circumvented this problem using a heuristic and intuitive approach whereby variables were specified that the AI should seek to minimize or maximize (e.g. aggregate height, number of complete lines, bumpiness). While this approach worked fantastically for the project, the goal was to produce an intelligence that could indefinitely clear lines without topping out[3] in a generic clone of *Tetris*[4]. This research aims to analyse how an AI plays specifically *NES Tetris* in

[2]Training refers to the process of adjusting the weights and biases of an ANN in order to bring its output for any given input closer to the desired output.

[3] *Topping out* is a phrase used amongst players of *Tetris* to describe the event of the tetrominoes reaching the top of the screen and causing the game to end.

[4]It should be noted that a key difference between the model used by Lee (2013) and *NES Tetris* is that Lee's does not vary in drop speed, meaning that the difficulty does not increase over time.

comparison with how humans play, and the goal for humans is to optimise the number of points scored rather than to clear lines indefinitely. Figure 3 demonstrates visually that disregarding achieving combos - particularly of four (known as a *tetris*) - would not be an optimal strategy for scoring points in *NES Tetris*.

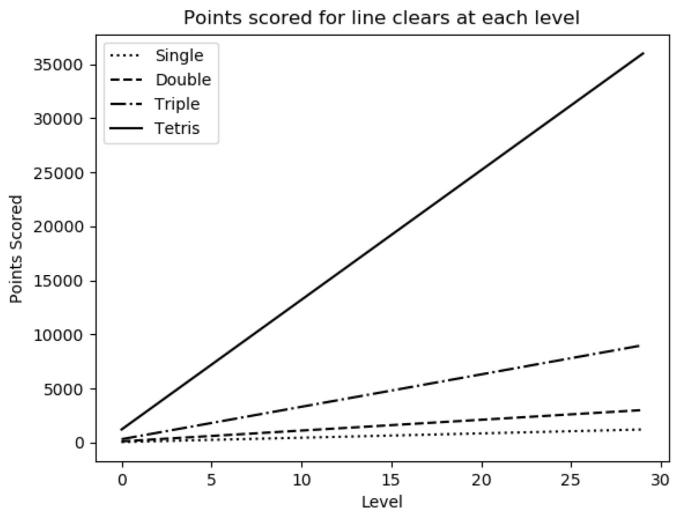


Figure 3: In plotting the data from Table 2, the efficiency gained from scoring points only by tetris becomes more apparent.

An alternative had to be used. A successful method of network training that allows for non-classifiable input is NeuroEvolution. NeuroEvolution, the process of changing a network's weights and biases in a way inspired by Darwinian evolution (Floreano, Dürr, & Mattiussi, 2008), is said to have originated in the 1990s (Hoekstra, 2011) with renowned NeuroEvolution researcher Kenneth O. Stanley, who worked on ground-breaking algorithms such as *ES-HyperNEAT* (Risi & Stanley, 2012), having claimed that the first NeuroEvolution algorithms were made as early as the 1980s (Stanley, 2017).

As for the key idea behind NeuroEvolution, a 2008 overview paper summarizes it quite succinctly:

Instead of using a conventional learning algorithm, the characteristics of neural *networks can be encoded in artificial genomes and evolved according to a performance criterion.* (Floreano et al., 2008).

In the simplest case, this could mean measuring the performance of an agent as the number of points it acquires.

3.3 NeuroEvolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies (NEAT) (Stanley & Miikkulainen, 2002) is the specific model of genetic algorithm that will be implemented. The main reason this model has been chosen is due to the defining feature of NEAT that gives it its name - *augmenting topologies*. In a standard neural network, the topological design is determined beforehand and remains static throughout training. A common problem with this is that the designing of the topology requires prior knowledge of the problem's complexity as well as iteration (Chen et al., 1993). By evolving the topology along with the weights the function is optimised and complexified simultaneously, potentially resulting in more efficient computation (Stanley & Miikkulainen, 2002).

Although augmenting topology as well as weights has been accomplished by other methods (Whitley et al., 1995; Zhang & Muhlenbein, 1993; Angeline, Saunders, & Pollack, 1994; Fullmer & Miikkulainen, 1992), NEAT has been chosen because it addresses problems such as genetic crossover between genomes of differing size and the protection of new, un-optimised, innovations in topology.

4 NEAT Implementation

As it is the original source on the subject, and is very in-depth in the coverage of the workings of NEAT, this section will extensively reference the third section of the NEAT paper by Stanley and Miikkulainen (2002).

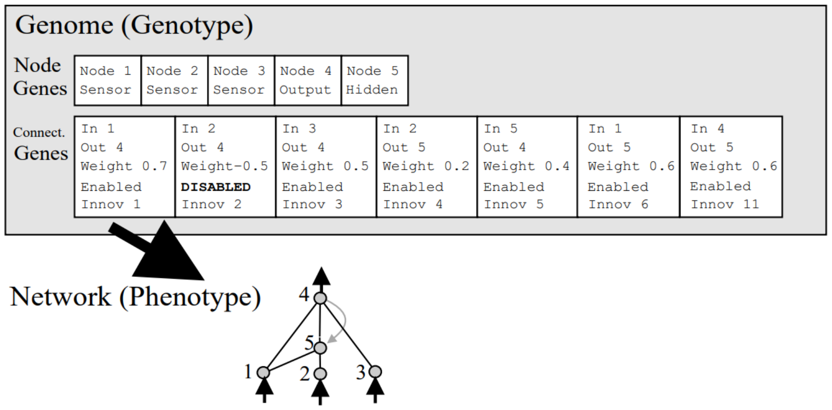

Genome (Genotype)
Node Genes
Node 1 Sensor
Node 2 Sensor
Node 3 Sensor
Node 4 Output
Node 5 Hidden
Connect. Genes
In 1 Out 4 Weight 0.7 Enabled Innov 1
In 2 Out 4 Weight-0.5 DISABLED Innov 2
In 3 Out 4 Weight 0.5 Enabled Innov 3
In 2 Out 5 Weight 0.2 Enabled Innov 4
In 5 Out 4 Weight 0.4 Enabled Innov 5
In 1 Out 5 Weight 0.6 Enabled Innov 6
In 4 Out 5 Weight 0.6 Enabled Innov 11
Network (Phenotype)
4
5
1
2
3

Figure 4: Illustration of the correspondence between the genotype (values) and the phenotype (network topology) of an example genome (Stanley & Miikkulainen, 2002).

4.1 Genome

Harking back to the aforementioned analogy with Darwinian evolution and biology in general, a specific network in regards to NeuroEvolution is referred to as a genome (Floreano et al., 2008). In NEAT, each genome consists of a list of *node genes* and a list of *connection genes* corresponding to the nodes and connections in that genome respectively (Stanley & Miikkulainen, 2002). Each node gene has an ID that is unique to the network and a type which determines which types of nodes can be connected to it, and each connection gene has information that determines what nodes it connects to, what weight it applies, whether it is enabled, and its *innovation number* (Figure 4). This system was implemented in the Tetris model using classes[1] that store the values of the genome and its respective genes as data. The innovation number simply refers to the global index of a particular connection when all evolved connections are ordered by first appearance, which is vital for speciation (Section 4.2) and *genetic crossover*.

Genetic crossover is one of the ways by which evolution occurs, and refers to the process by which genetic information from two *parent* genomes is transferred to a new *child* genome. In NEAT, it is possible to cross genomes of differing sizes due to the existence of innovation

[1]A class refers to a feature of Object Oriented Programming (OOP) that is a template for creating like *objects*.

numbers. The method for doing so is given as follows:

> *Matching* [innovation] *genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent.* (Stanley & Miikkulainen, 2002)

In the Tetris model, this was accomplished by creating an empty genome (the *child*), then iteratively copying each node gene from the more fit parent into it. This works because every node from the more fit parent will end up a part of the child regardless of its presence in the less fit parent, because disjoint and excess genes are always and only inherited from the more fit parent. Secondly, each connection gene in the more fit parent is iterated over and it is determined whether the innovation number from each exists in the less fit parent. If it does, then a random one of the two is selected to be inherited. If it doesn't, then the gene is inherited straight away. This has the desired effect of disregarding any excess or disjoint genes from the less fit parent while preserving those of the more fit parent. (Figure 5)

Also as according to the paper, structural mutation was implemented in two different ways: *add connection* and *add node*. *Add connection* works simply by creating a new connection with random weight between two nodes. *Add node* works by *deactivating* an existing connection and creating a new node where it was, with the connection leading out of it having the weight of the original connection and the connection leading into it having a weight of one. New connections created by this method receive a unique innovation number. Nodes do not require innovation numbers as their genome-specific IDs are sufficient in identifying where connections go.

Finally, mutation of the weights was implemented. The paper provided no specific direction for this, however I used the probabilities given in their pole balancing test (Stanley & Miikkulainen, 2002). A genome had an 80% chance of mutating. If it did, each weight had a 90% chance of having its weight multiplied by some value and a 10% chance of being assigned a completely new value. I chose the mutation multiplier to be a random number

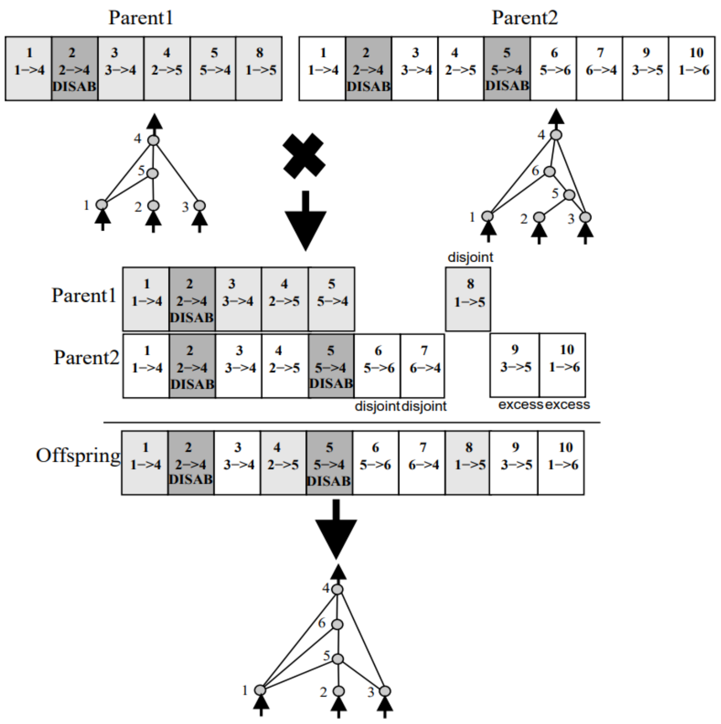


Figure 5: Illustration of crossover of two example genomes and the resultant child genome. In this example, the parent genomes have equal fitness. (Stanley & Miikkulainen, 2002).

between -2 and 2, and similarly the completely new value would also be between -2 and 2.

4.2 Speciation

Speciation, or *niching*, is presented in NEAT as a solution to the side-effect of augmenting topologies; that is genomes with novel topologies are likely to become extinct before the innovation has a chance to be optimised (Stanley & Miikkulainen, 2002). With genomes competing within a niche rather than the whole population, and sharing fitness within the niche, it becomes a lot harder for a particular genome to dominate. Genomes are separated into species when their difference is above a certain threshold. The function for determining the difference between two genomes was derived from the following equation from Stanley and Miikkulainen:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{7}$$

where c_1, c_2, and c_3 are specified coefficients used to weight the importance of each variable on a problem-to-problem basis, N is the size[2] of the largest genome, E and D are the number of excess and disjoint genes respectively, and $\overline{W}$ is the weight difference in matching genes (Stanley & Miikkulainen, 2002).

4.3 Testing

4.3.1 Many Connections Test

To ensure that the network topology was capable of evolution, a test was conducted wherein a population of 100 individuals was simulated with fitness given by the amount of connections

[2]Number of genes in the genome.

in the genome. Parent 1 from Figure 5 was arbitrarily selected as the initial genome from which all others evolved from. The conclusive results of this experiment can be seen in Figure 6.

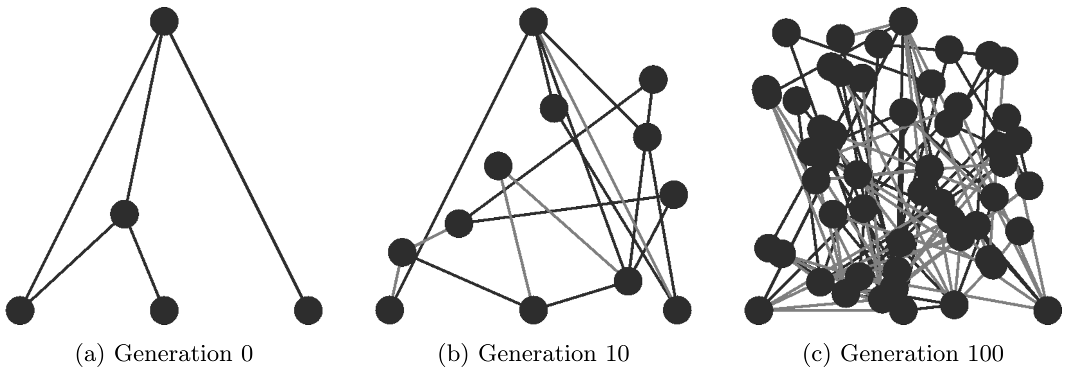


Figure 6: Networks a, b, and c represent the most fit phenotype from generation 0, 10, and 100 respectively from the *many connections* test. The trend toward more connections is obvious, showing that the implementation can successfully evolve topology.

However, it became apparent that there would not be a way to evaluate the output of networks such as the ones evolved in this test due to the fact that there are no measures in place to prevent the creation of connections that result in a neuron's activation being dependent on itself (Figure 7). Fortunately, the solution I found was simple: whenever a connection is added, it must first ensure that either the node being connected to is further from the sensor nodes or the node being connected from is a sensor node itself. The test was repeated, with results that can be seen by Figure 8. This concept of calculating the distance from the sensor nodes was reused to improve the coherency of the renderings of networks by sorting the nodes into layers based on that distance.

4.3.2 XOR Operator

XOR[3] is an operator used in Boolean logic, taking two inputs and receiving one output. XOR is used to test that non-linearity can be evolved by the model as XOR cannot be

[3]XOR outputs a signal when only one of its inputs is on.

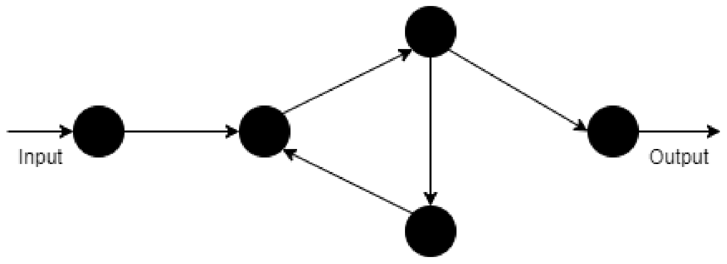


Figure 7: An example of a recursive network. In this network, each of the three hidden nodes are dependent on themselves; resulting in a network that can't be evaluated.

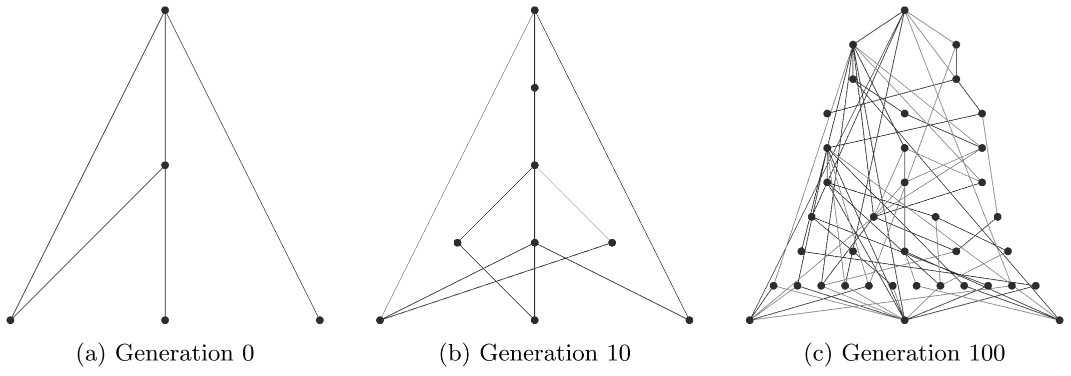
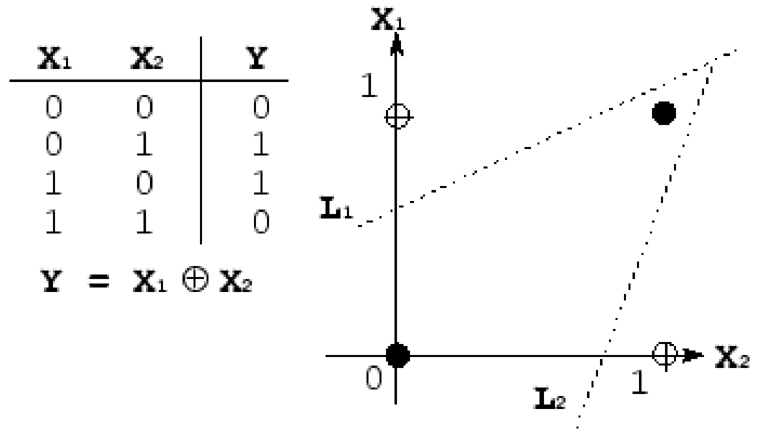


Figure 8: Networks a, b, and c represent the most fit phenotype from generation 0, 10, and 100 respectively from the second *many connections* test. The networks produced by this test are possible to evaluate due to the lack of recursion. These networks had fewer connections than in the first test because of a lowered chance of mutation.

computed by linear operations alone (Rashwan, Sadek, Al Ez, Hessian, et al., 2017). This is demonstrated graphically by Figure 9.

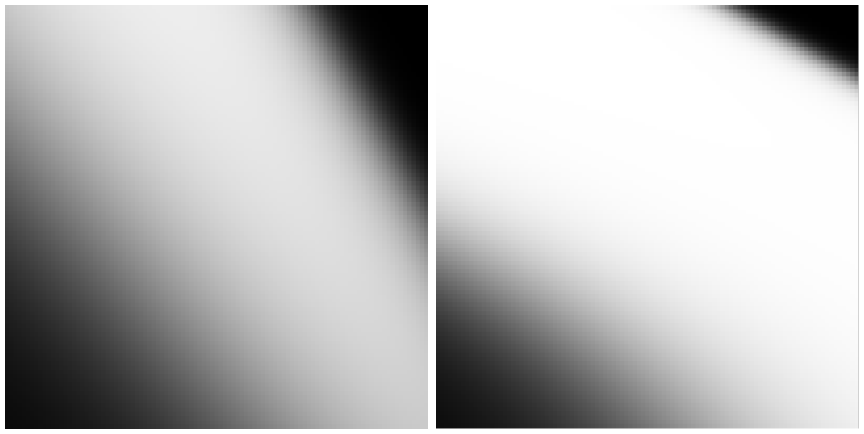
Upon attempting to train XOR on my network, it took far too long to train. This was fixed by scaling the sigmoid activation function used at each node, and the working results were rendered as a graphic texture seen in Figure 10.



X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

$Y = X_1 \oplus X_2$

Figure 9: The figure shows the output of the XOR operator and how it cannot be separated by a single line. This demonstrates why non-linearity is needed to compute it. From Rashwan et. al (2017)



(a) Halfway through training

(b) End of training

Figure 10: The figure demonstrates the success of the NEAT implementation at computing XOR. Each axis of the graph represents a gradient from 0 to 1, from left to right and from bottom to top. A lighter pixel represents a more activated output.

5 Tetris Training

5.1 Integration

Integrating the NEAT system with the Tetris engine was simple due to earlier decisions. The engine runs separately to whatever class controls it, so in order for it to be controlled by the AI a class was created for the Tetris agent that uses a neural network produced by the NEAT system. Every frame the agent class requests the current game state from the engine class, and determines the input that should be made. After brief tests with totally random starting topologies, I decided on the layout Figure 11 depicts.

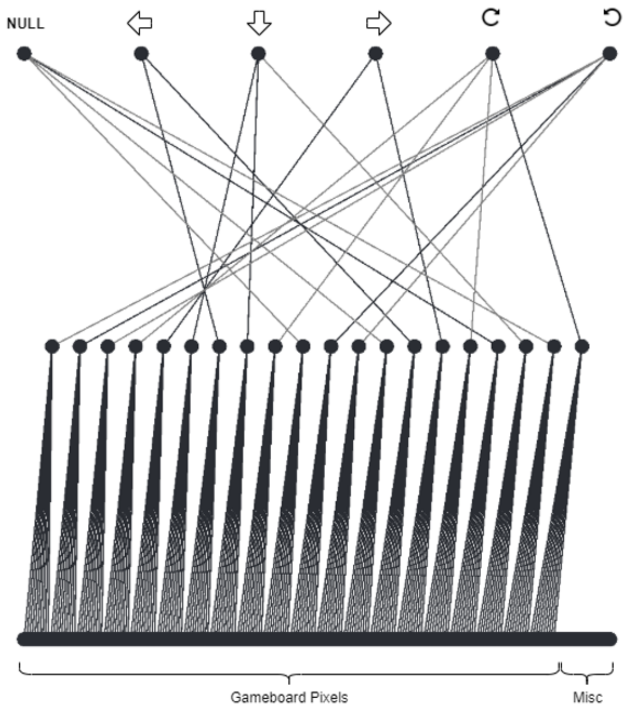


Figure 11: The initial topology used for training the neural network. The input nodes are at the bottom, with output nodes at the top. The output nodes represent all of the buttons on a typical NES controller, sparing the "Up" button, and "Start" and "Select" buttons as they are not required for playing Tetris. In addition, there is an output node which denotes that the network has chosen to take no action. There are input nodes for each pixel on the game board, as well as additional information that a player would receive: a node indicating the next tetromino and two nodes for the falling tetromino's position on the board. There is also a node for the DAS counter, as the network otherwise has no way to experience change in time. Finally, there is a bias node that always has its value at 1.

As seen in the figure, the network begins already having a hidden layer and many connections. I elected to do this in order to provide the network with an initial, very basic sense of the board's geometry. There are twenty nodes in the hidden layer, and each of them has a positive connection to all of the input nodes representing a particular vertical layer of the board. In this way, the network begins with a node for each layer. Once this setup has been generated, a single randomly weighted connection is created between each node in the hidden layer and a randomly selected output node to provide the network with a basic function to build upon.

5.2 Result

The network was trained for 72 hours of real-world time, producing one thousand generations in total. Each generation had three hundred individuals separated into ten different species on average. The best performing neural network from each generation was saved as a binary file, and the software was made such that these binary files can be loaded to be used at a later date for examination or display. In addition, the software can render the phenotypes of the genomes stored in the binary files as a network. The fitness of each genome was the amount of points it scored in its run of Tetris.

The result of this training was very unsuccessful in terms of point scoring, but there are interesting things to note from it. The first is that the fitness function may have been set too naively, as the AI learned very quickly that it can score points by holding down the down button constantly. It is clear for a human to see that this is not optimal, as the player will never clear any lines by this method and be limited to an abysmal number of points; never exceeding even 150. For the AI, however, this strategy is far greater than doing random actions and so mutations that cause the agent to deviate will perform far worse on average to the extent that they will not be protected by the speciation system. At the end of the 1000 generations, this was still the best performing strategy that the AI had learned.

Another interesting point of note was that the AI learned to distinguish between different kinds of tetrominoes. At around generation 250, I personally witnessed the active agent moving L and J tetrominoes to the far right, and and dropping all other tetrominoes immediately. Unfortunately, this behaviour was not more successful than dropping every tetromino and as such it was lost in future generations. I had hopes that if this behaviour were developed, the spreading out of tetrominoes could eventually lead to a line clear and then the behaviour could be refined further. Although this event was live-streamed, all evidence of it has been lost.

The final thing of note was that although the topology of the networks became more complex (Figure 12), the behaviour displayed by the agents did not in a proportional way.

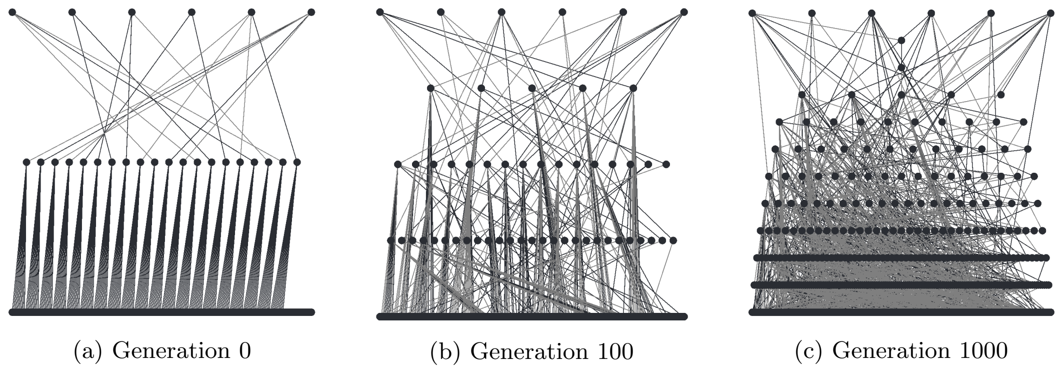

(a) Generation 0 (b) Generation 100 (c) Generation 1000

Figure 12: Figures a, b, and c show the evolution of the network topology of the Tetris AI.

6 Conclusions and Future Work

The problem of the AI getting stuck on only pressing the down button could potentially be solved in a number of ways. The first is to produce a fitness function that is more reflective of the success of the agent than simply its score. This would mean taking a heuristic approach, and such an approach was experimented with briefly after the main training by taking the total number of tetrominoes that have been instantiated into account for the fitness. This may be a good approach as it provides some way to measure how long the agent has survived, and is better than using the real time because that would unnecessarily disadvantage playing in a fast-paced way. With this solution, the fitness function may look something like $c_1S + c_2T$ where c represents some constant and S and T represent the score and number of dropped tetrominoes, respectively. Another solution considered for this problem would be to refine the speciation system such that individuals that deviated from the strategy of holding the down button weren't punished so harshly. In general it was very difficult to optimise this system and so refining it further could make a large difference to the performance of the AI, especially if put in conjunction with the other proposed solution.

It is my suspicion that another major cause of the lack of success seen in the model was the AI's failure to understand the geometry of the game board. This was due to the fact that it received the board as 200 individual points with no inherent relation to one another, and had evolve to understand their relation as a natural side effect of increasing fitness. As one could imagine this task would be difficult to do with the model as it is. I attempted to compensate for this by initialising the genome with a hidden layer for each row, but it was not enough to be effective. A possible solution to this problem comes from another paper by Kenneth O. Stanley, with the method *HyperNEAT* (Stanley, D'Ambrosio, & Gauci, 2009). This model was developed with the intention of helping to bridge the gap between artificial and natural neural networks by encoding information in geometry to ”produce connectivity patterns with symmetries and repeating motifs”. The nodes of the network are placed in a substrate which has geometrical properties that can vary depending on the problem (Figure 13).

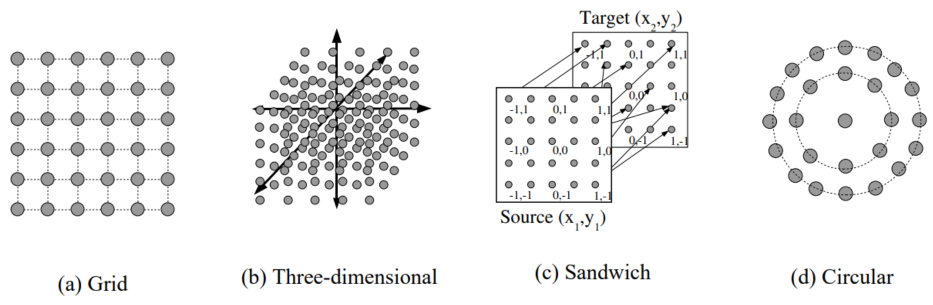


Figure 13: From Stanley et. al. (2009). The figures show some possible substrate configurations that could be suited to problems with different geometric properties.

To conclude, the answer to whether an AI could teach itself to play Tetris with similar strategies to humans was not definitively answered. However, results such as the computer learning to recognise different tetrominoes and even developing a (relatively poor) strategy to earn points could suggest that it is a possibility with the failures of this experiment considered.

Acknowledgement

I would like to thank my supervisor, Dr. Dolton, for providing me with incredible advice and support. Only through this was I able to complete the paper to a standard that I am very proud to present.

References

Angeline, P. J., Saunders, G. M., & Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, *5*(1), 54–65.

Arcorann, & Simonlc. (2019). *Tetris.wiki: Scoring.* Retrieved from `https://tetris.wiki/Scoring` (Accessed: 2019-10-22)

Chen, D., Giles, C., Sun, G., Chen, H., Lee, Y., & Goudreau, M. (1993). Constructive learning of recurrent neural networks. In *Ieee international conference on neural networks* (pp. 1196–1201).

Edo. (2010). *Tetris.wiki: Drop.* Retrieved from `https://tetris.wiki/Drop` (Accessed: 2019-10-27)

Floreano, D., Dürr, P., & Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary intelligence*, *1*(1), 47–62.

Fullmer, B., & Miikkulainen, R. (1992). Using marker-based genetic encoding of neural networks to evolve finite-state behaviour. In *Toward a practice of autonomous systems: Proceedings of the first european conference on artificial life* (pp. 255–262).

Haykin, S. (1994). *Neural networks: a comprehensive foundation.* Prentice Hall PTR.

Hebden, P. N. (2020). *Piturnah/tetris-ann-epq: Epq release.* Zenodo. Retrieved from `https://doi.org/10.5281/zenodo.3732039` doi: 10.5281/zenodo.3732039

Hendrickson, S. (2015). *Mari/o - machine learning for video games.* Retrieved from `https://www.youtube.com/watch?v=qv6UVOQ0F44` (Video File)

Hoekstra, V. (2011). An overview of neuroevolution techniques.

Jjdb210, Tepples, Nicholas, Colour thief, DIGITAL, Ghett0, ... MatMayuga (2018). *Hard Drop: Tetris (NES, Nintendo).* Retrieved from `http://harddrop.com/wiki/Tetris_(NES,_Nintendo)` (Accessed: 2019-10-09)

Kasabov, N. K. (1996). *Foundations of neural networks, fuzzy systems, and knowledge engineering.* Marcel Alencar.

Kitaru. (2010). *Nes tetris: Delayed autoshift technical demonstration.* Retrieved from https://www.youtube.com/watch?v=Aew60kov0gw (Video File)

Kitaru, Sval, Arcorann, & Simonlc. (2019). *Tetris.wiki: Tetris (NES, Nintendo).* Retrieved from https://tetris.wiki/Tetris_(NES,_Nintendo) (Accessed: 2019-10-08)

Lague, S. (2018). *Neural networks (e01: introduction).* Retrieved from https://www.youtube.com/watch?v=bVQUSndDllU (Video File)

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., & Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems* (pp. 396–404).

Lee, Y. (2013). *Tetris ai – the (near) perfect bot.* Retrieved from https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

Mothrayas, Ilari, Cyorter, & brunovalads. (2018). *TASVideos: Platform Framerates.* Retrieved from http://tasvideos.org/PlatformFramerates.html (Accessed: 2019-10-08)

Nielsen, M. A. (2015). *Neural networks and deep learning* (Vol. 25). Determination press San Francisco, CA, USA.

Rashwan, A., Sadek, M., Al Ez, A., Hessian, R., et al. (2017). Computational intelligent algorithms for arabic speech recognition. *Journal of Al-Azhar University Engineering Sector*, *12*(44), 886–893.

Risi, S., & Stanley, K. O. (2012). An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial life*, *18*(4), 331–363.

Rosasco, L., Vito, E. D., Caponnetto, A., Piana, M., & Verri, A. (2004). Are loss functions all the same? *Neural Computation*, *16*(5), 1063–1076.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Sanderson, G. (2017). *But what is a neural network? — deep learning, chapter 1.* Retrieved from https://www.youtube.com/watch?v=aircAruvnKk (Video File)

Simonlc. (2019). *Tetris.wiki: Nintendo Rotation System.* Retrieved from https://tetris.wiki/Nintendo_Rotation_System (Accessed: 2019-10-09)

Stanley, K. O. (2017). *Neuroevolution: A different kind of deep learning.* Retrieved from https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/

Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, *15*(2), 185–212.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*(2), 99–127.

Svozil, D., Kvasnicka, V., & Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, *39*(1), 43–62.

TASVideos. (2019). *Bizhawk.* Retrieved from https://github.com/TASVideos/BizHawk (Accessed: 2019-12-17)

Tepples, DIGITAL, Nicholas, Jason.tetris, & Tokihiko. (2019). *Tetris Wiki: DAS.* Retrieved from https://tetris.fandom.com/wiki/DAS (Accessed: 2019-10-14)

Tepples, Nicholas, DIGITAL, Colour thief, LFC, R., CaptainBlue777, ... Tokihiko (2019). *Tetris Wiki: Scoring.* Retrieved from https://tetris.fandom.com/wiki/Scoring (Accessed: 2019-10-22)

Trinklein, J. (2015). *Snes controller emulator using arduino.* Retrieved from https://github.com/jtrinklein/SConE (Accessed: 2019-11-30)

u/Poppinfreshzero. (2019). *Reddit: NES 1989 Tetris - How many points do you get for soft dropping? Misconceptions.* Retrieved from https://www.reddit.com/r/Tetris/comments/b5u8li/nes_1989_tetris_how_many_points_do_you_get_for/ (Accessed: 2019-11-01)

u/x34l. (2019). *Reddit: NES 1989 Tetris - How many points do you get for soft dropping? Misconceptions.* Retrieved from https://www.reddit.com/r/Tetris/comments/b5u8li/nes_1989_tetris_how_many_points_do_you_get_for/ (Accessed: 2019-11-01)

Verhulst, P. (1845). La loi d'accroissement de la population. *Nouv. Mem. Acad. Roy. Soc. Belle-lettr. Bruxelles*, *18*(1).

Whitley, D., et al. (1995). Genetic algorithms and neural networks. *Genetic algorithms in engineering and computer science*, *3*, 203–216.

World of Longplays. (2014). *Nes longplay [483] tetris.* Retrieved from `https://www.youtube.com/watch?v=-FAzHyXZPm0` (Video File)

Zhang, B.-T., & Muhlenbein, H. (1993). Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex systems*, *7*(3), 199–220.

A Using the software produced for this research

A.1 Acquisition

The software is available in two forms. At the time of this writing, a *WebGL* build of the software is available to use at the author's personal website (https://piturnah.xyz/tetris-epq/). This version is not likely to be up to date but will include the main functionality. With this build, it is possible to play the Tetris model as a human starting at any level from 00 to 19 and possible to train the AI with the Tetris model. The training may cause issues if used for an extended period as the software will attempt to store binary genome files as cookies, so it is not recommended. The option to load a genome file into the web build is currently not supported, and as such the middle button on the title screen has no functionality. In general, this version of the software is only recommended as a quick way to test the Tetris model as a human player.

The second form is by using the source code within the Unity Engine. The source code can be found from two sources; the author's GitHub page (https://github.com/Piturnah/tetris-ann-epq) and by its DOI (Digital Object Identifier). The DOI can be found in the References section of the essay (Hebden, 2020). The DOI refers to the version that the software was at by the time of submission of the essay, therefore future versions may only be found at the GitHub link. Therefore, it is recommended that most readers to use the GitHub page for the most up-to-date version whereas readers who seek specifically the snapshot of the software at the time of submission use the DOI.

A.2 Usage

This section pertains to using the DOI version of the software within the Unity Engine. To avoid compatibility issues, it is recommended that the software be used with version 2019.3.4f1 of the Unity Engine. The three main options of the software (playing as a human, displaying a neural network playing, evolving the neural network's genome) are available from buttons after pressing the play button with the *Menu* scene open. When choosing the display option, a prompt will appear to select a genome file. The files required have the extension *tetro* and are generated by the software when it writes a genome to long term storage. The training button simply begins training with the initial genome as described in the paper. What you see on screen is two individuals from each generation, from the total 300. This is because the 300 are batched into two groups of 150 to improve performance.

To render the phenotype of a genome from a tetro file, the process is more involved. First, the user must switch to the *NEAT_Testing* scene. By default, this will attempt to render ten of the 1003 tetro files that were saved from the training process described in the paper. As these will not be on your drive, it will cause errors. To render a particular tetro file, the user must enter the script *Assets/Scripts/NEAT/NEATTester.cs* and remove the code from line 73 to line 75. In its place, the user must write the following:

*renderer.DrawGenome(TetrisNEAT.OpenGenome("FILE_NAME.tetro", false), Vector3.zero, Vector3.one * scale);*

Replacing *FILE_NAME* with the file path of the desired tetro file.

I will not go into detail on how to use other functionality of the software such as using the TextureFromANN class, as they were only created to aid with the development of the software. If the user should require assistance in utilising these additional features, the author's email is on the front of the paper.