OPTIMISING TETRIS GAMEPLAY WITH A GENETIC ALGORITHM

Ilan Iwumbwe

THE NATIONAL MATHEMATICS AND SCIENCE COLLEGE

Terms and Definitions:

Below are terms that are used in this documentation some directly from Tetris nomenclature. Other technical terms and terms used to describe certain aspects of this implementation will be explained when they are introduced

- Skew pieces: These are special names sometimes used to identify the S and Z pieces. The S piece is sometimes known as the 'right skew' the Z piece is sometimes known as the 'left skew'.
- Hard drop: When a piece is dropped instantly from its current position to a possible final position on the terrain
- Terrain: The set of filled cells in the grid
- Computational complexity:
    - P problems (Polynomial time): A set of problems whose solution can be found in a reasonable amount of time by a computer. Examples: multiplication and sorting
    - NP problems (Non-Deterministic Polynomial time): A set of problems that take a long time to solve but whose solution can be checked in a reasonable amount of time. Examples: the travelling salesman problem, the circuit satisfiability problem, protein folding
    - A lot of NP problems are essentially the same core problem. There are the NP-complete problems. A fast program to solve a problem in the NP-complete class could be used to solve any other problem in the NP class.

A: ANALYSIS

A.1: BACKGROUND

The Genesis of Tetris:

Tetris was first made by Alexey Pajitnov in the summer of 1984 at the Soviet Academy of Sciences at their Computer Centre in Moscow. Being a puzzle enthusiast, he decided to implement his favourite puzzle, Pentominoes using Pascal on the Electronika 60. His friend Vadim Gerasimov then ported it to the IBM PC, where it quickly made its way out of the Computer Centre. By this time, it had evolved into Tetrominoes, and involved a well, where the pieces needed to be placed in a specific way. The game has now become a cultural juggernaut present in the memories of many adults and children alike.

Tetris Curiosities:

Tetris represents both a challenging and interesting problem from a mathematical and computer science point of view. From a mathematical standpoint, one could ask whether there's an optimal set of moves that will always guarantee success for any player, or whether there are certain sequences that will inevitably lead to a loss.

Tetrominoes are dealt randomly, and a certain sequence of pieces can be considered, that are easier to place than others. For example, the skew pieces do not fit well with each other, and hence if dealt in the sequence (S, Z, S, Z, S, Z, S, Z, ...) would lead to a loss however strategically they are placed. This is because the best strategy is to create lanes of S and Z blocks stacked upon each other. However, as the dimensions of a standard Tetris board are 10x20, and each lane would take up 2

columns, only 5 lanes can be fit. As 5 is odd, there will be either more of the S lane than Z lane (3 S, 2 Z) and vice versa. Even though the pieces will be the same in number, they are distributed differently, so the double lane will fill up faster, causing a loss. The only way to prevent this would be to place the piece in a wrong lane, but that creates more holes, which also lead one closer to a loss. It has been proven that a game can be lost after at most 69,600 of these blocks. If pieces are generated randomly, then given an infinite number of games, any possible sequence of pieces will eventually be dealt, such as the one above, although it would certainly take a lot of time.

Tetris is an NP-complete problem with high complexity and an enormous configuration space. A standard Tetris board contains 10×20 cells, which is 200 cells, each of which can either be filled, or not filled. This means that there are 2^{200} possible board configurations. Every piece that is played adds 4 filled cells, and every line that is cleared removes 10 filled cells. This means that the number of filled cells are always even, which means that the possible board configurations can be halved, to give 2^{199} possible states. The goal of this project is to use a set of heuristics that can effectively describe certain states and use a neural network to determine which cells should be filled in order to maximise the number of lines cleared from that state.

Project Summary

The goal of this project is to combine the complexity of Tetris with the power of neural networks to produce a machine learning agent that can play the game effectively. The project will borrow inspiration from research into how this has been achieved through other methods such as Monte Carlo Tree Search, to create a new approach to solving Tetris with a genetic algorithm, which will be inspired by readily available implementations such as NEAT-python, and other online resources.

Finally, the project aims to explain in detail how all the pieces work together to make it work using foundational principles in machine learning and optimisation, in a bid to give the reader a baseline understanding of the concepts used, and an appreciation for the power the techniques used hold.

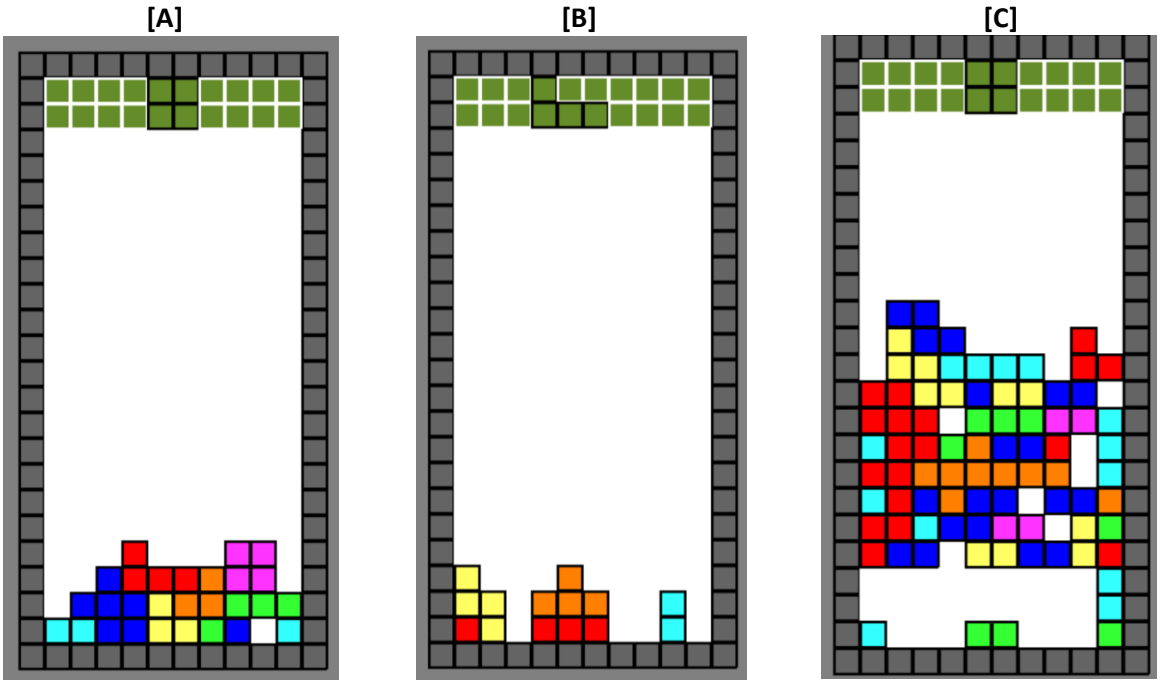
A.2: RESEARCH INTO PROJECT

Solving Tetris as an Optimisation problem

Tetris can be represented as an optimisation problem, where we try to optimise a function that maximises the net reward of an agent. We can then try to work out what the inputs and outputs of this function need to be, and this would define this function. The inputs to this function (features of a state) give the agent an understanding of the game state, and therefore how to gain the maximum possible reward from it. Through training, we can find an optimal function, which when given a set of new features can output a score that accurately describes how optimal the action that produced that state was. These features include: the number of holes in the board, the cumulative sum of the height per column, the bumpiness of the terrain, the number of lines that would be cleared, the number of row transitions, the number of column transitions, the standard deviation of the column heights, number of pits, deepest well, to mention but a few.

1. Number of holes

These need to be minimised because the more holes in your terrain, the faster the board fills up, and therefore, you are more likely to lose a game. Moreover, having holes in the terrain lowers the chance of clearing lines and getting tetrises. A *hole* is defined as an unoccupied space in the terrain that cannot be reached by any piece from above.



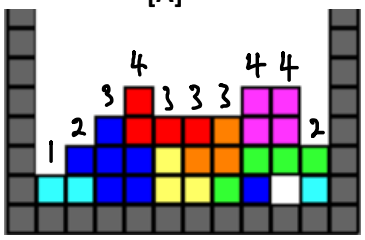
[A] :There's 1 hole in this state
[B]: There's no holes in this state
[C]: There are multiple holes in this state. The AI needs to learn how to minimise those holes to avoid reaching such a state

2. Sum of heights

The column height is defined as the number of occupied and unoccupied spaces in the column before the highest point in that column

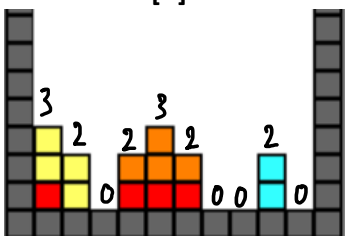
This feature gives us information about how high the terrain is. Generally, we would want to minimise the height, as having a high terrain increases the chances of losing the game. However, if we play strategically, we can aim to make the height as high as possible, leaving an empty column for the "I" piece. This strategy has been known by human players for a long time, and it will be interesting to see whether the AI learns to use this strategy. We can conclude that this feature doesn't really have a linear relationship with the game progression, which could make it harder to optimise with linear regression models. This among other reasons that will be discussed later justify the use of a *neural network* model, which is a powerful function approximator for non-linear functions. To solidify this idea, consider the situation in *[A].* The terrain isn't that high, so it is possible to stack some pieces to make it higher and leave a free column in which to slot the "I" piece, which would multiply the game score. Now consider the situation in *[C]*. The agent needs to clear lines quickly, or else the game is lost. This implies that we do not *always* want to maximise or minimise the height, the AI needs to learn *approximately when* should these features be maximised or minimised.

[A]



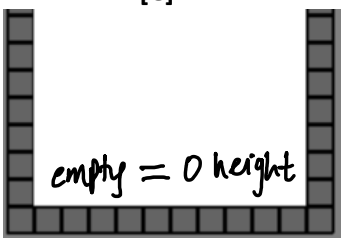
Total Height = 29

[B]



Total Height = 14

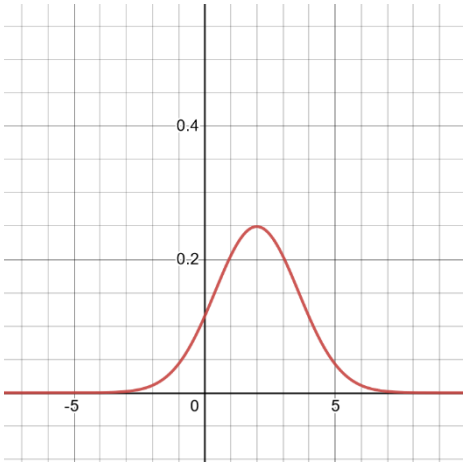
[C]



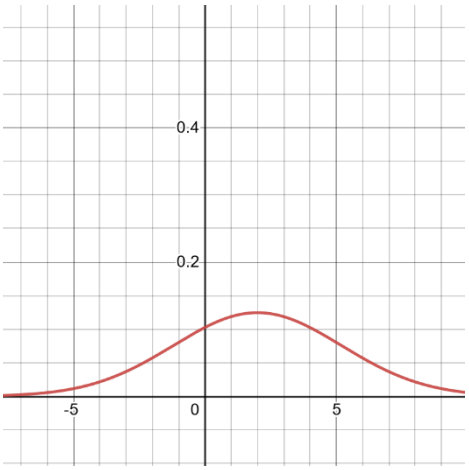
Total Height = 0

3. Standard deviation of heights

Standard is an idea from statistics that denotes the amount by which data is spread out, or in other words, measures the dispersion of the values in a distribution. The mean is normally denoted by the symbol μ, and the standard deviation is normally denoted by the symbol σ.



μ = *2*, σ = 1.6



μ = *2*, σ = 3.2

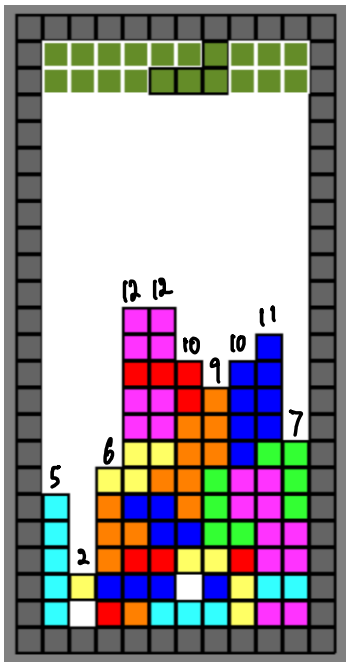
Given a list of N values, each value represented by the variable x, in this case the heights of each column, the standard deviation statistically describes their variation, where:

$$\sigma = \sqrt{\frac{\Sigma x^2}{N} - \left(\frac{\Sigma x}{N}\right)^2}$$

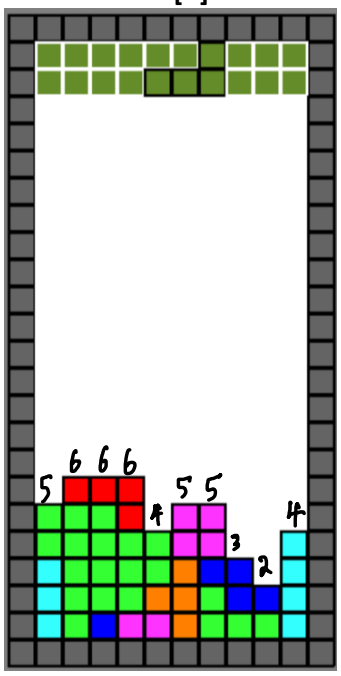
4. Bumpiness

Bumpiness is defined as the variation in column heights in a terrain. The higher the variation, the "bumpier" the terrain is. Bumpiness needs to be minimised because clearing lines in a bumpy terrain is harder. This is because bumpy terrains usually have *"wells",* which if covered, destroy the opportunity to clear all the lines that that well spans.

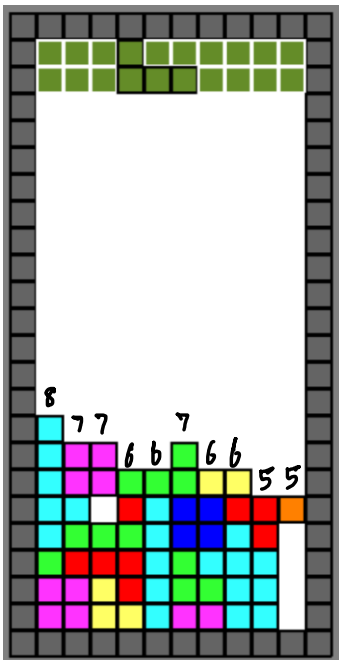
[A]



[B]



[C]



[A]: The 2nd column requires an "I" piece for the maximum decrease in height.
[B]: All columns have low height variation, so it's easier to clear lines and decrease height faster
[C]: If the well gets closed off, there's no way to clear the 4 lines anymore

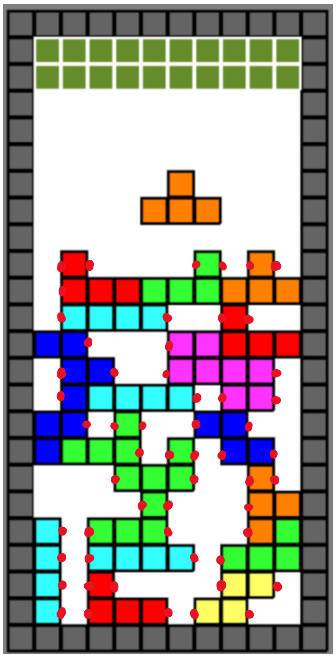
Bumpiness is calculated by taking the sum of the absolute difference between adjacent columns.
The bumpiness of [A] would be
$|5-2|+|2-6|+|6-12|+|12-12|+|12-10|+|10-9|+|9-10|+|10-11|+|11-7| = 22.$
The bumpiness of [B] would be
$|5-6|+|6-6|+|6-6|+|6-4|+|4-5|+|5-5|+|5-3|+|3-2|+|2-4| = 9$

The bumpiness of [C] would be
$|8-7|+|7-7|+|7-6|+|6-6|+|6-7|+|7-6|+|6-6|+|6-5|+|5-5| = 5.$

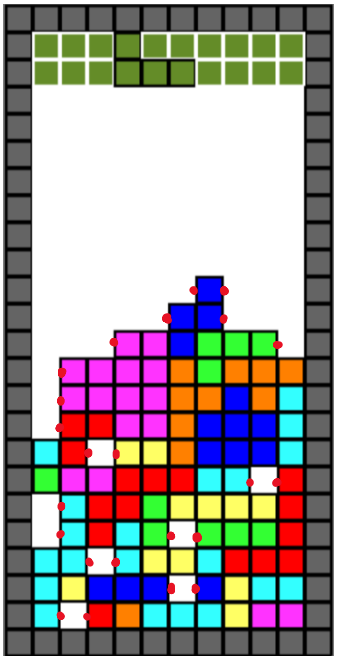
We can now conclude quantitatively that terrain [C] is smoother than [A] or [B], and it is therefore easiest to clear lines in terrain [C].

Bumpiness along with column transitions and row transitions features give an idea of the roughness of the terrain.

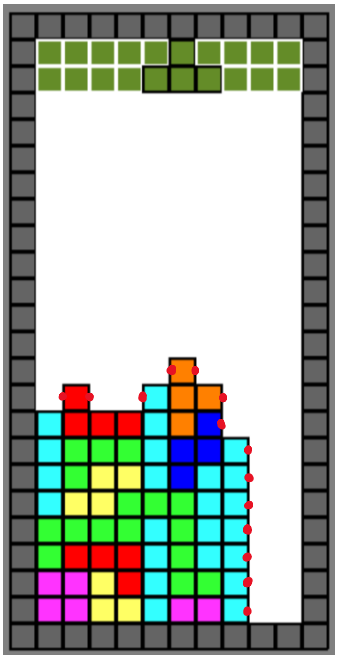
5. <u>Row Transitions</u>
For each row, this is the number of times there's a transition between occupied and unoccupied blocks. This feature gives an idea of the solidity of the terrain; a more solid terrain is a better one as it makes it more likely to clear lines.



Low Solidity



Moderate Solidity

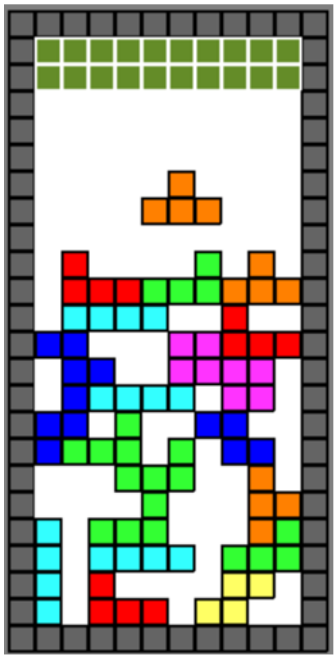


High solidity

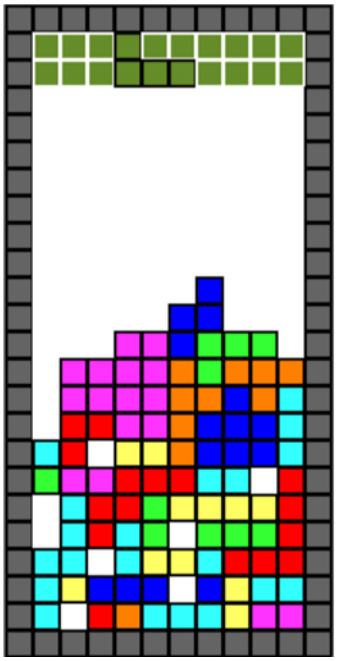
The first state has 56 row transitions, the second has 23, and the third has 14. Note that these features are brought together when computing how good a move is, so although the 2nd and 3rd may have quite similar row transitions, the 3rd is clearly better as there are no holes in it

6. Column Transitions

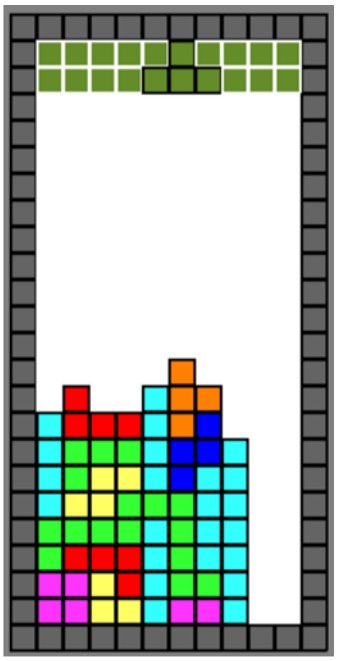
Like row transitions, these are the number of transitions from occupied to uncopied blocks and vice versa in each column in the board.



Low Solidity



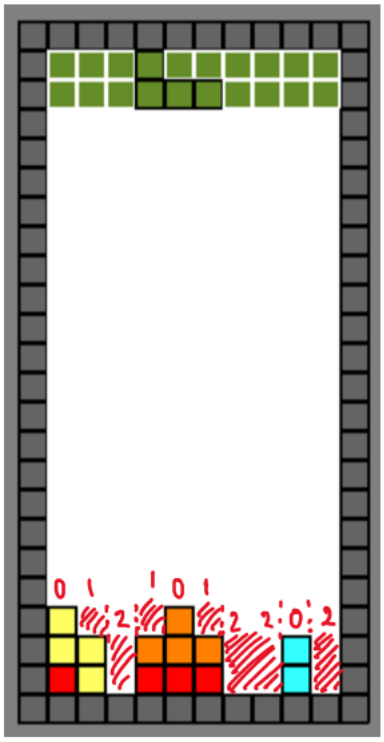
Moderate Solidity

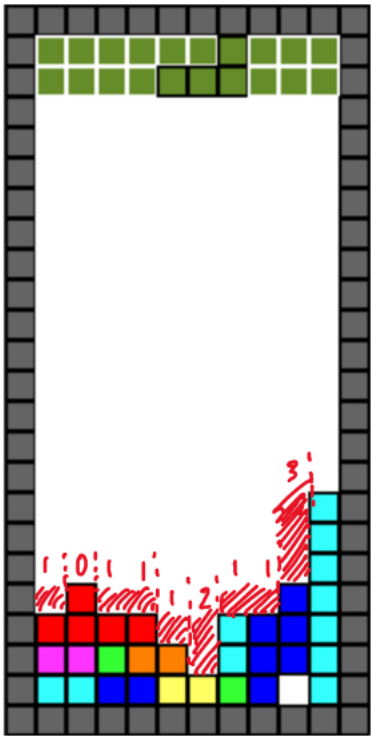


High Solidity

7. Deepest Well

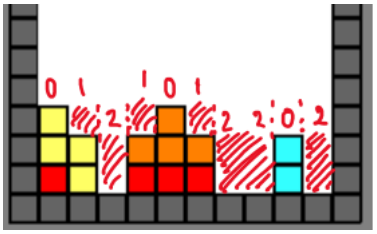
A well is defined as an area in the Tetris board surrounded by blocks of Tetris pieces. A feature of the Tetris board is the deepest of such wells.





The algorithm for computing this is a bit more involved so it will be explained below.

Consider:



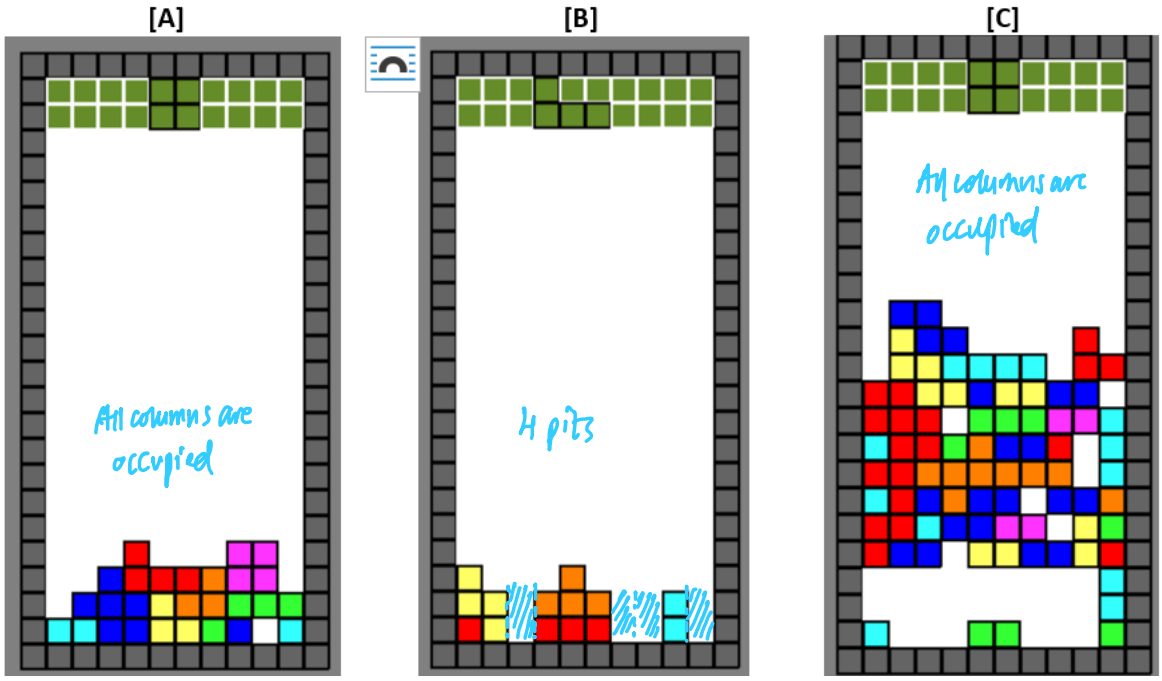
Algorithm:

1. Search each column in the board
2. Considering the extreme cases; the first and last row,
if the current column is the first column, subtract its height from the height of the next column
if the result is positive, then it is the well depth of the first column
else, let the depth of the first column be 0
if the current column is the last column, subtract its height from the height of the second last column
if the result is positive, then it is the well depth of the last column
else, let the depth of the last column be

```
3. Considering all other columns in between, for each column we search, we need to search to the left and to the
right of that column
    Given a column,
    Subtract its height from the height of the column to its left.
        if the result is positive, set it as a possible depth, from the left of the current column
            else, set the possible depth to the left of the current column to 0
    Subtract its height from the height of the column to its right.
        if the result is positive, set it as a possible depth, from the right of the current column
            else, ser the possible depth to the right of the current column to 0

    Finally, compare the two possible depths,
    if possible depth from the left > the one from the right, depth of current column = possible depth from left
    if possible depth from the right > the one from the left, depth of current column = possible depth from right

4. Output the maximum depth from the depth of each column.
```

8. Number of pits

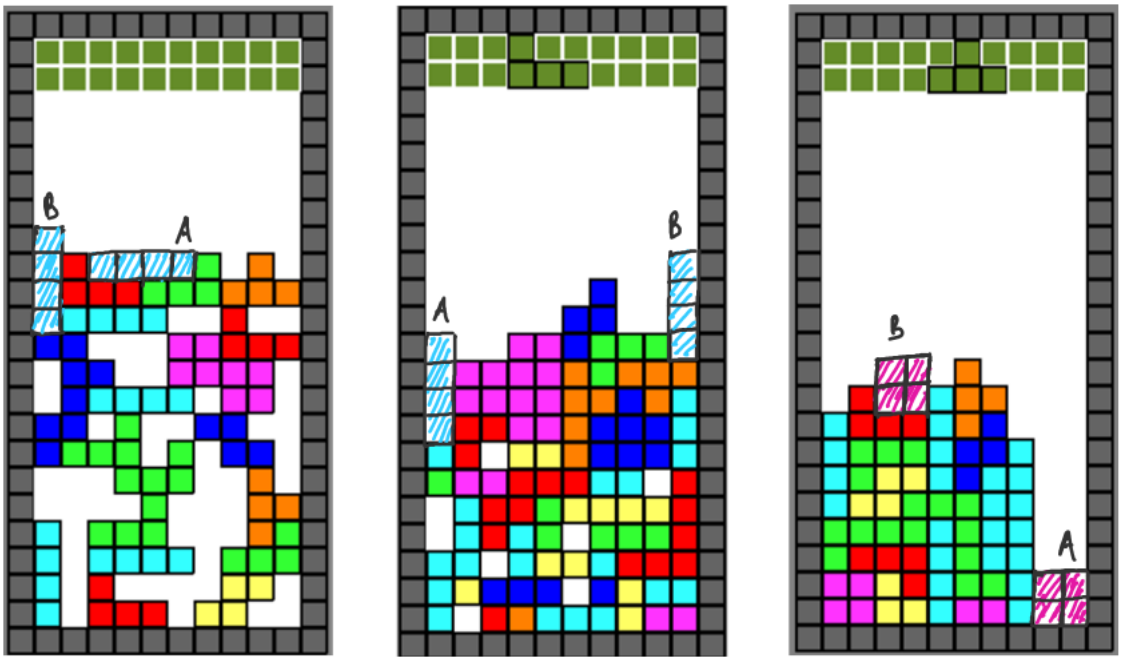
A pit is defined as a column that has no occupying blocks in it.



A has 0 pits
B has 4 pits
C has 0 pits

9. Number of lines cleared

This feature describes the number of lines that would be cleared after a certain move is made. This feature needs to be maximised, as it reduces the height of the terrain, and hence minimises the probability of losing the game and multiplies the score, with the multiplier increasing with the number of lines that are cleared at once.



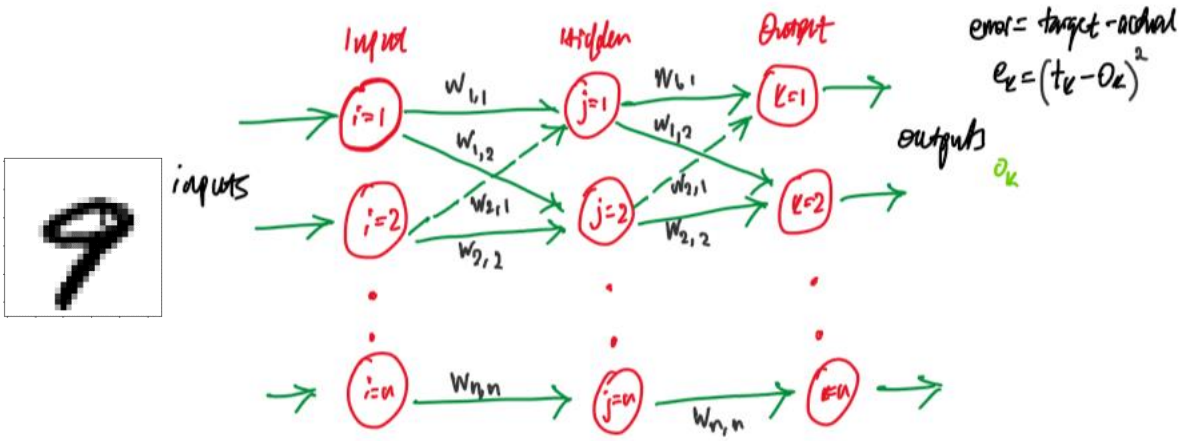
1st grid:
If the piece is placed in position A, 0 lines are cleared, while if it is placed in position B, 1 line is cleared

2nd grid:
If the piece is placed at position A, 3 lines are cleared, while if it is placed in position B, 0 lines are cleared

3rd grid:
If the piece is placed at position A, 2 lines are cleared, while if it is places in position B, 0 lines are cleared

Neural Network:

The idea of neural networks is inspired by biology, specifically the neurons in the brain. The most basic neural network is a *perceptron,* which has a series of integer inputs, and one output. The idea of modelling human decision-making processes with *weights* was introduced in the 1950s by Francis Rosenblatt, and it still carries on today, even with more complex techniques such as deep learning. These weights put an importance on each of the perceptron's inputs, and therefore they influence the output in different ways. The output of the perceptron can be a 0 or 1 depending on some threshold. These perceptrons can be used to make more complex structures that can make more complex decisions.



- *Note that the neurons above **do not** have multiple outputs, but rather a single output split as input to multiple other neurons*
- *There are 784 input neurons as the image is a 28 by 28 greyscale, so there are 784 pixels*
- *There are 10 output neurons for each of the numbers 0 to 9*
- *The weights are represented as matrices. For example, given there are 784 nodes in the input layer and 100 nodes in one hidden layer, the dimensions of the matrix that represent the weights between the input layer and hidden layer are (hidden nodes by input nodes) -> (100 by 784) matrix. This is so we can do matrix multiplication with the 784 inputs to obtain an output, which is then passed into the sigmoid function(explained below), and passed to subsequent layers*
- *The neural network is a very complex function. Consider a network with 784 input neurons, 100 hidden neurons and 10 output neurons. There would be (784*100) + (100*10) = 79400 parameters*

We can devise learning algorithms that solve complex problems such as identifying handwritten digit, by tuning the weights that connect these perceptrons, based on external stimuli. The problem here is that small changes in the weights of the perceptron cause large changes to its output. Suppose the network above incorrectly identified a 4 as a 9. While it may be possible to tune the weights of the perceptron to make the network's output closer to 4, it will have profound effects on the whole network, therefore the outputs for other images are harder to control. The solution to this is to use a different type of neuron, whereby small changes to its weights will have small changes to its output. Introducing the *sigmoid neuron.*

The sigmoid neuron is like a perceptron, with the major difference being that the inputs and outputs can take any value between 0 or 1, not just be 0 or 1. Given the input ***x*** with weight ***w,*** the output would be $\sigma(w.x)$, where $\sigma(z)$ is the sigmoid function.

The Sigmoid function:

$$\sigma(z) = \frac{1}{1 - e^{-z}}$$

As z -> ∞^ +, $\sigma(z)$ -> 1

As z -> ∞^ −, $\sigma(z)$ -> 0

The statements above show that the sigmoid neuron accurately approximates the perceptron model

For the neural network above, the output from the first layer would more formally be:

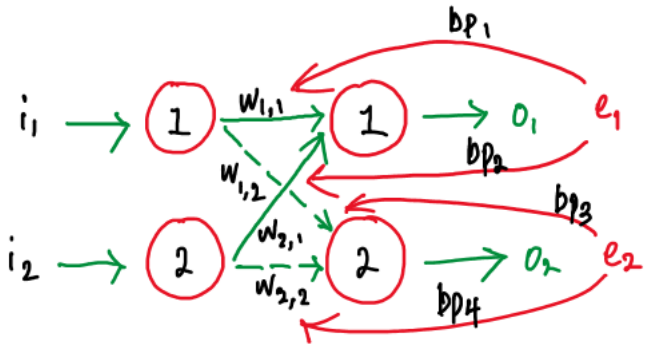
$$\sigma(z) = \frac{1}{1 - e^{-\sum_{i=1}^{i=784} w_i x_i}}$$

The learning process takes place in the *hidden layer* of the neural network, often through *backpropagation,* and *gradient descent.*

Backpropagation:

Neural networks normally use backpropagation of errors to tune their weights in such a way that its outputs better match the solution needed. Backpropagation involves working out which weight links contribute the most to the error. The error function for a single neuron and its output could be the squared difference between the actual output and the target output. For a complex network, such as the one above, we need to figure out a way of decreasing the error function, which is normally done through *gradient descent.*

Bp -> backpropagation



$$bp_1 = e_1 \times \frac{w_{1,1}}{w_{1,1} + w_{2,1}}$$

$$bp_2 = e_1 \times \frac{w_{2,1}}{w_{1,1} + w_{2,1}}$$

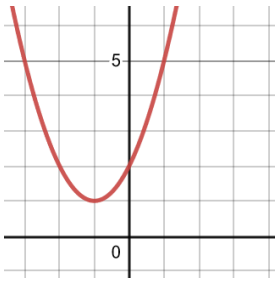
$$bp_3 = e_2 \times \frac{w_{1,2}}{w_{1,2} + w_{2,2}}$$

$$bp_4 = e_2 \times \frac{w_{2,2}}{w_{1,2} + w_{2,2}}$$

The errors are then recombined at the hidden nodes, and this backpropagation process can then be repeated for more layers. Like the feedforward process of the neural network, the backpropagating process can be represented by matrix multiplications, making it easier for programming languages to work out results quickly and efficiently.

Gradient descent

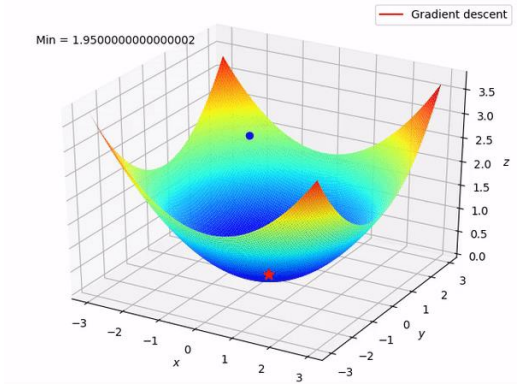
Mathematically, gradient descent is defined as a way of minimising the gradient, or the slope of a function. This is because, the gradient is viewed as a vector that gives the direction of greatest increase in a function. Suppose this function was the error function, we would want to decrease its slope to decrease the error function with respect to the connection weights to find a local minimum of this function. Normally functions are easy to find gradients for, it's just calculus.



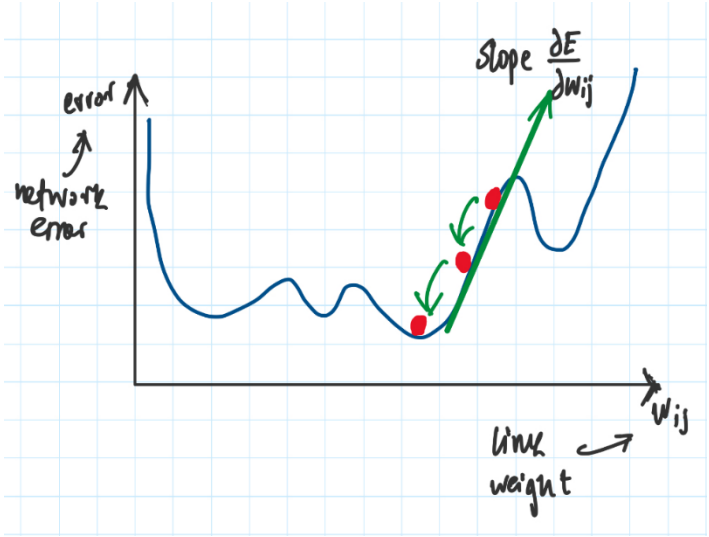
$$y = (x+1)^2 + 1$$

Gradient = $dy/dx = 2(x+1)$

Consider an error function for the 79400 parameters in the neural network above (the sum of the squared differences between the target output and the actual output), it would be infeasible to mathematically work out the gradients, and how to change them for each parameter. In this case, we use gradient descent, which is the idea of following the direction in a function that gets closer and closer to a local minimum.



In our case, we want to minimise the error function of the neural network, where we take small steps towards a local minimum.



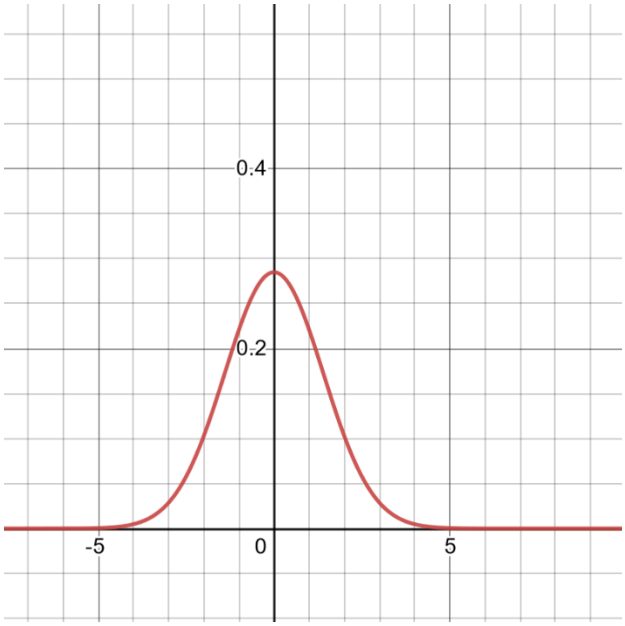
The real network with thousands of linked weights will have a more complex landscape, that needs to be optimised through gradient descent to create better connection weights.

Notice that we needed to work out the error of the neural network using the target output, which implies that we have a dataset of training data for the neural network to use, which is certainly available for the problem presented here of recognising handwritten digits (MNIST database). However, we might not have the same leisure with Tetris, and moreover, labelling such a dataset would be much more complex than labelling MNIST. *Labelling refers to associating each data point with what its output should be.* We need to use another method to optimise the weights in the neural network model we choose for solving Tetris.

Initialising Weights in a Neural Network

It is important to think about exactly how the weights of the neural network will be initialised. Although it is valid to generate random weights, a lot of research has been carried out that shows improved performance in neural networks depending on how well their weights are initialised. One such way involves sampling the weights from a normal probability distribution with its mean at 0, and its standard deviation depending on the number of links coming into the node.

Remember:



The normal distribution to the left was a mean μ and a standard deviation of σ. The standard deviation is a measure of the spread of the values in the distribution.

A node in the neural network with N incoming links can have its outgoing links with weights sampled from a normal distribution with:

$$\sigma = \frac{1}{\sqrt{N}}$$

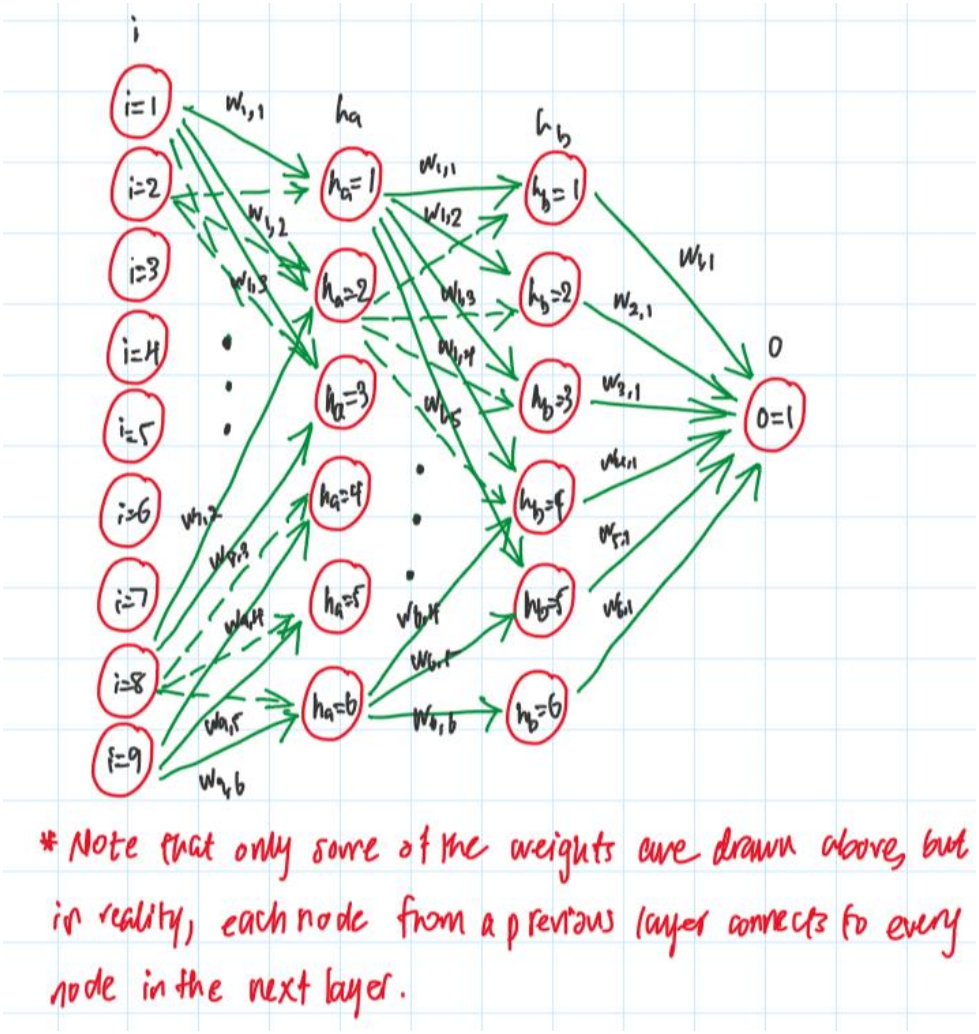
Optimising Neural Networks with Darwinian Evolution:

Notation overview:

- $S\_t \rightarrow$ *the state at a time* t
- $S\_(t+1) \rightarrow$ *the state at a time* $t+1$
- $s \rightarrow$ *the current state*
- $s\text{^}' \rightarrow$ *the next state*
- $a \rightarrow$ *the action made in the current state*
- $A\_t \rightarrow$ *the action at a time* t

Evolution is the idea that species with desirable traits survive longer than those without those traits in an environment, and then pass on these traits to their offspring, creating a generation of individuals that are more *fit.* Fitness here refers to how capable an individual is of surviving in an environment, and desirable traits are those that facilitate this survival. How can this idea be converted into an algorithm, that allows us to create an agent that plays Tetris efficiently? Well, we must start by understanding how evolution occurs, which is mainly through reproduction, and some mutation.

Let's start by defining an individual. Let an individual be a neural network, that takes in features from a Tetris State as input after a move has been made, and returns a number as output, which is an indicator of how good that move was. The input layer of the neural network consists of the 9 features explained above, and the output layer consists of a single sigmoid neuron. There are 2 hidden layers each consisting of an arbitrary number of neurons, 6 in this example.



A population will consist of 1000 of these neural networks, each neural network corresponding to an agent. The next stage is to have each of the agents play a game of Tetris, and record its fitness, which will be used for *crossover* later. The fitness of an agent will be based on the score it attains in the game, as well as the number of tetrises it makes it the game. The score is a good measure of how much time passes before a game ends, so an agent that achieves a higher score is a good agent. The number of tetrises show development of some good strategy, so higher tetrises indicate a good

agent. To setup an agent to play a game, we start from a Tetris state $S\_t = s$ then generate all possible moves a in that state for the current piece, in each column for each of the piece's rotation states. Each of these actions is tested by simulating the action, which creates a new state $s\text{^}'$. This state is represented in terms of its features, which are then passed as input to the agent's neural network. The neural network outputs a score for that action. At the end of this testing, the agent has a move set, with each move scored by its neural network. The final step is to make the move with the maximum score, leading to a new state $S\_(t+1)$

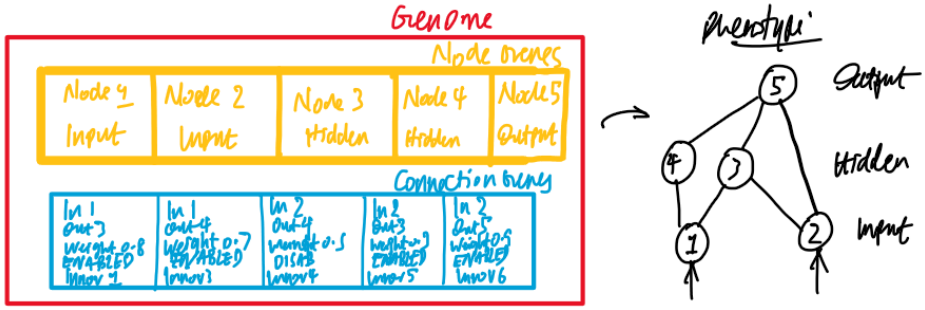
Using the ideas presented above, it is possible to write a genetic algorithm that learns to optimise the weights of a neural network, and over multiple training generations, contains a population with an individual that has learnt to maximise its Tetris score better than any of its predecessors. As we will see in the evaluation of this method, the agents eventually converge to an optimal solution. One existing implementation of genetic algorithms is NEAT-Python, which will be described below, as some of its ideas will be borrowed when writing our genetic algorithm.

NueroEvolution of Augmenting Topologies (NEAT):

NEAT was introduced in 2002 by Kenneth. O. Stanley and Risto Miikkulainen, as a solution to three major challenges for Topology and Weight Evolving Neural Networks (TWEANNs); 1. Finding a genetic representation in which dissimilar neural network structures can cross-over meaningfully, 2. Protecting topological innovation that needs a few generations to optimise, 3. Minimising complexity of topologies throughout evolution without a special fitness function that measures complexity. NEAT uses clever methods to solve each of these problems. Genetic encoding is used to track genes through historical markings, which allows the algorithm to keep track of closely related genomes and use that information for crossover. Speciation preserves genomes from premature extinction by pitting them against closely related genomes. The algorithm starts from a basic neural network structure, and gradually mutates this structure when need arises.

Genetic Encoding

Through genetic encoding, genomes are used to describe a neural network topology, while storing information about its close relatives requiring very little computation. A Genome consists of Node Genes and Connection Genes. There is a Node Gene for each node in the neural network, and information about its index, and in which layer it is stored. The Connection Genes store the bulk of the information needed to create a final network; the In and Out Nodes which are the 2 nodes being connected (the connection running from the In Node to the Out Node), the weight of the connection, whether the connection is enabled, and an innovation number (this is how historical markings are done to track genes throughout mutation). From this a phenotype(network) is made.



Mutation is how simple structures such as the ones above grow into more complex ones throughout the evolution process. There are 2 forms of this: Weight Mutation, which involves changing the weights of connections based on the network output in comparison to the desired output. This is

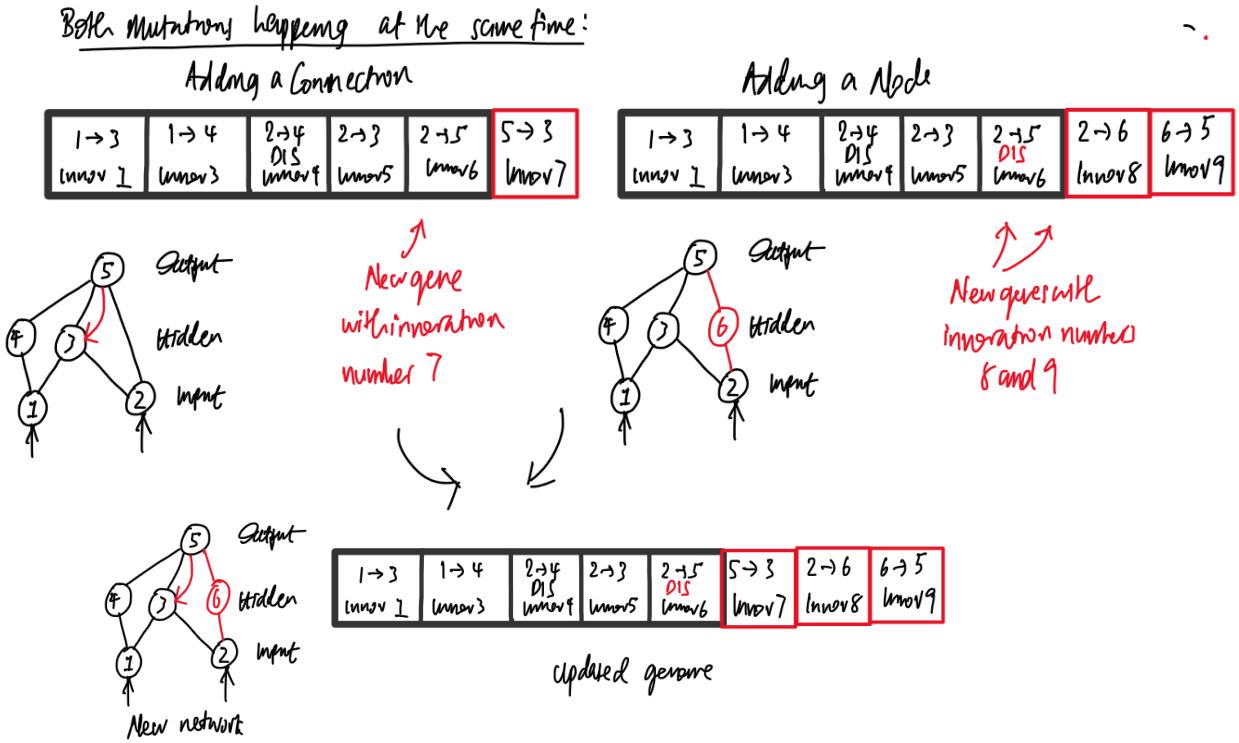
done through backpropagation as in a traditional neural network. For example, the outputs of a neural network and the target output can be used to calculate an error between the input and hidden nodes, and the hidden and output nodes. An Error function can then be created, which defines how the error changes as a function of the weights between each layer. This error function along with the calculated errors is used to come up with a new weight for the connection.

The second form is Structural Mutation, which has 2 sub-groups: Adding a Connection, Adding a Node. When adding a connection, a new Connection Gene is added, enabling a connection between 2 previously unconnected nodes. When adding a Node, a new Node Gene is created, and replaces a connection. The connection between the 2 nodes becomes disabled, and 2 new connections are made from either node to the new node and added to the genome.

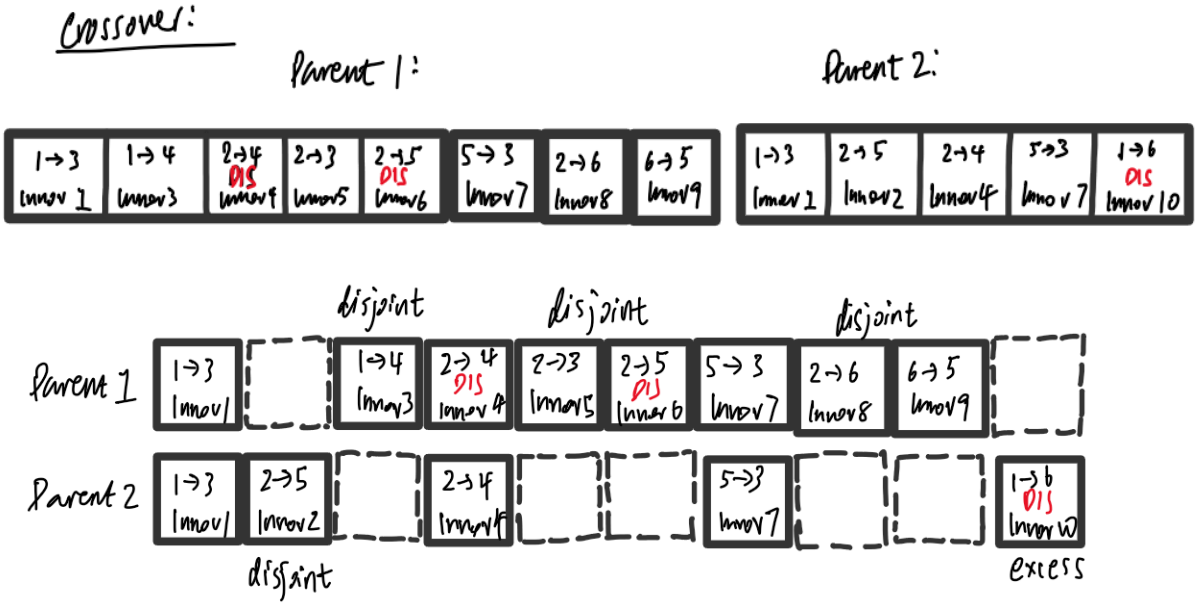
Tracking genes through historical markings

When mutations happen, neural networks of different topologies and complexity are created, and this describes the problem that was faced by TWEANNs before NEAT; finding a way to meaningfully cross-over disparate topologies to produce better offspring. NEAT solves this by giving the system information of closely related networks, through a low-computation method that uses an innovation number. Whenever a new gene is made, a global innovation number is increamented and assigned to that gene.

During cross-over, the offspring inherits the innovation number from its parent gene as they are never changed. As a result, the historical origin of every gene in the population is known.



Cross-over then becomes a more tractable problem with the aid of historical markings, as no topological analysis is needed now to work out which genes match up. From historical markings, the historical origin of the genes can be inferred, which is useful as it helps the algorithm know which genes match up with which



The genomes are then lined up for the crossover process. Genes that are out of range with the innovation number of the parents are excess genes, and those within range but without a pair are disjoint genes. For genes with matching innovation number, the gene inherited by the offspring is chosen randomly from the parent's genes. This solves the problem of meaningful crossover in TWEANNS, without need for topological analysis.

Protecting innovation through speciation

Speciation is used in the NEAT algorithm to protect innovation of species that need some time to optimise. Some structures when added to the population may initially have a low fitness, and therefore are eliminated from the population. Through speciation, competition may only be limited to genomes of the same niche, giving them time to optimise to their best potential. To carry out speciation, the system needs to have information of which genes are closely related, which is done through historical markings discussed above. The number of disjoint genes between 2 genomes is an inherent indicator of their compatibility. A compatibility distance can be computed, and each genome in the whole population is checked, and compared randomly with another genome. If their compatibility distance is less than a certain threshold, they are placed in the same species. The genome is placed in the first species that satisfies this condition, so that one genome is not placed in 2 species.

Excess genes → E
Disjoint genes → D
Average weight difference between matching genes → $\bar{W}$

compatibility distance → δ
compatibility threshold → δ_t
c_1, c_2, c_3 → weight factors

N → number of genes in genome with more genes

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

N_j → old number of genomes in species j

N_j' → new number of genomes in species j

t_{ij} → fitness of individual i in species j

$\bar{f}$ → mean adjusted fitness in entire population

$$N_j' = \frac{\sum_{i=1}^{N_j} t_{ij}}{\bar{f}}$$

Reproduction in NEAT

Reproduction in NEAT is done through explicit fitness sharing, where all genomes in the same species must share the same fitness as their niche, making it difficult for any one species to dominate the population. Their original fitness is first divided by the number of individuals in the species, and if the new fitness is below the species average, then that genome is expelled from the species. The best performing individuals in each species are randomly selected for mating and produce offspring that replaces the entire population.

A.3: REQUIREMENTS GATHERING

The Tetris Guideline is a set of rules that govern how Tetris ought to be implemented. The indispensable rules were considered in this Tetris implementation, except for the ghost piece functionality, because the game will be played primarily by the genetic algorithm, and not a human, where this functionality would become more important.

INTERVIEW:

I had a short interview with one of my classmates, Daehee, to discuss their introduction to Tetris, and what they would expect to see in a Tetris game, as well as the AI aspect of it.

Q1: What is your history with Tetris and how were you introduced to it?

A1: Tetris is a classic and even though I haven't played in a long time, I still remember the aesthetic, which is falling pieces of various shapes and colours, where you must place them in a certain way to win. The game itself isn't too complicated, the difficulty for me is in the speed increments that happen during the game, giving you limited thinking time.

Q2: What are the main things you would expect a Tetris implementation to have?

A2: I would expect it to have basic resemblance to the original and feature some memorable tactics such as stacking pieces as high as possible and leave space for an "I" piece to gather more points for a Tetris.

Q3: Would it be fun to have the AI tell you which placement is the best, and then let the human player perform that placement?

A3: Well, I'm not sure that that would be fun, especially due to the low thinking time you get on higher levels. Even if it told you, you probably wouldn't have enough time to place the piece anyway. Besides, most people can play Tetris reasonably well with much practise. I think it would be more fun to go *against* the AI and have a go at defeating it.

Q4: Do you think it would be fun to have different types of pieces in the game? For example, when Tetris was first conceived it started out as Pentominoes, which are 5 blocked variations of Tetris.

A4: Yes, I think that would be very interesting. In fact, it would be nice to go against the AI with pentomino pieces and remove the time increments entirely, to test the placement ability of either player without the time pressure. Then would the AI predictions on piece placement be of better use to the player.

SUMMARY:
Daehee would like any Tetris implementation to look like its original predecessors and allow for tactics to be developed during play. The AI would be fun to go against, especially if it was a challenge of placement tactics with new variations of pieces, rather than a race against time.

A.4: OBJECTIVES

There are many examples of Tetris implementations done with reinforcement learning among other machine learning techniques, but partly due to the time constraint of the project, and partly due to curiosity, I decided to make a program that could play Tetris using a genetic algorithm. This algorithm is from 2002 and yet solves problems with early Topology and Weight Evolving Neural Networks with clever, but simple solutions. Such simplicity is quite reminiscent of techniques used to 'solve' Go, Chess and Shogi among others, by AlphaGo, AlphaGo Zero and Alpha Zero. Even though these were much more complex that this project, they showed that often the simpler ideas are the most powerful. Even more exciting, is that there may well be simpler ideas that solve even bigger problems in the future, such as the NP problems, which would indeed be a phenomenal achievement. Tetris being an NP-complete problem, I wanted to discover for myself whether a genetic algorithm, that could learn Flappy Bird in about 3 generations, could also tackle Tetris, to a decent level of play.

The objectives below are derived from the interview and the indispensable Tetris guideline.

Tetris Objectives:

1. The game should be playable, in the sense that the user can carry out any moves without the game crashing or ending abruptly
    a. The game should have error catching capability, while outputting relevant and understandable error messages for the user
    b. Player should be able to pause the game
    c. Player should be able to hard drop
    d. Player should be able to carry out all translational moves
    e. Player should be able to carry out all possible piece rotations
    f. Player should be shown help instructions for controls

2. The game should follow the indispensable Tetris guidelines
    a. Playfield should be 10 cells wide and 20 cells tall
    b. The game should allow for natural rotation of a Tetromino piece
    c. Piece preview should be available
    d. Player should be able to hold a piece
    e. Hard drop should be an option
    f. Colours of the Tetrominoes should be as follows:
        i. I -> light blue
        ii. J -> dark blue
        iii. L -> orange
        iv. O -> yellow
        v. S -> green
        vi. Z -> red
        vii. T -> magenta
    g. Pieces should be generated with the 7-bag algorithm

h. Game must include a version of Korobeiniki music
i. The game is only over when a piece is spawned overlapping at least one block above the playable field

AI Objectives:

3. AI should achieve a score of at least 200,000 by the end of the project
a. This is under the condition that this high score is achieved in under 10 generations of training
b. For each training game, agent has no cap on the number of pieces it is dealt

4. AI should be reasonably good at the game, making choices most human players would
a. Agent should demonstrate good Tetris play by emulating known human strategies
b. Agent should demonstrate good Tetris play by placing pieces in reasonable positions given the situation of the game e.g., shouldn't make a move that would make the structure taller when the game is about to be lost

5. Data visualisation tools should be used to graph AI progress
a. Program should be able to graph progress over 10 epochs of training for the average fitness per epoch over the 10 epochs
b. Program should graph all fitnesses for each of the 10 epochs of training
c. Program should graph the average score achieved over each epoch for the 10 epochs
d. Program should graph each of the scores for each game played over each epoch for all 10 epochs.

6. AI should accurately follow the ideas behind genetic algorithms
a. The agent should learn to play the game by using a genetic algorithm and its ideas such as crossover and mutation as opposed to backpropagation

7. Program should have useful AI testing features
a. The program should save populations after ever set number of epochs
b. The program should be able to load these populations back, carry out mutation and crossover accurately from them, and continue training
c. The program should offer an option to carry out ablation tests for any of the heuristics the user wants

In the sections below, each of these objectives will be referred to in the following format: the first under *Tetris Objectives* will be T.1, second T.2 and so on. The first under *AI Objectives* will be AI.1, second AI.2 and so on.

Scope:

These are components that I have decided to remove from this implementation of the Tetris AI.

- The Agent will **not** be able to move a piece after it has been placed. This limits the search space of the game, which reduces computational burden, because this would probably require a pathfinding algorithm

- The Agent will **not** be able to animate the piece as it places it. Once it calculates the best move, the piece will be hard-dropped

- The Agent will **not** be able to hold and swap a piece

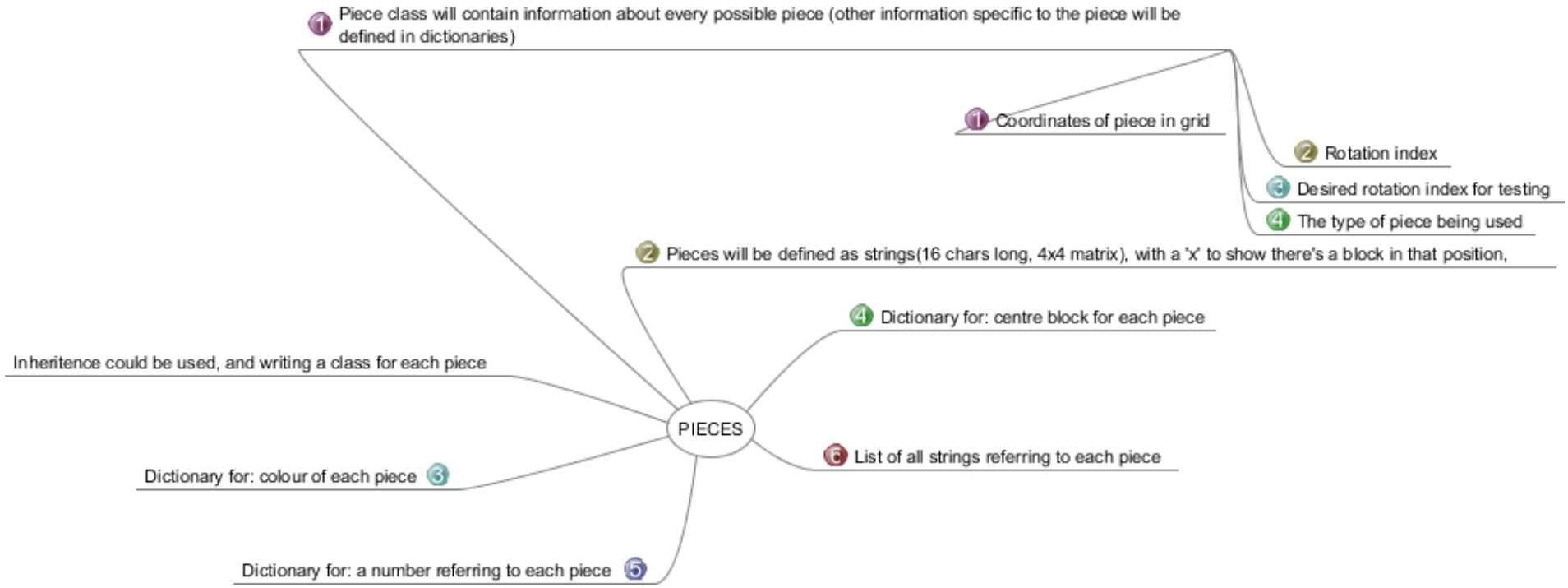
B. DOCUMENTED DESIGN

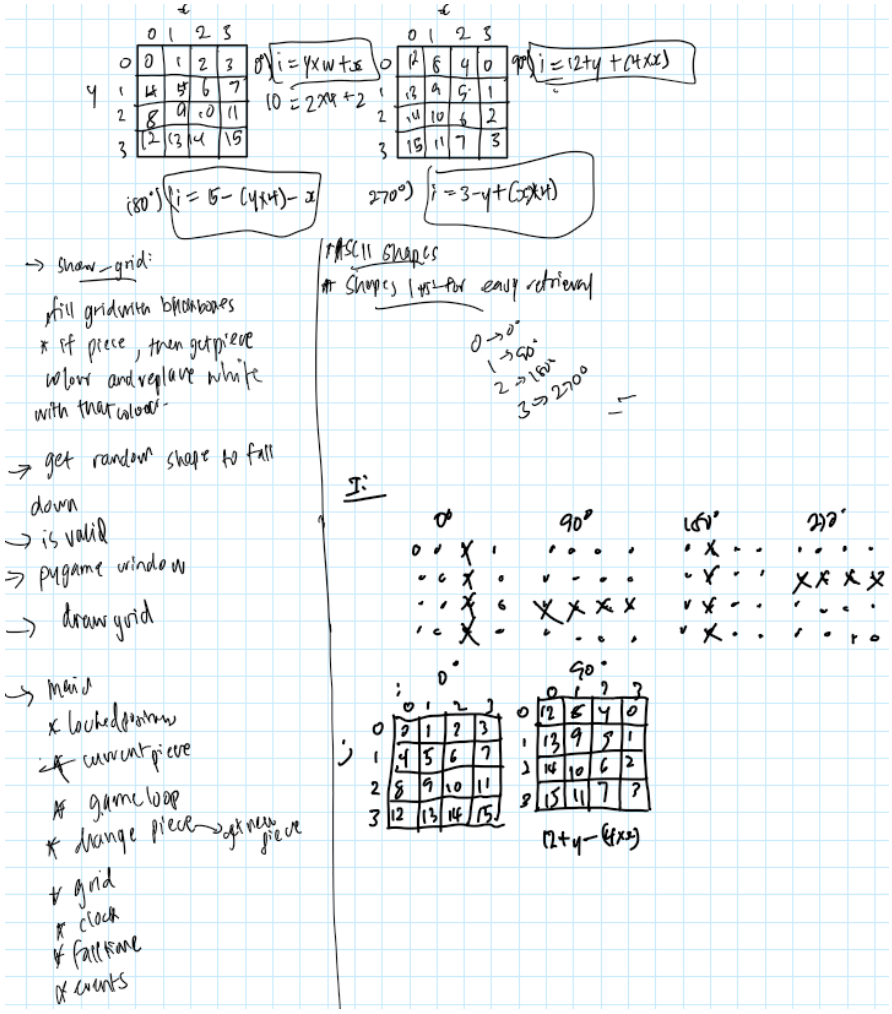
Design Overview:

This project can be split into 2 sub-sections: Tetris implementation and genetic algorithm. Tetris involves piece generation, collision detection and game rendering among other things. We also need to write code for a genetic algorithm as well as represent a population of neural networks to train using the genetic algorithm.

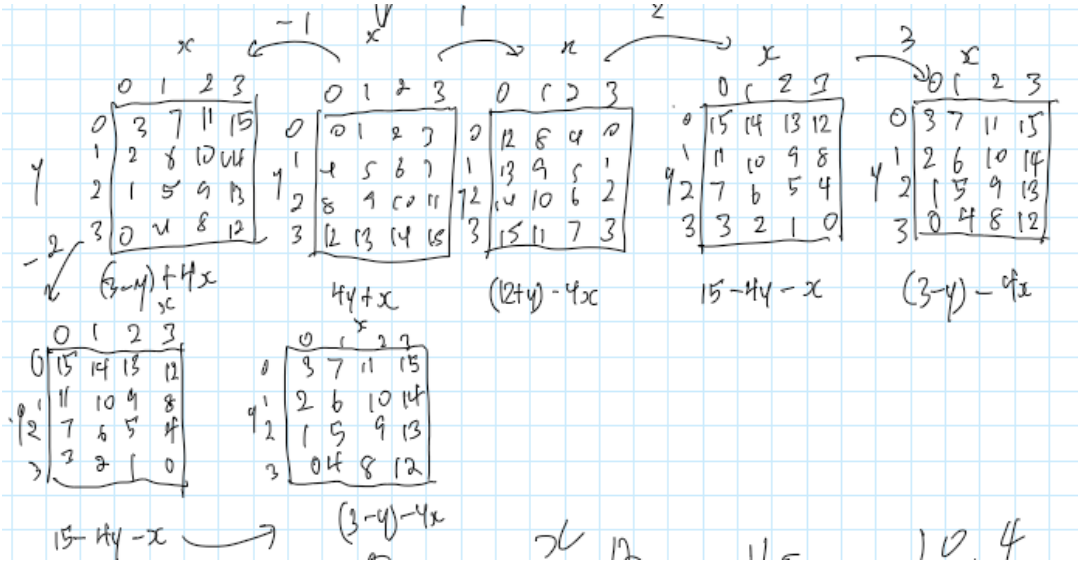
Initial ideas:

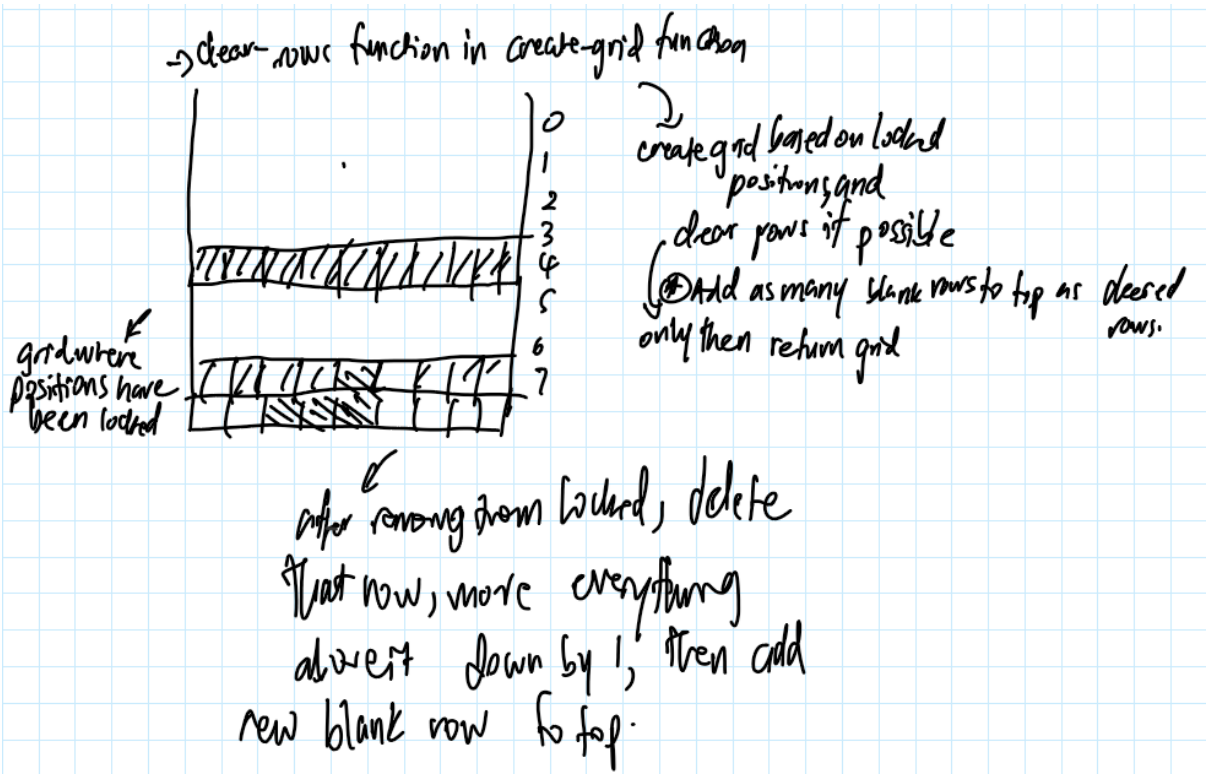
The next few pages are of my initial thoughts on how I would go about doing this project.


PIECES
1 Piece class will contain information about every possible piece (other information specific to the piece will be defined in dictionaries)
1 Coordinates of piece in grid
2 Rotation index
3 Desired rotation index for testing
4 The type of piece being used
2 Pieces will be defined as strings(16 chars long, 4x4 matrix), with a 'x' to show there's a block in that position,
4 Dictionary for: centre block for each piece
Inheritence could be used, and writing a class for each piece
Dictionary for: colour of each piece 3
Dictionary for: a number referring to each piece 5
6 List of all strings referring to each piece

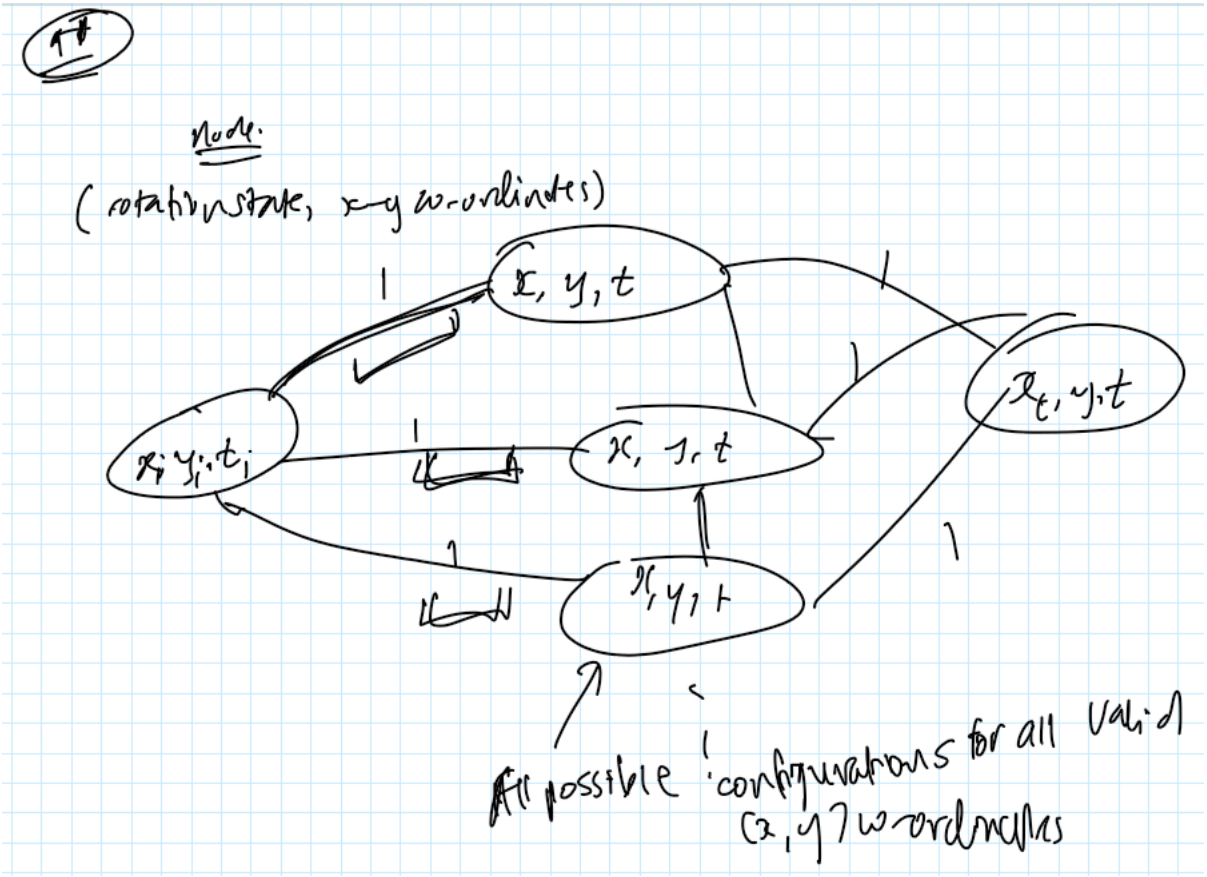


The above shows some plans for functions to code and piece rotation representation. Initially, I decide to use the idea of generating each piece state using a formula from my research, I later discovered the Super Rotation System which I decided to implement instead.

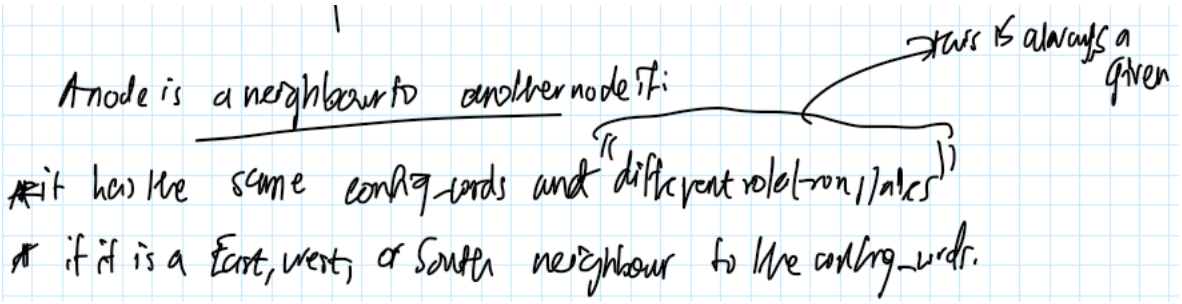


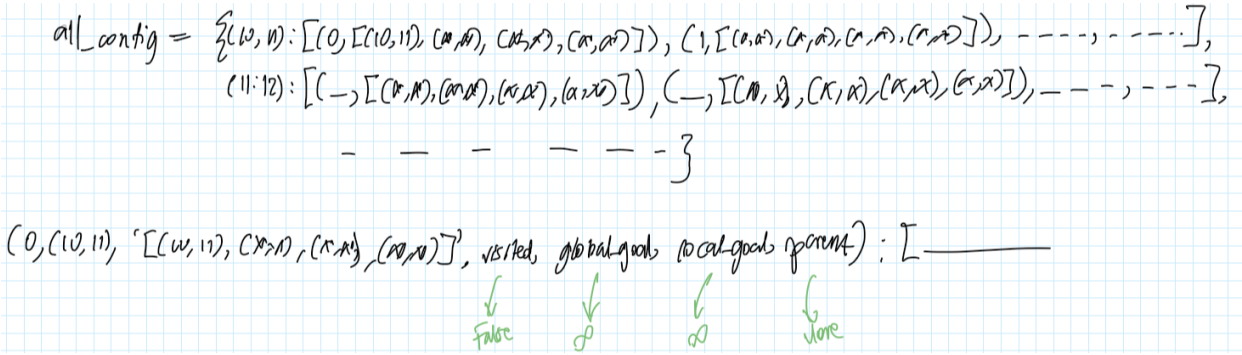


Idea for how board updates will be made and how lines will be cleared.

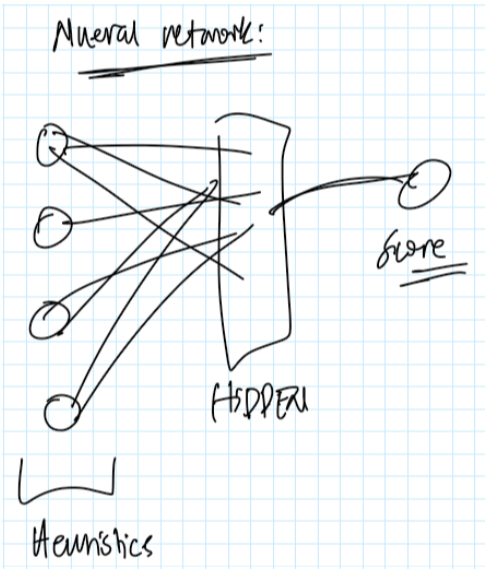


An idea I had for the AI agent was to try having it animate the piece and place it in the best move position with the fewest number of moves, which I tried to do using A* search.





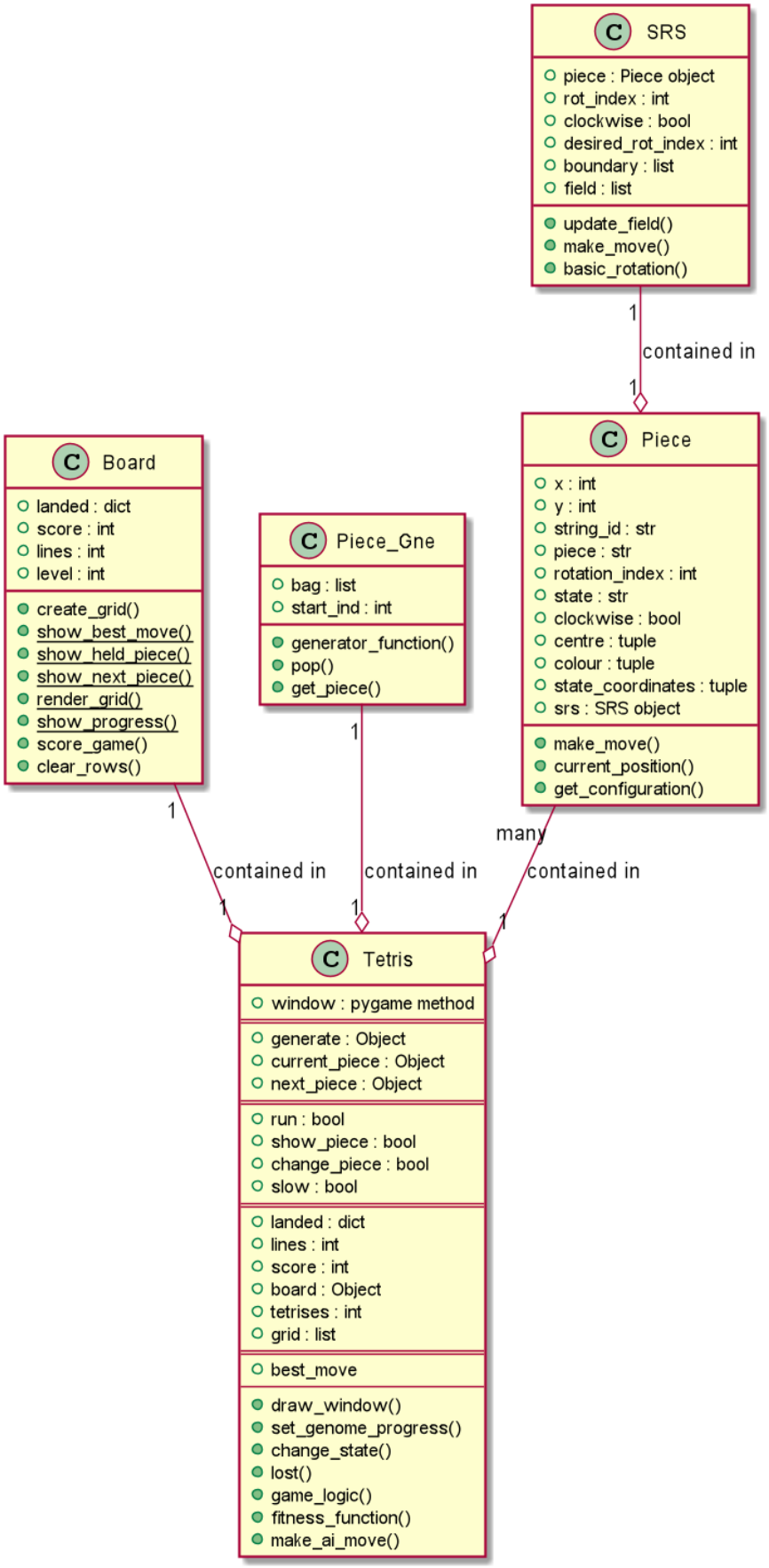
The idea was to generate all possible valid areas the piece could fill throughout the whole board, and use those as nodes, then traverse through the graph with vertices of weight 1 and use the cartesian distance between the area at the node being traversed and the target area (best move) to direct the search. In the end I did not carry on with this idea, but it gave me a good starting point for thinking about how to generate possible configurations of pieces, which came in handy for the agent training code.



To the left is my initial idea for how the structure of the neural network. This hasn't changed much in the final implementation.

The pages after this detail the ideas that I decided to use in the final implementation.

B.1: CORE GAME FUNCTIONALITY



*** The idea explained below assume that the program will be controlled by the AI hence some aspects will miss from it that would be expected in a normal Tetris implementation, such as key presses. However, extensions to this that make it human controllable are explained, as they are essential for testing. Specific functions that need extension will be prepended by an asterisk and explained further**

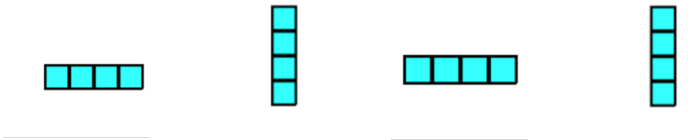
****Parameters that are used only in the human playable version are prepended with a double asterisk**

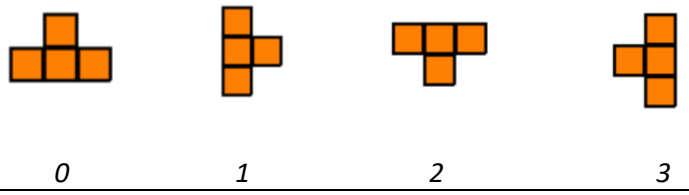
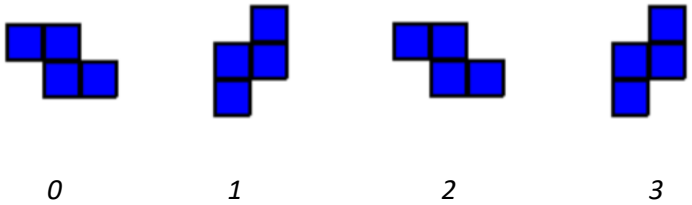
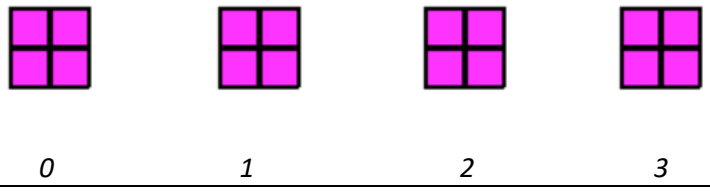
Tetromino Pieces

Defining the rotation states:

The rotation index of a piece identifies the rotation state it is in. This index becomes very important as described later in this section when we need to work out the number of rotations clockwise or counter-clockwise that need to be made for a piece to change from its current configuration to the target configuration *(the best move).* Below are the conventions used in this project for the state of each piece.

All pieces spawn from the 0th rotation index

I	
J	
L	
S	

T	 *0* *1* *2* *3*
Z	 *0* *1* *2* *3*
O	 *0* *1* *2* *3*

Block Coordinates in 4x4 grid

As introduced above, the pieces are stored as strings of length 16, but to interpret and manipulate the string, it needs to be represented as coordinates. Each piece and all its rotation states therefore have a unique representation that defines that state, and piece. Rotation and mapping to the grid among other tasks then become a matter of matrix multiplication, and arithmetic respectively.

```
   0 1  2  3
 0 .  .  .  .
 1 .  x  .  .
 2 x  x  x  .
 3 .  .  .  .
```

Such data will be stored in a separate file for each piece, and its rotation indices. The coordinates for each new rotation state were produced with the *SRS* class explained below and stored in the ***data.py*** file for ease of access when this data needs to be used later for the AI agent.

```
def T(self):
T = {0: [(0, 2), (1, 1), (1, 2), (1, 3)],
           1: [(0, 2), (1, 1), (1, 2), (2, 2)],
           2: [(1, 1), (1, 2), (1, 3), (2, 2)],
           3: [(0, 2), (1, 1), (1, 2), (1, 3)]
    }
return T[self.rot_index]
```

This format is used throughout this Tetris implementation, as it is handy way to use the index positions for rendering, finding relative coordinates to any block in the piece, among many other uses.

The problem presented here is how to represent Tetris pieces in a data-structure in such a way that it can be made to move around on the board, rotate and store colour. The solution is to use a class to represent all the pieces, where instances of this class are created when a piece needs to be spawned. Each piece needs to have a basic representation that can then be translated onto the board that is consistent for all piece types. The way it is done in this implementation is through length 16 string, which can be split into a 4 by 4 array to represent the piece

Example:
This string stores the "T" piece and is then split into a 4 by 4 array when the coordinates of the blocks (represented by 'x') are needed as shown in the code snippet below.

```
T = '....'
    '.x..'
    'xxx.'
    '....'
```

Pieces need to have some useful information in the game and for the AI use. These are all nicely encapsulated into a Piece class.

```
class Piece:
    def __init__(self, x=None, y=None, piece=None):
```

To initialise the Piece class, the spawn (x, y) coordinate is passed as well as the piece that needs to be represented.

PROPERTIES	DATA TYPE	PURPOSE
self.x **self.y**	Tuple	Store the current position of the Tetris piece in the board
self.piece	String	Store the current **type** of piece that's on the board i.e., "J" "L" "S" "T" "Z" "O" "I"
self.colour	Tuple	Store the colour of the current piece
self.rot_index	integer	Store the current rotation index of the piece i.e., 0 1 2 3. These are really used to identify the different rotation states
self.state	String	Store the new form of the rotated piece
self.centre	Tuple	Store the (x, y) coordinate of the centre block in the piece, which will be the point about which the piece is rotated.
self.str_id	String	Store a string that matches the type of piece in play. This is for convenience and ease of access of specific Piece objects

`self.state_cords`	List	This stores the 4x4 grid coordinates of the current state of the piece, which are updated for new rotations, and are used to update the state of the piece
`self.state`	String	Stores the state of the piece, which is a string representation of the rotation state it is in. This variable will only ever change if the piece is rotated successfully. If a translational move only is made, this variable will **not** change, only the (x, y) coordinates of the piece will
`self.srs`	Object	Handles all collision detection and move validation in the game. This will be explained in detail below with reference to wall kicks, their uses, and results. For the AI version, complex moves are taken out, and therefore the class will be expanded upon to include that functionality

METHODS	SCHEME OF WORK
`def make_move(self, move)`	This function receives a prospective move, and if it is validated by the srs system, then it will be allowed, and the state of the piece will change. If it is a translational move, and is valid, the (x, y) coordinate is changed. For example, if the move is *"right",* and it is valid, the x coordinate is incremented by 1 If it is a rotational move, the rotation will only be made if it is validated by the srs object
`def current_position(self):`	Get the coordinates of each block as defined by the 4x4 grid in which the piece is confined Add each x coordinate to the x coordinate of the piece, and add each y coordinate to the y coordinate of the piece

`def get_config(self):`
 	
`A configuration of a piece (essentially a move)in this implementation is defined as follows:`
 	
`(rotation_index, (x, y), current position)`
	Return a tuple of the current rotation index, the (x, y) coordinate, and the current position of the piece

Rotation and Wall Kicks

Rotation and collision detection is implemented in a dedicated class called SRS, hence the SRS object in the Piece class. SRS stands for “Super Rotation System”, and is derived from the Tetris guidelines, where it outlines how Tetris Pieces should rotate, and which states they should rotate to. Wall-kicks although not implemented for this project are also specified under this scheme.

Wall kicks were first implemented in the 2001 Tetris Worlds game and have now become on the most loved and useful game mechanics. The wall-kicks are a useful mechanic, in that they ‘kick’ the piece to an alternative position when they are rotated in an invalid position. This can be useful for T-spins and other complex manoeuvrers. The idea is that for each transition from a state to another, there’s certain offsets that can be made to a piece to have it move into another position. There are 4 possible states, and for each state 2 final states and one can rotate clockwise or anti-clockwise. Therefore, 8 sets of wall-kick data need to be used to work out an alternative position for the pieces.

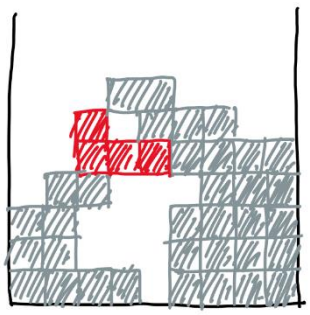
The state of a piece is defined by its (x, y) coordinate and its rotation state. Translational moves change the coordinate, but maintain the state, rotational moves maintain the coordinate by change the state. The goal of wall kicks is for a rotational move that produces an invalid state (basic rotation), find alternative (x, y) coordinates that would make that move valid. However, if none of the alternatives work, the move will not be made. **Note that the shifts are made relative to the current (x, y) coordinate (the coordinate at the basic rotation) not one after another.**

When a rotation is attempted, 5 positions are sequentially tested, including the basic rotation (hence the (0,0) starting each data set). If no new positions yield a valid rotation, then the rotation is impossible and is not allowed. The positions to be tested depend on the initial rotation index and desired rotation index. The positions are described as a sequence of *(x, y)* co-ordinates. By convention *(2,3)* would mean 2 blocks to the right and 3 blocks up. The J, L, S, T, Z pieces share the same data, I with its own set, and O with no data as it doesn’t kick. Each tuple in the wall kick data represents a shift value. For example, (-1, 1) would shift the x coordinate of the piece to the left by 1, and shift the y coordinate up by 1.

Example:

J, L, S, T, Z Tetromino Wall Kick Data

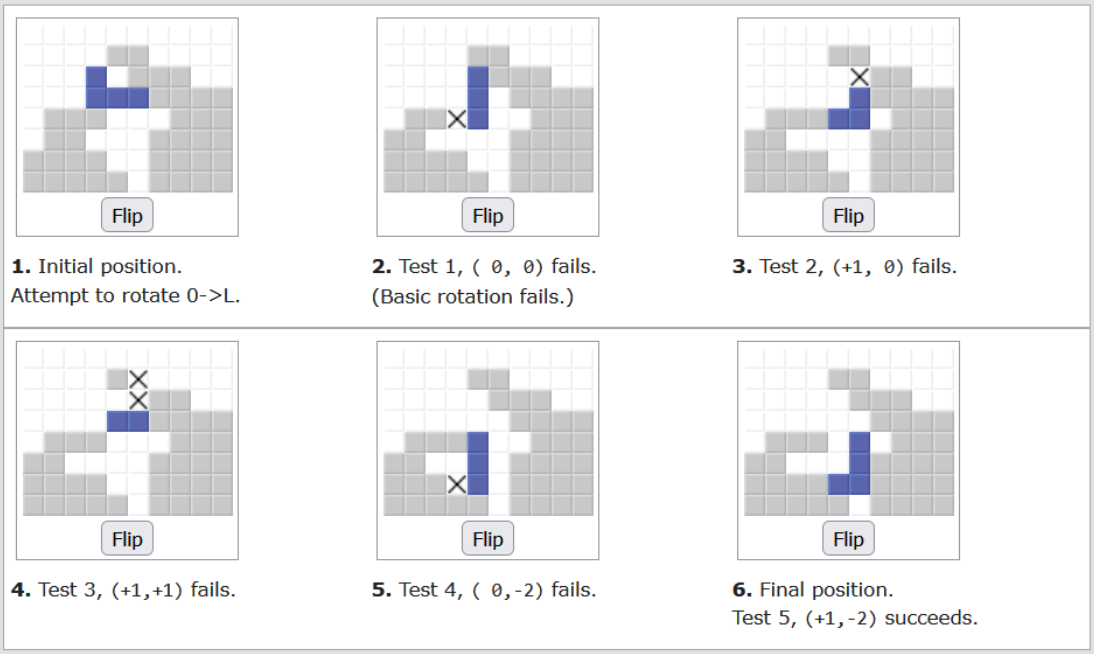
	Test 1	Test 2	Test 3	Test 4	Test 5
0->R	(0, 0)	(−1, 0)	(−1,+1)	(0,−2)	(−1,−2)
R->0	(0, 0)	(+1, 0)	(+1,−1)	(0,+2)	(+1,+2)
R->2	(0, 0)	(+1, 0)	(+1,−1)	(0,+2)	(+1,+2)
2->R	(0, 0)	(−1, 0)	(−1,+1)	(0,−2)	(−1,−2)
2->L	(0, 0)	(+1, 0)	(+1,+1)	(0,−2)	(+1,−2)
L->2	(0, 0)	(−1, 0)	(−1,−1)	(0,+2)	(−1,+2)
L->0	(0, 0)	(−1, 0)	(−1,−1)	(0,+2)	(−1,+2)
0->L	(0, 0)	(+1, 0)	(+1,+1)	(0,−2)	(+1,−2)



Consider a rotation from state (0) to state (L) with the J piece starting the position above.

Series of tests carried out to reach a valid final state:

A wall kick example:
The desired rotation is 0->L, and from the table above, the wall kick test order is (0, 0), (+1, 0), (+1,+1), (0,-2), (+1,-2).



1. Initial position. Attempt to rotate 0->L.

2. Test 1, (0, 0) fails. (Basic rotation fails.)

3. Test 2, (+1, 0) fails.

4. Test 3, (+1,+1) fails.

5. Test 4, (0,-2) fails.

6. Final position. Test 5, (+1,-2) succeeds.

This move is invalid and will not be allowed.

In this implementation, the rules for piece rotation around a centre are followed and explained below.

Super Rotation System

The information required to rotate a piece is its centre, rotation index, and which type of piece it is. This is gathered in the constructor function of the SRS class. The rotation method then handles the piece rotation.

PROPERTIES	DATA TYPE	PURPOSE
`self.piece`	String	Store the kind of piece that is being referred to
`self.centre`	Tuple	Store the (x, y) coordinates of the centre block of the piece that will be rotated about
`self.rot_index`	Integer	Store the rotation index of the piece
`self.clockwise`	Boolean	Set to *"True"* if a clockwise rotation is desired, else it is set to *"False"*

`self.desired_rot_index`
 	
`Will only be used in the human playable version`
	Integer	Stores the desired rotation index

`self.boundary`	List	Stores coordinates that define the boundary of the field
`self.field`	List	Stores a representation of the state so that move validity can be tested

METHODS	SCHEME OF WORK

`def basic_rotation(self, clockwise):`
 	
`A Boolean is passed in, 'True' for clockwise rotation and 'False' for counter-clockwise rotation`
	Split the state of the piece, stored in *"self.piece.state"* into a 4 by 4 array, populate a list of all its global coordinates. Global coordinates here are the locations of all "x" in the string.
	
	Populate a new list with all positions relative to the current piece. This is done by subtracting the centre coordinates from the block coordinates.
	
	Based on which direction is desired, choose the correct rotation matrix from the data.py file. i.e., $\begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix}$ for clockwise rotation and
	
	$\begin{matrix} 0 & -1 \\ 1 & 0 \end{matrix}$ for counterclockwise rotation.
	
	For each of the relative coordinates, apply a dot product of the matrix with that coordinate, as many times as the rotation index. This way, the correct coordinate is produced.
	
	For each new relative coordinate, add the centre coordinates to get the new global coordinates, and hence new state after rotation.
	
	Return these new global coordinates.

`def update_field(self, landed):`	Fill the grid with '0's Check if there are landed positions (there won' t be at the start of the game). If there are, fill in '1's at those positions in the field, and shift them by 1, to take account of the boundary

	Go through the boundary positions and fill those positions with '1's
`*def make_move(self, move):`	If the move is translational (right, left, down), get the current position of the piece Test the wanted move on the field, remembering to offset each coordinate by the width of the boundary If all the resulting positions store 0s, the move is valid and should be allowed If the move is rotational, check whether it is a clockwise or anticlockwise rotation. If clockwise, get the 4x4 grid coordinates of the next rotation state from the *basic_rotation* class, passing an argument of *"True",* do the opposite if the move is anticlockwise Using these coordinates, test the move, by checking whether those positions store a 0, just as we've done above. If all of them do, change the state coordinates of the piece to the coordinates used for testing At this point, we also need to change the state of the piece, so for each (x, y) coordinate in the state coordinates, convert it into an index for a 16-length string, using the formula *(4*y) + x,* and place an "x" and each of those indices Finally, if the move was clockwise, increment the rotation index of the piece by 1, else decrement it by 1

Piece Generation

Tetris pieces dealt pseudo-randomly from a computer might lead to prolonged periods where the same piece is spawned consecutively, or draughts, where some pieces take up to 15 spawns to finally appear on the board. To solve this problem, an algorithm was devised and coined the 7-bag algorithm, as it is analogous to picking pieces from a bag until it is empty.

a. To implement it, I start with an initial bag of pieces `["I","S","O","Z","T","L","J"]`, and get all possible permutations of this list, and store them in a list.
b. As we need to **generate** a new piece each time an old one lands, this offers an opportunity to use Python generators, so each piece is **yielded** from a random permutation. This makes up the **generator function.**

   The **while** loop here runs this generator function for as long as the program is running, therefore we never "run out" of pieces to deal. However, we need to control this output,

and only pick pieces when we need them. Therefore, a function is needed that removes an element from a buffer, shifts a pointer to the next available slot in the buffer, then generates a piece into that slot

```
def generator_function(self):
    permu = list(permutations(self.bag))

    while True:
        for piece in random.choice(permu):
            yield piece
```

c. The **pop** function removes an element at the start of the buffer list, then shifts the pointer by 1, using the modulus operator to make sure it doesn't go over the length of the buffer. It is passed a generator, and **buffer** list, from which a piece is returned.

```
def pop(self, buffer, generator):
    popped = buffer[self.start_ind]
    self.start_ind = (self.start_ind + 1) % len(buffer)
    buffer[self.start_ind] = next(generator)
    return popped
```

The start index denotes the index of the piece that needs to be popped.

d. The *get_piece* function then sets up a 7 sized buffer, which will empty the 1st permutation. Therefore, the *pop()* function will return the first element in the buffer, and then place a new piece into the next slot using the generator function, which will pick from a different permutation.

The last step is to simply use the piece string generated to instantiate a piece object, which is then returned

```
def get_piece(self):
    gen = self.generator_function()
    buffer = [next(rng) for _ in range(7)]

    popped = self.pop(buffer, rng)


    piece_obj = Piece(4, 0, popped)

    return piece_obj
```

"Popped" holds the string of the piece we need, and it is passed into the piece class to instantiate a piece object, which is then returned.

Tetris Board

Representing Tetris visually and in data structures used throughout the project poses many questions such as: *"How do we create the illusion of movement?", "How do we render that board on the screen?", "How do we implement line clearing?".* The solutions to these problems are solved methodically by different methods that work together in the *Board* class that will be explained in this section.

To initialise a *Board* class, the landed positions dictionary, number of lines and score are passed into the constructor. These are updated during the game as needed

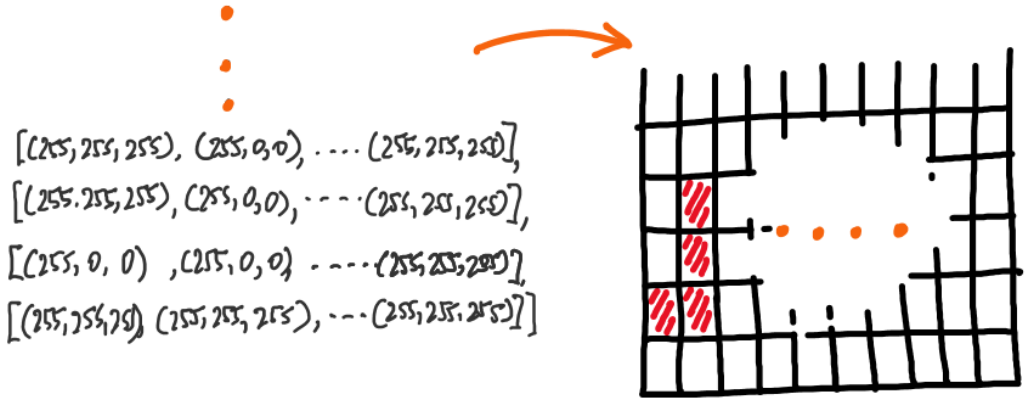
PROPERTIES	DATA TYPE	PURPOSE
`self.landed`	Dictionary	Store the coordinates in the board that are occupied, as well as the colour associated with it
`self.score`	Integer	Store the store accumulated during the game
`self.level`	Integer	Store the level in the game that the player has reached
`self.lines`	Integer	Store the number of lines cleared in the game played
`self.paused_effect`	List	Stores the list of all coordinates in the board the make up the word "PAUSED" that will be rendered when the user decides to pause the game

The questions posed above, and more, are discussed below and how solutions are coded. To create the illusion of movement, the *create_grid()* function below clears the grid except the positions where pieces landed and does this continuously under a game loop during the game. Therefore, any movement of the piece is projected onto the board in real time.

Line clearing uses a simple algorithm that detects whether there's no white blocks in the grid, then deletes those landed positions from the *self.landed* dictionary. All lines above the cleared line are then moved down by the number of cleared lines. Rendering is entirely handled by the pygame module.

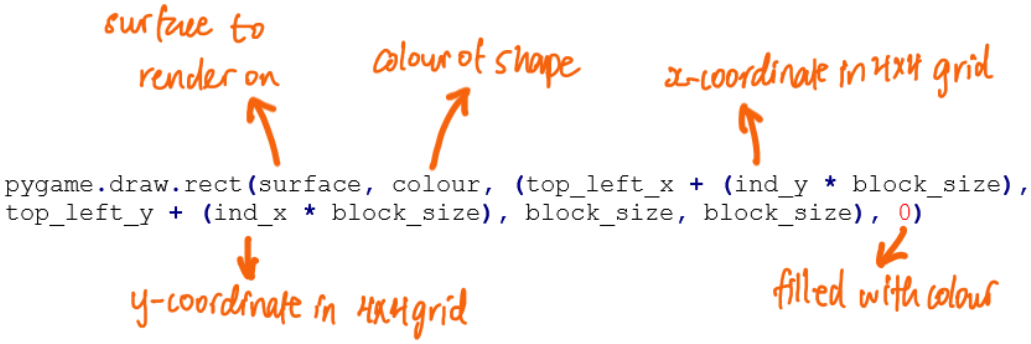
Rendering squares mapped onto the surface

Squares need to be rendered onto the screen that correspond to the positions specified by the 2-D list that stores the grid.



To do this, a simple formula can be used. All renders in pygame are done starting from the top-left conner of that render, specified by a given coordinate.

To render a piece block by block, we need to specify a new x-y starting coordinate for each block, that correctly makes up the whole piece. To get the top left x coordinate, the x coordinate of the block in the 4x4 grid layout id first multiplied by the specified block size, then added to the x coordinate that's passed in the rendering method.



Using the pygame module, a rectangle can be drawn with the `pygame.draw.rect(surface, colour,(x, y, width, height ), 0),` where x and y are the coordinates on the surface of the rectangle, 0 is the thickness of the border. 0 implies it is filled with the colour specified

This will be referred to as the *block rendering technique* throughout this documentation.

METHODS	SCHEME OF WORK
`def create_grid(self):`	a. Set up a 22 by 10 grid with each position storing a tuple corresponding to the colour white i.e., (255, 255, 255) b. Search through the grid coordinates, if that coordinate is in *self.landed,* set that position to store a tuple corresponding to the colour

	stored as the value in the *key:value* pair in the dictionary.
`def paused_grid(self):`	a. Set up a 22 by 10 grid with each position storing a tuple corresponding to the RGB value for black (0, 0, 0) b. Get a random colour using the *random* module c. Search through the grid, if the (x, y) coordinate at that position is in *self.paused_effect*, then that position's tuple is changed to that of the random colour d. Since this function will execute under the game loop, this creates a strobe light effect showing the word "PAUSED" when the game is paused

`@staticmethod`
`def show_held_piece(surface,held_piece):`
 	
`and`
 	
`@staticmethod`
`def show_next_piece(surface, next_piece):`
	a. Receives the surface that needs to be rendered on, which is defined by its width and height, creating a pop-up window on the screen. (width = 800 pixels, height = 650 pixels
	
	b. The coordinates on the surface where the piece needs to be rendered are also stored
	
	c. The string version of the piece is passed, then converted into the 4x4 grid form, to get the block coordinates, and use them for rendering.
	
	d. Go through the 4x4 grid version of the piece, for every 'x', render a square on the surface, mapped to the coordinates specified with the *block rendering technique*

`@staticmethod` `def render_grid(surface, grid):`	a. Pass the surface to be rendered on, as well as the grid that needs to be rendered b. The grid is 10 columns wide, and 22 rows high. The 2 extra rows at the top are used for spawning the piece. The data structure used is a list of lists, where each sub-list is a row, and each element in the sub-list stored a colour, which is rendered onto the screen c. Every position that has a block of a piece stores the colour of that piece, while all other positions store *(255, 255, 255)* i.e., white boxes

	d. To render a boundary around the 10x22 grid, first render a grid that's *(10+BOUNDARY WIDTH)X(22+BOUNDARY WIDTH).* Then go through the grid, and render a square with the *block rendering technique* e. For the black boundaries surrounding the blocks for each piece, simply iterate through the grid, if that position doesn't store white, then there's a block there, so render a black boundary with the *block rendering technique,* but the fill parameter should now be set as a border thickness value i.e., not 0.
`def clear_rows(self, grid):`	a. Set up 2 variables to keep track of the index of the row that has been cleared, as well as the total number of rows that have been cleared b. Search through the grid row by row, which will be a list. If that list doesn't have the tuple *(255, 255, 255)* - *white* stored in it, then it must be cleared c. Increment the number of rows cleared and record the row index that needs to be cleared d. Go through each position in the row, and delete it from the *self.landed* dictionary, which means that row will no longer be rendered. The final step if to move all rows above the cleared row down by the number of cleared rows e. To do this, sort the *self.landed* dictionary in descending order based on the y-coordinates that it stores (rows). f. If the row index is less than the index of the cleared row (if above it), then delete that position from the *self.landed* dictionary, and replace it with a new position, where the y-coordinate has been incremented by the number of cleared rows. Set this position to store the colour that the deleted position was storing

`def score_game(self, cleared):`	Score a board state based on the table below | Level	Points for 1 line	Points for 2 lines	Points for 3 lines	Points for 4 lines
---	---	---	---	---
0	40	100	300	1200
1	80	200	600	2400
2	120	300	900	3600
...				
9	400	1000	3000	12000
n	40 * (n + 1)	100 * (n + 1)	300 * (n + 1)	1200 * (n + 1)	 *This follows the original Nintendo scoring mechanism*
`@staticmethod` `def show_progress(surface, genome_count, pop_size):`	This function will show a chart to visually give an idea of how many genomes have trained so far in a population *Pop_size* will be in integer for the number of genomes in a population, *genome_count* is a counter keeping track of genomes that have finished training a. Choose a block size, and an (x, y) coordinate from which the chart will be rendered b. Split the population list into rows c. For each block in each row, if its numerical index is less than or equal to the genome count, then it should be coloured orange - The numerical index can be calculated using the formula: $index = \#\ of\ columns * y - coordinate + x - coordinate$

Tetris!

Each of the classes described above are brought together in the *Tetris* class, which is the main game class that encapsulates all the components of the game together from each of the classes described above.

Class Attributes:

`current_gen`
`genome_count`
`pop_size`
`h_score`
`best_fitness`
`av_fitness`
	These will be initialised at 0, and will constantly get updated with information from training the genome that will get rendered onto the screen

Defining the action space:

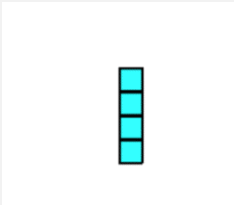
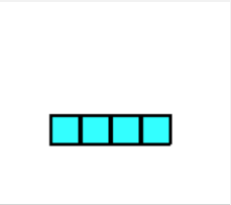
This is a global variable that maps a key press with the expected action

```
action_space = {pygame.K_DOWN:'down', pygame.K_RIGHT:'right',
pygame.K_LEFT:'left', pygame.K_UP: 'cw', pygame.K_w: 'ccw', pygame.K_TAB: 'hd',
pygame.K_SPACE: 'hold', pygame.K_BACKSPACE: 'unhold', pygame.K_p:'pause'}
```

PROPERTIES	PURPOSE
self.win **The game window is setup to a given width and height using the *pygame* module.** **Width = 800** **Height = 650**	Set up a pygame window with a width of 800 by a height of 650 pixels
self.generate **self.current_piece** **self.next_piece**	Set up a piece generator object passed with the 7 possible pieces. Pieces will be generated from this object Get the current and next piece of the game from this generator object
# game board setup **self.landed** **self.lines** **self.score** **self.board** **self.tetrises** **self.grid**	Set up a board class with initialised variable Set up the collision object for collision detection Initialise the grid data structure from the board object
# control parameters **self.run** **self.change_piece** ****self.held_piece** ****self.unhold_piece** ****self.hold_piece**	Set up control parameters for the game that control when the game ends, when the current piece needs to be rendered, the piece object that is being held, when a new piece needs to be spawned
# gravity setup ****self.fall_time** ****self.fall_speed** ****self.clock =** **pygame.time.Clock()**	Initialise fall time to 0, and the speed to 0.002 (which is fast as the agent training time needs to be quick) Set up a clock object using *pygame*
self.best_move	Set up a variable to store the best move in the game, initialised to None. This will change based on what the agent thinks the best move is

METHODS	SCHEME OF WORK
`def draw_window(self):`	Contains calls to methods that carry out rendering to the screen Initialise pygame font object and setup coordinates to the right of the board on the screen. Use the *render* method from the pygame object to render text onto the screen to show the score, lines, number of tetrises, next piece Call the *show_next_piece()* method to render the next piece onto the screen Call the *render_grid()* function to render the board onto the screen Update the pygame display
`def set_genome_progress(self, pop, gen, h_score, best_fitness, av_fitness):`	This function will update each of the relevant variables with the information they need
`def lost(self):`	To check whether a game has been lost, check every coordinate in the landed dictionary, and if any if the y-coordinates is less than or equal to 1, then a piece must have gone into the spawning region, therefore the game should end
`def change_state(self):`	This function handles a state change, which only ever happens if a piece lands Increase the score of the game by 1 Update the *landed* dictionary with the coordinate of each block on the current piece and its colour Call the *clear_rows()* function from the board object, which returns the number of rows that were cleared If 4 lines were cleared at once, then increases the *self.tetrises* variable by 1. Pass the *landed* dictionary to the collision object method *create_field(),* so that its board can be updated with the current state of the board Set the current piece to be the next piece Use the *Piece_Gne* object to get a new next piece

	Set *self.change_state* to False
`*def game_logic(self):`	The code under this function will run under a while loop the controls the runtime of the game Set *self.grid* to *self.board.create_grid(),* which under the game loop will continually re-create the grid ,which creates the illusion of piece movement Increment the variable *self.fall_time* by a certain number of milliseconds using the inbuilt clock method *get_rawtime().* If the fall time is higher that the set fall speed, then the piece should move down by 1. If it can no longer do so (the piece has landed), then the score is incremented by 1 and the state is changed Fill the window with the set background colour, and set the window caption to 'Tetris' Get the coordinates of each block of the current Tetris piece in its current position and place the colour of the piece into the grid at those positions, which will render it onto the screen Call the *self.draw_window()* function, which will set up the pop-up pygame window and render all aspects on the board If the game has been lost, set *self.run* to True
`def fitness_func(self):`	The fitness function is defined as a function of the number of tetrises cleared in the game and the score attained in the game $$f(s) = tetrises * 50 + score * 5$$ The fitness given a state s
`def make_ai_move(self):`	To animate the movement of the piece, we need to compare the current configuration of the piece and change that current configuration to match the configuration of the best move. a. Find the difference between the current rotation state of the piece and the target rotation state. This difference tells us whether to rotate clockwise or anticlockwise, and the number of turns to make $$delta = current\ rotation\ index - target\ rotation\ index$$

	b. If delta is negative, then a series of clockwise turns will be made as many times as delta, i.e., if delta is 1, 1 clockwise turn is made. This movement is done by calling the *make_move()* function described above. *3rd rotation index*  *0th rotation index*  c. Using a similar idea, we can compare the current x-coordinate, and the target x-coordinate. $$delta = target\ x\ coordinate - current\ x\ coordinate$$ d. Delta tells us the number of translations that need to be made. If the target x coordinate is greater than the current x coordinate, a translation to the right is made as many times as delta, i.e., if delta is 2, then 2 right translations will be made. If the reverse is true, a left translation is made e. The final step is to translate the piece downwards, hence landing the piece in the position declared as the best move

Extensions to make the game human playable

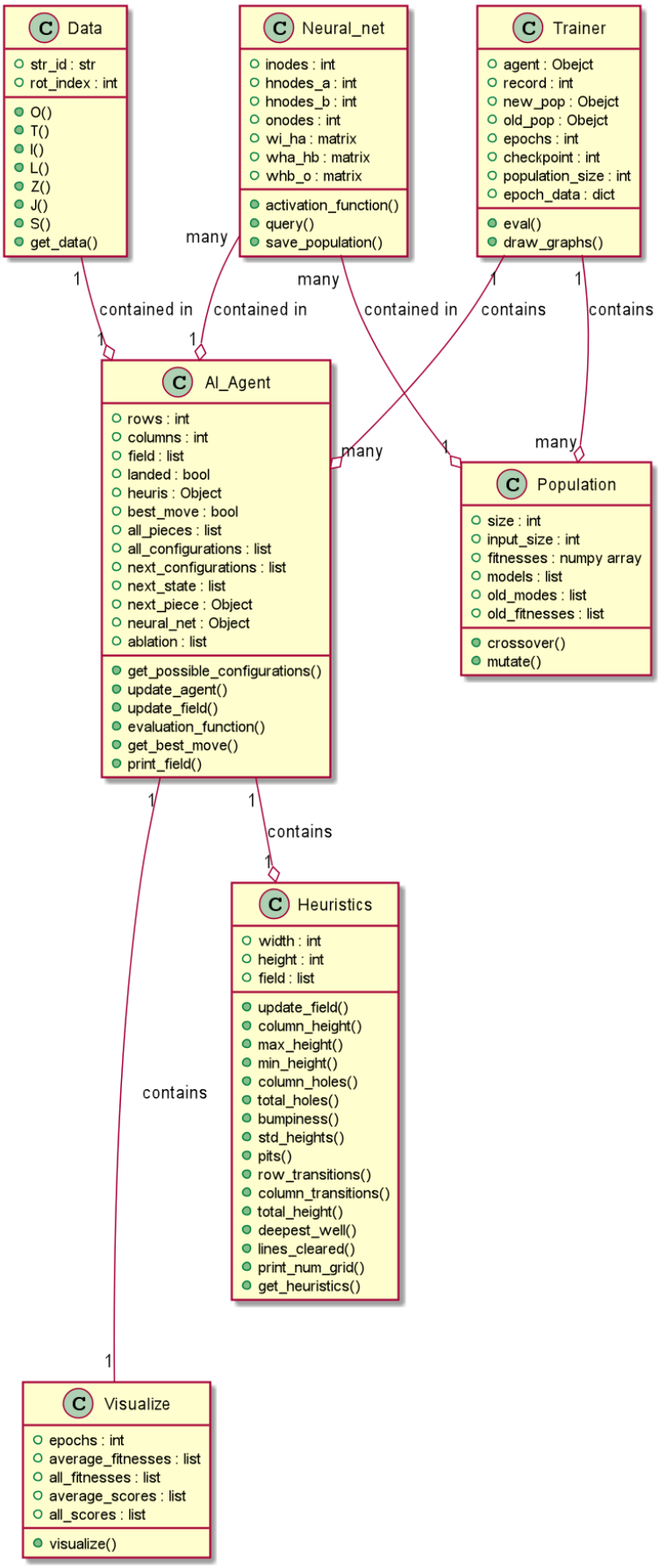
Game logic:

We only need to add the ability to detect key presses and carry out a move in the *game_logic()* function. Additionally, we can also implement the ability for the user to hold a piece. The tables below will explain additions to the code to make it human playable.

METHODS	ADDITIONS
`def game_logic(self):`	Now, we need to check for key presses, which can be done in pygame by checking whether the game event is a KEYDOWN. We then simply make whatever move corresponds to the key that has been pressed, which will be done only if the move is valid. If the user wants to hold piece, set the Boolean variable to True, but only if they aren't already holding a piece

	If the user wants to use the piece they were holding, set the Boolean variable to True, but only if they have a piece held, and set the (x, y) co-ordinates of the held piece to those of the current piece to make it spawn exactly where current piece was, then make swap If the user wants to hold piece, store it in held piece, change current to next, generate new piece to replace next piece set hold piece back to false If the user wants to unhold the piece, set the (x, y) coordinates of the held piece to those of the current piece, so that it respawns exactly where the current piece was. Set *self.held_piece* to "*None" ,* then set *self.unhold_piece* to "*False"*

B.2 : AI IMPLEMENTATION



Rotation data

Rotation data refers to the coordinates of each block for a particular rotation state of a piece. This data is used when the agent is testing all possible moves in a rotation state. Remember that we have a function called *rotation()* that can return these coordinates given a state of a piece under the *SRS* class, explained under *B.1.* The states are passed in sequentially and the data outputs are recorded and stored in one file for each piece under a class. Given any piece and its rotation state, the class can return the correct data. This also creates a boundary between the Tetris implementation and the Agent implementation.

PROPERTIES	PURPOSE
`self.str_id`	A string that stores the piece id so that we know which function to use
`self.rot_index`	Stores the rotation index of the piece so that we know which data under the function to use

Functions corresponding to a piece are of the same format; a dictionary with each of the 4 rotation states as keys and lists corresponding to a list storing the coordinates of each block in a piece with that rotation state as values.

```
def T(self):
T = {0: [(0, 2), (1, 1), (1, 2), (1, 3)],
           1: [(0, 2), (1, 1), (1, 2), (2, 2)],
           2: [(1, 1), (1, 2), (1, 3), (2, 2)],
           3: [(0, 2), (1, 1), (1, 2), (1, 3)]
    }
return T[self.rot_index]
```

Corresponding to:

```
   0 1 2 3       0 1 2 3       0 1 2 3       0 1 2 3
0  . . . .    0  . . . .    0  . . . .    0  . . . .
1  . x . .    1  . x . .    1  . . . .    1  . x . .
2  x x x .    2  . x x .    2  x x x .    2  x x . .
3  . . . .    3  . x . .    3  . x . .    3  . x . .
```

Returning the correct data:

The function called depends on the type of piece, so if it is a “T” piece, the program should call the *T()* function, which will automatically return the list corresponding to the current rotation state of that “T” piece.

```
def get_data(self):
    if self.str_id == 'T':
        return self.T()
```

Calculating heuristics from a board

Under the research section of the analysis, the heuristics needed were explained. We need to write a class that encapsulates these calculations, whereby given any board state, this information can be returned.

PROPERTIES	PURPOSE
`self.width`	Stores the number of columns in the board
`self.height`	Stores the number of rows in the board
`self.field`	Stores an internal representation of the grid from which the heuristics can be calculated

METHODS	SCHEME OF WORK
`def update_field(self, field):`	Receives a 2-D list with 1s for every position that contains a block and 0s for every empty position. This accurately describes the board state from which heuristics can be calculated
`def column_height(self, column):`	- This function receives an integer denoting the column that should have its height measured. - To do this in code, each row is searched from the top of the grid, and the position corresponding to the (x, y) coordinate defined by (column, row) is checked against the number 1 - If they match, then: $column\ height = self.height - row$ - If the column has no rows with 1 in them, then its height is 0 This function can now be used to find the maximum height of the grid, and standard deviation of heights, among others
`def max_height(self):`	Finds the height of each column in the grid, and uses the Python *max()** function to return the highest of these values *A similar thing could be done to find the minimum height, by simply using the *min()* function instead

`def column_holes(self, column):`	Finds the total number of holes in a given column - First find the height of that column, which can be done using the *column_height()* function - Search the rows of the grid from the bottom of the grid. If the position at the (x, y) coordinate corresponding to (column, row) stores a 0 and the height of the column to that empty space (*self.height - row*) < the column height, then that empty position must be a hole - Return the total number of holes for that column
`def bumpiness(self):`	- Initialise a variable *total* that will store a value for the bumpiness - Search through each column of the grid, using the Python *range()* function to *self.width-1*. This is because we need to find the height of contiguous columns, and when we get to the last column, there's no column to its right. - For every 2 columns next to each other, find the absolute value of the distance between them and add it to *total* - Return *total*
`def std_heights(self):`	- Find the height of each column using the *column_height()* function - Numpy has an inbuilt function which when given a list of values, returns their standard deviation. This function is employed here, and the result is returned
`def pits(self):`	- Remember that a pit is defined as a column without any blocks - To find them we need to search through each column in the grid - If all rows contained in that column are empty (store a 0), then that column is a pit
`def row_transitions(self):`	- Initialise a variable for the total number of row transitions - Search each row in the field - If there's a 1 stored in that row, implying a block in that row, search through each column. If the current column stores a 1 and the next stores a 0, or vice versa, then there must be a transition on that row - If the condition above is satisfied, increment the *row_transitions* variable.

`def col_transitions(self):`	- Initialise a variable for the total number of column transitions - For each column, search all rows contained in it - If a position contains a 1 and the next contains a 0 or vice versa, then there must be a transition - Increment the *column_transitions* variable if the above condition is satisfied
`def total_height(self):`	Go through each column in the grid, and find its height using the *self.column_height()* function, then sum up these heights
`def deepest_well(self):`	The algorithm for this is explained in the ***Analysis*** section under the *Deepest Well* section
`def lines_cleared(self):`	Search each row in the grid. If there's no "0" in that row, this implies that that line would be cleared if that move were made
`def get_heuristics(self):`	Returns a list of all function calls for heuristics about a certain board state, which will then act as the input to the neural network

The neural network and genetic algorithm

The structure of the neural network that will be used in this program is shown above. Under this class, all its functionality will be encapsulated. There needs to be a certain number of input nodes, number of nodes in the 1st hidden layer(hidden a), number of nodes in the 2nd hidden layer(hidden b), number of nodes in the output layer. An action function is needed, the sigmoid function will be used in this program. Weight matrices for the weights between layers. We have 4 layers, so we need 3 matrices for: weights between the input layer and hidden a, weights between hidden a and hidden b, and weights between hidden b and the output layer.

We also need a function, which when passed a list of heuristics will propagate them through the neural network and return a value; the score of the move being tested

For these project, I arbitrarily chose to have 8 nodes in each of the 2 hidden layers. We'll have one output node as there will be one score. The input size of the neural network can change if the user decides to carry out ablation tests, so this is accounted for in the *Population* class.

```
The number of input nodes, nodes in hidden a, nodes in hidden b and output
nodes are passed into the __init__ function
```

PROPERTIES	PURPOSE
`self.inodes = input_nodes`	Stores the number of input nodes the neural network will have
`self.hnodes_a = hidden_a_nodes`	Stores the number of hidden a nodes the neural network will have

`self.hnodes_b = hidden_b_nodes`	Stores the number of hidden b nodes that the neural network will have
`self.onodes = output_nodes`	Stores the number of output nodes that the neural network will have
`self.wi_ha`	Stores the weight matrix for weights between the input layer and hidden a
`self.wha_hb`	Stores the weight matrix for weights between hidden a and hidden b
`self.whb_o`	Stores the weight matrix for weights between hidden b and the output layer

METHODS	**SCHEME OF WORK**
`self.activation_func = lambda x : scipy.special.expit(x)`	The sigmoid function will be provided by the ***SciPy*** library The function will be passed an array for the sigmoid function. The scipy library applies the sigmoid function element-wise to each value in the array, and outputs an array of the same dimensions as x. This output array acts as the output from the layer
`def query(self, input_list):`	The input list will be a normal Python list. We need to convert it into a matrix form so that it can be multiplied with the weight matrices. a. First we convert it into a 2-D NumPy array, then transpose it. This will make it have the same structure as the weight matrices. b. Get the inputs into hidden a by finding the cross product of the matrix of inputs with the matrix of weights between the input layer and hidden a. Get the outputs from hidden a by passing the cross product into *self.activation_func().* c. Get the inputs into hidden b by finding the cross product of the matrix of hidden a outputs with the matrix of weights between hidden a and hidden b. Get the outputs from hidden b by passing the cross product into *self.activation_func()*

	d. Get the inputs into the final layer by finding the cross product of the matrix of hidden b outputs with the matrix of weights between hidden b and the output layer. Get the outputs from the final layer by passing the cross product into *self.activation_func()* e. Return the final outputs. This will be the score output from the neural network

Parameters of the genetic algorithm:

The genetic algorithm only implements cross-over and mutation. We need to set a ***mutation probability***, which is the chance of tuning the weights slightly for a new generation. The ***mutation power***, which is the amount by which the weights are tuned. The ***elitism*** threshold, which is a certain percentage of the best of the population that will continue to the next generation without crossover with another one of the best from the population.

```
The size of the population, the list of all neural networks from the old
population and the input size of each neural networks are passed into the
__init__ function of the population.
```

PROPERTIES	PURPOSE
`self.size = size`	Stores the size of the population
`self.fitnesses`	Stores a list of the fitnesses of each neural network in the population. After each generation of training, these fitnesses are used to sort the neural networks by performance, and cross-over the best ones
`self.input_size = input_size`	Stores the number of nodes in the input layer of each neural network
`self.old_models`	Stores the list of all neural networks from the previous generation
`self.old_fitnesses`	Stores the list of all fitnesses for each neural network in the old population
`self.models`	Stores a list of all neural networks from the current population

If there's no old population passed, then the program initialises a list of *n* Neural network objects, *n* being *self.size.* If there's an old population passed, load its fitnesses and list of models into

self.old_fitnesses and *self.old_models* respectively, then carry out crossover and mutation with them. This should populate the current models list with a new set of Neural networks.

METHODS	SCHEME OF WORK
`def crossover(self):`	a. Setup a list of higher probabilities for higher performing neural networks and vice versa using the fitness of each neural network and the sum of fitnesses over the whole population b. Sort the neural networks by performance, the best performers coming first and so on. If within the elitism region, append the neural network as it is to the models list as a child.] c. If outside the elitism region, randomly choose 2 models *a* and *b*, but over the probability distribution set up initially. This ensures that the 2 random models chosen are 2 of the highest performing ones. Ensure that that pair cannot be reselected again. This will not jeopardize the crossover because there's a huge space from which to choose a pair. Assume about 250 of the 1000 networks are in the best performing region, then you have $\binom{250}{2}$ combinations, which is 31125. We certainly have more than enough options to refill the population. This offers some explanation as to why having a high population of genomes is better. If the population was lower, say 50, then maybe 10 were the best performers. You now only have $\binom{10}{2}$ combinations which is only 45. Considering you aren't taking duplicates; you'll have to pull into the worst performer's region to refill the population of 50. d. Get the fitnesses of each of the models *a* and *b.* Initialise a child neural network object. The weights of this object will be from its parents. If both fitnesses of the parents are 0 (which will only happen at the very beginning of training), the we need to manually set the weights of the child. If the opposite is true, we take a weighted sum of the weight matrices from each parent and set them to those of the child. The weight is the fraction of the parent's fitness of the sum of both parents' fitnesses. e. Append the new child object to the population

`def mutate(self):`	Go through each of the new models, for each of their weight matrices, add some noise to the weight
`def save_population(self, epoch_number):`	a. Append a tuple of the weight matrices of each model to a list b. Set up a path to a folder that will store all populations saved. Save the weights list to a file in this folder. c. Set up a path to a folder that will store all populations saved. Save the fitnesses list to a file in this folder d. The epoch number is saved with the file name so it can be reloaded at a specific population

Graphing training progress

All graphs will be done with Matplotlib. After training, the data for each epoch will be passed into a class which encapsulates all plotting functionality.

The Visualisation class will be passed an epochs list, a list of average fitnesses for each of the epochs, a list of all fitnesses for all genomes in each of the 10 epochs, a list of average scores for all epochs, and a list of all scores for all genomes in each of the 10 epochs

PROPERTIES	PURPOSE
`self.epochs = epochs_list`	A list of epochs, acts as the x coordinates for the graph
`self.afl = av_fitnesses`	A list of average fitnesses per epoch
`self.fl = all_fitnesses`	A list of lists for all fitnesses per genome per epoch, used for plotting scatter plots
`self.asl = av_scores`	A list of average scores per epoch
`self.sl = all_scores`	A list of lists for all scores per genome per epoch, used for plotting scatter plots

Programming the Tetris Agent

A Tetris playing agent needs to do the following and have the following information to evaluate the best move given a Tetris state:

Information:

- A representation of the current state of the board
- Analytical information about that board state (calculating heuristics from the board)
- Knowledge of all possible pieces
- A function, which when given input (board heuristics) outputs a value (move score)

    This function is the neural network

Basic idea:

Given the current board state, current piece and next piece, start by finding all possible final positions the current piece can land in for all its rotation states. For each of these possible positions, calculate heuristics from the board of the new state created by making that move. Let's call this an *intermediate state.* Pass these heuristics into the neural network and record the score output. To give a one-piece look-ahead to the agent, we could find all possible final positions of the next piece for all its rotation states, and for each of them, calculate heuristics from the new state resulting from the intermediate state, then return the maximum possible score. Going back to the intermediate state, to find its score, we add the score calculated from the current piece to the maximum possible score from playing the next piece. The intermediate state with the highest score is chosen, and the move that led to that state is passed as the *best move.*

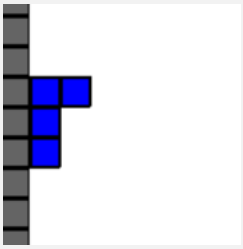
Now to represent this in code……

PROPERTIES	**PURPOSE**
`self.rows`	Stores the number of rows in a board
`self.columns`	Stores the number of columns in a board
`self.field`	Initialised representation of the board, with 0s for every empty space. 1s will be put at every position that gets filled with a block of a piece
`self.landed`	Stores the coordinates of all positions that are occupied in the board
`self.heuris`	Sets up an object that will be used to calculate heuristics from the board

`self.best_move`
 	
`A configuration of a piece (essentially a move)in this implementation is defined as follows:`
 	
`(rotation_index, (x, y), current position)`
	Stores the configuration of the best move

`self.all_pieces`	Stores all possible pieces in the game. These will be referenced to identify the current and next piece
`self.all_configurations`	This list will be populated with all possible configurations in the current state
`self.next_configurations`	This list will be populated with all possible configurations resulting from the next piece in the intermediate state
`self.next_state`	Stores a representation of the intermediate state
`self.next_piece`	Stores an object of the next piece
`self.nueral_net`	Stores an object of the neural network that the neural network will use
`self.ablation`	Stores a list of indices to the heuristics has chosen not to consider in the ablation test

METHODS	**SCHEME OF WORK**
`def get_possible_configurations(self, piece, field, is_next_piece):`	Given a piece, the board that it needs to be played to, this function should generate all possible configurations of the piece in that state - Set up a list of all valid (x, y) coordinates, these being those that are unoccupied - Set up a *data (from the Data class)* object, and a list to store all the configurations <u>The Algorithm:</u> a. Search all possible x-coordinates from -3 to *self.columns+3.* *** This is not done from 0 to *self.columns* because it is possible for a piece to have a negative x-coordinate, which would place it towards the very left of the board**



The 4x4 grid coordinates are:

```
1: [(1, 1), (1, 2), (1, 3), (2, 1)]
```

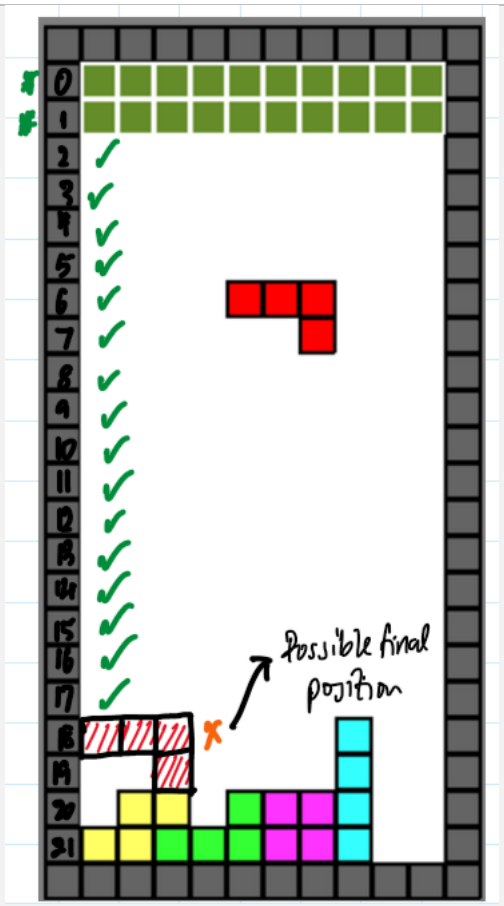
The configuration:

```
(1, (-1, 11), [(1, 12), (0, 12), (0, 13), (0, 14)])
```

Remember that adding the (x, y) coordinates to each of the 4x4 coordinates produces the global coordinates that define the current position of the piece

b. First we need to check whether there are any rotation states for the piece that are valid with the x-coordinate being checked. If there are none, we simply move on to the next x-coordinate, conversely, if there's at least one possible rotation state, we move on to step c. To check, we do not need to find whether a rotation state is possible for all rows, we only need to check an arbitrary row for that x-coordinate, by checking whether all positions that the piece would get are unoccupied.

c. For each rotation index, 0 through 3, set the rotation index of the data object that was set up above, and then call the *get_data()* function, which will return the 4x4 grid coordinates of the current piece for the current rotation state. Now, we don't choose an arbitrary y coordinate, we start by initialising a variable *pos_y = 0*, and a variable *done = False*. Done being a flag that tells us when the piece has landed and cannot go any further down. We then record the position it is in when this happens, as it is a possible final position



The first 2 rows are stared because if a piece does have this as its final position, it will not matter if it turns out to be the best move and is played, because the game will end immediately. This is done just for completeness, to make sure there's always a best move to play, in cases where the board is almost completely filled up. However, just to make sure, if there's no possible final configuration for the piece, it will default to picking its current position as the best move.

d. There are 2 conditions we need to consider when choosing a possible final configuration:

- When the global coordinates formed by a +1 increment to the variable *pos_y* are not valid. This means that those formed by the current *pos_x, pos_y* coordinates are the final position of the piece.

- When the global coordinates formed by the current *pos_x, pos_y* are not valid. This means that those formed by a -1

	decrement to the variable *pos_y* are valid, and are the final position of the piece If none of the above conditions is satisfied, the variable *pos_y* is incremented by 1, and under a while loop, the search continues If either of the above conditions is satisfied, then *done* is set to True, and the while loop is terminated. e. If the flag *is_next_piece* passed as an argument is False, then populate *self.all_configurations* with all the possible configurations found after the search. If the flag is set to True, then populate *self.next_configurations,* as this is an intermediate state, and a next piece.
`def update_field(self):`	This function will update the field with all positions that have been occupied. a. First reset the board to all 0s. This accommodates the fact that this field doesn't have the ability to clear lines, so to we cannot assume that the state hasn't changed from the last time we added the landed positions b. If there are positions in *self.landed,* if the colour corresponding to that position is not white, then it is occupied, therefore, the corresponding position in *self.field* must be assigned to a 1
`def update_agent(self, current_piece, next_piece, landed):`	a. First update *self.landed* with the new landed positions, then call *self.update()* field, so that it is using the correct landed positions b. Call the *get_possible_configurations()* function, passing the current piece, updated field, and setting the *is_next_piece* flag to False c. Update *self.next_piece* with the new next piece
`def evaluation_function(self, current_piece):`	a. First check whether there are any configurations in the *self.all_configurations.* If there isn't, then the *self.get_all_configurations()* function was unable to find possible final positions for the piece. (This is probably an extremely rare occurrence). In this case, set *self.best_move* to be the current configuration of the piece

	b. If there are possible moves, iterate through all the possible configurations, and for each, place a 1 in a position where a block would occupy if that move were made. Set the variable *self.next_state to self.field,* as it will be used as the intermediate state when scoring the next piece. c. Calculate heuristics from *self.field,* then use *self.ablation* go sort through the ones that are considered. Get the move score by passing the heuristics as input, the output will be the score of that move d. Add the score of the attained from the next state by calling *self.next_piece_knowledge()* e. The best move is the move with the highest score
`def next_piece_knowledge(self):`	a. The functionality here is the same as that in *self.evaluation_function().* The difference is that we look at all possible configurations from the intermediate state with that next piece and return the highest attainable score. All configurations from the intermediate state are got using the *self.get_all_configurations()* function, but this time passing the next piece(*self.next_piece*), the next state(*self.next_state*) and setting the *is_next_piece* flag to *True.* b. This score is added to the move score of the current piece. Therefore, if 2 moves have approximately equal scores, if one move is more superior considering the next state after the current state, then it will get a significant boost in its score c. If there are no possible configurations in the intermediate state, then the function returns 0
`def get_best_move(self, current_piece):`	a. Call *self.evaluation_function(),* this will obtain a best move b. Return that best move

Training the AI Agent

This class will handle the training of the agent in terms of setting up populations from the ***Population*** class and a Tetris game from the ***Tetris*** class, printing prompts to set up ablation tests, passing information about the training progress to get rendered onto the screen and printed in the console, calling the ***Visualize*** function to plot graphs from the stored progress data, among other tasks. This essentially brings everything discussed above together.

PROPERTIES	**PURPOSE**
`self.agent = AI_Agent()`	Stores an object for the agent that will be trained
`self.record`	Stores an integer value for the high-score achieved by any agent over any of the epochs of training
`self.new_pop`	Stores the current population being trained
`self.old_pop`	Stores the previous population that was trained
`self.epochs`	Sets the number of generations over which training is done
`self.checkpoint`	A population will be saved every n generations, n will be set by this variable
`self.epoch_data`	A dictionary that stores data about each epoch that will be used to graph training progress. Such data includes average fitness over a generation, maximum fitness of a generation, high-score gained in a generation, average score gained in a generation
`self.population_size`	Sets the size of the population

METHODS	**SCHEME OF WORK**
`def eval(self, load_population, epoch_number, ablation)`	a. If the user has decided to ablate some heuristics, create a list of the indices of those heuristics. This list will be passed to the agent object in *self.agent* so that it knows which heuristics to leave out when passing input into the neural network b. If the user has decided to load a population, they will have passed an epoch number from which to load. Use this to obtain the path of the

	weight matrices and use them to set up a population. Load the fitnesses of each model in that epoch and pass them into the population object as well. Set this population object to *self.old_pop* c. This will trick the program into thinking that the previous population is the one the user manually loaded. This means that cross-over and mutation carried out will produce offspring from the loaded population, which will then start training. d. For each generation of training: - Set up a population object for the current population to be trained. Set it to *self.new_pop.* - Pass into the object *self.old_pop* and the input size of the neural network, which will be 9(maximum possible inputs) – number of ablations. e. For each neural network in the population: - Set up a Tetris game object, and an object for the agent that will train. Set the ablation list for that agent, and set its neural network to be the current neural network in the iteration - Run a game loop, update the agent with information it needs, get the best move from the agent, make the move. - Calculate the fitness of the neural network based on the outcome of making that move. - When the game is lost, set the fitness of the current neural network, store the score obtained, then move on to the next f. After all epochs store the data used for graphing g. Set the old population to the current population

	h. If the saving checkpoint has been reached, save the population

C: IMPLEMENTATION

File Directory Structure

```
.
└── tetris/
    ├── images/
    │   ├── xp.png
    │   └── bg.png
    ├── populations/
    │   ├── 2fitness.pkl
    │   ├── 2population.pkl
    │   ├── 4fitness.pkl
    │   ├── 4population.pkl
    │   ├── 6fitness.pkl
    │   ├── 6population.pkl
    │   ├── 8fitness.pkl
    │   ├── 8population.pkl
    │   ├── 10fitness.pkl
    │   └── 10population.pkl
    ├── controls.txt
    ├── tetris-gameboy-02.ogg
    ├── heuristics.py
    ├── neuralnet.py
    ├── tetris.py
    ├── tetris_ai.py
    ├── train_agent_scratch.py
    └── visualize.py
```

Implementing Tetris: (tetris.py)

```
import random
import pygame
from pygame import mixer
from itertools import permutations
import numpy as np
```

Imports

- The random module is used when creating the paused game effect on the board
- The pygame module handles rendering of the pieces, the grid and text on the screen
- The pygame module also contains a *mixer* function that will be used to play music continuously while the player plays
- The Itertools module contains a function, which when given a list outputs all permutations of that list
- The NumPy module is used to carry out matrix multiplication used when rotating the piece

```
# game assets

I = '....' \
    'xxxx' \
    '....' \
    '....'

S = '....' \
    '.xx.' \
    'xx..' \
    '....'
```

Game assets

```
O = '....' \
    '.xx.' \
    '.xx.' \
    '....'

Z = '....' \
    'xx..' \
    '.xx.' \
    '....'

T = '....' \
    '.x..' \
    'xxx.' \
    '....'

L = '....' \
    '...x' \
    '.xxx' \
    '....'

J = '....' \
    'x...' \
    'xxx.' \
    '....'
```

Game assets

- A string representation for each piece in its 0th rotation index state
- From these initial states, the other states can be derived as explained in the *DESIGN* section
- This is what is defined as the 4x4 grid of the piece

```
CLOCKWISE_MATRIX = [[0, 1], [-1, 0]]
ANTICLOCKWISE_MATRIX = [[0, -1], [1, 0]]

pieces = {'I':I, 'S': S, 'O': O, 'Z': Z, 'T': T, 'L': L, 'J': J}
centres = {'I': (1, 1), 'S': (1, 2), 'O': (1, 1), 'Z': (1, 2), 'T': (1, 2),
'L': (2, 2), 'J': (1, 2)}
```

Global variables

- Clockwise and anticlockwise matrices will be used to generate new global coordinates in the 4x4 grid of the piece, hence creating a new state

- Each piece is keyed to a string of that piece to access it easily. This is defined as the piece's string ID
- Each piece has a centre about which it rotates

```
CYAN = (51, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (225, 21, 132)
YELLOW = (255, 255, 100)
ORANGE = (255, 128, 0)
RED = (255, 0, 0)
GREEN = (51, 255, 51)
WHITE = (255, 255, 255)
GREY = (100, 100, 100)
BLACK = (0, 0, 0)
SPAWN_ROWS = (100, 140, 40) #(50, 100, 255)
BG = (128, 128, 128)
```

Global variables

- All colours that are used in the game are defined as global variables, which we can easily change at leisure

```
colours = {'I': CYAN, 'S': GREEN, 'O': YELLOW, 'Z': RED, 'T': PURPLE, 'L':
ORANGE, 'J': BLUE}

action_space = {pygame.K_DOWN: 'down', pygame.K_RIGHT: 'right',
pygame.K_LEFT: 'left', pygame.K_UP: 'cw',
                pygame.K_w: 'ccw', pygame.K_TAB: 'hd', pygame.K_SPACE:
'hold', pygame.K_BACKSPACE: 'unhold', pygame.K_p:'pause'}

BOUNDARY = 1
ROWS = 20 + 2  # 2 extra rows for spawning
COLUMNS = 10

# dimensions
width = 800
height = 650
block_size = 20
play_w = ROWS * block_size
play_h = COLUMNS * block_size

top_left_x = (width - play_w) // 2 - 100
top_left_y = height - play_h - 400

# font and music!
f = 'freesansbold.ttf'

mixer.init()
mixer.music.load('tetris-gameboy-02.ogg')
mixer.music.play(-1)
```

Global variables

- Each piece is stored with its corresponding colour
- The action space is defined as all possible actions in the game, mapped to the corresponding key presses that trigger them

- The board itself has 10 columns and 20 rows, with a boundary with of 1 block
- The dimensions of the pop-up window are defined by the width and height in pixels. The block size is arbitrarily chosen to be 20 by 20. We can then dynamically work out the width and height of the board in pixels, stored here with variable names *play_w* and *play_h.*
- All rendering in pygame is done from the top left corner, so this coordinate needs to be defined. It is chosen based on what fits well on the screen. Everything else is rendered relative to this coordinate, the text, all individual blocks, everything
- The font used for rendering can be changed by changing the variable *f,* and the mixer object can be used to play music on a loop during runtime

Super Rotation System Class

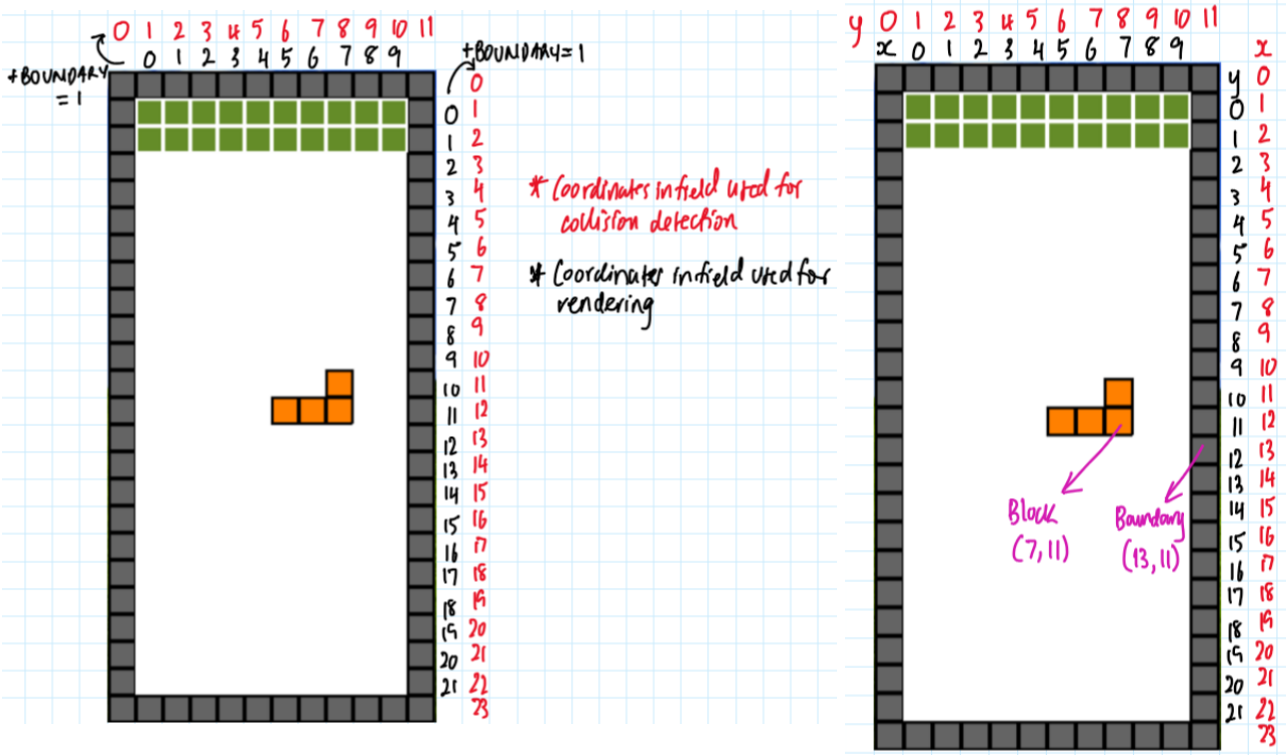
```
class SRS:
    def __init__(self, piece):  # piece is an object
        self.piece = piece
        self.centre = self.piece.centre
        self.rot_index = self.piece.rot_index
        self.clockwise = True
        self.desired_rot_index = None
        self.boundary = [(row, col) for col in range(1) for row in
range(24)] + \
                        [(row, col) for col in range(12) for row in
range(1)] + \
                        [(row, col) for col in range(11, 12) for row in
range(24)] + \
                        [(row, col) for col in range(12) for row in
range(23, 24)]
        self.field = [[0 for _ in range(COLUMNS + 2 * BOUNDARY)] for _ in
range(ROWS + 2 * BOUNDARY)]


    def update_field(self, landed):
        self.field = [[0 for _ in range(COLUMNS + 2 * BOUNDARY)] for _ in
range(ROWS + 2 * BOUNDARY)]

        # create landed positions
        if landed != {}:
            for (x, y) in landed.keys():
                try:
                    self.field[y+BOUNDARY][x+BOUNDARY] = 1
                except IndexError:
                    print('Error on landed piece placement')
                    pass

        # create boundary
        for x, y in self.boundary:
            try:
                self.field[x][y] = 1
            except IndexError:
                print('Boundary positions are wrong')
```

- As discussed in the *DESIGN* section, the rotations, collision, and wall kicks are handled by this class. All the elements needed are initialised in the constructor method

- **Notice:**

The yellow highlighted parts are indexed differently for 2 reasons:

- Firstly, the *+BOUNDARY* accounts for the fact that the indices of the blocks will have to be shifted by the width of the boundary when placed into the field, so that they are in the correct place. Remember that the coordinates of the blocks of the piece do not consider this boundary at all, i.e., the piece class has no knowledge of the boundary. However, we need to take it into account to correctly detect collisions. When we do so, this shifts the width of the board



- Secondly, the landed positions are updated in the form *self.field[y][x] = 1,* yet the boundary positions are updated in the form *self.field[x][y] = 1.* These both work because the block positions of the pieces are defined differently from the boundary positions. This discrepancy can be spotted by looking at the blue highlighted parts and realising that the tuple will correctly index the row (x) then column (y) of the 2-D array storing the field, while the block coordinates need to be flipped to correctly index the 2-D array.

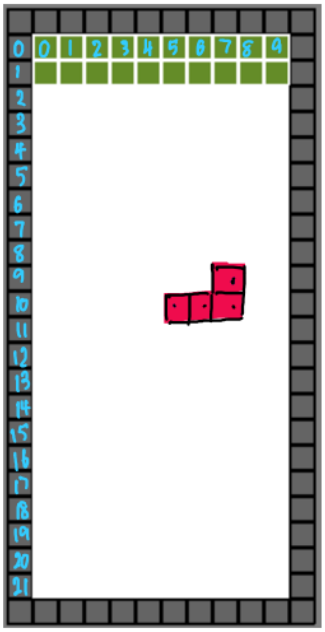
```
    def basic_rotation(self, clockwise):
        mat = None

        piece = [self.piece.state[i:i + 4] for i in range(0,
len(self.piece.state), 4)]
        c_x, c_y = self.centre

        global_cords = []
        new_global_cords = []

        relative_cords = []
        new_relative_cords = []

        # get global cords
        for ind_x, row in enumerate(piece):
            for ind_y, col in enumerate(row):
                if col == 'x':
                    global_cords.append((ind_y, ind_x))

        # get relative cords to centre
        for x, y in global_cords:
            relative_cords.append((x - c_x, y - c_y))

        if clockwise:
            self.clockwise = True
            mat = CLOCKWISE_MATRIX
        if not clockwise:
            self.clockwise = False
            mat = ANTICLOCKWISE_MATRIX

        # calculate new relative cord to centre
        for cord in relative_cords:
            new_r_cord = np.dot(cord, mat)
            new_relative_cords.append(new_r_cord)

        # get new global cords
        for x, y in new_relative_cords:
            new_global_cords.append((x + c_x, y + c_y))

        return new_global_cords
```

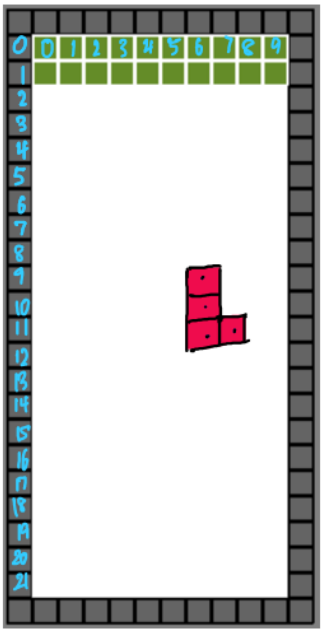
- From a given set of state coordinates, new rotation states can be formed using the method above
- The piece state will be in the form on the right and are converted into the form on the left. This enables us to create grid coordinates, which will be called *global coordinates.*

		0 1 2 3
		0 . . . .
". . . . . x . . x x x . . . . ."	->	1 . x . .
		2 x x x .
		3 . . . .

- After getting relative coordinates to the centre, we apply the matrix to produce new coordinates relative to the centre. To get the new global coordinates, we simply add the coordinates of the centre to each coordinate.

Testing basic rotations





```
Current state: (1, (4, 8), [(5, 10), (6, 10), (7, 10), (7, 9)], 0)
Coordinates of next state: [(7, 11), (6, 9), (6, 10), (6, 11)]
```

The Piece class and SRS class work hand in hand to describe the movement, rotation, colour, and type of Tetris piece being used on the board.

Notice the highlighted coordinates (in pink). These remain the same, because these define the centre block of the piece i.e., the block around which the piece rotates. For further confirmation, we can subtract the (x, y) coordinate of the piece in its current state to find out what this centre is and check with the defined coordinate in the code.

$$(6, 10) - (4, 8) = (2, 2)$$

```
centres = {'I': (1, 1),'S': (1, 2),'O': (1, 1),'Z': (1, 2),'T': (1,
2),
'L': (2, 2),'J': (1, 2)}
```

It matches the defined value for the centre of the 'L' piece, further confirmation that the rotation methods work as expected.

```
def wall_kick_data(self):
    if self.clockwise:
        self.desired_rot_index = (self.rot_index + 1) % 4
    else:
        self.desired_rot_index = (self.rot_index - 1) % 4

    JLSTZ = {(0,1):[(0,0),(-1,0),(-1,1),(0,-2),(-1,-2)],
             (1,0):[(0,0),(1,0),(1,-1),(0,2),(1,2)],
             (1,2):[(0,0),(1,0),(1,-1),(0,2),(1,2)],
             (2,1):[(0,0),(-1,0),(-1,1),(0,-2),(-1,-2)],
             (2,3):[(0,0),(1,0),(1,-1),(0,-2),(1,-2)],
             (3,2):[(0,0),(-1,0),(-1,1),(0,2),(-1,2)],
             (3,0):[(0,0),(-1,0),(-1,1),(0,2),(-1,2)],
             (0,3):[(0,0),(1,0),(1,-1),(0,-2),(1,-2)]}

    I_piece = {(0,1):[(0,0),(-2,0),(1,0),(-2,-1),(1,2)],
               (1,0):[(0,0),(2,0),(-1,0),(2,1),(-1,-2)],
               (1,2):[(0,0),(-1,0),(2,0),(-1,2),(2,-1)],
               (2,1):[(0,0),(1,0),(-2,0),(1,-2),(-2,1)],
               (2,3):[(0,0),(2,0),(-1,0),(2,1),(-1,-2)],
               (3,2):[(0,0),(-2,0),(1,0),(-2,-1),(1,2)],
               (3,0):[(0,0),(1,0),(-2,0),(1,-2),(-2,1)],
               (0,3):[(0,0),(-1,0),(2,0),(-1,2),(2,-1)]}

    if self.piece.str_id == 'I':
        return I_piece[(self.rot_index, self.desired_rot_index)]
    else:
        return JLSTZ[(self.rot_index, self.desired_rot_index)]
```

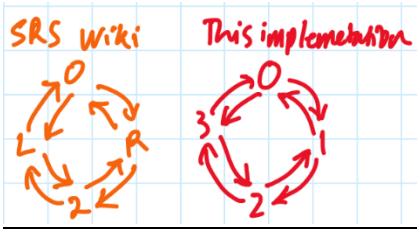
Super Rotation System class

- All the data above is from an online resource, where the series of tests that need to be carried out defined

J, L, S, T, Z Tetromino Wall Kick Data

	Test 1	Test 2	Test 3	Test 4	Test 5
0->R	(0, 0)	(-1, 0)	(-1,+1)	(0,-2)	(-1,-2)
R->0	(0, 0)	(+1, 0)	(+1,-1)	(0,+2)	(+1,+2)
R->2	(0, 0)	(+1, 0)	(+1,-1)	(0,+2)	(+1,+2)
2->R	(0, 0)	(-1, 0)	(-1,+1)	(0,-2)	(-1,-2)
2->L	(0, 0)	(+1, 0)	(+1,+1)	(0,-2)	(+1,-2)
L->2	(0, 0)	(-1, 0)	(-1,-1)	(0,+2)	(-1,+2)
L->0	(0, 0)	(-1, 0)	(-1,-1)	(0,+2)	(-1,+2)
0->L	(0, 0)	(+1, 0)	(+1,+1)	(0,-2)	(+1,-2)

- The data in the table above is for the J, L, S ,T and Z pieces. 0 is the spawn state. R is the rotation state after 1 clockwise rotation from the spawn state. L is the state after 1 anticlockwise rotation from the spawn state. 2 is the state after 2 rotations either direction from the spawn state.



Definition of rotation states comparison

```
    def make_move(self, move):
        if move == 'cw' or move == 'ccw':
            if move == 'cw':
                # get basic rotation cords after 1 clockwise rotation
                basic_rot_cords = self.basic_rotation(True)
            else:
                # get basic rotation cords after 1 anticlockwise rotation
                basic_rot_cords = self.basic_rotation(False)

            new_state = ['.' for _ in range(16)]

            # get correct wall kick data
            wk_data = self.wall_kick_data()

            trials = 0

            # let the final coordinates be the current coordinates
            final_x, final_y = self.piece.x, self.piece.y

            # try every shift value (5 of them)
            for x_shift, y_shift in wk_data:
                # increment trials made
                trials += 1

                # set up grid coordinates of the piece, shifting the
coordinates
                grid_cords =[(x+self.piece.x+x_shift, y+self.piece.y-
y_shift) for x, y in basic_rot_cords]

                try:
                    if all([self.field[y + BOUNDARY][x + BOUNDARY] == 0 for
x, y in grid_cords]):
                        # if the move works, set final x and final y to be
the coordinates after the shift
                        final_x = self.piece.x + x_shift
                        final_y = self.piece.y - y_shift
                        break
                    else:
                        # if move is invalid move on to next test
                        continue
                except IndexError:
                    continue

            """
            Move is made if the basic rotation worked, which is trials = 1
and coordinates are not shifted
            OR
            If more than one trial was made and the variables final_x and
final_y have changed, they must have produced
            a valid move
            """
            if (trials == 1 and ((final_x == self.piece.x) and (final_y ==
self.piece.y))) or \
                    (trials > 1 and ((final_x != self.piece.x) or (final_y
!= self.piece.y))):
                # set the new state coordinates of the piece, as this move
works
                self.piece.state_cords = basic_rot_cords
                # this move works under these coordinates, final_x and
final_y
                self.piece.x, self.piece.y = final_x, final_y
```

```
                # set the new state of the piece
                for x, y in self.piece.state_cords:
                    ind = 4 * y + x
                    new_state[ind] = 'x'

                self.piece.state = ''.join(new_state)

                # change rotation index accordingly

                if move == 'cw':

                    self.piece.rot_index = (self.piece.rot_index+1) % 4
                else:
                    self.piece.rot_index = (self.piece.rot_index-1) % 4
            else:
                # the move is impossible, do noting
                pass

        elif move == 'down':
            pos = self.piece.current_position()

            try:
                return all([self.field[y + BOUNDARY + 1][x + BOUNDARY] == 0
for x, y in pos])
            except IndexError:
                print('WIll try again')
                pass

        elif move == 'right':
            pos = self.piece.current_position()

            try:
                return all([self.field[y + BOUNDARY][x + BOUNDARY + 1] == 0
for x, y in pos])
            except IndexError:
                print('Will try again')
                pass

        elif move == 'left':
            pos = self.piece.current_position()

            try:
                return all([self.field[y + BOUNDARY][x + BOUNDARY - 1] == 0
for x, y in pos])
            except IndexError:
                print('Will try again')
                pass

        return False
```

Piece Class

```
class Piece:
    def __init__(self, x=None, y=None, str_piece=None):
        self.x = x
        self.y = y
        self.str_id = str_piece
        self.piece = pieces[self.str_id]
        self.rot_index = 0
        self.state = self.piece
        self.clockwise = None
        self.all = [(j, i) for i in range(4) for j in range(4)]
        self.centre = centres[self.str_id]
        self.colour = colours[self.str_id] if self.piece is not None else
None
        self.state_cords = []

        piece = [self.state[i:i + 4] for i in range(0, len(self.state), 4)]
        for ind_x, row in enumerate(piece):
            for ind_y, col in enumerate(row):
                if col == 'x':
                    self.state_cords.append((ind_y, ind_x))
        self.srs = SRS(self)

    def make_move(self, move):
        if move == 'right':
            if self.srs.make_move('right'):
                self.x += 1
            else:
                print('Move is invalid')
        elif move == 'left':
            if self.srs.make_move('left'):
                self.x -= 1
            else:
                print('Move is invalid')
        elif move == 'down':
            if self.srs.make_move('down'):
                self.y += 1
            else:
                return False
        elif move == 'cw':
            if self.str_id == 'O':
                pass
            else:
                if self.str_id == 'I':
                    self.srs.make_move('ccw')
                else:
                    self.srs.make_move('cw')

        elif move == 'ccw':
            if self.str_id == 'O':
                pass
            else:
                self.srs.make_move('ccw')

    def current_position(self):  # get grid positions of a passed piece
object
        return [(r_x+self.x, r_y+self.y) for r_x, r_y in self.state_cords]

    def get_config(self):
        return self.rot_index, (self.x, self.y), self.current_position()
```

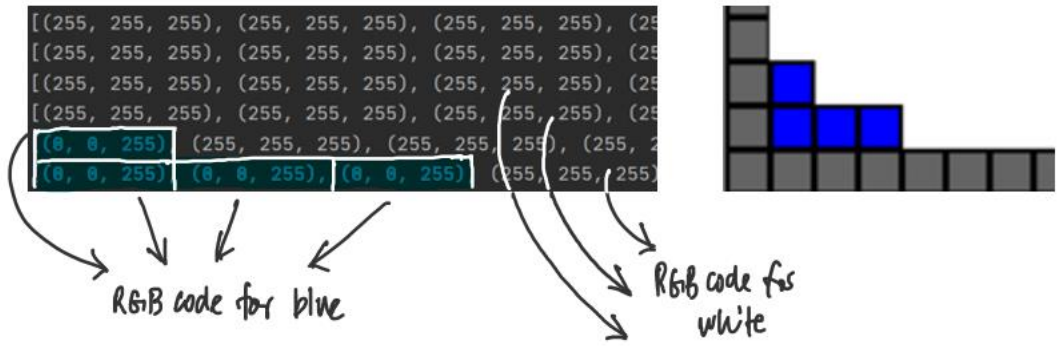
Board Class (with Rendering)

```
class Board:
      def __init__(self, landed, lines, score):
            self.landed = landed
            self.score = score
            self.lines = lines
            self.level = 0
            self.paused_effect=
      [(2,18),(2,19),(2,20),(3,18),(3,20),(4,18),(4,19),(4,20),(5,20),(6,20
      ),(5,17),(6,17),
      (3,17),(4,17),(4,15),(4,16),(2,16),(3,15),(5,15),(6,15),(2,14),(2,12)
      ,(3,12),(3,14),
      (4,12),(4,14),(5,12),(5,14),(6,13),(3,11),(2,10),(2,9),(4,9),(4,10),(
      5,8),(6,9),(6,10),(6,11),(2,8),(2,7),(2,6),(4,8),(4,7),(4,6),(6,8),(6
      ,7),(6,6),(3,8),(2,5),(3,5),(4,5),(5,5),(6,5),(6,4),(2,4),(3,3),(4,3)
      ,(5,3)]
      def create_grid(self):
      # if you want to change colour of grid, change _____ to desired colour!
      (except tetromino colour)
      GRID = [[WHITE for column in range(COLUMNS)] for row in range(ROWS)]

      for i in range(ROWS):
            for j in range(COLUMNS):
                  if (j, i) in self.landed:
                  # set colour if position is landed i.e there is a piece
                  there
                  GRID[i][j] = self.landed[(j, i)]

      return GRID

    def paused_grid(self):
        GRID = [[BLACK for column in range(COLUMNS)] for row in
range(ROWS)]
        random_colour =
colours[random.choice(['J','L','S','T','Z','O','I'])]
        for i in range(ROWS):
            for j in range(COLUMNS):
                if (j, i) in self.paused_effect:
                    GRID[i][j] = random_colour

        return GRID

    @staticmethod
    def show_held_piece(surface, held_piece):
        pos_x = top_left_x + play_w - 100
        pos_y = top_left_y

        n_p = [held_piece.piece[i:i + 4] for i in range(0,
        len(held_piece.piece), 4)]

        for ind_x, row in enumerate(n_p):
            for ind_y, column in enumerate(row):
                if column == 'x':
                    pygame.draw.rect(surface, held_piece.colour, (
                                                          pos_x + ind_y *
                                    block_size + 20, pos_y + ind_x *
                                    block_size + 20, block_size,
                                    block_size), 0)
                    pygame.draw.rect(surface, BLACK, (
```

```
            pos_x + ind_y * block_size + 20, pos_y + ind_x * block_size +
            20, block_size, block_size), 2)

    @staticmethod
    def show_next_piece(surface, next_piece):
        pos_x = top_left_x + play_w - 220
        pos_y = top_left_y

        n_p = [next_piece.piece[i:i + 4] for i in range(0,
        len(next_piece.piece), 4)]

        # next piece
        for ind_x, row in enumerate(n_p):
            for ind_y, column in enumerate(row):
                if column == 'x':
                    pygame.draw.rect(surface, next_piece.colour, (
                                        pos_x + ind_y * block_size + 20,
                  pos_y + ind_x * block_size + 20, block_size, block_size),
                  0)
                    pygame.draw.rect(surface, BLACK, (
                                pos_x + ind_y * block_size + 20, pos_y +
      ind_x * block_size + 20, block_size, block_size), 2)

    @staticmethod
    def render_grid(surface, grid):
        # boundary
        for i in range(ROWS + BOUNDARY + 1):
            for j in range(COLUMNS + BOUNDARY + 1):
                pygame.draw.rect(surface, GREY, (top_left_x - (BOUNDARY *
block_size) + j * block_size,
                                                 top_left_y -
(BOUNDARY * block_size) + i * block_size,
                                                 block_size,
block_size), 0)
                pygame.draw.rect(surface, BLACK, (top_left_x - (BOUNDARY *
block_size) + j * block_size,
                                                  top_left_y -
(BOUNDARY * block_size) + i * block_size, block_size,
                                                  block_size), 2)

        # convert grid colours to output onto surface
        for ind_x, rows in enumerate(grid):
            for ind_y, colour in enumerate(rows):
                pygame.draw.rect(surface, colour, (
                    top_left_x + (ind_y * block_size), top_left_y + (ind_x
* block_size), block_size, block_size), 0)

        # draw black boundaries
        for i in range(ROWS):
            for j in range(COLUMNS):
                if i == 0 or i == 1:  # first 2 rows are for spawing
                    pygame.draw.rect(surface, SPAWN_ROWS,
                                     (top_left_x + j * block_size,
top_left_y + i * block_size, block_size, block_size),
                                     0)
                    pygame.draw.rect(surface, WHITE,
                                     (top_left_x + j * block_size,
top_left_y + i * block_size, block_size, block_size),
                                     2)
                if grid[i][j] != WHITE:
```

```
                    pygame.draw.rect(surface, BLACK,(top_left_x + j *
block_size, top_left_y + i * block_size, block_size, block_size),
                                     2)

    def clear_rows(self, grid):
        cleared_row = 0
        cleared_rows = 0

        for index in range(ROWS - 1, -1, -1):
            if WHITE not in grid[index]:
                cleared_rows += 1
                cleared_row = index
                for column in range(COLUMNS):
                    try:
                        del self.landed[(column, cleared_row)]  # deletes
colours off cleared rows
                    except KeyError:
                        pass

        # sort all landed positions based on the rows in the grid, then
        reverse that as we are
        # searching grid from below

        for position in sorted(list(self.landed), key=lambda pos: pos[1],
        reverse=True):
            col, row = position

            if row < cleared_row:  # if row is above index of row that was
            cleared:
                new_pos = (col, row + cleared_rows)  # make new position
                that moves itm down by number of cleared rows

                self.landed[new_pos] = self.landed.pop(
                    position)  # .pop here removes colours from all rows
above, and places them in their new positions

        self.lines += cleared_rows
        self.score += self.score_game(cleared_rows)

        return cleared_rows

    def score_game(self, cleared):
        if cleared == 1:
            return 40 + (40 * self.level)
        elif cleared == 2:
            return 100 + (100 * self.level)
        elif cleared == 3:
            return 300 + (300 * self.level)
        elif cleared == 4:
            return 1200 + (1200 * self.level)
        else:
            return 0
```

Checking that landed dictionary updates are accurately mapped onto the board when it updates.



The screenshots above show the bottom left section of the board. Notice that as explained above the boundary has no internal representation in the grid data structure but is directly rendered on the board as it never changes and has no bearing on the game. However, it is considered when checking for collision under the *SRS* class.

Piece Generation Class:

```
class Piece_Gne:
    def __init__(self, bag):
        self.bag = bag
        self.start_ind = 0

    def generator_function(self):
        permu = list(permutations(self.bag))

        while True:
            for piece in random.choice(permu):
                yield piece

    def pop(self, buffer, gen):
        popped = buffer[self.start_ind]
        self.start_ind = (self.start_ind + 1) % len(buffer)
        buffer[self.start_ind] = next(gen)

        return popped

    def get_piece(self, landed):
        gen = self.generator_function()
        buffer = [next(gen) for _ in range(7)]

        popped = self.pop(buffer, gen)

        p = Piece(4, 0, popped)
        p.srs.update_field(landed)
        return p
```

Tetris!

```
class Tetris:
    def __init__(self):
        self.win = pygame.display.set_mode((width, height))
        # piece generation setup
        self.generate = Piece_Gne(['I', 'S', 'O', 'Z', 'T', 'L', 'J'])

        # game board setup
        self.landed = {}
        self.lines = 0
        self.score = 0
        self.board = Board(self.landed, self.lines, self.score)
        # self.collision = Collision()
        # self.collision.create_field(self.landed)
        self.tetrises = 0
        self.grid = self.board.create_grid()

        # get starting piece object
        self.current_piece = self.generate.get_piece(self.landed)

        # control parameters
        self.run = True
        self.show_piece = True
        self.held_piece = None
        self.change_piece = False
        self.hold_piece = False
        self.unhold_piece = False
        self.paused = False

        # get next piece
        self.next_piece = self.generate.get_piece(self.landed)

        # gravity setup
        self.fall_time = 0
        self.fall_speed = 0.3

        # game clock
        self.clock = pygame.time.Clock()

    def draw_window(self):  # pass instance of board
        pygame.font.init()

        font = pygame.font.Font(f, 15)

        pos_x = top_left_x + play_w
        pos_y = top_left_y + play_h // 2

        score = font.render(f'Score: {self.board.score}', True, BLACK)
        lines = font.render(f'Lines: {self.board.lines}', True, BLACK)
        level = font.render(f'Level: {self.board.level}', True, BLACK)
        tetrises = font.render(f'Tetrises: {self.tetrises}', True, BLACK)
        next_text = font.render('NEXT PIECE', True, BLACK)
        hold_text = font.render('HOLD PIECE', True, BLACK)
        bg = pygame.image.load("./images/xp.jpg", "background")

        self.win.blit(pygame.transform.scale(bg, (width, height)), (0, 0))
        game_texts = [score, lines, level, tetrises]

        for ind, t in enumerate(game_texts):
            self.win.blit(t, (pos_x - 200, pos_y + ind * 40))
```

```
            self.win.blit(next_text, (pos_x - 200, pos_y - 90))
            self.win.blit(hold_text, (pos_x - 90, pos_y - 90))

        with open('controls.txt', 'r') as file:
            for ind, line in enumerate(file.readlines()):
                c_r = font.render(line[:-1], True, BLACK)
                self.win.blit(c_r, (pos_x-10, pos_y+(20*ind)))

        self.board.show_next_piece(self.win, self.next_piece)
        self.board.render_grid(self.win, self.grid)

        if self.held_piece is not None:
            self.board.show_held_piece(self.win, self.held_piece)

        pygame.display.update()
```

```
    def lost(self):

        # if piece touches top of grid, it's a loss
        return any([pos[1] <= 2 for pos in self.landed])

    def change_state(self):
        # lock position
        for i in self.current_piece.current_position():
            self.landed[i] = self.current_piece.colour

        # clear rows
        cleared = self.board.clear_rows(self.grid)

        if cleared == 4:
            self.tetrises += 1

        # update game level
        self.board.level = self.lines // 10

        self.board.show_next_piece(self.win, self.next_piece)

        self.lines, self.score = self.board.lines, self.board.score

        self.next_piece.srs.update_field(self.landed)
        self.current_piece = self.next_piece

        if [WHITE] * 10 == self.grid[1] and [WHITE] * 10 == self.grid[0]
and [WHITE] * 10 == self.grid[2]:
            self.next_piece = self.generate.get_piece(self.landed)
            self.change_piece = False
```

```
    def game_logic(self):
        if self.paused:
            self.grid = self.board.paused_grid()
            self.show_piece = False
        else:
            self.grid = self.board.create_grid()
            self.show_piece = True

        if not self.paused:
            self.fall_time += self.clock.get_rawtime()
            self.clock.tick()


            if self.fall_time / 1000 > self.fall_speed:
                self.fall_time = 0

                if self.current_piece.make_move('down') is False:
                    self.board.score += 1
                    self.change_piece = True
                else:
                    pass

        self.win.fill(BG)

        pygame.display.set_caption('Tetris')

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.run = False
            if event.type == pygame.KEYDOWN:
                try:
                    if not self.paused:
self.current_piece.make_move(action_space[event.key])

                    #print(self.current_piece.get_config())

                    if action_space[event.key] == 'hold':
                        self.hold_piece = True if self.held_piece is None
else False

                    if action_space[event.key] == 'unhold':
                        self.unhold_piece = True if self.held_piece is not
None else False

                    if action_space[event.key] == 'pause':
                        self.paused = not self.paused

                    if action_space[event.key] == 'hd':
                        for _ in range(100):
                            self.current_piece.make_move('down')
                except KeyError:
                    print('WRONG KEY!')

        piece_positions = self.current_piece.current_position()

        # render on board
        if self.show_piece:
            for x, y in piece_positions:
                if (x, y) not in self.landed:
                    try:
                        self.grid[y][x] = self.current_piece.colour
```

```
                    except IndexError:
                        self.run = False

        if self.hold_piece:
            self.held_piece = self.current_piece
            self.current_piece = self.next_piece
            self.next_piece = self.generate.get_piece(self.landed)
            self.hold_piece = False

        if self.unhold_piece:
            self.held_piece.x, self.held_piece.y = self.current_piece.x,
self.current_piece.y
            self.current_piece = self.held_piece

            self.held_piece = None
            self.unhold_piece = False

        self.draw_window()

        if self.lost():
            self.run = False
```

```
tetris_game = Tetris()

while tetris_game.run:
    tetris_game.game_logic()

    if tetris_game.change_piece:
        tetris_game.change_state()
```

Main game loop

Tetris for Agent training (tetris_ai.py)

Most of the code below is the same as that in *tetris.py* however, there are some parts removed from the training version such as wall-kicks, gravity, piece holding capability, as well as some additions, such as a genome counter, and training statistics rendering on the screen. All procedures that are different from the human playable version file will be highlighted blue

```
import random
import pygame
from itertools import permutations
import numpy as np
```

Imports

```
# game assets

I = '....' \
    'xxxx' \
    '....' \
    '....'

S = '....' \
    '.xx.' \
    'xx..' \
    '....'

O = '....' \
    '.xx.' \
    '.xx.' \
    '....'

Z = '....' \
    'xx..' \
    '.xx.' \
    '....'

T = '....' \
    '.x..' \
    'xxx.' \
    '....'

L = '....' \
    '...x' \
    '.xxx' \
    '....'

J = '....' \
    'x...' \
    'xxx.' \
    '....'

CLOCKWISE_MATRIX = [[0, 1], [-1, 0]]
ANTICLOCKWISE_MATRIX = [[0, -1], [1, 0]]

pieces = {'I':I, 'S': S, 'O': O, 'Z': Z, 'T': T, 'L': L, 'J': J}
centres = {'I': (1, 1), 'S': (1, 2), 'O': (1, 1), 'Z': (1, 2), 'T': (1, 2),
'L': (2, 2), 'J': (1, 2)}

CYAN = (51, 255, 255)
BLUE = (0, 0, 255)
PURPLE = (225, 21, 132)
YELLOW = (255, 255, 100)
ORANGE = (255, 128, 0)
RED = (255, 0, 0)
GREEN = (51, 255, 51)
WHITE = (255, 255, 255)
GREY = (100, 100, 100)
BLACK = (0, 0, 0)
SPAWN_ROWS = (100, 140, 40) #(50, 100, 255)
BG = (128, 128, 128)

colours = {'I': CYAN, 'S': GREEN, 'O': YELLOW, 'Z': RED, 'T': PURPLE, 'L':
ORANGE, 'J': BLUE}
```

Game assets and global variables

```
RED = (255, 0, 0)
GREEN = (51, 255, 51)
WHITE = (255, 255, 255)
GREY = (100, 100, 100)
BLACK = (0, 0, 0)
SPAWN_ROWS = (100, 140, 40) #(50, 100, 255)
BG = (128, 128, 128)

colours = {'I': CYAN, 'S': GREEN, 'O': YELLOW, 'Z': RED, 'T': PURPLE, 'L':
ORANGE, 'J': BLUE}

action_space = {pygame.K_DOWN: 'down', pygame.K_RIGHT: 'right',
pygame.K_LEFT: 'left', pygame.K_UP: 'cw',
                pygame.K_w: 'ccw', pygame.K_TAB: 'hd', pygame.K_SPACE:
'hold', pygame.K_BACKSPACE: 'unhold', pygame.K_p:'pause'}

BOUNDARY = 1
ROWS = 20 + 2  # 2 extra rows for spawning
COLUMNS = 10

# dimensions
width = 1000
height = 650
block_size = 20
play_w = ROWS * block_size
play_h = COLUMNS * block_size

top_left_x = (width - play_w) // 2 - 200
top_left_y = height - play_h - 400

# font and music!
f = 'freesansbold.ttf'
```

Global variables

Width is higher compared with version above to allow space for the genome counter.

Super Rotation System Class

```
class SRS:
    def __init__(self, piece):  # piece is an object
        self.piece = piece
        self.rot_index = self.piece.rot_index
        self.clockwise = True
        self.boundary = [(row, col) for col in range(1) for row in
range(24)] + [(row, col) for col in range(12) for row in range(1)] + [(row,
col) for col in range(11, 12) for row in range(24)] + [(row, col) for col
in range(12) for row in range(23, 24)]
        self.field = [[0 for _ in range(COLUMNS + 2 * BOUNDARY)] for _ in
range(ROWS + 2 * BOUNDARY)]

    def update_field(self, landed):
        self.field = [[0 for _ in range(COLUMNS + 2 * BOUNDARY)] for _ in
range(ROWS + 2 * BOUNDARY)]
        # create landed positions
        if landed != {}:
            for (x, y) in landed.keys():
                try:
                    self.field[y+BOUNDARY][x+BOUNDARY] = 1
                except IndexError:
                    print('Error on landed piece placement')
                    pass

        # create boundary
        for x, y in self.boundary:
            try:
                self.field[x][y] = 1
            except IndexError:
                print('Boundary positions are wrong')

    def make_move(self, move):
        if move == 'cw' or move == 'ccw':
            if move == 'cw':
                state_cords = self.basic_rotation(True)
            else:
                state_cords = self.basic_rotation(False)

            if all([self.field[y + BOUNDARY][x + BOUNDARY] == 0 for x, y in
state_cords]):
                self.piece.state_cords = state_cords
                new_state = ['.' for _ in range(16)]

                for x, y in self.piece.state_cords:
                    ind = 4 * y + x
                    new_state[ind] = 'x'

                self.piece.state = ''.join(new_state)


                if move == 'cw':

                    self.piece.rot_index = (self.piece.rot_index+1) % 4
                else:
                    self.piece.rot_index = (self.piece.rot_index-1) % 4

        elif move == 'down':
            pos = self.piece.current_position()
```

```
            try:
                  return all([self.field[y + BOUNDARY + 1][x + BOUNDARY] ==
            0 for x, y in pos])
            except IndexError:
                print('WIll try again')
                pass

        elif move == 'right':
            pos = self.piece.current_position()

            try:
                return all([self.field[y + BOUNDARY][x + BOUNDARY + 1] == 0
for x, y in pos])
            except IndexError:
                print('Will try again')
                pass

        elif move == 'left':
            pos = self.piece.current_position()

            try:
                return all([self.field[y + BOUNDARY][x + BOUNDARY - 1] == 0
for x, y in pos])
            except IndexError:
                print('Will try again')
                pass

        return False

    def basic_rotation(self, clockwise):
        mat = None
        piece = [self.piece.state[i:i + 4] for i in range(0,
len(self.piece.state), 4)]
        c_x, c_y = self.piece.centre

        global_cords = []
        new_global_cords = []

        relative_cords = []
        new_relative_cords = []

        # get global cords
        for ind_x, row in enumerate(piece):
            for ind_y, col in enumerate(row):
                if col == 'x':
                    global_cords.append((ind_y, ind_x))

        # get relative cords to centre
        for x, y in global_cords:
            relative_cords.append((x - c_x, y - c_y))

        if clockwise:
            self.clockwise = True
            mat = CLOCKWISE_MATRIX
        if not clockwise:
            self.clockwise = False
            mat = ANTICLOCKWISE_MATRIX

        # calculate new relative cord to centre
        for cord in relative_cords:
            new_r_cord = np.dot(cord, mat)
```

```
    # get new global cords
    for x, y in new_relative_cords:
        new_global_cords.append((x + c_x, y + c_y))

    return new_global_cords
```

Piece Class

```
class Piece:
    def __init__(self, x=None, y=None, str_piece=None):
        self.x = x
        self.y = y
        self.str_id = str_piece
        self.piece = pieces[self.str_id]
        self.rot_index = 0
        self.state = self.piece
        self.clockwise = None
        self.all = [(j, i) for i in range(4) for j in range(4)]
        self.centre = centres[self.str_id]
        self.colour = colours[self.str_id] if self.piece is not None else
None
        self.state_cords = []

        piece = [self.state[i:i + 4] for i in range(0, len(self.state), 4)]
        for ind_x, row in enumerate(piece):
            for ind_y, col in enumerate(row):
                if col == 'x':
                    self.state_cords.append((ind_y, ind_x))
        self.srs = SRS(self)

    def make_move(self, move):
        if move == 'right':
            if self.srs.make_move('right'):
                self.x += 1
        elif move == 'left':
            if self.srs.make_move('left'):
                self.x -= 1
        elif move == 'down':
            if self.srs.make_move('down'):
                self.y += 1
            else:
                return False
        elif move == 'cw':
            if self.str_id == 'O':
                pass
            else:
                if self.str_id == 'I':
                    self.srs.make_move('ccw')
                else:
                    self.srs.make_move('cw')

        elif move == 'ccw':
            if self.str_id == 'O':
                pass
            else:
                self.srs.make_move('ccw')

    def current_position(self):  # get grid positions of a passed piece
object
        return [(r_x+self.x, r_y+self.y) for r_x, r_y in self.state_cords]

    def get_config(self):
        return self.rot_index, (self.x, self.y), self.current_position()
```

Board Class

```
class Board:
    def __init__(self, landed, lines, score):
        self.landed = landed
        self.score = score
        self.lines = lines
        self.level = 0

    def create_grid(self):
        GRID = [[WHITE for column in range(COLUMNS)] for row in
range(ROWS)]

        for i in range(ROWS):
            for j in range(COLUMNS):
                if (j, i) in self.landed:
                    # set colour if position is landed i.e there is a piece
there
                    GRID[i][j] = self.landed[(j, i)]

        return GRID

    @staticmethod
    def show_best_move(grid, best_move):
        if best_move:
            for x, y in best_move[2]:
                try:
                    grid[y][x] = GREY
                except IndexError:
                    print('Problem')

    @staticmethod
    def show_next_piece(surface, next_piece):
        pos_x = top_left_x + play_w - 220
        pos_y = top_left_y

        n_p = [next_piece.piece[i:i + 4] for i in range(0,
len(next_piece.piece), 4)]

        # next piece
        for ind_x, row in enumerate(n_p):
            for ind_y, column in enumerate(row):
                if column == 'x':
                    pygame.draw.rect(surface, next_piece.colour, (
                        pos_x + ind_y * block_size + 20, pos_y + ind_x *
block_size + 20, block_size, block_size), 0)
                    pygame.draw.rect(surface, BLACK, (
                        pos_x + ind_y * block_size + 20, pos_y + ind_x *
block_size + 20, block_size, block_size), 2)
```

```
    @staticmethod

    def show_progress(surface, genome_count, pop_size):
        bl_size = 12
        pos_x = top_left_x + 360
        pos_y = top_left_y
        group = 40

        for ind_x in range((pop_size//group) + 1):
            if pop_size < group:
                val = pop_size
            else:
                val = min(group, abs(pop_size-group*ind_x))

            for ind_y in range(val):
                if group*ind_x + ind_y+1 < genome_count:
                    pygame.draw.rect(surface, ORANGE,
                                     (pos_x + (ind_y * bl_size), pos_y +
(ind_x * bl_size), bl_size, bl_size), 0)
                elif group*ind_x + ind_y+1 == genome_count:
                    pygame.draw.rect(surface, GREEN,
                                     (pos_x + (ind_y * bl_size), pos_y +
(ind_x * bl_size), bl_size, bl_size), 0)
                else:
                    pygame.draw.rect(surface, GREY,
                                     (pos_x + (ind_y * bl_size), pos_y +
(ind_x * bl_size), bl_size, bl_size), 0)
                pygame.draw.rect(surface, BLACK,
                                 (pos_x + (ind_y * bl_size), pos_y + (ind_x
* bl_size), bl_size, bl_size), 1)
```

```
    @staticmethod
    def render_grid(surface, grid):
        # boundary
        for i in range(ROWS + BOUNDARY + 1):
            for j in range(COLUMNS + BOUNDARY + 1):
                pygame.draw.rect(surface, GREY, (top_left_x -
      (BOUNDARY * block_size) + j * block_size,top_left_y - (BOUNDARY *
      block_size) + i * block_size, block_size, block_size), 0)
                pygame.draw.rect(surface, BLACK, (top_left_x - (BOUNDARY *
block_size) + j * block_size, top_left_y - (BOUNDARY * block_size) + i *
block_size, block_size, block_size), 2)

        # convert grid colours to output onto surface
        for ind_x, rows in enumerate(grid):
            for ind_y, colour in enumerate(rows):
                pygame.draw.rect(surface, colour, (
                    top_left_x + (ind_y * block_size), top_left_y + (ind_x
* block_size), block_size, block_size), 0)

        # draw black boundaries
        for i in range(ROWS):
            for j in range(COLUMNS):
                if i == 0 or i == 1:  # first 2 rows are for spawing
                    pygame.draw.rect(surface, SPAWN_ROWS,(top_left_x + j *
block_size, top_left_y + i * block_size, block_size, block_size),0)
                    pygame.draw.rect(surface, WHITE,(top_left_x + j *
block_size, top_left_y + i * block_size, block_size, block_size),2)
                if grid[i][j] != WHITE:
                    pygame.draw.rect(surface, BLACK, (top_left_x + j *
block_size, top_left_y + i * block_size, block_size, block_size),2)
```

```
    def clear_rows(self, grid):
        cleared_row = 0
        cleared_rows = 0

        for index in range(ROWS - 1, -1, -1):
            if WHITE not in grid[index]:
                cleared_rows += 1
                cleared_row = index
                for column in range(COLUMNS):
                    try:
                        del self.landed[(column, cleared_row)]  # deletes
colours off cleared rows
                    except KeyError:
                        pass

        # sort all landed positions based on the rows in the grid, then
reverse that as we are
        # searching grid from below

        for position in sorted(list(self.landed), key=lambda pos: pos[1],
reverse=True):
            col, row = position

            if row < cleared_row:  # if row is above index of row that was
cleared:
                new_pos = (col, row + cleared_rows)  # make new position
that moves itm down by number of cleared rows

                self.landed[new_pos] = self.landed.pop(
                    position)  # .pop here removes colours from all rows
above, and places them in their new positions

        self.lines += cleared_rows
        self.score += self.score_game(cleared_rows)

        return cleared_rows

    def score_game(self, cleared):
        if cleared == 1:
            return 40 + (40 * self.level)
        elif cleared == 2:
            return 100 + (100 * self.level)
        elif cleared == 3:
            return 300 + (300 * self.level)
        elif cleared == 4:
            return 1200 + (1200 * self.level)
        else:
            return 0
```

Piece Generation Class

```
class Piece_Gne:
    def __init__(self, bag):
        self.bag = bag
        self.start_ind = 0

    def generator_function(self):
        permu = list(permutations(self.bag))

        while True:
            for piece in random.choice(permu):
                yield piece

    def pop(self, buffer, gen):
        popped = buffer[self.start_ind]
        self.start_ind = (self.start_ind + 1) % len(buffer)
        buffer[self.start_ind] = next(gen)

        return popped

    def get_piece(self, landed):
        gen = self.generator_function()
        buffer = [next(gen) for _ in range(7)]

        popped = self.pop(buffer, gen)

        p = Piece(4, -1, popped)
        p.srs.update_field(landed)
        return p
```

Tetris Class

```
class Tetris:
    current_gen = 0
    genome_count = 0
    pop_size = 0
    h_score = 0
    best_fitness = 0
    av_fitness = 0
    def __init__(self):
        self.win = pygame.display.set_mode((width, height))
        # piece generation setup
        self.generate = Piece_Gne(['I', 'S', 'O', 'Z', 'T', 'L', 'J'])

        # game board setup
        self.landed = {}
        self.lines = 0
        self.score = 0
        self.board = Board(self.landed, self.lines, self.score)
        self.tetrises = 0
        self.grid = self.board.create_grid()

        # get starting piece object
        self.current_piece = self.generate.get_piece(self.landed)

        # control parameters
        self.run = True
        self.show_piece = True
        self.change_piece = False

        # get next piece
        self.next_piece = self.generate.get_piece(self.landed)

        # gravity setup
        self.fall_time = 0
        self.fall_speed = 0.3

        # game clock
        self.clock = pygame.time.Clock()

        # AI
        self.best_move = None
```

```
    def draw_window(self):
        pygame.font.init()

        font = pygame.font.Font(f, 15)

        pos_x = top_left_x + play_w
        pos_y = top_left_y + play_h // 2

        score = font.render(f'Score: {self.board.score}', True, BLACK)
        lines = font.render(f'Lines: {self.board.lines}', True, BLACK)
        tetrises = font.render(f'Level: {self.tetrises}', True, BLACK)
        next_text = font.render('NEXT PIECE', True, BLACK)

        # AI stuff
        gen = font.render(f'Generation : {Tetris.current_gen}', True,
BLACK)
        h_score = font.render(f'Highscore: {Tetris.h_score}', True, BLACK)
        bf = font.render(f'Best fitness: {Tetris.best_fitness}', True,
BLACK)
        avf = font.render(f'Average Fitness: {Tetris.av_fitness}', True,
BLACK)

        self.win.blit(next_text, (pos_x-200, pos_y-80))

        game_texts = [score, lines, tetrises]
        ai_texts = [gen, h_score, bf, avf]

        for ind, t in enumerate(game_texts):
            self.win.blit(t, (pos_x - 200, pos_y + ind*40))

        for ind, t in enumerate(ai_texts):
            self.win.blit(t, (pos_x-60, top_left_y +
(Tetris.pop_size//40)*15 + ind*30 + 20))

        self.board.show_next_piece(self.win, self.next_piece)
        self.board.render_grid(self.win, self.grid)
        self.board.show_progress(self.win, Tetris.genome_count,
Tetris.pop_size)

        pygame.display.update()

    @staticmethod
    def set_genome_progress(genome_count, gen, pop_size, h_score,
best_fitness, av_fitness):
        Tetris.current_gen = gen
        Tetris.genome_count = genome_count
        Tetris.pop_size = pop_size
        Tetris.h_score = h_score
        Tetris.best_fitness = best_fitness
        Tetris.av_fitness = av_fitness

    def lost(self):
        # if piece touches top of grid, it's a loss
        return any([pos[1] <= 2 for pos in self.landed])
```

```
    def change_state(self):
        # clear rows
        cleared = self.board.clear_rows(self.grid)

        if cleared == 4:
            self.tetrises += 1

        # update game level
        self.board.level = self.lines // 10

        self.board.show_next_piece(self.win, self.next_piece)

        self.lines, self.score = self.board.lines, self.board.score

        self.next_piece.srs.update_field(self.landed)
        self.current_piece = self.next_piece

        self.next_piece = self.generate.get_piece(self.landed)
        self.change_piece = False
```

```
    def game_logic(self):
        self.grid = self.board.create_grid()

        self.win.fill(BG)

        pygame.display.set_caption('Tetris')

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.run = False

        piece_positions = self.current_piece.current_position()

        # render on board
        if self.show_piece:
            for x, y in piece_positions:
                if (x, y) not in self.landed:
                    try:
                        self.grid[y][x] = self.current_piece.colour
                    except IndexError:
                        self.run = False

        self.draw_window()

        if self.lost():
            self.run = False
```

```
    def fitness_func(self):
        return self.tetrises*50 + self.score*5 + 10

    def make_ai_move(self):
        target_config = self.best_move  # exp: (0, (7, 21), [(6, 19), (7,
19), (7, 20), (7, 21)])

        for i in target_config[2]:
            self.landed[i] = self.current_piece.colour

        self.change_piece = True
```

Neural network and genetic algorithm (neuralnet.py)

The code below contains a class encapsulating functionality for a neural network. This code is written in raw Python without many Machine Learning libraries such as Pytorch. The implementation for the neural network is a repurposed class from an earlier project I worked on which was using a neural network to identify hand-written digits, following from Tariq Rashid's book (Make your own Neural Network). The code also contains a class encapsulating functionality for a population of neural networks, which involves crossover and mutation implementations; the genetic algorithm part.

```
import numpy as np
import scipy.special
import pickle
```

Imports

Neural network Class

```
class Nueral_net:
    def __init__(self, input_nodes, hidden_a_nodes, hidden_b_nodes,
output_nodes):
        self.inodes = input_nodes
        self.hnodes_a = hidden_a_nodes
        self.hnodes_b = hidden_b_nodes
        self.onodes = output_nodes

        self.activation_func = lambda x : scipy.special.expit(x)

        # weight matrix between input and hidden
        self.wi_ha = np.random.normal(0.0, pow(self.inodes, -0.5),
(self.hnodes_a, self.inodes))
        self.wha_hb = np.random.normal(0.0, pow(self.hnodes_a, -0.5),
(self.hnodes_b, self.hnodes_a))
        self.whb_o = np.random.normal(0.0, pow(self.hnodes_b, -0.5),
(self.onodes, self.hnodes_b))
```

```
    def query(self, input_list):
        inputs = np.array(input_list, ndmin=2).T

        hidden_a_inputs = np.matmul(self.wi_ha, inputs)
        hidden_a_outputs = self.activation_func(hidden_a_inputs)
        hidden_b_inputs = np.matmul(self.wha_hb, hidden_a_outputs)
        hidden_b_outputs = self.activation_func(hidden_b_inputs)

        final_inputs = np.matmul(self.whb_o, hidden_b_outputs)
        final_outputs = self.activation_func(final_inputs)

        return final_outputs



hidden_a_size = 8
hidden_b_size = 8
output_size = 1

# genetic algorithm parameters
mutation_prob = 0.08
elitism = 0.2
mutation_power= 0.1
```

Population Class

```
class Population:
    def __init__(self, size, old_population, input_size):
        self.size = size
        self.fitnesses = np.zeros(self.size)
        self.input_size = input_size
        # at the start of the game, there's no old population
        if old_population is None:
            # set up population of nueral nets
            self.models = [Nueral_net(self.input_size, hidden_a_size,
hidden_b_size, output_size) for _ in range(self.size)]
        else:
            # get all nueral nets from previous iteration
            self.old_models = old_population.models
            self.old_fitnesses = old_population.fitnesses
            # setup models for this iteration, will fill list in mutation
and crossover phase
            self.models = []
            self.crossover()
            self.mutate()


    def crossover(self):
        print('Crossover Process')
        # setup higher probabilities for higher performing neural nets
        sum_of_fitnesses = np.sum(self.old_fitnesses)
        probs = [self.old_fitnesses[i]/sum_of_fitnesses for i in
range(self.size)]

        for i in range(self.size):
            if i < self.size*elitism:
```

```
                # sort by order of fitness (descending)
                fitness_indices = np.argsort(probs)[::-1]
                # if model within elitism critical region, select it as is
                child = self.old_models[fitness_indices[i]]
            else:
                # select 2 best performing fitness indices using probs, do
not replace,
                # so indices cannot be selected twice
                a, b = np.random.choice(self.size, size=2, p=probs,
replace=False)

                # setup models from those indices
                parent_a, parent_b = self.old_models[a], self.old_models[b]
                child = Nueral_net(self.input_size, hidden_a_size,
hidden_b_size, output_size)
                a_fitness, b_fitness = self.old_fitnesses[a],
self.old_fitnesses[b]

                if a_fitness == 0 and b_fitness == 0:
                    child.wi_ha = np.zeros((hidden_a_size,
self.input_size))
                    child.wha_hb = np.zeros((hidden_b_size, hidden_a_size))
                    child.whb_o = np.zeros((output_size, hidden_b_size))

                    prob_from_a = 0.5

                    # cross-over weights between inputs and hidden a
                    for row_ind, row in enumerate(child.wi_ha):
                        for col_ind, col in enumerate(row):
                            if np.random.random() < prob_from_a:
                                child.wi_ha[row_ind][col_ind] =
parent_a.wi_ha[row_ind][col_ind]
                            else:
                                child.wi_ha[row_ind][col_ind] =
parent_b.wi_ha[row_ind][col_ind]

                    # cross-over weights between hidden a and hidden b
                    for row_ind, row in enumerate(child.wha_hb):
                        for col_ind, col in enumerate(row):
                            if np.random.random() < prob_from_a:
                                child.wha_hb[row_ind][col_ind] =
parent_a.wha_hb[row_ind][col_ind]
                            else:
                                child.wha_hb[row_ind][col_ind] =
parent_b.wha_hb[row_ind][col_ind]

                    # cross-over weights between hidden b and output
                    for row_ind, row in enumerate(child.whb_o):
                        for col_ind, col in enumerate(row):
                            if np.random.random() < prob_from_a:
                                child.whb_o[row_ind][col_ind] =
parent_a.whb_o[row_ind][col_ind]
                            else:
                                child.whb_o[row_ind][col_ind] =
parent_b.whb_o[row_ind][col_ind]

                else:
                    child.wi_ha =
(a_fitness/(a_fitness+b_fitness))*parent_a.wi_ha +
(b_fitness/(a_fitness+b_fitness))*parent_b.wi_ha
```

```
                    child.wha_hb =
                (a_fitness/(a_fitness+b_fitness))*parent_a.wha_hb +
                (b_fitness/(a_fitness+b_fitness))*parent_b.wha_hb
                    child.whb_o =
                (a_fitness/(a_fitness+b_fitness))*parent_a.whb_o +
                (b_fitness/(a_fitness+b_fitness))*parent_b.whb_o

            # add new object to population
            self.models.append(child)



    def mutate(self):
        print('Mutation Process')
        for model in self.models:
            for row in model.wi_ha:
                for ind, col in enumerate(row):
                    if np.random.random() < mutation_prob:
                        row[ind] += np.random.uniform(-mutation_power,
mutation_power)

            for row in model.wha_hb:
                for ind, col in enumerate(row):
                    if np.random.random() < mutation_prob:
                        row[ind] += np.random.uniform(-mutation_power,
mutation_power)

            for row in model.whb_o:
                for ind, col in enumerate(row):
                    if np.random.random() < mutation_prob:
                        row[ind] += np.random.uniform(-mutation_power,
mutation_power)

    def save_population(self, epoch_number):
        weight_matrices = []

        for model in self.models:
            weight_matrices.append((model.wi_ha, model.wha_hb,
model.whb_o))

        path1 = f"./populations/{epoch_number+1}population.pkl"
        with open(path1, "wb") as f:
            pickle.dump(weight_matrices, f)

        path2 = f"./populations/{epoch_number+1}fitness.pkl"

        np.save(path2, self.fitnesses, allow_pickle=True)
```

Heuristics file (heuristics.py)

This code contains a class which receives a board state and returns a list of heuristics that describe the state quantitatively, in such as a way that useful information from the state is encoded accurately for the neural network to determine a best move in that state

```
import numpy as np
```

Imports

Heuristics Class

```
class Heuristics:
    def __init__(self, columns, rows):
        self.width = columns
        self.height = rows
        self.field = [[0 for _ in range(self.width)] for _ in
range(self.height)]

    def update_field(self, field):  # get new field with updated 1s
        self.field = field

    # height of one column
    def column_height(self, column):
        for row in range(self.height):
            if self.field[row][column] == 1:
                return self.height - row
        return 0

    # max height
    def max_height(self):
        return max([self.column_height(col) for col in range(self.width)])

    def min_height(self):
        return min([self.column_height(col) for col in range(self.width)])

    # number of holes in a column
    def column_holes(self, column):
        col_height = self.column_height(column)
        holes = 0

        for row in range(self.height - 1, -1, -1):
            if self.field[row][column] == 0 and (self.height - row) <
col_height:
                holes += 1

        return holes

    # number of holes in all columns
    def total_holes(self):
        return sum([self.column_holes(col) for col in range(self.width)])
```

```
    # bumpiness of terrain
    def bumpiness(self):
        total = 0
        for col in range(self.width-1):
            total += abs(self.column_height(col)-self.column_height(col+1))

        return total

    # std dev of heights
    def std_heights(self):
        heights = [self.column_height(col) for col in range(self.width)]
        return np.std(heights)

    # number of pits
    def pits(self):
        pits = 0
        for col in range(self.width):
            for row in range(self.height):
                if all([self.field[row][col] == 0]):
                    pits += 1

        return pits

    # row transitions
    def row_transitions(self):
        transitions = 0
        for row in self.field:
            if 1 in row:
                for col in range(self.width-1):
                    if (row[col] == 0 and row[col+1] == 1) or (row[col] ==
1 and row[col+1] == 0):
                        transitions += 1
        return transitions

    # col transitions
    def col_transitions(self):
        transitions = 0

        for col in range(self.width):
            for row in range(self.height-1):
                if (self.field[row][col] == 1 and self.field[row+1][col] ==
0) or (self.field[row][col] == 0 and self.field[row+1][col] == 1):
                    transitions += 1
        return transitions

    def total_height(self):
        return sum([self.column_height(i) for i in range(self.width)])
```

```
    # deepest well
    def deepest_well(self):
        depths = []
        for col in range(self.width):
            if col == 0:
                possible_depth = self.column_height(col+1) -
self.column_height(col)
                depths.append(possible_depth) if possible_depth > 0 else
depths.append(0)
            elif col == self.width-1:
                possible_depth = self.column_height(col-1) -
self.column_height(col)
                depths.append(possible_depth) if possible_depth > 0 else 0
            else:
                pl = self.column_height(col-1) - self.column_height(col)
                possible_depth_left = pl if pl > 0 else 0
                pr = self.column_height(col+1) - self.column_height(col)
                possible_depth_right = pr if pr > 0 else 0
                depths.append(possible_depth_right) if possible_depth_right
>= possible_depth_left else depths.append(possible_depth_left)

        return max(depths)

    # lines cleared
    def lines_cleared(self):
        lines = 0

        for row in self.field:
            if 0 not in row:
                lines += 1

        return lines

    def print_num_grid(self):
        print(' NEW FRAME .....................................')
        for i in self.field:
            print(i)

    def get_heuristics(self):
        return [self.deepest_well(), self.total_height(),
self.total_holes(), self.bumpiness(), self.lines_cleared(),
self.row_transitions(), self.std_heights(), self.pits(),
self.col_transitions()]
```

Graph plotting (visualize.py)

The matplotlib module is used to plot graphs from data recorded while the agents train. The code here contains a class that receives the data after training and passes it as arguments to matplotlib functions which then plot graphs.

```
import matplotlib.pyplot as plt
from matplotlib import style
```

Imports

Visualisation Class

```
class Visualize:
    def __init__(self, epochs_list, av_fitnesses, all_fitnesses, av_scores,
all_scores):
        self.epochs = epochs_list
        self.afl = av_fitnesses
        self.asl = av_scores
        self.sl = all_scores
        self.fl = all_fitnesses

    def visualize(self):
        style.use('ggplot')

        max_fitnesses = []
        for ind, e in enumerate(self.epochs):
            max_fitnesses.append(max(self.fl[ind]))
            plt.scatter([e for _ in range(1000)], self.fl[ind],
color='lightskyblue')

        plt.plot(self.epochs, max_fitnesses, color='red',label='Max fitness
per epoch',marker='.')

        plt.plot(self.epochs, self.afl, marker=".", label='Average fitness
per epoch',color='gold')
        plt.xlabel('Epochs')
        plt.ylabel('Fitness')
        plt.legend()
        plt.show()

        plt.plot(self.epochs, self.afl, marker=".", label='Average fitness
per epoch', color='blue')
        plt.xlabel('Epochs')
        plt.ylabel('Fitness')
        plt.legend()
        plt.show()

        max_scores = []
        for ind, e in enumerate(self.epochs):
            max_scores.append(max(self.sl[ind]))
            plt.scatter([e for _ in range(1000)], self.sl[ind],
color='lightseagreen')

        plt.plot(self.epochs, max_scores, color='red',label='Max score per
epoch',marker='.')

        plt.plot(self.epochs, self.asl, marker=".", label='Average score
per epoch',color='gold')
        plt.xlabel('Epochs')
        plt.ylabel('Score')
        plt.legend()
        plt.show()

        plt.plot(self.epochs, self.asl, marker=".", label='Average score
per epoch', color='blue')
        plt.xlabel('Epochs')
        plt.ylabel('Score')
        plt.legend()
        plt.show()
```

The Agent and how it is trained (train_agent_scratch.py)

```
import tetris_ai
import pickle
from nueralnet import Population
from visualize import Visualize
from heuristics import Heuristics
from time import process_time
import numpy as np
```

Imports

Agent Class

```
class AI_Agent:
    def __init__(self):  # piece is an object
        self.rows = tetris_ai.ROWS
        self.columns = tetris_ai.COLUMNS
        self.field = [[0 for _ in range(self.columns)] for _ in
range(self.rows)]
        self.landed = None
        self.heuris = Heuristics(self.columns, self.rows)  # this is a
heuristics obj
        self.best_move = None
        self.all_pieces = ['I', 'S', 'O', 'Z', 'T', 'L', 'J']
        self.all_configurations = []
        self.next_configurations = []
        self.next_state = None
        self.next_piece = None
        self.nueral_net = None
        self.ablation = None

    def get_possible_configurations(self, piece, field, is_next_piece):
        positions = [[(ind_y, ind_x) for ind_y in range(self.columns) if
field[ind_x][ind_y] == 0] for ind_x in range(self.rows)]
        all_positions = [tupl for li in positions for tupl in li]

        data_obj = Data(piece.str_id, None)
        all_configurations = []

        for pos_x in range(-3, self.columns+3):
            possible = 0

            for ind in range(4):
                data_obj.rot_index = ind
                ascii_cords = data_obj.get_data()

                abb_y = 2
                if not all([(pos_x + x, abb_y + y) in all_positions for x,
y in ascii_cords]):
                    possible += 0
                else:
```

```
                    possible += 1

            if possible != 0:
                for index in range(4):
                    data_obj.rot_index = index
                    ascii_cords = data_obj.get_data()
                    pos_y = 0
                    done = False

                    while not done:
                        if not all([(pos_x + x, pos_y + 1 + y) in
all_positions for x, y in ascii_cords]):
                            final_global_cords = [(x + pos_x, y + pos_y)
for x, y in ascii_cords]
                            # check validity of rotation states
                            if all([cord in all_positions for cord in
final_global_cords]):
                                all_configurations.append(((pos_x, pos_y),
index, final_global_cords))

                            done = True

                        elif not all([(pos_x + x, pos_y + y) in
all_positions for x, y in ascii_cords]):
                            final_global_cords = [(x + pos_x, y + pos_y -
1) for x, y in ascii_cords]
                            # check validity of rotation states
                            if all([cord in all_positions for cord in
final_global_cords]):
                                all_configurations.append(((pos_x, pos_y -
1), index, final_global_cords))

                            done = True

                        else:
                            pos_y += 1
            else:
                continue

        if not is_next_piece:
            self.all_configurations = all_configurations
        else:
            self.next_configurations = all_configurations



    def update_agent(self, current_piece, next_piece, landed):
        self.landed = landed
        self.update_field()
        self.get_possible_configurations(current_piece, self.field, False)
        self.next_piece = next_piece
```

```
    def update_field(self):
        self.field = [[0 for _ in range(self.columns)] for _ in
range(self.rows)]
        if self.landed != {}:
            for pos, val in self.landed.items():
                if val != (255, 255, 255):
                    x, y = pos
                    try:
                        self.field[y][x] = 1
                    except IndexError:
                        print('random index error is random')
                        pass

    def next_piece_knowledge(self):
        score_next_moves = []

        self.get_possible_configurations(self.next_piece, self.next_state,
True)

        if len(self.next_configurations) != 0:
            for next_cord, next_ind, next_positions in
self.next_configurations:
                for x, y in next_positions:
                    self.next_state[y][x] = 1

                self.heuris.update_field(self.next_state)
                board_state = self.heuris.get_heuristics()
                board_state_2 = [board_state[k] for k in
range(len(board_state)) if k not in self.ablation]
                next_move_score = self.nueral_net.query(board_state_2)[0]

                for x, y in next_positions:
                    self.next_state[y][x] = 0

                score_next_moves.append(next_move_score)

            return max(score_next_moves)
        else:
            return 0


    def get_best_move(self, current_piece):
        self.evaluation_function(current_piece)

        return self.best_move


    def print_field(self):
        for i in self.field:
            print(i)
```

```
    def evaluation_function(self, current_piece):
        if len(self.all_configurations) != 0:
            score_moves = []

            for cord, index, positions in self.all_configurations:
                move_score = 0
                for x, y in positions:
                    self.field[y][x] = 1

                self.next_state = self.field

                move_score += self.next_piece_knowledge()

                self.heuris.update_field(self.field)

                board_state = self.heuris.get_heuristics()

                board_state_2 = [board_state[k] for k in
range(len(board_state)) if k not in self.ablation]

                move_score += self.nueral_net.query(board_state_2)[0]

                for x, y in positions:
                    self.field[y][x] = 0

                score_moves.append((index, cord, positions, move_score))

            best_move = max(score_moves,  key= lambda x: x[-1])

            self.best_move = best_move[:-1]

        else:
            self.best_move = current_piece.get_config()
```

Rotation data Class

```
class Data:
    def __init__(self, piece_id, test_rot_index):
        self.str_id = piece_id  # its a string
        self.rot_index = test_rot_index

    def O(self):
        O = {0: [(1, 1), (2, 1), (1, 2), (2, 2)],
             1: [(1, 1), (2, 1), (1, 2), (2, 2)],
             2: [(1, 1), (2, 1), (1, 2), (2, 2)],
             3: [(1, 1), (2, 1), (1, 2), (2, 2)]
             }
        return O[self.rot_index]

    def T(self):
        T = {0: [(0, 2), (1, 2), (1, 1), (2, 2)],
             1: [(1, 1), (1, 2), (1, 3), (2, 2)],
             2: [(0, 2), (1, 2), (2, 2), (1, 3)],
             3: [(1, 1), (1, 2), (1, 3), (0, 2)]
             }
        return T[self.rot_index]
```

```
def I(self):
    I = {0: [(0, 1), (1, 1), (2, 1), (3, 1)],
         1: [(1, 0), (1, 1), (1, 2), (1, 3)],
         2: [(0, 1), (1, 1), (2, 1), (3, 1)],
         3: [(1, 0), (1, 1), (1, 2), (1, 3)]
         }
    return I[self.rot_index]

def L(self):
    L = {0: [(1, 2), (2, 2), (3, 2), (3, 1)],
         1: [(2, 1), (2, 2), (2, 3), (3, 3)],
         2: [(1, 2), (2, 2), (1, 3), (3, 2)],
         3: [(1, 1), (2, 2), (2, 1), (2, 3)]
         }
    return L[self.rot_index]

def Z(self):
    Z = {0: [(0, 1), (1, 2), (1, 1), (2, 2)],
         1: [(2, 1), (1, 2), (2, 2), (1, 3)],
         2: [(0, 2), (1, 2), (1, 3), (2, 3)],
         3: [(0, 2), (1, 2), (1, 1), (0, 3)]
         }
    return Z[self.rot_index]

def J(self):
    J = {0: [(0, 1), (1, 2), (0, 2), (2, 2)],
         1: [(1, 1), (1, 2), (1, 3), (2, 1)],
         2: [(0, 2), (1, 2), (2, 2), (2, 3)],
         3: [(1, 1), (1, 2), (1, 3), (0, 3)]
         }
    return J[self.rot_index]

def S(self):
    S = {0: [(0, 2), (1, 2), (1, 1), (2, 1)],
         1: [(1, 1), (1, 2), (2, 2), (2, 3)],
         2: [(1, 1), (1, 2), (2, 2), (2, 3)],
         3: [(1, 3), (1, 2), (0, 3), (2, 2)]
         }
    return S[self.rot_index]

def get_data(self):
    if self.str_id == 'O':
        return self.O()
    elif self.str_id == 'I':
        return self.I()
    elif self.str_id == 'Z':
        return self.Z()
    elif self.str_id == 'S':
        return self.S()
    elif self.str_id == 'T':
        return self.T()
    elif self.str_id == 'L':
        return self.L()
    elif self.str_id == 'J':
        return self.J()
```

Training Class

```
class Trainer:
    def __init__(self):
        self.agent = AI_Agent()
        self.record = 0
        self.new_pop = None
        self.old_pop = None
        self.epochs = 10
        self.checkpoint = 2
        self.epoch_data = {}
        self.population_size = 1000

    def eval(self, load_population, epoch_number, ablation):

        if ablation is None:
            li = []
        else:
            li = [int(j) for j in ablation.split(',')]

        if load_population:
            try:
                self.old_pop = Population(self.population_size, None, 9)

                path1 = f"./populations/{epoch_number}population.pkl"
                with open(path1, "rb") as f:
                    weight_matrices = pickle.load(f)

                for ind, model in enumerate(self.old_pop.models):
                    model.wi_ha = weight_matrices[ind][0]
                    model.wha_hb = weight_matrices[ind][1]
                    model.whb_o = weight_matrices[ind][2]

                path2 = f"./populations/{epoch_number}fitness.pkl"

                self.old_pop.fitnesses = np.load(path2, allow_pickle=True)

            except FileNotFoundError or FileExistsError:
                print('File not found, or it does not exist')

        for epoch in range(self.epochs):
            print('_____________________________________________')
            scores = []
            self.new_pop = Population(self.population_size, self.old_pop,
9-len(li))
            print(f'EPOCH: {epoch+1}')
            print(f'HIGHSCORE: {self.record}')
            for neural_index in range(self.new_pop.size):
                current_fitness = 0

                tetris_game = tetris_ai.Tetris()

                self.agent = AI_Agent()
                self.agent.ablation = li
                self.agent.nueral_net = self.new_pop.models[neural_index]

                while tetris_game.run:
                    tetris_game.game_logic()
                    # update the agent with useful info to find the best
move
```

```
                    self.agent.update_agent(tetris_game.current_piece,
                tetris_game.next_piece, tetris_game.landed)

                    tetris_game.best_move =
self.agent.get_best_move(tetris_game.current_piece)

                    # make the move
                    tetris_game.make_ai_move()

                    current_fitness += tetris_game.fitness_func()

                    if tetris_game.change_piece:
                        tetris_game.change_state()

                    if not tetris_game.run:
                        self.new_pop.fitnesses[neural_index] =
current_fitness
                        break

                    tetris_game.set_genome_progress(neural_index+1, epoch +
1, self.new_pop.size, self.record, max(self.new_pop.fitnesses),

sum(self.new_pop.fitnesses) / self.new_pop.size)

                scores.append(tetris_game.score)
                if tetris_game.score > self.record:
                    self.record = tetris_game.score
                    print(f'HIGHSCORE: {self.record}')

            self.epoch_data[epoch+1] =
(np.sum(self.new_pop.fitnesses)/self.population_size,
list(self.new_pop.fitnesses), sum(scores)/self.population_size, scores)
            self.old_pop = self.new_pop

            if (epoch+1) % self.checkpoint == 0:
                print('Saving models//////.....')
                self.old_pop.save_population(epoch)
                print('Saved successfully/////////////////')

            print(f'Best fitness: {max(self.epoch_data[epoch+1][1])}')
            print(f'Average fitness: {self.epoch_data[epoch+1][0]}')
            print(f'Average score :
{sum(self.epoch_data[epoch+1][3])/self.population_size}')
```

```
if __name__ == '__main__':
    trainer = Trainer()

    print('_______ Ablation _______')
    print("0 -> Deepest well\n1 -> Total height\n2 -> Total holes\n3-> 
Bumpiness\n4 -> Lines Cleared\n5 -> Row transitions\n6 -> Standard 
deviation of heights\n7 -> Number of pits\n8 -> Column transitions")
    print('Input should be between 0 and 8, follow list above ¯\_(ツ)_/¯')
    i = input('Type one number to index to the heuristic you want to remove 
or type a series of numbers separated by commas, Enter not to ablate: ')

    epochs = []
    av_fitnesses = []
    all_fitnesses = []
    av_scores = []
    all_scores = []

    if i:
        start_time = process_time()

        trainer.eval(False, 0, i)

        end_time = process_time()

        print(f'Training time(hrs): {(end_time-start_time)//3600}')

        for e, data in trainer.epoch_data.items():
            epochs.append(e)
            av_fitnesses.append(data[0])
            all_fitnesses.append(data[1])
            av_scores.append(data[2])
            all_scores.append(data[3])

        # in case visualisation doesn't work automatically, all data is 
printed to console, so
        # it can be passed manually into a visualisation object
        print(f'Epochs:{epochs},\n\nAverage 
fitnesses\n{av_fitnesses},\n\nAll fitnesses\n{all_fitnesses},\n\nAverage 
scores\n{av_scores},\nAll scores{all_scores}')

        vis = Visualize(epochs, av_fitnesses, all_fitnesses, av_scores, 
all_scores)
        vis.visualize()

    else:
        print('_____________________________________')
        load = input('LOAD POPULATION(L): Enter not to load ')
        if load == 'L':
            start_time = process_time()

            trainer.eval(True, int(input('From which epoch(2,4,6,8,10): 
')), None)

            end_time = process_time()

            print(f'Training time(hrs): {(end_time-start_time)//3600}')

            for e, data in trainer.epoch_data.items():
                epochs.append(e)
                av_fitnesses.append(data[0])
                all_fitnesses.append(data[1])
```

```
                av_scores.append(data[2])
                all_scores.append(data[3])

            # in case visualisation doesn't work automatically, all data is
printed to console, so
            # it can be passed manually into a visualisation object
            print(f'Epochs:{epochs},\n\nAverage
fitnesses\n{av_fitnesses},\n\nAll fitnesses\n{all_fitnesses},\n\nAverage
scores\n{av_scores},\nAll scores{all_scores}')

            vis = Visualize(epochs, av_fitnesses, all_fitnesses, av_scores,
all_scores)
            vis.visualize()
        else:
            start_time = process_time()
            trainer.eval(False, 0, None)
            end_time = process_time()

            print(f'Training time(hrs): {(end_time-start_time)//3600}')

            for e, data in trainer.epoch_data.items():
                epochs.append(e)
                av_fitnesses.append(data[0])
                all_fitnesses.append(data[1])
                av_scores.append(data[2])
                all_scores.append(data[3])

            # in case visualisation doesn't work automatically, all data is
printed to console, so
            # it can be passed manually into a visualisation object
            print(f'Epochs:{epochs},\n\nAverage
fitnesses\n{av_fitnesses},\n\nAll fitnesses\n{all_fitnesses},\n\nAverage
scores\n{av_scores},\nAll scores{all_scores}')

            vis = Visualize(epochs, av_fitnesses, all_fitnesses, av_scores,
all_scores)
            vis.visualize()
```

D: TESTING

The following tests are for specific modules of the program. Those of which were showcased in the testing video will be referenced, while others introduced for the first time or somewhere else in this documentation will be supported by screenshots and references respectively.

The tests below are for modules used to satisfy *AI Objectives* 3 and 4. The results for *AI Objectives* 5, 6 and 7 and *Tetris Objectives* 1 and 2 are shown in this video.

Link: https://youtu.be/OZKNdm6125U?t=476
Video: Tetris Agent with Genetic Algorithm

MODULE and TEST	**EXPECTED RESULT**	**OBSERVED DATA**
Getting all possible piece configurations **1.** Are they correct?	A list of tuples. Each tuple contains the x, y coordinate of the piece in its final position, the rotation index, and the list of each block of the piece at its final position	*See Test1, Test3 and Test4* Data is correctly output, and is logically correct https://youtu.be/OZKNdm6125U?t=676 Timestamp: 11:16
Calculating heuristics given a state **2.** Are the heuristics correct?	A list of values, each a heuristic calculated from the state	*See Test2* Data is correctly output and is logically correct
Getting and making the best move **3.** Is the best move chosen made correctly on the board? **4.** Is the best move a valid move?	A tuple of the best configuration of the piece The configuration should be physically possible	*See Test1, Test3 and Test4* https://youtu.be/OZKNdm6125U?t=676 Timestamp: 11:16
Ablation tests **5.** Are the heuristics the use chooses to ablate correctly ablated?	A list of values, each a heuristics calculated from the state apart from those the user decided to leave out	*See Test5* Data is correctly output and is logically correct https://youtu.be/OZKNdm6125U?t=1298. Timestamp: 12:38
6. Updating landed positions to pass into the	A 2-D list of the representation of the	*See Test6*

SRS class to check for collisions	current state of the board when a piece lands	
7. Saving and loading a population	After a set number of generations, information about the current population must be saved and be retrievable at a later time to continue training from that population	It is possible to set a checkpoint every after which a generation will be saved. This generation can be loaded after https://youtu.be/OZKNdm6125U?t=1065 Timestamp: 17:45

Test1, Test3 and Test4 :

To get all possible configurations of a piece in a given state, the function receives a representation of the current state (boundary isn't considered) and the piece object from which it gets the 4x4 grid coordinates of the piece.

The state passed into the function is shown below by the coloured *'x'*s, showing a filled cell. The possible final configuration is shown as the grey piece.

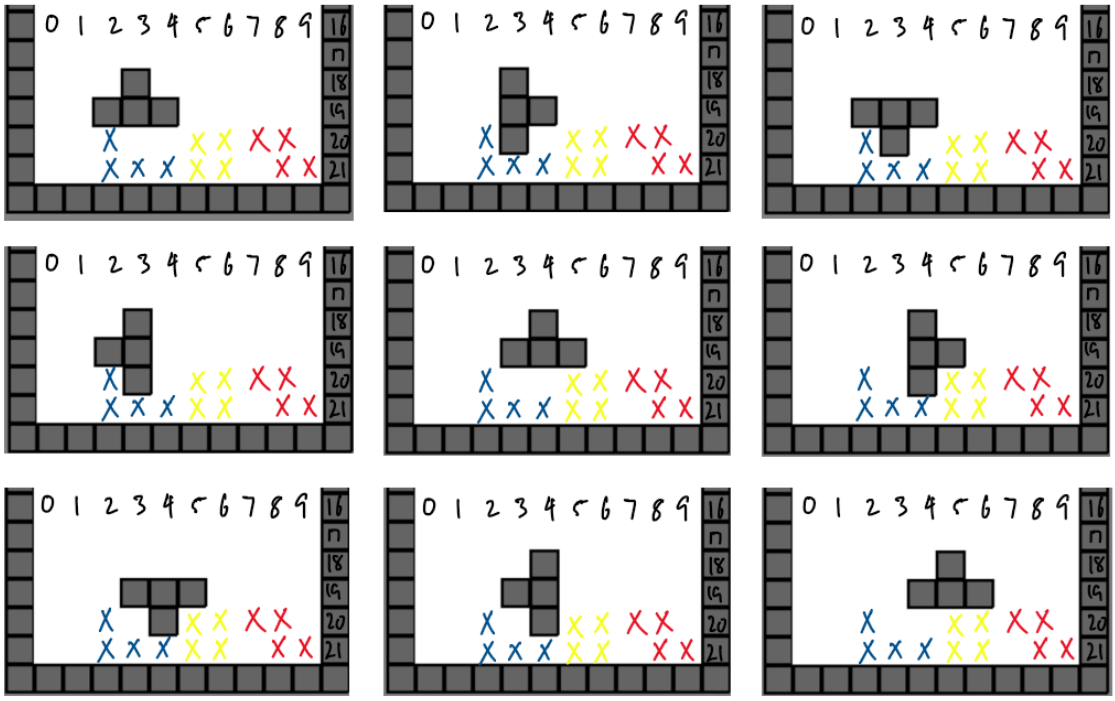
```
((-1, 18), 1, [(0, 19), (0, 20), (0, 21), (1, 20)])
((0, 17), 0, [(0, 19), (1, 19), (1, 18), (2, 19)])
((0, 17), 1, [(1, 18), (1, 19), (1, 20), (2, 19)])
((0, 17), 2, [(0, 19), (1, 19), (2, 19), (1, 20)])
((0, 18), 3, [(1, 19), (1, 20), (1, 21), (0, 20)])
((1, 17), 0, [(1, 19), (2, 19), (2, 18), (3, 19)])
((1, 16), 1, [(2, 17), (2, 18), (2, 19), (3, 18)])
((1, 16), 2, [(1, 18), (2, 18), (3, 18), (2, 19)])
((1, 16), 3, [(2, 17), (2, 18), (2, 19), (1, 18)])
((2, 17), 0, [(2, 19), (3, 19), (3, 18), (4, 19)])
((2, 17), 1, [(3, 18), (3, 19), (3, 20), (4, 19)])
((2, 17), 2, [(2, 19), (3, 19), (4, 19), (3, 20)])
((2, 17), 3, [(3, 18), (3, 19), (3, 20), (2, 19)])
((3, 17), 0, [(3, 19), (4, 19), (4, 18), (5, 19)])
((3, 17), 1, [(4, 18), (4, 19), (4, 20), (5, 19)])
((3, 17), 2, [(3, 19), (4, 19), (5, 19), (4, 20)])
((3, 17), 3, [(4, 18), (4, 19), (4, 20), (3, 19)])
((4, 17), 0, [(4, 19), (5, 19), (5, 18), (6, 19)])
((4, 16), 1, [(5, 17), (5, 18), (5, 19), (6, 18)])
((4, 16), 2, [(4, 18), (5, 18), (6, 18), (5, 19)])
((4, 16), 3, [(5, 17), (5, 18), (5, 19), (4, 18)])
```

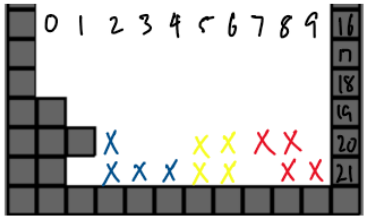
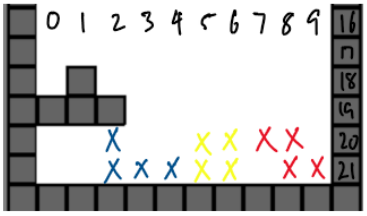
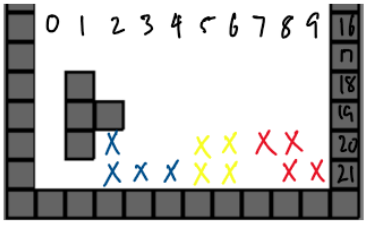
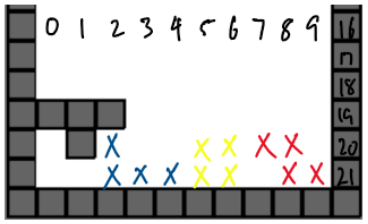
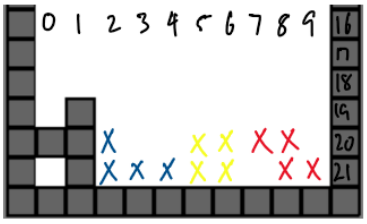
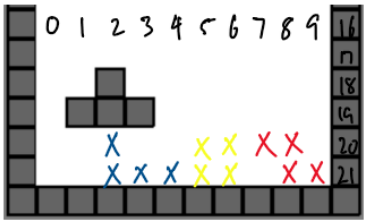
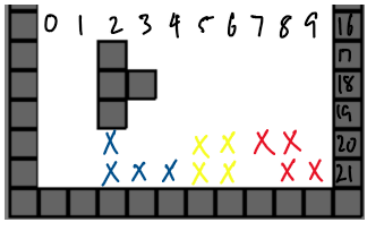
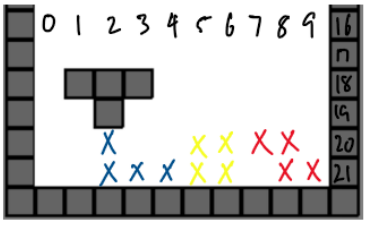
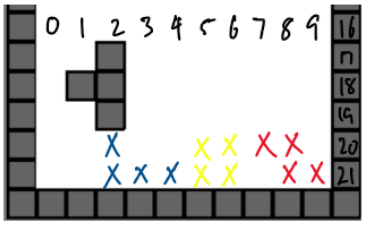
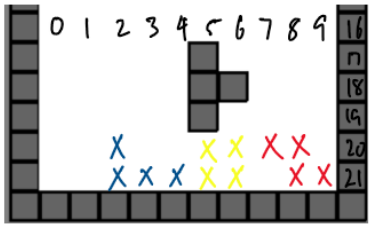
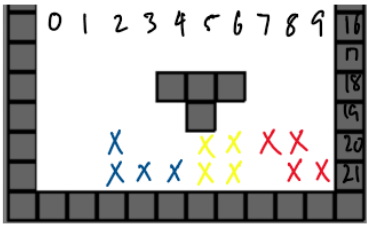
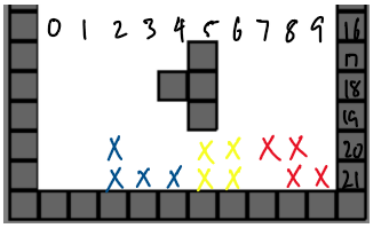
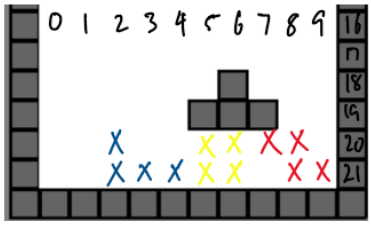
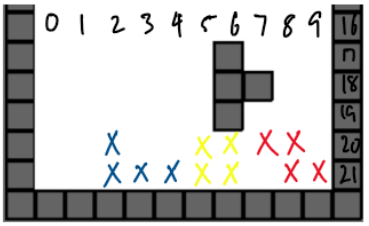
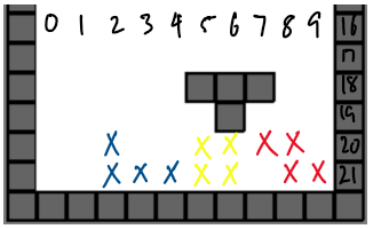
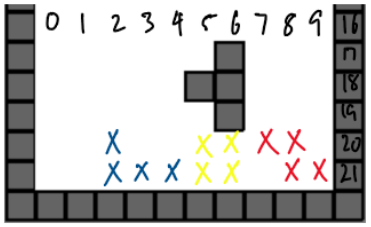
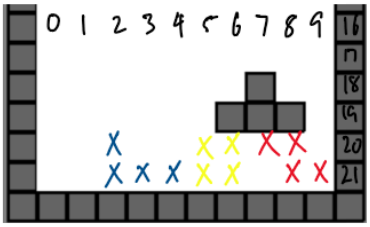
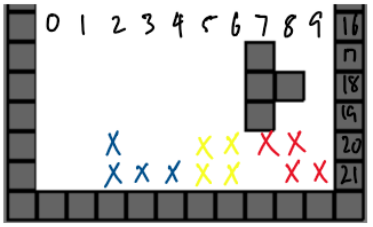
Taking the *'T'* piece as the current piece, the output of the function is:

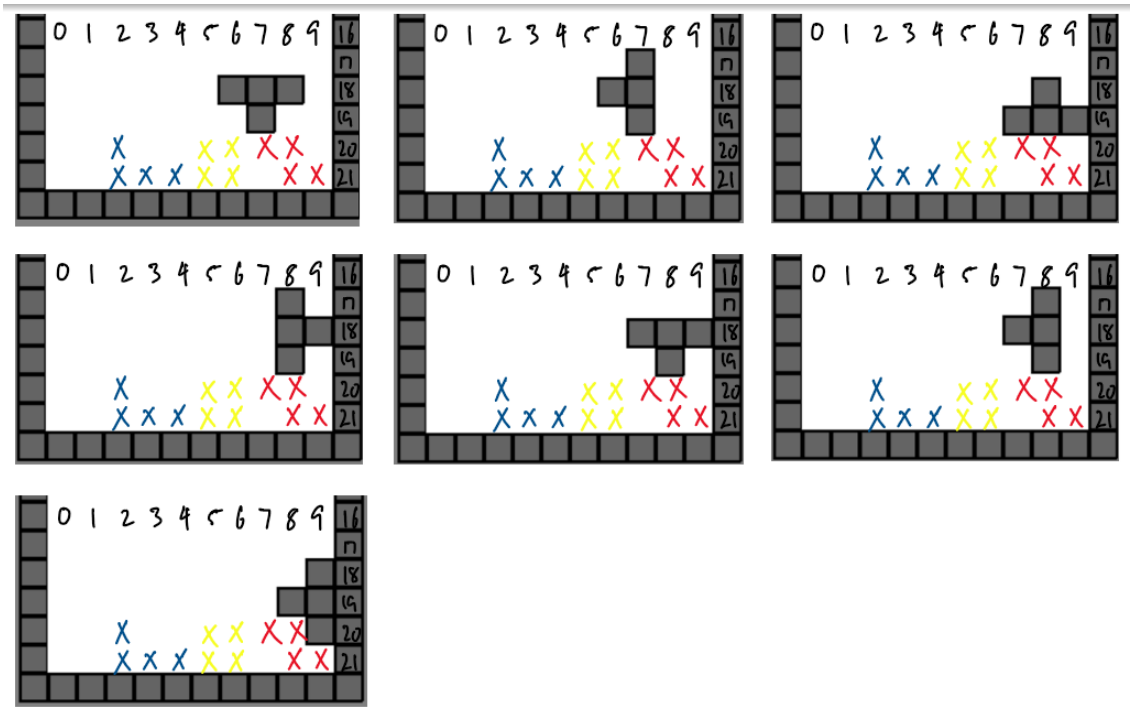
```
((5, 17), 0, [(5, 19), (6, 19), (6, 18), (7, 19)])
((5, 16), 1, [(6, 17), (6, 18), (6, 19), (7, 18)])
((5, 16), 2, [(5, 18), (6, 18), (7, 18), (6, 19)])
((5, 16), 3, [(6, 17), (6, 18), (6, 19), (5, 18)])
((6, 17), 0, [(6, 19), (7, 19), (7, 18), (8, 19)])
((6, 16), 1, [(7, 17), (7, 18), (7, 19), (8, 18)])
((6, 16), 2, [(6, 18), (7, 18), (8, 18), (7, 19)])
((6, 16), 3, [(7, 17), (7, 18), (7, 19), (6, 18)])
((7, 17), 0, [(7, 19), (8, 19), (8, 18), (9, 19)])
((7, 16), 1, [(8, 17), (8, 18), (8, 19), (9, 18)])
((7, 16), 2, [(7, 18), (8, 18), (9, 18), (8, 19)])
((7, 16), 3, [(8, 17), (8, 18), (8, 19), (7, 18)])
((8, 17), 3, [(9, 18), (9, 19), (9, 20), (8, 19)])
```

From this list, a best move is chosen. By showing that all these moves are valid moves, we also show that the best move will always be valid, which satisfies test 4. To show the move being made in real time to test whether the move made is correct, refer to this test, and this link which shows a video recording of this action.

Each of these positions corresponds to:

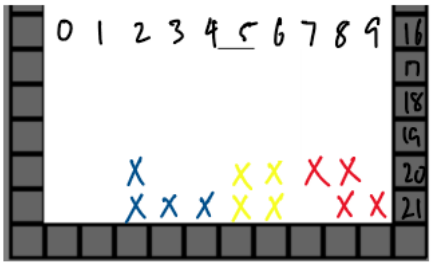




Test2:

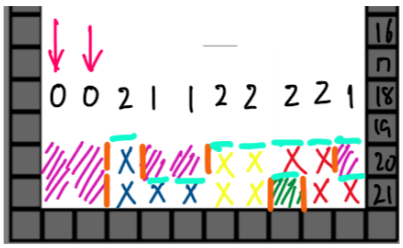
Consider the state:



When passed into the *Heuristics* class, the following data is returned:

```
[2, 13, 1, 5, 0, 7, 0.7810249675906655, 2, 9]
```

The data corresponds to the heuristics: Deepest well, Total height, Total holes, Bumpiness, Lines cleared, Row transitions, Standard Deviation of Heights, Number of pits, Number of column transitions



→ Deepest well = 2 (purple)

→ Total height = 0+0+2+1+1+2+2+2+2+1 = 13

→ Total holes = 1 (green)

→ Bumpiness = |0-0|+|0-2|+|2-1|+|1-1|+|1-2|+|2-2|+|2-2|+|2-2|+|2-1| = 0+2+1+0+1+0+0+0+1 = 5

→ Lines cleared = 0 (No full rows)

→ Row transitions = 7 (Orange lines)

→ Std of heights $= \sqrt{\frac{0^2+0^2+2^2+1^2+1^2+2^2+2^2+2^2+2^2+1^2}{10} - \left(\frac{13}{10}\right)^2}$

$= 0.7810249676$

→ Number of pits = 2 (pink arrows)

→ Column transitions = 9 (teal lines)

The hand calculations above show that the heuristics are calculated correctly

Test5

This test shows the series of steps taken by the user to carry out ablation

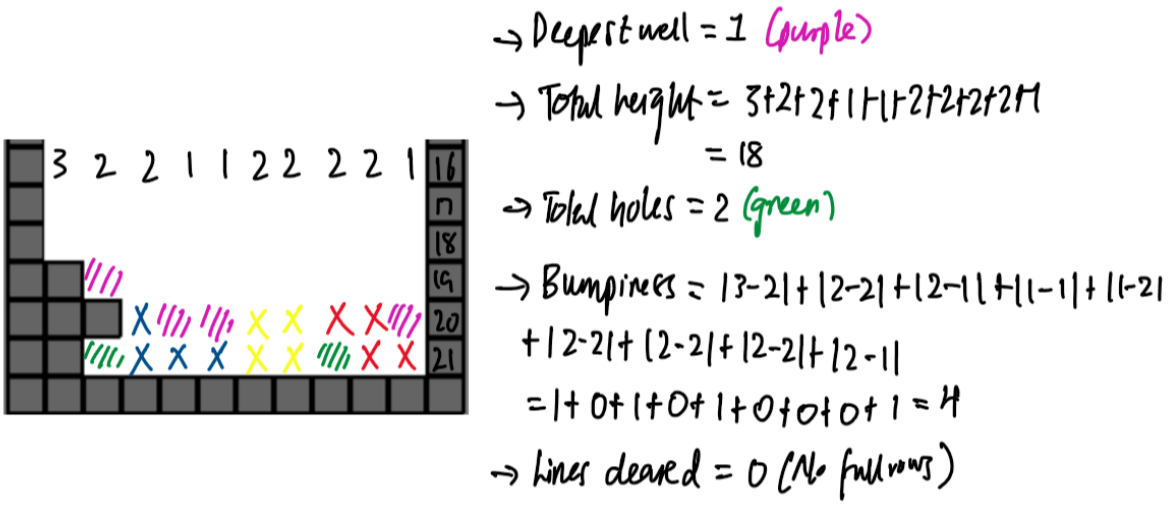
Ablation list:

```
_______ Ablation _______
0 -> Deepest well
1 -> Total height
2 -> Total holes
3-> Bumpiness
4 -> Lines Cleared
5 -> Row transitions
6 -> Standard deviation of heights
7 -> Number of pits
8 -> Column transitions
```

Using the same state and piece as that above, I will ablate heuristics at index 5 through 8, returning data for the first possible move and checking whether the correct heuristics were ablated.

```
[1, 18, 2, 4, 0] ((-1, 18), 1, [(0, 19), (0, 20), (0, 21), (1, 20)])
```

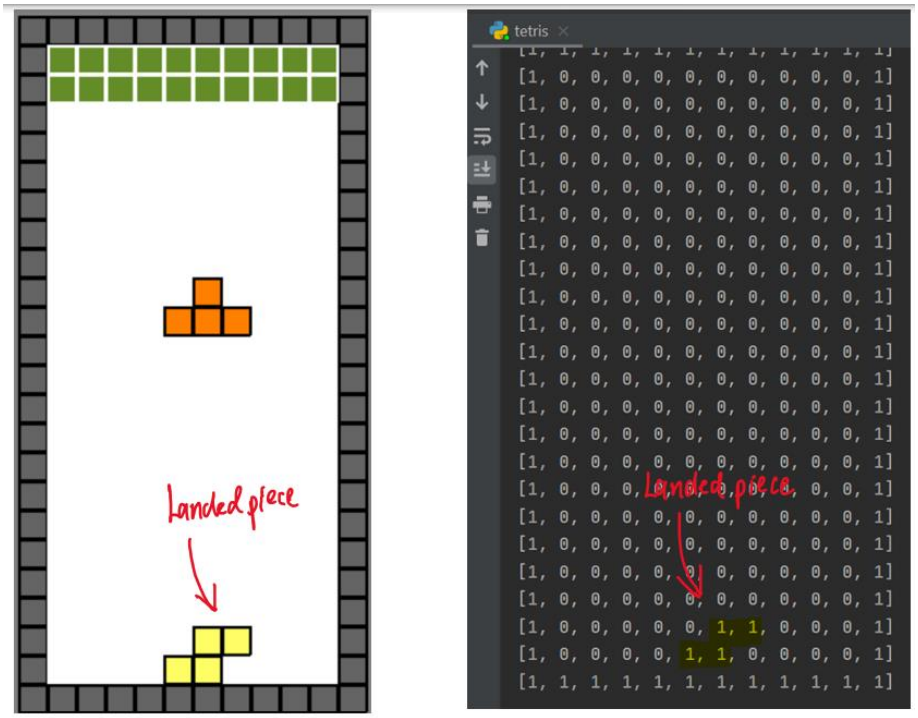
The first list are the heuristics, the tuple is the piece configuration



This matches with the data above, showing that ablation works correctly.

Test6

Every time a piece lands, the *landed* dictionary in the *Tetris* class needs to be updated with the coordinates of the board corresponding to colour of the block in that piece.
This information needs to be passed on to the collision detector, so that it can accurately detect whether a new piece will collide with anything in the current state of the board.



To the left is the Tetris board rendered onto the screen, and to the right is the representation obtained by the collision detection class, stored in the variable *self.field.*
Notice that the current piece isn't represented in this board, as it doesn't really make sense to test collision with itself. We only need information about the boundary, and the landed pieces.

E: EVALUATION

Objectives set out at the start of the project will be compared with final output, and comments will be made on whether they were met. A discussion will be had about possible improvements for the future, and a final reflection about the project.

How well were objectives met ?

1. The game should be playable, in the sense that the user can carry out any moves without the game crashing or ending abruptly
    a. The game should have error catching capability, while outputting relevant and understandable error messages for the user

Verdict: This was achieved and demonstrated through the testing section. The player could play for as long as they wanted without worrying about abrupt crushes or unnatural placements. Moreover, the user had feedback on what was happening in the game i.e., why certain moves weren't possible. User also got needed information such as next piece knowledge, game rules, and controls as well as the ability to pause a game from the UI, aiding them in playing the game smoothly , even as beginners.

2. The game should follow the indispensable Tetris guidelines
    a. Playfield should be 10 cells wide and 20 cells tall
    b. The game should allow for natural rotation of a Tetromino piece
    c. Piece preview should be available
    d. Player should be able to hold a piece
    e. Hard drop should be an option
    f. Colours of the Tetrominoes should be as follows:
        i. I -> light blue
        ii. J -> dark blue
        iii. L -> orange
        iv. O -> yellow
        v. S -> green
        vi. Z -> red
        vii. T -> magenta
    g. Pieces should be generated with the 7-bag algorithm
    h. Game must include a version of Korobeiniki music
    i. The game is only over when a piece is spawned overlapping at least one block above the playable field

Verdict: All guidelines stated above were followed, although not all were considered in the analysis from the official guideline. Nonetheless, the ones stated were achieved, giving the game an authentic Tetris feel

AI Objectives:

3. AI should achieve a score of at least 200,000 by the end of the project
    a. This is under the condition that this high score is achieved in under 10 generations of training

b. For each training game, agent has no cap on the number of pieces it is dealt

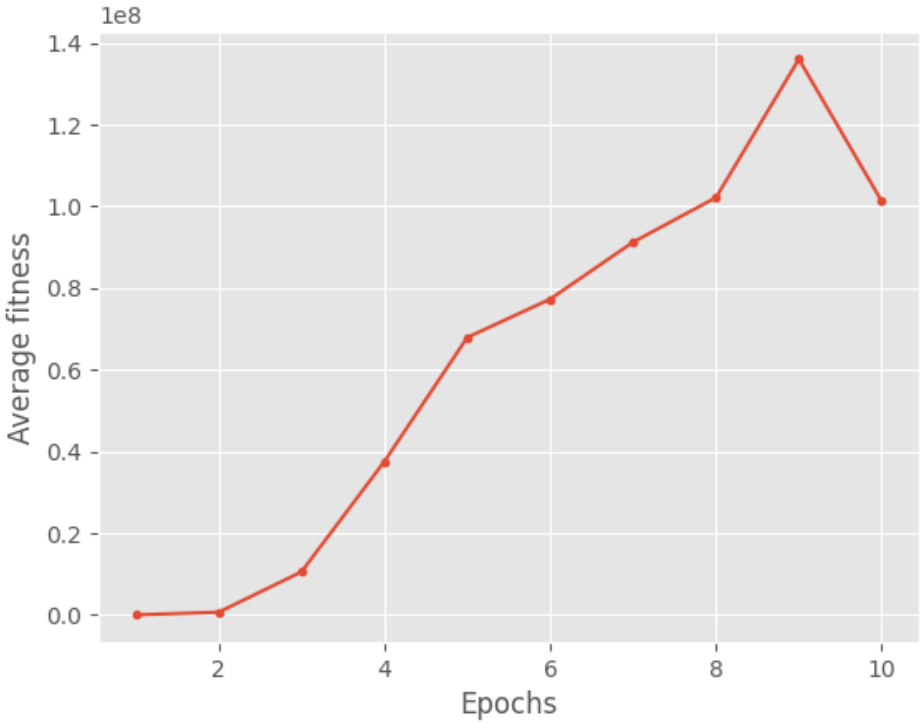
Verdict: The trained agent beat this target and surpassed it to the point of reaching a high-score of over 2 million by the 12th generation. This goes to show the power of machine learning techniques in spotting patterns in messy data and utilising those patterns to generalise strategies for the game of Tetris. Tetris itself holds a large state space, with 10^{60} possible stable(without full rows) states, making this achievement more impressive considering that the neural networks start off with random weights, and through natural selection, a set of weights that maximise the score are found. Using heuristics helps massively, because heuristics essentially allow the agent to get as much information as possible from a given Tetris state, which would otherwise be hard to do due to the enormous search space of Tetris.

4. AI should be reasonably good at the game, making choices most human players would

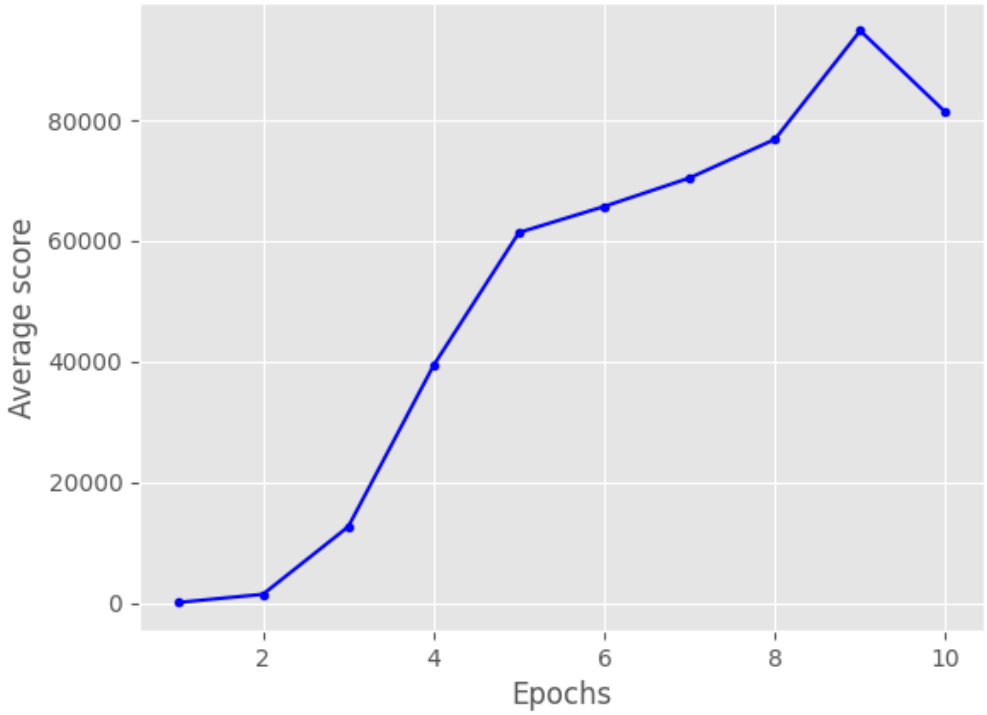
a. Agent should demonstrate good Tetris play by emulating known human strategies
b. Agent should demonstrate good Tetris play by placing pieces in reasonable positions given the situation of the game e.g., shouldn't make a move that would make the structure taller when the game is about to be lost

Verdict: The agent not only learns how to effectively play Tetris, but also learns strategies that are used by humans in only its first 3 generations of training. One predominant trait in its play style is one that is famous among Tetris players; creating a structure that is compact and moderately tall, allowing for an empty column for the "I" piece to occupy. In fact, when observing the agent play, it seems to be overly dependent on this strategy, failing to use alternatives even when the structure built is too high, and will lose the game soon. Nonetheless, this effect is less and less noticeable with more training.

Results with no ablation:



The graph above shows an increase in average fitness for the agents over the 10 epochs of training. Notice the steepest increase in fitness between epochs 3 and 5, and a curious drop from epoch 9 to 10



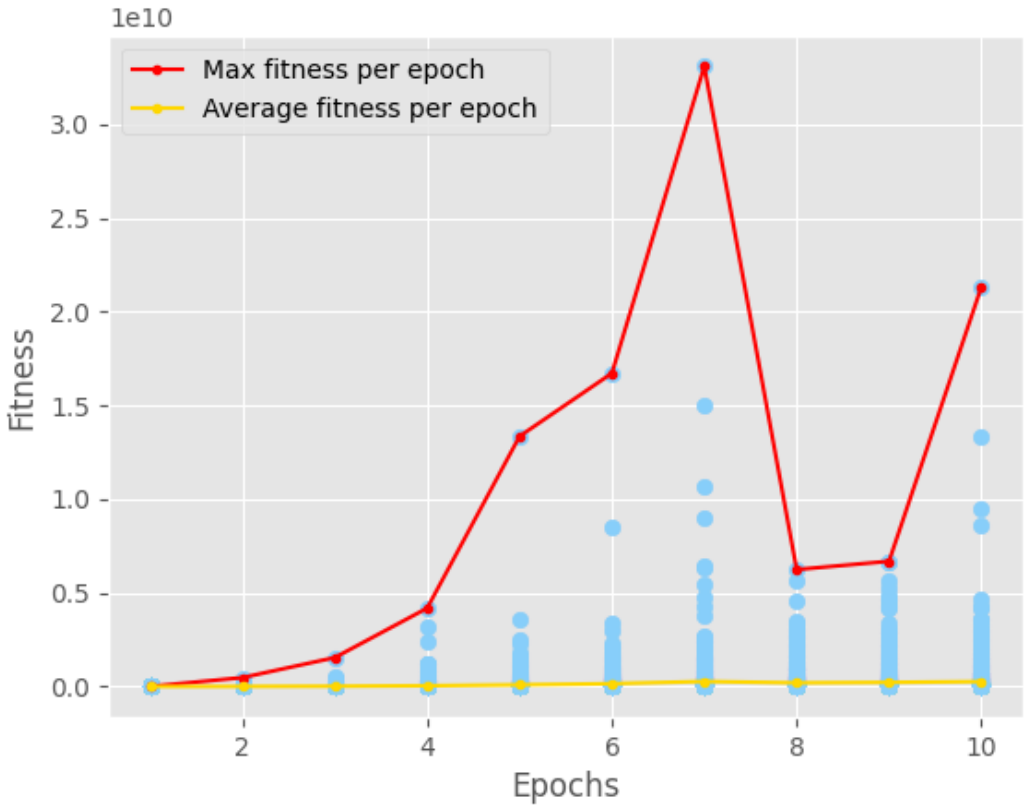
The graph above shows the relationship between average score and epochs. This shows an increase in score with a peak at about 900,000.

Ablation tests:

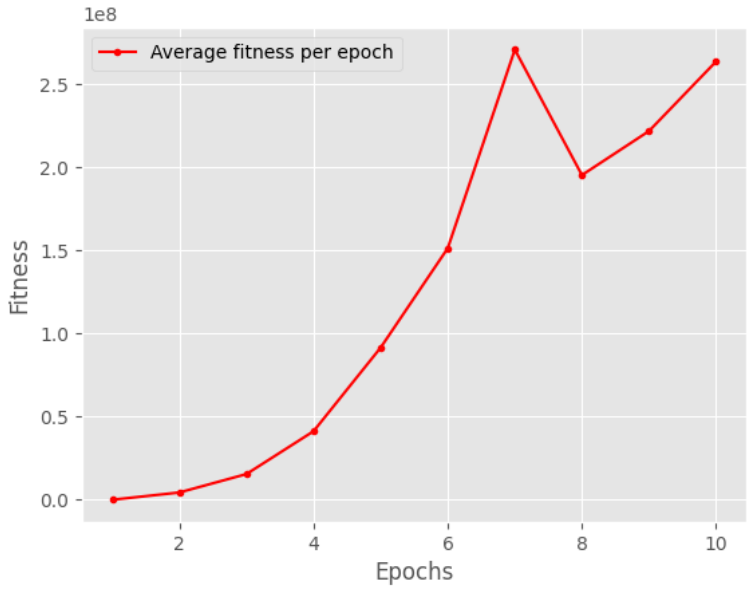
Some heuristics were removed and weren't considered when calculating the score for a move. These are the results of those tests.

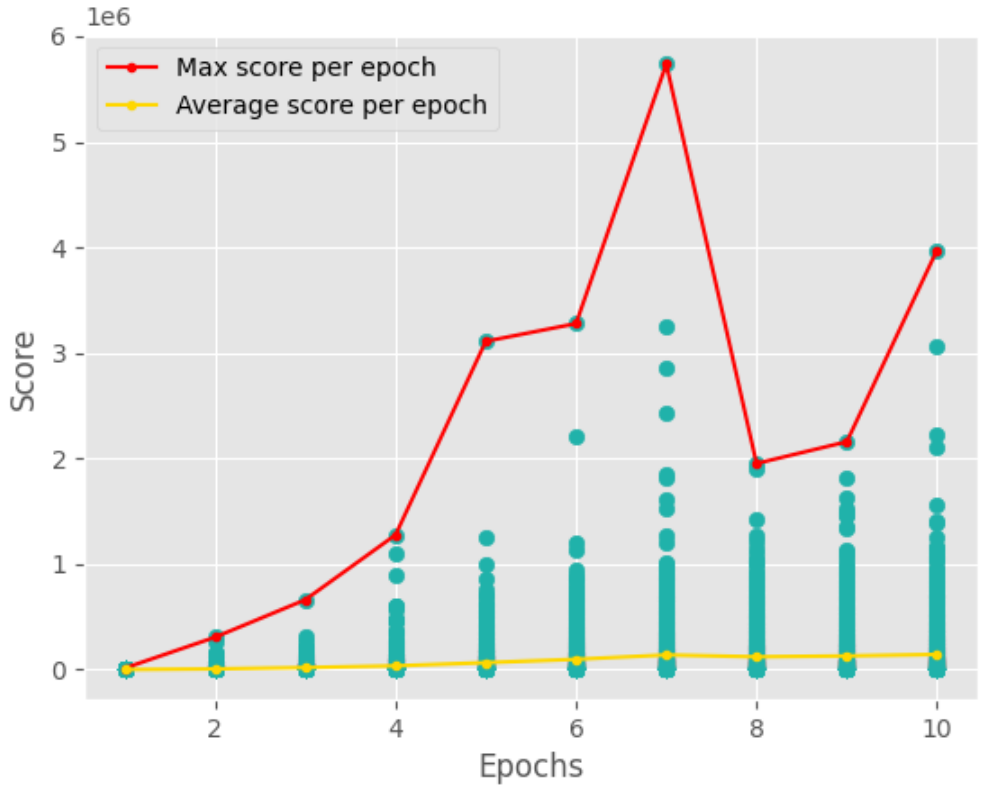
```
_______ Ablation _______
0 -> Deepest well
1 -> Total height
2 -> Total holes
3-> Bumpiness
4 -> Lines Cleared
5 -> Row transitions
6 -> Standard deviation of heights
7 -> Number of pits
8 -> Column transitions
```

Results when 5, 6, 7, and 8 are ablated:

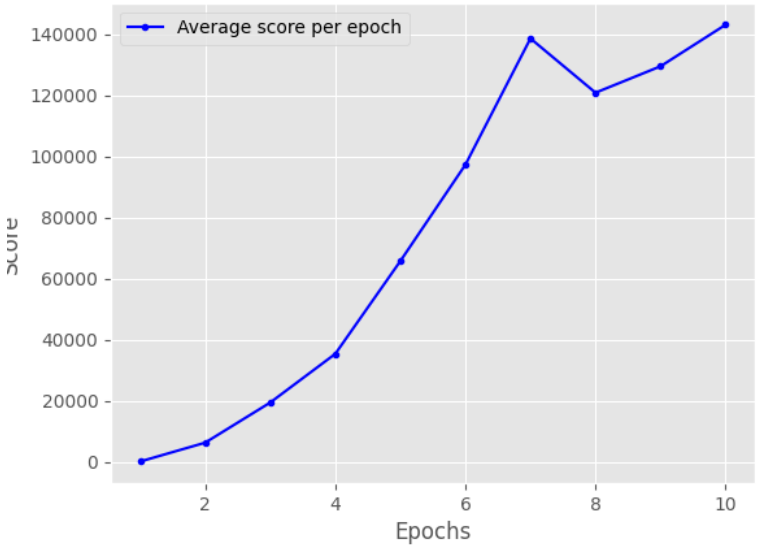


The graph above shows fitness of genomes against epoch number. There's a rapid increase from epoch 0 to epoch 7 in fitness, reaching a peak at about 33 million, then dropping dramatically from the 7th to 8th epoch. This was probably because the best genome from the previous population was made worse through mutation. (I don't think that crossover was the issue because the elitism threshold should've passed the best genome without any crossover, so the mutation is the culprit). The agent then finds its feet into the 9th to 10th generation and shows promise of getting even better with more training time.





The graph above shows game score against epoch number for all genomes. As expected, the graph looks similar to the fitness graph, as fitter genomes will perform better at the task. The maximum score gained over training was about 5.8 million at the 7th epoch, with an average best of about 200,000 for all training. In comparison, the average score with all heuristics for the genomes was at a best of about 950,000 at the 8th epoch *(see Figure1)*.



5. Data visualisation tools should be used to graph AI progress
    a. Program should be able to graph progress over 10 epochs of training for the average fitness per epoch over the 10 epochs
    b. Program should graph the average score achieved over each epoch for the 10 epochs

Verdict: This objective has been achieved as the graphs used above are using data from the game and are produced after training by the program

6. AI should accurately follow the ideas behind genetic algorithms
    a. The agent should learn to play the game by using a genetic algorithm and its ideas such as crossover and mutation as opposed to backpropagation

Verdict: The concepts of genetic algorithms are used by the agent and their power is on full display. We started from neural networks that had random weights and knew nothing about Tetris, and only used mutation and crossover to train over multiple generations and eventually learn to play good Tetris. It is very fascinating to consider that in the incredibly large state space of Tetris, where it would be impossible to write rules encoding any given state, an idea as simple as natural selection can find those states that correspond to high reward.

7. Program should have functionality to save and load populations at the user's will
    a. The program should save populations after ever set number of epochs
    b. The program should be able to load these populations back, carry out mutation and crossover accurately from them, and continue training

Verdict: This has been achieved as shown in the tests above

How could the project be better?

Although a human user had the ability to perform complex moves such as T-spins, it would've been nice to implement that same capability for the AI agent. In this implementation, I decided not to because I thought that the agent would be able to showcase its strength and that of genetic algorithms well enough if it can make the near-optimal moves each time. If it can do this, making complex moves is only a small increase in performance.

Allowing the user to choose different sizes of board and piece types for the agent to train on would have been a fun addition to see whether the agent had reached the point of generalising its knowledge. Moreover, this would have given the human players more choice on the games they wanted to play. To add to this, a multiplayer system would have been nice, perhaps one using web-sockets to allow computers in the same network to communicate and play each other online.

A good idea would be to test this agent against other agents already implemented by other people, by getting each agent to play against an adversary that tries to deal the worst scoring pieces in each state, while the agent tries to maximise their score for that given piece and state. This could be done with a minimax algorithm.

The agent could have been better if more complex machine learning techniques had been employed, such as Q-learning or using a Convolutional Neural Network. These would take the agent to world record territory, which is at about a score of 100 million. Combining other optimisation algorithms such as Particle Swarm Optimisation would be a good addition as well.

Possible extensions to the project:

- Multiplayer mode using sockets for computers on the same network to be able to play each other
- Multiplayer mode, with option to play against the Agent at different difficulty levels (making use of different saved populations

Reflection

This has been a wonderfully fun project to work on because it balanced challenge with learning experience. I have learnt about neural networks and how they operate at a deeper level, and this excites me to continue researching this field and learning more about methods such as Convolutional Neural Networks. The most rewarding part was seeing the results of the agent performing far better than I thought it would. Moreover, the process of figuring out why certain aspects of the project didn't work was rewarding in the sense that I had to think about the problem and work out a solution to it. This strengthened my belief that working on projects such as these can be an irreplaceable form of learning.

The project has also introduced me to other forms of machine learning and what they are being used for. For example, while reading a paper about using Monte Carlo Tree Search to optimise Tetris, I learnt about its use in DeepMind's AlphaGo and AlphaGo Zero systems that made huge achievements in building purely "intelligent" AI. This then led me to read the papers on these systems which broadened my scope on the research and work that is being done around the world to improve upon AI systems by making them learn by themselves.

This project has inspired me to look at the machine learning space with a sense of optimism for its ability to help humans solve many of the problems we face, such as climate change, pollution, energy storage and manufacture, the protein folding problem in biology, among many others. I hope that whoever reads this also feels that sense of optimism and fascination for AI and is encouraged to read more about the field.

Sources and references:

- Used a gameboy emulator and Pytorch to build a Tetris AI from scratch: https://towardsdatascience.com/beating-the-world-record-in-tetris-gb-with-genetics-algorithm-6c0b2f5ace9b
    - This was my starting point when thinking about how to code a genetic algorithm in raw Python.
- Explains use of a genetic algorithm without neural networks: https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/
    - This helped me understand how genetic algorithms work to optimise a function
- The neural network was built upon an example in Tariq Rashid's book (Build your own Neural Network) which uses a neural network to classify handwritten digits from the MNIST database: https://www.goodreads.com/en/book/show/29746976-make-your-own-neural-network
    - I repurposed the neural network by adding more layers and writing code for an agent that would use that neural network. The agent had most of the hard work to do, the neural network just received input and provided output for the agent to do yet more processing to choose a best move

- Explanation of genetic algorithms: https://lethain.com/genetic-algorithms-cool-name-damn-simple/
    - An article that explains how genetic algorithms work with some basic examples
- Ideas for implementation from Greer Viau, who also used a genetic algorithm to build a Tetris agent: https://www.youtube.com/watch?v=1yXBNKubb2o&t=482s
    - This video was my foundation for how the agent and neural network could work together to produce as functional Tetris playing program
- The original paper on NEAT-Python: http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf
    - The first test I carried out was to use NEAT Python which worked out well as a proof of concept that it was indeed possible to use a genetic algorithm to play Tetris
- Using NEAT-Python for Sonic the Hedgehog: https://www.youtube.com/watch?v=pClGmU1JEsM&list=PLTWFMbPFsvz3CeozHfeuJIXWAJMkPtAdS
    - An example of genetic algorithms being used on other games
- Explanation of wall kicks and the Super Rotation System: https://harddrop.com/wiki/SR
    - This website provided the data I used when programming the Super Rotation System
- Uses a local search algorithm to build a Tetris agent: https://github.com/saagar/ai-tetris/blob/master/paper/tetrais.pdf
    - This paper provided ideas on which heuristics to use, and why they were useful
- How to write a line clearing algorithm: https://www.youtube.com/watch?v=HcGQB1nHOOM&t=934s
    - This video helped me understand how to structure my Tetris implementation in such a way that it was broken down into modules that made problems like this easier to solve
- Foundational ideas for Tetris Implementation by Javidx9 on the OneLoneCoder channel: https://www.youtube.com/watch?v=8OK8_tHeCIA&t=1083s
    - The first video I watched which goes through a simplistic Tetris implementation in C++
- Complexity zoo: https://www.youtube.com/watch?v=YX40hbAHx3s