Playing Tetris using Reinforcement Learning Methods

Phase 3

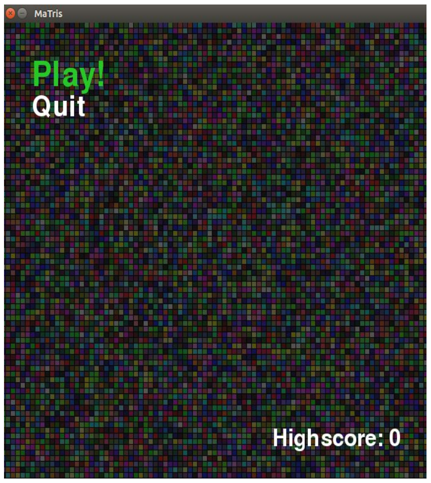
Alperen Tercan
Redion Xhepa

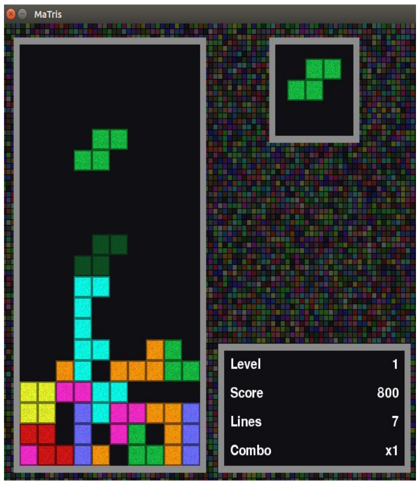
1. Introduction

The aim of this project is to be able to train a bot which will be able to play Tetris. Tetris is a demanding game in terms of playing because the falling tetromino is random and has a multitude of variety.
Our implementation of the AI Agent is expected to independently develop long-term strategies to cope with the inherent stochasticity of the falling tetromino. In this way by having gained insight on different strategies on how to play Tetris, AI agent should perpetually interact with the GUI of the Tetris game and achieve satisfactory results in terms of scores gained and the number of lines cleared. This AI agent will act as if a human was playing the game.

2. Dataset Description

Considering that we are using unsupervised machine learning techniques we do not need to have ready labelled data.In order to be able to make our "AI agent" play itself the game, we had first to find and then adjust a Tetris emulator according to our need. For this reason, we tried to use the open-source code found in [1].This code includes three main classes namely; Matris, Game, and Menu. This game implementation is based upon the "pygame" library and it is GUI based.Even though the code was written in a very neat fashion it was a bit difficult for us to understand and then change it.There are in total 7 **.py** files in the Tetris game project and more than 1000 lines of codes to be fully understood.At first, after we were able to run the game(even this part was very problematic since a lot of libraries had to be downloaded and imported) we got the following output.



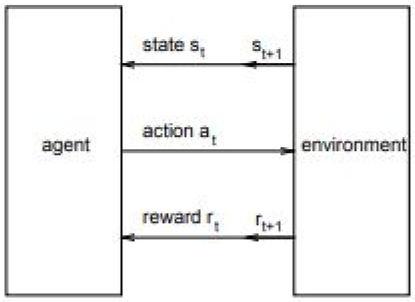


Open-up screen Instance game

As we can see the player has to click to the "Play" or "Quit" button.So our first duty was to skip the pressing of the button. At least 100 lines of codes had to be changed written in this part.Eventually the game would be played in the following manner: At first, the user would press the "play" button and then the bot would play on its own until the user would interrupt.In this case, no external interference would be needed to restart a new game, the bot would itself start the game.Later on, we had to alter the code even more in order to adjust the speed since we needed a lot of time to train and test our system.The final step for preprocessing before specific methods were written, was to write some function where to place the tile according to the output of the method or other helping methods such as whether is a particular movement feasible or not.

3. Description of the methods used
Three different unsupervised learning techniques are used to train the AI agent to play the Tetris game, namely Q-Learning,Genetic Algorithm and Neural Network with Genetic Algorithm.
Q Learning
Tetris game is proven to be NP-complete[2]. This result implies that it is impossible to find an optimal policy in an effective manner. That's why reinforcement learning techniques such as Q-Learning might be used to find an approximate solution.In order to make it easier for the reader to follow let us clarify some of the terms related to this algorithm[3]:

- Agent: The entity that plays the game and takes the actions.
- Action: The set of possible movements that an agent can perform.
- Environment: The setting where the agent performs, in our case the Tetris game and its rules.
- Policy: strategy followed by the agent to choose an action based on the current state.
- Reward: Feedback that serves as an indicator of how good or bad was a particular action of the agent.
- State: Situation in which the agent is.
- Q-Value: The expected long-term return of state s, performing the action under a policy π.



Q Learning is an elegant way to derive the optimal policy in a free model way. As its name states, this algorithm tries to learn the Q value. Q_π (s, a), as we previously mentioned, represents the long-term return of the current state s while performing a particular action s by following the policy π [4]. In a more formal manner we can mathematically express what we previously mentioned as following:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Since we want to maximize the discounted reward we act as following[4]:

$$Q^\pi(s,a) = \mathbb{E}_\pi\Big[r_{t+1}+\gamma r_{t+2}+\gamma^2 r_{t+3}+...|s_t = s, a_t = a\Big] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\Big]$$

$$Q^\pi(s,a) = \mathbb{E}_\pi\Big[r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_t = s, a_t = a\Big]$$

$$Q^\pi(s,a) = \sum_{s'}\mathcal{P}^a_{ss'}\Bigg[\mathcal{R}^a_{ss'} + \gamma\mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s'\Big]\Bigg]$$

$$Q^\pi(s,a) = \sum_{s'}\mathcal{P}^a_{ss'}\Bigg[\mathcal{R}^a_{ss'} + \gamma\sum_{a'}\mathbb{E}_\pi\Big[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2}|s_{t+1} = s', a_{t+1} = a'\Big]\Bigg]$$

$$Q^\pi(s,a) = \sum_{s'}\mathcal{P}^a_{ss'}\Big[\mathcal{R}^a_{ss'} + \gamma\sum_{a'}\pi(s',a')Q^\pi(s',a')\Big]$$

An iterative process is needed to be able to update the Q values for each state. As the algorithm further explores the environment, Q-Learning algorithm will lead to nicer approximations.
The generic format of the Q-Learning algorithm can be succinctly summarized as following[4]:

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize $s$
  Repeat (for each step of episode):
    Choose $a$ from $s$ using policy derived from $Q$
    Take action $a$, observe $r$, $s'$
    Update
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$;
  Until s is terminal
```

State Reduction for Q-Learning

Performing Q-Learning in its crude form is almost infeasible due to its computational complexity since there are $\mathbf{2^{200}}$different raw states. Therefore, we tried to encode the information contained by these states in a much smaller state space. In literature, there are different ways how this can be done. Usually, a feature space that consists of a combination of average height, bumpiness, roughness, number of holes in the game matrix is used. However, again in literature results of such an implementation is not successful[3]. Therefore, we have taken a different approach which we perform non-uniform quantization of the difference between heights of adjacent columns and divide the game matrix into three overlapping pieces. Tetrominoes has a maximum depth of 4, and it is less than 3 usually. Hence, most of the significant information about positioning tetrominoes lies in range of (-2,2). So we map height difference between consecutive columns 0 to 0,1 to 1,-1 to 2,2 to 3, -2 to 4 and the rest to 5. In this way, we distinguish the states that we can fit a tetromino successfully. without creating holes etc.In order to further reduce number of states, we divided the game table to 3 overlapping subtables. The underlying assumption is when positioning a tetromino, most important information is available in the 5 columns around the desired position. Hence, we created 3 tables of width 5 columns with limits: (0,4),(2,6),(5,9). First reduction decreased the number of states from $\mathbf{2^{200}*7}$ to $\mathbf{6^{10}*7}$. Second reduction decreases this number to $\mathbf{6^{5}*7}$.Considering that there are up to 5 different possibilities for translations and 4 possibilities for rotation, it makes possible a total of 20 actions that an agent can perform.Now we have the final view of the Q matrix, with dimensions **9072 by 20** .

Rewards for Q-learning

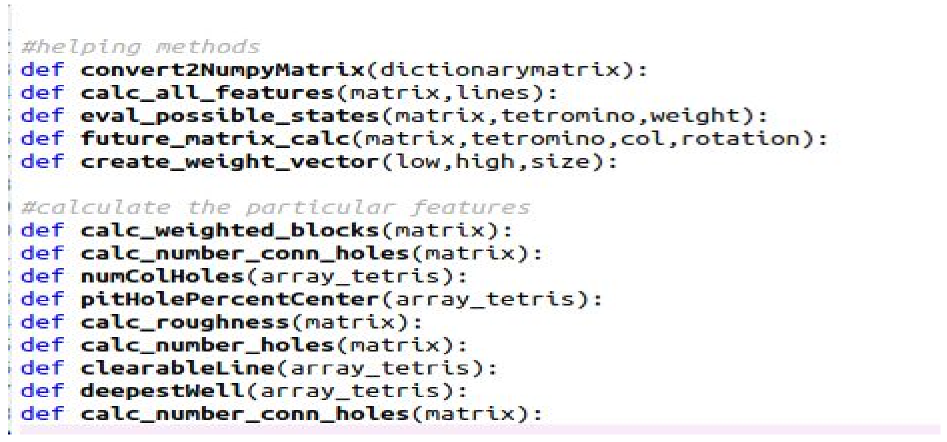
It is very important to have a good reward function which rewards good actions and punishes bad ones, preferably in a way that also expresses how good or bad an action is. Although the Tetris game has a built-in score metric, immediate rewards are only available when lines are cleared. This could be a sufficient function if enough computation time is given. However, we would like introduce intermediary rewards to make convergence faster. More importantly, as we have introduced a new state space that treats each part of the game matrix independently; we have to introduce a reward function which can evaluate how good a particular part of game matrix is. For this we have used combination of a novel concept, partial lines, and well-known average height. Partial lines is the number of completed rows in a particular sub-game matrix. In our case, it means the number of rows for which all 5 columns of sub-matrix is filled. Hence, final reward function is as below.

$$Reward = 400 * (linesCleared)^2 + \Delta partialLine - 2 * \Delta averageHeight - 10$$

Genetic Algorithm

This is the first algorithm which was implemented.Before the implementation, a thorough literature review was done.For sake of completeness let us make a short review of the Genetic Algorithm. Genetic Algorithm is a way to solve optimization problems when it is difficult mathematically model the problem. It uses genetic reproduction process and evolution concept that we encounter in nature. For starting, an initial population of genetically coded solutions is created. In literature, there are several ways to implement this[5].It can be done uniformly or gaussian random. Also mean and variance of the population should be determined based on the problem. After this next generations(iteration), will be obtained from previous generation based on their fitness scores. A proportion of best individuals in a generation is usually directly copied to next generation. This approach is called elitism and very helpful particularly in preserving good individuals from mutations and crossovers . Then again a proportion of best individuals are used in crossovers where they are mixed with each other in a stochastic way to obtain hybrid solutions. Lastly, there is a mutation process where some individuals are altered in a stochastic way. This alterations can be done by substituting some genes with random numbers or adding a random noise to all elements. Mutation process is important for keeping the solution space large and escaping local optima. Proportion of individuals coming from each of these processes, their implementations, probability of mutation and crossovers are some of the parameters that should be adjusted according to problem. This is usually where the contextual information comes in. Our choices will discussed later. The first part of our implementation in python was writing the methods which would evaluate/calculate some of features and other helping methods.

Creating the Feature Vector



```
#helping methods
def convert2NumpyMatrix(dictionarymatrix):
def calc_all_features(matrix,lines):
def eval_possible_states(matrix,tetromino,weight):
def future_matrix_calc(matrix,tetromino,col,rotation):
def create_weight_vector(low,high,size):

#calculate the particular features
def calc_weighted_blocks(matrix):
def calc_number_conn_holes(matrix):
def numColHoles(array_tetris):
def pitHolePercentCenter(array_tetris):
def calc_roughness(matrix):
def calc_number_holes(matrix):
def clearableLine(array_tetris):
def deepestWell(array_tetris):
def calc_number_conn_holes(matrix):
```

At first we had to understand how the Tetris game implemented would give us the current state matrix.That matrix was saved as a dictionary type.In order to perform operations in a more feasible and faster way we had to convert it to a numpy array(convert2NumpyMatrix).In this way the current state matrix consists of only 1's and 0's.Most of the features that we decided to use and implement are concisely described as following[3]:
Hole:an empty space which has a filled area above it(compare it with pit)
Block:s set of tetrominoes which stand together
Pit:an empty space which can be at the corners or it is closed both in its right and left sides(a hole without a filled area above it).

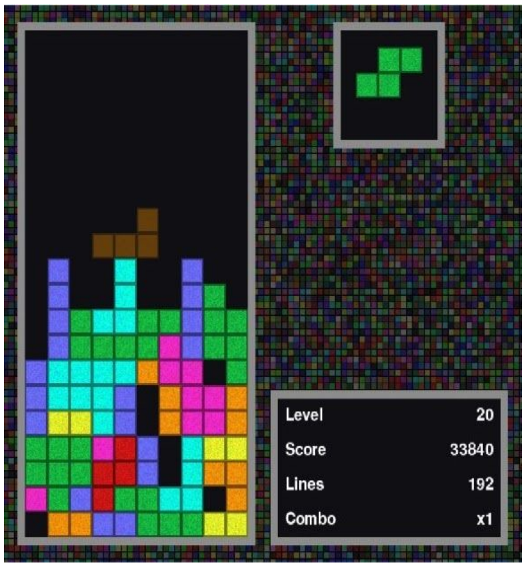
- **Blocks Weighted :** weighted summation where a relevant weight corresponds to the number of the row which the block belongs to.
- **Holes Connected :** number of vertically connected holes
- **Number of holes** : Counting every single hole
- **Cleared Lines**: how many lines are cleared with last move
- **Tetris :** is a tetris done with last move
- **Roughness** :summation over the absolute difference among heights of adjacent columns
- **Percentage Hole and Pit**:Ratio of pits over number of the total of pits and holes
- **Lines clearable**:prediction of the maximum lines clearable in the next state by a straight line.
- **Deepest Well**:the row which belongs to the deepest pit
- **Column Holes**:check how many columns have holes(at least one)

The full implementation is give in the Appendix.Before supplying the output to the other methods we built some test cases to check for the validity of our implementations.After checking everything we proceeded with the most important part of the Genetic Algorithm.

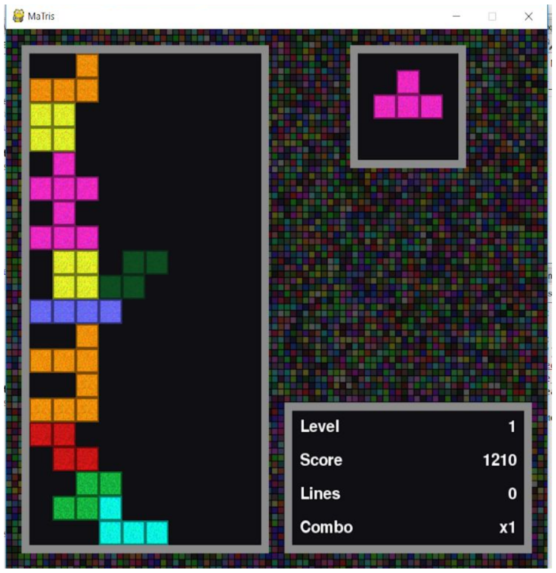
Creating the Genetic Code and Evolution Policy
We have modeled weights corresponding to these features as genes. Hence, each individual has 10 genes. We initialize that these genes as random number sampled from a uniform distribution with zero mean and 5 range.
We used average of 10 games and population size of 10.(See section 5.1) Since the population size is too low to successfully span the solution space, we implemented an initial step where among 80 individual we choose the best 10. This helps us with avoiding local optima.

In a regular iteration, we have tried two different policy of ratios to form next generation. Generation Forming 1(GF1) consists of best %40 of previous generation, crossovers of best %20 of previous generation and best %40 of previous generation with mutation. For the Generation Forming 2 (GF2) the numbers are %30,%40, and %30, respectively.
We have tried two different methods to perform crossover. Averaging Crossover(AC) substitutes randomly selected genes with random weighted average of parents where weights are uniform random numbers in range (0,1) and independent of each other. Substitution Crossover (SC) creates a 0.5 mean Bernoulli random matrix which indicates the genes to be crossed over. Then, crossover is done with substituting those genes with corresponding ones of another random individual. For mutation, again two methods are investigated. Substitution Mutation(SM) determines the genes to be mutated with a 0.5 mean Bernoulli random matrix, and substitute them with new random genes from initialization distribution. Noise Mutation(NM) adds 0 mean 1.5 standard deviation Gaussian noise to all genes.

With this setup we obtained individuals with different behaviors. An example can be seen below. The first one is an individual who penalizes building high towers as human usually do. However, second one prefers high towers; and because of this bad strategy it scores much less than former. Therefore, the first one is much more likely to advance to next generation.



Level 20
Score 33840
Lines 192
Combo x1



Level 1
Score 1210
Lines 0
Combo x1

Specific results and further implementation technicalities are explained in the Implementation and results sections.

Genetic Algorithm with Neural Network

This algorithm was proposed by Alperen without any previous knowledge of the literature.In order to support his claim, we checked whether researchers of the field had done something similar. Apparently, this idea to combine Neural Networks with Genetic Algorithms was being used in [6].Let us explain the theoretical basis of this method.
Neural Networks can be considered simply as universal function approximators[7].
The Universal Approximation Theorem states that with the proper number of layer,proper number of neurons and enough time steps Neural Networks can approximate any function with satisfactory performance.But unfortunately, this does not mean that there is a specific training algorithm which will make the relevant task.That’s why most of the researchers of the field use the famous back-propagation algorithm to train the weights of the network.Indeed we thought what if we use could Genetic Algorithm

as a method to train the neural network.We anticipated that the Neural Network would capture more nonlinear relation of the data(adding fidelity to the system) whereas just the simple linear relation that the Genetic algorithm alone would do.
Genetic algorithms are optimization techniques which can explore a huge and complex space in effective way to converge to global minimum[5]. Consequently, they are fitted to solve the issue of training neural networks.
In a fully connected neural network, the number of weights and biases can be calculated with following equation:

$$\#W = \sum_{k=0}^{\#layers-1} a(k) * a(k+1) + \sum_{k=1}^{\#layers} a(k)$$

We altered the first solution, and set the number of genes to total number of weights and biases. For each individual in the population, we use these genes as weights and biases of our neural network. Then, we use this neural network to evaluate the quality of a state by feeding the feature vector of that state as input values. After performing forward-propagation; the output value gives the quality score of the state. Then, we continued our greedy playing strategy using these scores. Average of the final scores of 10 games is used as fitness score of that individual. Then, same procedure is repeated for new individual. After evaluation of each individual in a generation; evolution algorithms which are described under genetic algorithms is used to form next generation. We tried several Neural Network configurations, their analysis can be found in Section 5. We used sigmoid as activation function of hidden layers and linear function for output layer.

.

4)Performance metrics

Tetris is a game which is proved that in its original form cannot be won[3].For this reason as a measure of success the Agent should try to increase the total score gained before game-over. Score gained is a good metric since it means that the agent should try to deploy long-term strategies and stay as longer in the game.
But different implementation of Tetris have different scoring systems and choice of that would affect the performance of the Agent.We have selected the following game score system calculation:
As performance metric we used the game score which is calculated by the following formula:

$$Score = \sum_{i=0}^{N} (100 * lines(i)^2 + 20)$$

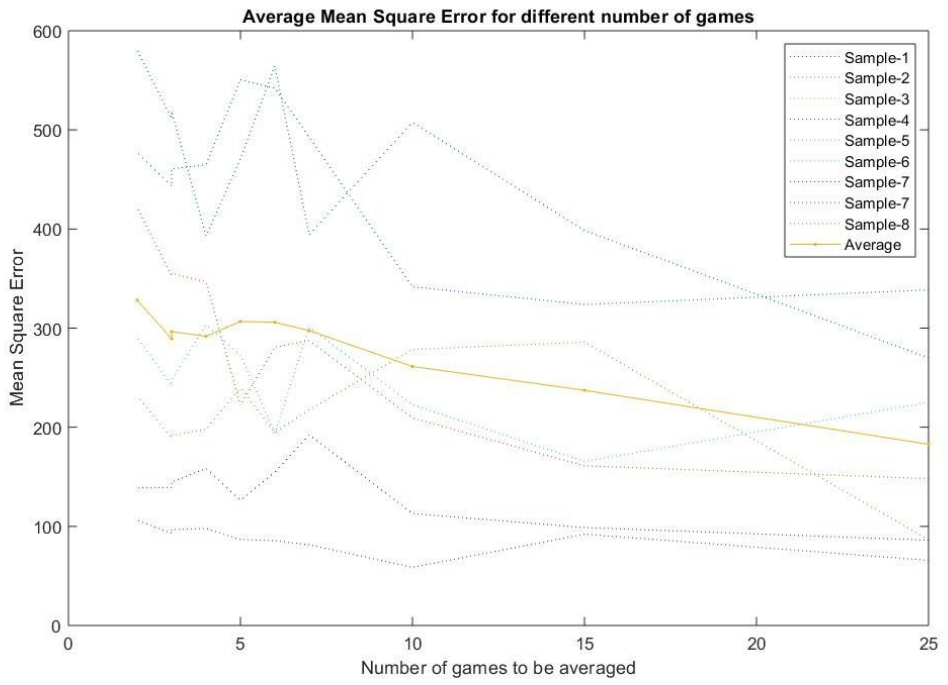
where lines(i)= lines cleared with i-th tetromino
where N=#of tetrominoes placed

In order to make a valid comparison between different algorithms, we used predetermined seeds for random tetromino sequences in the validation and testing. Moreover, we averaged scores in 20 games in order to avoid fitness to particular tetromino sequences. For training, seeds wasn’t predetermined; in order to preserve stochastic nature of the game, which makes it challenging and interesting to apply learning algorithms. For the final comparison between methods, a box plot will be provided too.

5)Implementation and results

Discussion of Variance in Performance

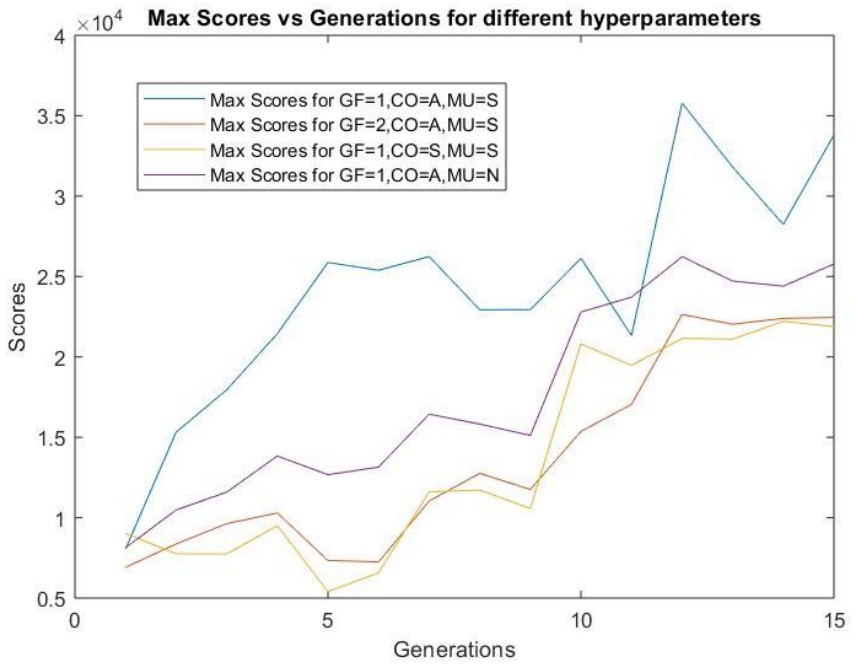
Tetris game can be considered as a difficult problem to solve with evolutionary algorithms. Since the environment is stochastic, fitness values corresponds to same solution can be different. More problematically, when there is a ‘unlucky’ sequence of tetrominoes a good solution can perform worse than a bad solution. Although we enjoyed dealing with learning in a such stochastic environment, in order to reach convergence within our limited computation time; we tried to alleviate this problem by averaging multiple games for each individual, hence reducing variance of the environment. That helps; however, there is a trade-off for computation power. Since the number of the games can be played is limited; we should balance number of games for each individual, size of the population and number of iterations.



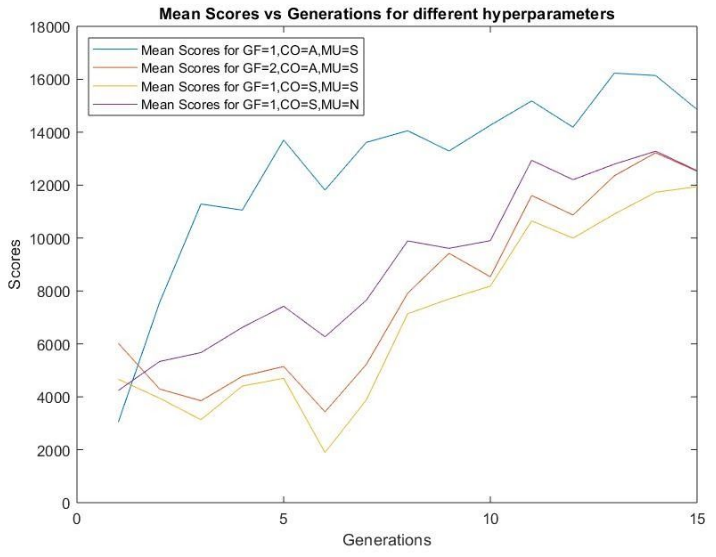
To decide this amount, we have chosen 8 different agents; make them play 75 games by each. Then, mean of 75 games is compared with means of partitions of several lengths. As can be seen above, as number of games to be averaged increase the MSE decreases. We chose 10 games, as slope of MSE decreases after this value. This analysis is valid for both Genetic Algorithms and Neural Networks methods.

Choosing Hyper-parameters

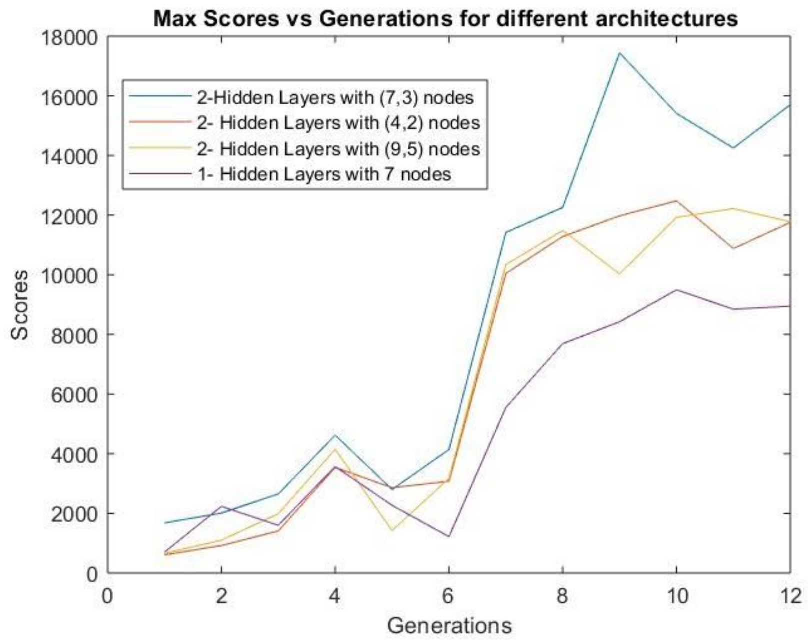
Genetic Algorithms



By altering 3 hyperparameters, we determined 4 different set of hyperparameters. These are (GF1, AC, SM), (GF2, AC, SM), (GF1, SC, SM) and (GF1, AC, NM). To decide which set of hyperparameters is better, we run a validation step as explained in Section 4. At each run, the algorithms are trained for 15 generations and these are the maximum scores of each generation. From results above, it can be said that (GF1, AC, SM) is the best choice of hyperparameters. Although we used maximum scores for comparison, mean scores for each generation can be seen below. The mean scores too supports our choice.

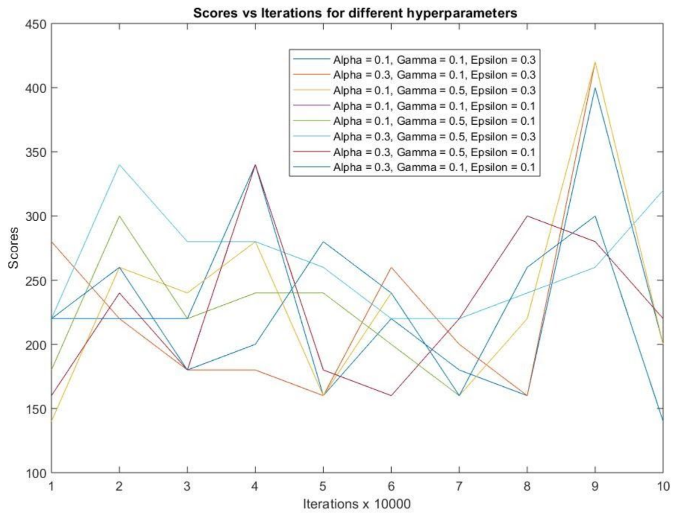


Neural Networks



We have run a validation with configurations above, As can be seen 2 hidden layers with 7 and 3 hidden nodes is the best configuration.

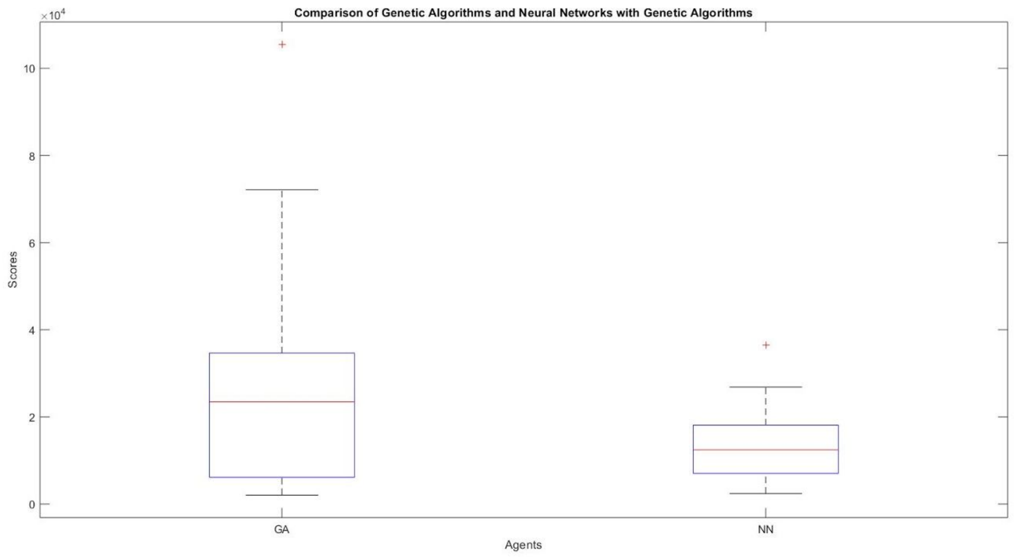
Q - Learning



We tried several hyperparameter sets with Q-learning. However, as can be seen from results above, Q-learning agent didn't show any significant progress or success. So we excluded it in our final test.

Cross-comparison of Methods

After determining optimal hyperparameters for each methods, we did a final test to compare Genetic Algorithms and Neural Networks with Genetic Algorithms. We took the best agent of each method in their 15th generation and compare their results in average of 25 games. To ensure a fair comparison, we used same random seed for tetromino sequences. Results can be seen below:



As the plot indicates, Genetic Algorithms agent performs better than Neural Networks agent in terms of many statistics concepts like maximum, median, third quartile, maximum outlier, mean. Hence, it is easy to decide GA as winner of this comparison.

6)Discussions and conclusion

We finished the implementation of three different methods to train the AI agent who plays Tetris,namely Q Learning,Genetic Algorithm and Neural Network with Genetic Algorithm.
Simulations done using the Q-Learning show a bad performance compared to other methods.Q Learning has several drawbacks.First of all if we would use the raw state space we would need to make a Q matrix of $\mathbf{2^{200}}$ **by 40** (where 40 is the to total number of actions during a particular state) it would be a computational overkill in terms of memory and time consumed to perform the operations. Decreasing the state space to approximately 9000 different states leads to a loss in the information representation of each space.This is the first drawback.But having a state reduction might not be a problem but finding the proper transformation is difficult.We tried to encode the state space in such a manner that the states which will lead to a higher chance of clearing lines (kind of a non-uniform quantization).But using such heuristic functions apparently does not help too much.This will be clearer when considering the supremacy of Genetic Algorithms since they are trying to find finely tuned and more complex heuristic functions.Another burden to be done is tuning the parameters such as alpha,gamma and epsilon.Their choice affects the speed at which the learning is done.These parameters are recommended to have small values in the range between 0.1 and 0.01.But in general using a state reduced Q Learning Algorithms is not effective when a simple heuristic function is being used.State-of-the art algorithms use Deep Convolutional Neural Networks in order to surpass this difficulty[8].Q Learning is effective only when the state space encodes a huge portion of the needed information from the system and most of the state-action pairs are iterated enough.

The second implemented method is Genetic algorithms.This method is superior in terms its overall success. Results indicate its effectiveness in learning comparison of different states. Other than its quantitative results, its playing style is very close to expert human players. Not only that it is able to clear hundreds of lines, it even performs advanced tactics like creating I valleys to clear multiple lines simultaneously. (Note that this boosts score, as score is proportional to square of lines cleared at each move.) Despite of its success in this particular game, whether it can generalize its learning to variants of this game or not, is an interesting question that should be addressed. Moreover, optimization of the Genetic Algorithms is another issue to be addressed. Although we have tried several evolution policies, these are only a small subset of what is offered in literature. A deeper research in this topic can be useful to shorten the convergence time of the algorithm.

Lastly, neural networks with genetic algorithms is also proved to be successful. Although it is outperformed by regular genetic algorithms' fast convergence in the short run; when enough time is given it will be better than genetic algorithms. Linear function that regular genetic algorithms suggests was not good enough to play 'almost endless' games. Maximum number of moves it achieved so far is less than 4000 moves. Hence, this suggests that the question of how good a state is might be more complex than to be address by a linear function. We have chosen our configuration as the best one among a couple of alternatives. Better configurations of neural networks should be investigated with further research.

This project confirms the literature that learning in stochastic processes is possible. However, it also shows that with limited computation power; human intervene through providing context related heuristics is almost crucial for effective learning. This research was mainly limited by computation power and time; however, we believe that methods that we propose are scalable. Hence, with more computation power; it should be possible to outperform our results.

References:

[1]https://github.com/Uglemat/MaTris
[2]Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. 2003. Tetris is hard, even to approximate. In Proceedings of the 9th annual international conference on Computing and combinatorics (COCOON'03), Tandy Warnow and Binhai Zhu (Eds.). Springer-Verlag, Berlin, Heidelberg, 351-363.
[3]Lundgaard , Nicholas, and Brian McKee. Reinforcement Learning and Neural Networks for Tetris.
[4]Understanding RL: The Bellman Equations, joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/.
[5] Lewis, Jason. Playing Tetris with Genetic Algorithms. 2015.
continuous optimization problems", Journal of Global Optimization, vol. 37, no. 3, pp. 405-436, 2006.
[6]Montana, David, and Lawrence Davis. “Training Feedforward Neural Networks Using Genetic Algorithms.”
[7]Hornik, Kurt et al. “Multilayer feedforward networks are universal approximators.” Neural Networks 2 (1989): 359-366.
[8]Sabeek Pradhan, and Matt Stevens. "Playing Tetris with Deep Reinforcement Learning."