Deep Reinforcement-Learning with Multiplayer Tetris

Ian Conceicao
UID: 505153981
IanCon234@gmail.com

Viacheslav Inderiakin
UID: 505430686
v.inderiakin.uk@gmail.com

Suraj Vathsa
UID: 205092512
surajvathsa@gmail.com

Abstract

Tetris 99 is a multiplayer battle-royal game extending classic Tetris gameplay. In this work, we create the game environment, implement training algorithms and optimize reinforcement learning agents to play Tetris 99 at the human level. In particular, we compare the performance of three Neural Network architecture classes: a Fully Connected Net, a Fully Connected Net with artificially constructed features and a Convolutional Neural Network. We also compare three learning algorithms: Deep Q-learning, Deep Q-learning with fixed targets and Double Deep Q-learning. In addition, we study how transfer learning affects resulting performance. We show that by using preprocessed features and keeping model size moderate it is possible to achieve high performance in both single-player and multiplayer tasks. We also show that transfer learning helps greatly enhance model's performance and guide training.

1 Introduction

Tetris 99 is a game where 99 players compete to be the last player standing. Each of the players controls a standard 20×10 board where they can sequentially place incoming pieces. Completion of certain conditions, for example, clearing 4 rows at once, not only increases score, but also gives an opportunity to attack other players by overloading their boards with incomplete lines called garbage rows, or simply garbage. A comprehensive list of Tetris99 rules can be found at [1]. Our team chose this game over other RL testbeds because it combines four features frequently present in real-life RL problems. First, it has an enormous state space of more than 2^{20} states for the single-player mode and an exponentially higher state space in the multiplayer scenario. Second, the environment is not deterministic, because new tetrominos spawn randomly. Third, it requires the agent to be aware of the other players and learn a strategy to survive in a competition [2]. Finally, Tetris 99 holds high similarity to its single-player version, which makes the usage of transfer learning possible. Simultaneously, to the best of our knowledge, this game has not been studied before in reinforcement learning context.

In this project, we approximate action-value function $Q(s, a)$ using Neural Networks. We explore three distinct approaches for Neural Network design. In the first scenario, we pass the raw representation of the players' boards through several fully connected layers with ReLU activation functions. While this architecture has an advantage of analyzing the full complexity of the game state, the feature space might be too large to provide high performance in acceptable training times. The second approach implies the usage of a set of artificially constructed features including: board height, number of cleared lines, "bumpiness" and "number of holes" [3]. The artificial features are then passed through several fully connected layers. Our assumption is that by drastically reducing the state space we will improve learning efficiency and speed at the cost of slightly decreasing the final performance. Finally, we try to pass the raw board state through a combination of convolutional

Preprint. Under review.

and fully connected layers, to compensate for the large state space complexity while simultaneously retaining high final performance.

Early on in multiplayer development we experimented with having the agent play against a random opponent. This worked fine for experimenting, but the random opponent would inevitably lose very quickly, and there was little room for improvement for our agent. Moving forward we decided to implement a self-play training process [4], where the agent would play against a past version of itself that is regularly updated. Self-play has been shown to allow agents to develop new tactics out of a necessity for continuous improvement.

2 Environment

A game environment simulating Tetris 99 gameplay and supporting Open AI gym API was implemented. The environment follows the standard OpenAI Gym API to make implementing training algorithms straightforward and to allow the results of our experiments to be easily reproducible. The environment was built in Python3, and heavily utilizes the Numpy library to handle the Tetris boards. Built next to the game was a renderer that creates frames for any number of players and game state.

2.1 Raw Actions

Originally we aimed to have actions in the environment perfectly mirror raw button presses in the real game: left, right, rotate CW, rotate CCW, switch attack mode, and no-op. However, after testing against a variety of RL algorithms and NN architectures, including fully connected networks and Deep Q-Learning with fixed targets, performance never improved. We hypothesize that this approach failed for three reasons. Firstly, it increased game time drastically because at the worst case approximately 20 time-steps would occur before the agent placed a single piece. Secondly, it made rewards extremely sparse, to the point where it could be troublesome for the agent to attribute which actions deserved reward. Lastly, nearly every move was completely reversible meaning that the agent was often just making noise to learn from.

2.2 Grouped Actions

Similar work done by Matt Stevens and Sabeek Pradhan of Stanford found a variant of raw button presses to fail as well [5]. Their paper implements a solution to combine all the actions involved in placing 1 piece into 1 action, which they called grouped actions. We followed their lead, and modified our idea of an action representing 1 single time step with a raw button press to instead represent the changes that occur between a piece being assigned and placed. For example the series of actions in English: [left, left, no-op, change attack mode, change attack mode, right] would instead be: move left once and set attack mode to Random. The downside is that dimensionality of action space becomes much higher. Because given a game state and a set of actions the end state is unique, a grouped action taken in a given state is logically equivalent to a next state. Therefore it was more convenient to have our neural network not output a favored action, but instead to analyze a potential next state and approximate the value of the state. The upside of this implementation is it works easily with grouped actions. All next possible states can be calculated, usually about 70 states are found, and the NN can pick the highest valued state. The downside of a value-based model is it is impossible to implement Policy Gradient approaches, where a policy that outputs an action is learned directly. The grouped action approach effectively kept the same functionality of the original Tetris 99, even though certain complex moves became impossible to perform. Grouped actions solved all three of the issues raw actions presented, and we began to see improvements in our algorithms.

2.3 Renderer

Coupled with the environment is a renderer that displays the game in pixel form. The renderer is heavily inspired from the real Tetris 99 game visuals, both in what is observable from a player's point of view and where boards and elements are placed. The player can see their own board, swap piece, incoming pieces, garbage row, and game position. In addition, the player can see every other player's board, as they appear much smaller and simpler. The frames are rendered using the drawing tools from the popular game engine Pygame. An additional tool was also built to convert any episode into a gif, which proved helpful for debugging and visualizing the playing style of the agents. For the renderer, stylistic influence and some program organization but not code was drawn from David Gurevich's Pygame Tetris implementation. [6]. An example of a frame with 99 players created using the renderer can be seen in figure 1.

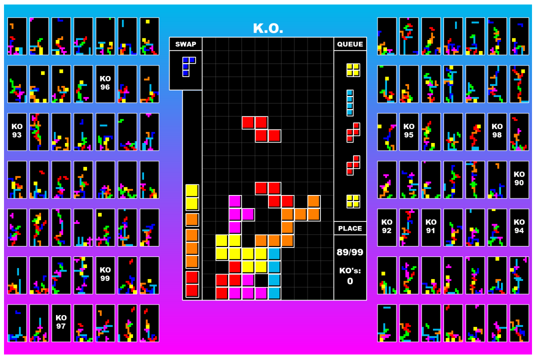


Figure 1: A snapshot of the environment.

3 Approach

3.1 Features

The feature extraction done in this project fits into 2 categories, raw and artificial features. Raw features in the Fully Connected net case appear as a flattened, binary version of the board, where 0 means that the cell is free and 1 means that the cell is occupied. Appended is also the number of lines the player has cleared, in the effort to let the agent learn the connection between clearing rows, and obtaining reward. Raw features in the Convulational Neural Network case is simply the board, in its 20x10 dimension, allowing the CNN to learn spatial relations between indices. Extended CNN additionally accounted for the number of lines cleared by the player.

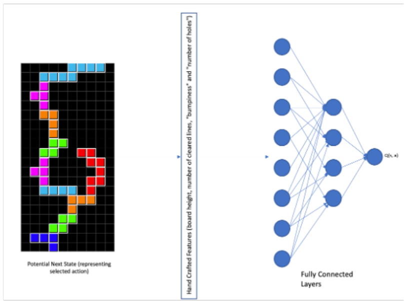
Artificial features are 4 key derived features calculated from the board. They all come from Yiyuan Lee's, Tetris AI – The (Near) Perfect Bot [3]. The first feature is the number of lines cleared by the player. This should have a positive correlation with good performance as a good playing is often clearing away blocks. The second feature is "holes", which is the number of empty squares with pieces above them. It is not preferred to have holes on your board because it means the row on which the hole lies can only be cleared after the rows on top of it are cleared. The next derived features is "total bumpiness", which is the sum of the absolute value of the differences in height of neighboring columns. The last feature is the sum of the height of all columns in the board. Together, these 4 derived features make a 4 integer vector to input as our artificial features.

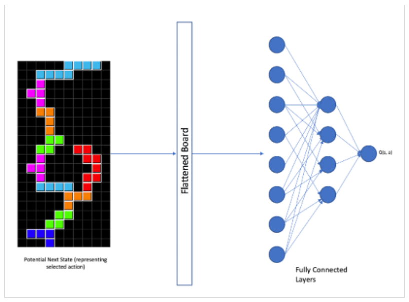
3.2 Networks

All neural networks of our project estimate the value of a particular state. The networks we created can fit into two categories: Fully Connected networks (FC nets) and Convolutional Neural Networks (CNNs). FC nets connect every node with every other node in sequential layers. CNNs capture spatial dependencies in the 2-d images, such as our board, using a combination of convolution filters and fully-connected layers.

3.2.1 Fully Connected Networks

For single player we implemented two different FC nets each for different applications. First, an FC net was designed to work for the artificial features described earlier. It takes 4 scalars representing the lines, holes, total bumpiness, and height sum. The architecture is 4 x 64 x 64 x 1 with 3 layers (figure 2a). Secondly, an FC net, referred to as FC Board Net, was created to capture the board input. It takes 201 values, including 200 binary values representing a flattened representation of the board where 0 means no piece and 1 means a piece. The remaining input is an integer indicating how many lines the player has cleared. This last input was added to try to teach the agent to clear more lines. The architecture is 201 x 64 x 64 x 1 (figure 2b).


Potential Next State (representing selected action)
Hand Crafted Features (board height, number of cleared lines, "bumpiness" and "number of holes")
Fully Connected Layers
Q(s, a)
(a) FC net with artificial features.



(b) FC net with flattened or ’raw’ features.

Figure 2: Single player fully connected network architectures.

For multiplayer we used FC net with artificial features described above and additionally implemented two new FC nets with identical architecture, except one implements transfer learning, FCNetTransfer, and one does not, FCNetMultiplayer. The design takes the artificial features from the players board, and passes it through two layers. Meanwhile, it takes all the artificial features from the other players' boards, and passes those through two different layers. Then on the third layer the two isolated subnets are combined into 64 nodes, and finally on the last layer a value is returned. This allows two events to happen. First, the agent can learn separately what it's own features mean, vs what other players' features mean. Second, because the agent has a subnet that is identical to the single player case pertaining to its own board, those weights can be initialized with the learned optimal weights found in the single player case. This is an example of transfer learning.

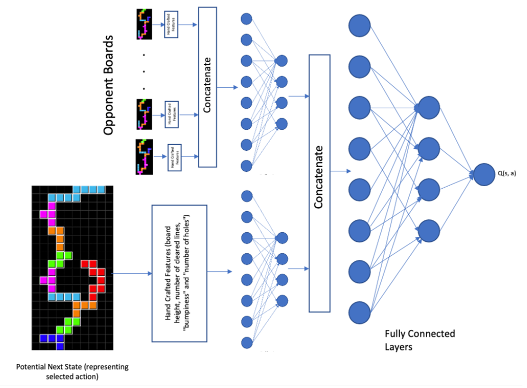

Opponent Boards
Hand Crafted Features
Hand Crafted Features
Hand Crafted Features
Concatenate
Concatenate
Hand Crafted Features (board height, number of cleared lines, "bumpiness" and "number of holes")
Potential Next State (representing selected action)
Fully Connected Layers
Q(s, a)

Figure 3: Fully Connected Network multiplayer architecture.

3.2.2 Convolutional Neural Networks

We implemented two Convolutional Neural Networks for value approximation in single-player mode. The first, simply denoted as CNN is built like a traditional vanilla CNN. It consists of 4 layers, the first three of which are convolutional. They have out channels of 3, 6, and 9 and kernel sizes of 3, 5, 4 respectively with a stride of 1 and no padding. The output is then flattened and connected to our one value output.

The other CNN we implemented is referred to as the Extended CNN because it is an extension of the previous CNN described. The Extended CNN is identical up until the last layer, where instead of immediately flattening the results and linearly combining them, it instead passes the results through 2 more fully connected layers and then an output layer. Before it does the extra two layers however, it appends the number of lines the player has cleared.

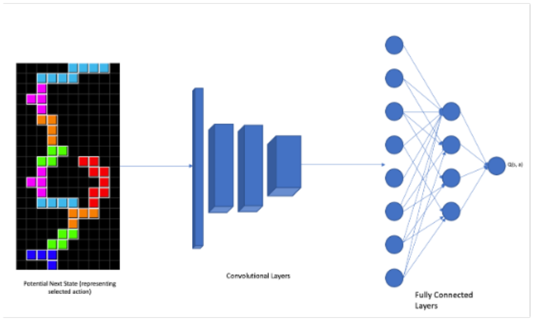


Figure 4: CNN architecture.

3.3 Algorithms

Deep Q-learning is a form of reinforcement learning that employs function approximation. Particularly, a Neural Network based architecture is used to approximate the Q-function for an unknown Markov Decision Process. Deep Q-Learning makes use of experience replay [7] from which batches are sampled to serve as training examples to learn the weights of the Deep Q-Network. The Deep Q-Network with fixed targets additionally incorporates a target network which is used to stabilize learning of the expected return. The target network uses the same architecture as the Q-network, i.e. the network used to learn the Q-function, but the weights of this network are updated at regular intervals instead of updating at each time step. Double DQN uses two separate Neural Network models. On each step, one network learns the weights from experience replay, and the other is used to find an action, which allows to mitigate the problem of action-value function overestimation [8].

We implemented a vanilla Deep Q-Network, a Deep Q-Network with fixed targets, and a Double Deep Q-Network. As discussed in section 2.2, instead of taking args (s_t, a) it is logical to build a Q-function that takes arg: s_{t+1} where s_{t+1} is the next state by taking action a in state s_t. Then, our policy is formed by calculating all possible next states and choosing the state with the maximum value according to our neural network.

3.4 Rewards

The rewards for every trial were:	Multiplayer also has rewards for:
• Survived: 0.001	• Sent Line: 0.02
• Cleared Line: 0.01	• Win: 10

The survival reward ensures the agent will prefer actions that do not end the game over ones that do, and it helps with a potential sparse rewards issues by constantly rewarding the agent for staying alive. Clearing lines is an extremely important skill in Tetris, because it allows the player to remove pieces from their board, so the high reward incentives the agent to do so. In multiplayer, sending lines to other players worsens the other player's board state, and can even knock them out of the game, so a high reward we believe is appropriate. And finally, winning the game is the inevitable goal, and the extreme reward of 10 is aimed to incentive actions and sequence of actions that result in a win.

4 Results

4.1 Single-player

We trained each algorithm for $100,000$ steps and automatically chose the model that achieved the highest cumulative reward in a given episode to save. We used the hyperparameters and rewards outlined in their respective sections. The saved model was then tested by running 50 consecutive episodes playing exclusively greedily. The results are summarized in table 1.

Out of all combinations of training algorithms and NN architectures, the DQN with fixed targets using FC nets with artifically constructed features performed the best. Being tested for more than 10000 of time steps, it made no errors and survived the whole time, so expectations of all performance metrics

		Training Aglorithm					
		Classic DQN		DQN fixed target		DDQN	
Architecture	Metric	$\mathbb{E}[\cdot]$	$\sigma[\cdot]$	$\mathbb{E}[\cdot]$	$\sigma[\cdot]$	$\mathbb{E}[\cdot]$	$\sigma[\cdot]$
FC net with raw features	survival time	15.84	2.32	26.00	3.55	11.68	1.05
	total reward	0.02	0.0	0.03	0.01	0.01	0.005
	lines cleared	0.0	0.0	0.78	1.17	0.0	0.00
FC net with artificially constructed features	survival time	239.48	222.19	∞	N/A	44.26	98.39
	total reward	1.44	1.43	∞	N/A	0.25	0.78
	lines cleared	84.76	88.46	∞	N/A	12.04	37.92
CNN with raw features	survival time	11.31	1.47	459.98	310.04	11.24	1.5
	total reward	0.011	0.001	2.33	2.79	0.011	0.0015
	lines cleared	0	0	170.72	123.44	0	0
Extended CNN with raw features	survival time	25.64	5.32	26.46	4.32	27.82	4.81
	total reward	0.05	0.02	0.04	0.02	0.04	0.02
	lines cleared	2.00	1.33	0.82	1.18	1.32	1.52

Table 1: RL algorithms' and architectures' performance comparison.

were deemed infinite. The very same reason prevented us from calculating variance of this extremely effective model. Training dynamics of this RL agent is provided in figure 5.

From table 1 we can see that FC net with raw features performed worse than all other models. We hypothesize that while having all necessary information for decision making, FC net can not fetch piece density and cleared lines given the model capacity and allowed training time. CNN with raw features performed considerably better, most likely due to efficient data handling by convolutional layers. The extended CNN, however, performed worse than the simpler CNN, possibly due to the 2 extra layers that need to be trained. It's performance also did not seem to depend on which algorithm was implementing it, indicating the bottle-neck was not the implementation but instead the model itself.

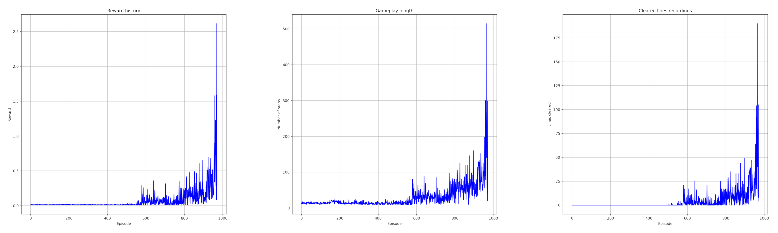


Figure 5: Reward (left), Episode length (middle) and Total number of cleared lines (right) of FC net with artificial features during training with "DQ learning with fixed target" algorithm.

4.2 Multiplayer

Based on our experiments in single-player (vanilla) Tetris, we attempted to train models for playing Tetris99. We restricted the number of players to two players given the limited time and resources available for training. Given the multiplayer nature of Tetris99, we sought to understand how the Deep Q Network that performed best in single-player Tetris and a modification to the Deep Q Network, inspired by this architecture and described in section 3.2.1, behaved when made to compete with an opponent skilled at vanilla Tetris at various levels.

We also sought to understand the effects of transfer learning and whether transferring weights learned by an agent playing vanilla Tetris would help boost performance or training speed. To carry out the experiments we trained two different architectures in three scenarios:

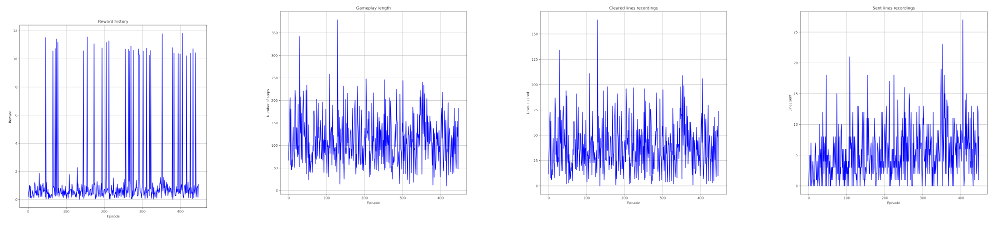
1. both the agent and its opponent start learning from scratch, and opponent's weights are regularly updated using agent's weights
2. the multiplayer agent initialized with random weights plays against the best single-player agent, which weights were transferred from section 4.1
3. both the agent and its opponent are initialized using the weights of the most performant agent from section 4.1; opponent's weights are regularly updated using agent's weights

In all scenarios described above, we used DQ learning with fixed target algorithm as it performed the best in single-player scenrio. To evaluate agent performance, we made it to play greedily against the best single-player agent from section 4.1. This opponent would never lose by itself and is very good at defense, so we can measure probability of victory and time required by multiplayer agent to defeat the single player agent. We also made a multiplayer agent to play with a copy of itself to measure how "aggressive" it is in its attacks. The results for all training settings and architectures are summarized in table 2.

From table 2 we can see the best performance in terms of both aggressiveness and win probability is achieved by FC net with artificially constructed features, without multiplayer enhancement (it's training dynamics is shown in figure 6). This might look counter-intuitive considering features of the FC net with multiplayer enhancement are more rich. However, this should not surprise us since capacity of the enhanced model is also higher, and given a limited number of steps for training it is not able to achieve its best performance.

While training both the agent and its opponent from scratch, we noticed an interesting behavior when training with the Enhanced FC net. The agent learns to play in the multiplayer setting initially while maximizing the number of lines cleared. However, after a certain number of epochs, the agent stops doing so and learns to get worse at clearing lines and playing the game. We hypothesize that since the opponent's weights were transferred from the agent at regular intervals, the agent learns "bad" weights in order to make the opponent worse at the game. This is explained by the decline in the number of steps the agent lasts and the number of lines the agent clears while the wins remains consistent. This is shown in figure 7.

Another important observation is that models trained with transfer learning (algorithm 3) performed considerably better for the same training time than those trained completely without it (algorithm 1) or by transferring single-player weights to the opponent only (algorithm 2). Further, transfer learning enables guided training and no anomalous behavior is observed such that in figure 7.

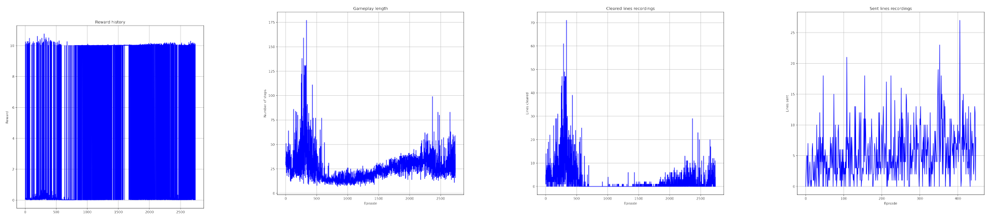


(a) Reward history. (b) History of episodes' length. (c) Total lines cleared per episode. (d) Total attack power per episode.

Figure 6: Training dynamics of FC net trained with transfer learning.

5 Discussion and Conclusion

During the course of this project, we trained RL agents that can play classic single-player Tetris and compete in multiplayer Tetris 99 at human level. A video showcasing agents' performance can be



(a) Reward history. (b) History of episodes' length. (c) Total lines cleared per episode. (d) Total attack power per episode.

Figure 7: Training dynamics of Enhanced FC net trained without transfer learning.

		Training Setting					
		1		2		3	
Architecture	Metric	$\mathbb{E}[\cdot]$	$\sigma[\cdot]$	$\mathbb{E}[\cdot]$	$\sigma[\cdot]$	$\mathbb{E}[\cdot]$	$\sigma[\cdot]$
FC net with artificially constructed features	probability of victory	0.76		0.38		0.92	
	Episode length	124.22	86.48	333.0	234.46	176.35	109.28
	Total attack power	6.24	3.44	8.54	6.42	18.15	8.89
Enhanced FC net with artificially constructed features	probability of victory	0		0		0.67	
	Episode length	38.4	21.26	21.58	2.47	136.46	78.53
	Total attack power	6.32	7.33	0.02	0.0	5.57	4.06

Table 2: RL algorithms' and architectures' performance comparison.

found at [9]. We compared the performance of several architectures and training algorithms on both single-player and multiplayer testbeds, and observed that increase in feature complexity strongly increases training time and degrades final performance. However, given a highly representative features and an architecture with sufficient capacity, an agent can be trained to play almost perfectly mimicking human behavior. We also compared how transfer learning affects training in multiplayer scenario. We found that transferring knowledge of game mechanics from single-player scenario to multiplayer is highly beneficial for training the agent in the given number of steps. In the future, we are planning to study ways of transferring knowledge of obtained RL agents for environments that use raw button presses as inputs. We also hope to extend these experiments to T99 with 99 players and observe if the behaviors observed in a two player setting is the same as that observed in a ninety nine player setting.

6 Group Member Contributions

- Ian Conceicao: implementation of game renderer, gif maker, automatizing of game environment testing, saving / load functionality, training / testing models;
- Viacheslav Inderiakin: implementation of game environment, DQN with fixed target / DDQN implementation for single player / multiplayer scenario, training / testing models;
- Suraj Vathsa: DQN implementation for single player / multiplayer scenario, NN architecture design, training / testing models;

References

[1] James, Ford (2019) "Tetris 99 tips - All the tips for Tetris 99 so you can build your way to victory", GamesRadar, https://www.gamesradar.com/tetris-99-tips/

[2] Tampuu A & Matiisen T & Kodelja D & Kuzovkin I & Korjus K &, et al. (2017) Multiagent cooperation and competition with deep reinforcement learning. PLOS ONE 12(4): e0172395. https://doi.org/10.1371/journal.pone.0172395

[3] L. Yiyuan. Tetris ai - the (near) perfect bot. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player.

[4] OpenAI. (2017). Competitive Self-Play https://openai.com/blog/competitive-self-play/

[5] Stevens, M. & Pradhan, S. (2016) Playing Tetris with Deep Reinforcement Learning.

[6] Gurevich, D. (2017) ICS3U Tetris Game. Github repository. https://github.com/davidgur/Tetris

[7] Long Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. Mach. Learn., 8:293–321, 1992.

[8] Hasslet, H. & Arthur, G. & David S. (2016) Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16).* AAAI Press, 2094–2100.

[9] Vathsa S., Inderiakin V., Conceicao I. (2021), Tetris 99 Reinforcement Learning Agent, https://github.com/SlavaInder/239AS_T99_RL_Agent/blob/main/gifs/multiplayer.gif