Multiplayer Tetris with Reinforcement Learning

https://github.com/arxk9/multiagent_tetris

ALAN ZHENG, BRANDON YANG, and BOHENG MU, School of Engineering and Applied Science, University of Virginia

Multiplayer Tetris is an extension to the standard version of the single player Tetris game where the goal is to outlast your opponents. Players can send opposing players "garbage," which are horizontal grey blocks with one randomly generated hole that fill up from the bottom of the screen, which makes the overall playing area smaller for your opponent.

We used Multi-Agent Reinforcement Learning (MARL) and Deep Q-Learning to train an agent to play multiplayer Tetris. Our work demonstrated that introducing parameter sharing between different agents in Tetris allows agents to accomplish state-of-the-art multiplayer Tetris strategies.

CCS Concepts: • **Computing methodologies** → **Multi-agent reinforcement learning**; **Markov decision processes**.

Additional Key Words and Phrases: Tetris, deep reinforcement learning

ACM Reference Format:
Alan Zheng, Brandon Yang, and Boheng Mu. 2022. Multiplayer Tetris with Reinforcement Learning: https://github.com/arxk9/multiagent_tetris. In . ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The primary motivation for this project is to create a Tetris agent that would also consider its opponent's Tetris board when playing multiplayer mode, similarly to how a human player would behave. Currently, conventional Tetris AI agents are trained to play single-player Tetris, and the same policy is applied to play multiplayer Tetris [3]. However, these agents do not take in the consideration of its opponent's board.

A human player would look at their opponent's board and decide when to send a small amount of garbage with a high frequency, and when to send large amount of garbage with lower frequency. Generally, when the opponent has a smaller playing area (due to large amount of uncleared lines or large amount of uncleared garbage), the best strategy is to send small amounts of garbage frequently. We want to create an agent that uses these information to play multiplayer Tetris.

Multiplayer Tetris can be formulated into a MDP environment, such that state can be represented as the current state of the Tetris board, action can be defined as the legal actions in Tetris (move left, move right, piece rotation) [15]. Since the goal of the agent is to play against other players, we utilized self-play from MARL to develop a smart agent.

2 RELATED WORK

There have been many Tetris agents trained using Reinforcement Learning (RL). Some of these algorithms [6] use traditional value and policy iterations, which improves overall single-player Tetris score, but do not converge to optimal policy. Deep RL has also been widely applied to solve single-player Tetris [13][4][11]. These algorithms use deep Q-networks (DQN) and temporal difference (TD) methods to train the agent, and some are able to achieve optimal policy, in which the agent is able to keep the game playing forever. Most of these deep RL algorithms follow a general heuristic configuration, in which the state vector representation of Tetris is formulated as holes, aggregate height, complete lines and bumpiness [1]. A hole is the determined by the number of empty cells below the occupied cells of each column; aggregate height is the sum of height of all columns in the game space; complete lines are the number of cleared lines; bumpiness is the variation of column heights.

MARL has also been widely applied to game settings [14]. Multi-agent RL algorithms fall under the centralized or decentralized learning [10]. Centralized learning revolves around the idea of using one centralized neural network to develop a joint policy between all agents interacting with the environment, and decentralized learning allows individual agents to learn through their own neural networks. Some environments and game settings use a combination of both [16]. Our work implements characteristics from both centralized and decentralized learning to achieve optimal performance.

However, there has not been extended work on MARL implementation on multiplayer Tetris. Our work is an extension of [18], which is a Tetris agent trained to play single-player Tetris with DQN.

Manuscript submitted to ACM

3 ENVIRONMENT

Our targeted environment for our project was the game of Tetris. Tetris is a game centered around a piece called the tetromino. A tetromino is a 4-polyomino, a shape composed of four squares connected orthogonally.

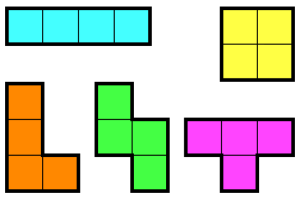

Fig. 1. All possible 7 tetrominos.

The playing field is 10 blocks wide and 22 blocks high. Tetrominos fall from the top of the screen, taken from a randomized sequence. The player can rotate and translate the tetromino as it falls. The objective is to arrange tetrominos so that a horizontal line is filled. Once that happens, that line is cleared. The game ends when the player "tops out," when the next tetromino cannot spawn in because spaces are already occupied there.

In modern Tetris, the sequence randomization is called 7-bag. Piece sequences will come in bags of the 7 distinct tetrominos, but each bag is internally randomized.

In multiplayer Tetris, there are additional mechanics. Players can send garbage lines to the other player's board by clearing lines. To remove garbage, players must rotate and place their blocks so that the hole in the garbage can be filled, clearing that line. Players can send garbage by stacking their tetrominoes in such a way so that their lines can be cleared. Players must clear more than one line at once to send garbage to their opponents. Two lines cleared corresponds to one garbage line sent, three lines cleared corresponds to two garbage lines sent, and four lines cleared, which is the greatest amount possible in Tetris, corresponds to four garbage lines sent. There are additional actions that yield additional garbage sent: t-spins, back-to-backs, combos, and perfect clears. These actions do not yield additional awards in classic single-player Tetris.

Tetris inherently is a Markov decision process [4], as all that needs to be considered at any point in time is the current state of the board(s) and all previous states and actions coalesce into this current state. We implemented our own limited multiplayer environment of Tetris, where t-spins, combos, and perfect clears are not considered.

3.1 States

For our project, we described the state of a single game board with four scalars: height, bumpiness, number of holes, and number of lines cleared. We chose this state encoding both to reduce the state space as well as to allow flexibility for our agent to deal with differing board dimensions and piece shapes.

In our multiplayer environment, we feed both game board state embeddings as the state to each agent. The idea of this is for the agent to take into account the state of the opponent board when choosing its actions. For example, if the opponent board is high and close to topping out, our agent should prioritize sending more garbage lines to "spike" the opponent.

3.2 Actions

One approach to defining the action space in a Tetris environment is to consider each action as a separate, atomic action simulating a single key press. In other words, the agent would be able to take any one of these actions at a time, and each action would be considered independently. This approach would result in a relatively large action space, as there would be many possible actions that the agent could take at any given time.

Alternatively, we defined the action space in a way that allows the agent to take grouped actions, such as placing a piece in a specific location with a specific orientation. This approach would result in a smaller action space, as there would be fewer possible grouped actions that the agent could take. This is less granular and would also reduce of range of states we would have to consider, as we would need to consider the position of the piece currently in play if our action space were atomic.

3.3 Rewards

The reward function for our multiplayer Tetris environment is as follows:

- +1 for each step not reaching a terminal state

- -2 for reaching a terminal state
- + $(cleared\_lines)^2 * board\_width$
- + $\frac{(opponent\_board\_height)-(board\_height)}{10}$
- +10 for the opponent reaching a terminal state

This reward function does incorporate some additional human knowledge (i.e. the difference in board heights and losing opponent) that a single-player Tetris environment does not have. This is to encourage keeping the opponent board high and the agent board low, as well as survival. However, due to the restraint of time and computing power, we add these elements to the reward function for faster convergence.

4 METHODS

4.1 Deep Q-Learning

For our project, we opted to use deep Q-learning, an off-policy, model-free reinforcement learning algorithm [5]. It uses a neural network, or deep Q-network (DQN), to approximate a function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots | s_t = s, a_t = a, \pi] \tag{1}$$

s_t represents the state at time t, a_t represents the action at time t, pi represents the action policy, r_t represents the reward at time t following action a_t from state s_t, and γ represents the discount factor. Since we know that the reward at each time step depends purely on the directly previous state and action, this Q-function obeys the Bellman equation and can be simplified to

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \tag{2}$$

s' is the state that results from taking action a given state s, and a' is the action taken given state s'.

Deep Q-learning also takes advantage of experience replay [20] to improve sample efficiency. It keeps a replay buffer of past states, actions, and rewards. After taking an action in the environment, it samples a mini-batch of experiences to update the DQN and perform gradient descent on. This sampling allows the DQN to learn on a more diverse set of experiences at a time and de-correlate sequential data, improving stability and convergence of the training process [9].

Deep Q-learning learns and approximates the value function from high-dimensional input, which is ideal for our environment with a large state space [7]. It also has improved sample efficiency [19] compared to traditional Q-learning algorithms, which can require a large number of samples to learn a good policy. There is also the potential for transfer learning [17] and continual learning [8], allowing the algorithm to adapt to new tasks and environments without forgetting previous knowledge.

4.2 Multi-Agent Reinforcement Learning

MARL has an advantage in competitive/cooperative setting over individual agents. Having a joint reward and joint game state allows the black box model to better formulate a strategy that will beat the game [12]. In our experiment, the single agent model has an solution for the single player game but when an additional agent was added to the environment it was not able to compete, despite its superior ability to clear lines compared to the Multi-Agent model.

Our training model adopts a decentralized structure. Each agent has their own neural network, which would output their own policies when facing against its opponent. However, we also adopted characteristics from a centralized structure such as implementing joint states and joint reward. In addition to its own states we have defined for the agent, the agent is also able to evaluate its opponents states and parameters. This way, the agent is able to make decisions and alter its policy based on the states of its opponent. Parameter sharing between agents interacting with the same environment is what allows our multiplayer agent to outperform a single player agent [2].

4.3 Behavior Policy

For the behavior policy of our deep Q-learning algorithm, we chose to use decaying ϵ-greedy. The value of our decaying ϵ was $\epsilon_t = \epsilon_f + \frac{max(N_d - t, 0)(\epsilon_0 - \epsilon_f)}{N_d}$, where ϵ_t is the value of ϵ at episode t, ϵ_0 and ϵ_f are the initial and final values of ϵ respectively, and N_d is the number of episodes we want the decay to occur. For our training, we used $\epsilon_0 = 1$, $\epsilon_f = 0.001$, and $N_d = 2000$.

4.4 Semi-Gradient TD

When training our DQN for Q-function approximation The parameter update is defined by the following equation when using mean squared error:

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla[v_\pi(s_t) - \hat{v}(s_t, w_t)]^2 \tag{3}$$

w_t represents the function approximator parameters (in our case, the neural network weights) at time t, α is the learning rate, v_π is the true Q-value, $\hat{v}$ is our function approximator, and s_t is the state at time t. For training, we need some "ground truth" Q-values as a placeholder for v_π in order to establish a gradient for tuning the function parameters.

This can be done with an isolated target network, a separate entity that estimates the true Q-values. For our project, we used one-step temporal difference for estimation, which updates the function parameters as so:

$$w_{t+1} = w_t + \alpha[r + \gamma\hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t)]\nabla\hat{v}(s_t, w_t) \tag{4}$$

Note that using this method is called semi-gradient TD(0), as this bootstrapping target depends on the current value of w, which breaks the key assumption of the gradient update rule. This ignores part of the gradient, and is not guaranteed to converge.

4.5 Network Architecture

For our DQN used for Q-function approximation, we opted for a vanilla neural network with 4 layers: a 5-dimensional input layer, two 64-neuron hidden layers, and a single-neuron output layer. We chose to use Leaky ReLU activation function for all layers.

We had initially tried feeding the entire state embedding of the opponent board as part of the input (resulting in an 8-dimensional input layer), but we did had neither the computing power (even on GPU servers) or time to train to convergence. Even after 100,000 episodes, the agent had only marginally improved. We opted to only feed the bumpiness of the opponent board in, which should still provide information about board vulnerability.

4.6 Training

We trained mutiplayer agents on a 20 by 10 sized Tetris board. Later we will show that our trained policy performs just as well on board of all sizes. The size of the experience replay is 30,000, and each time the algorithm takes 512 random batches to evaluate. Learning rate was set to 0.001. γ is 0.99. For our epsilon decay function, we chose initial $\epsilon = 1$, and final epsilon $\epsilon_{final} = 0.001$. We trained for 50,000 epochs.

5 RESULTS

5.1 Evaluation Metrics

The performance of the agent is evaluated on three metrics: tetrominoes played, lines cleared, and final score. Tetrominoes played is a measurement of survival fitness, the more tetrominoes played the longer the agent is able to go without topping out. Lines cleared is an assessment of the agent's understanding of the game, where clearing lines can both improve the score and send garbage to the opponent. Tetrominoes played and lines cleared come together to make the final score which is fine-tuned to value cleared lines more. These metrics will allow us to access the progress of the agents and compare them.

5.2 Training Results

We trained 50,000 epochs using hyperparameters mentioned above. The graphs depecit two lines, each corresponding to an agent. Both lines follow the same trend throughout the training process, which is a characteristic of self-play MARL. The number of tetrominoes is the same for both agents throughout training, which makes both lines overlap. The overall trend of both agents displays continuous increasing functions.

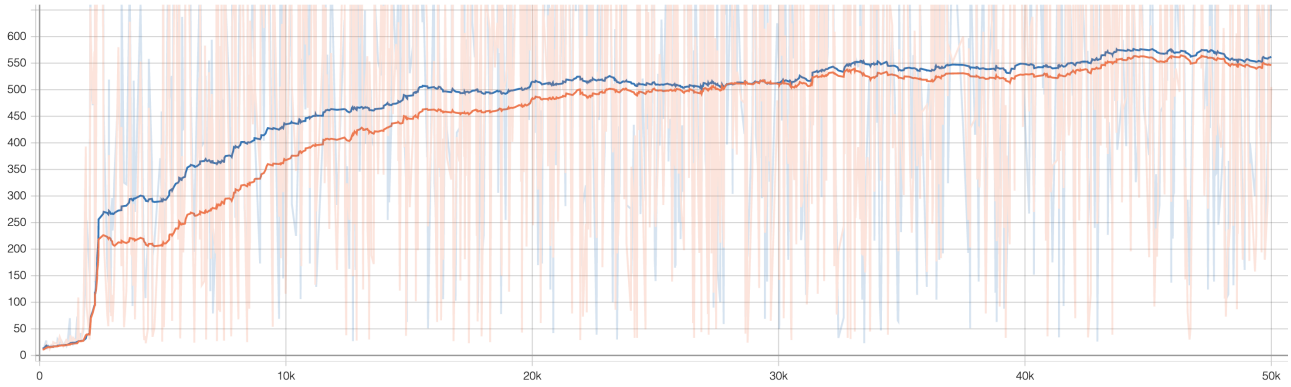

Fig. 2. Scores

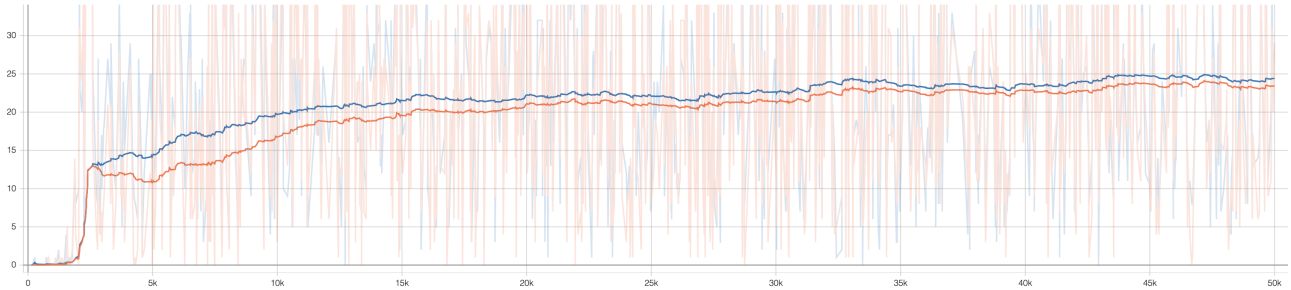

Fig. 3. Cleared Lines

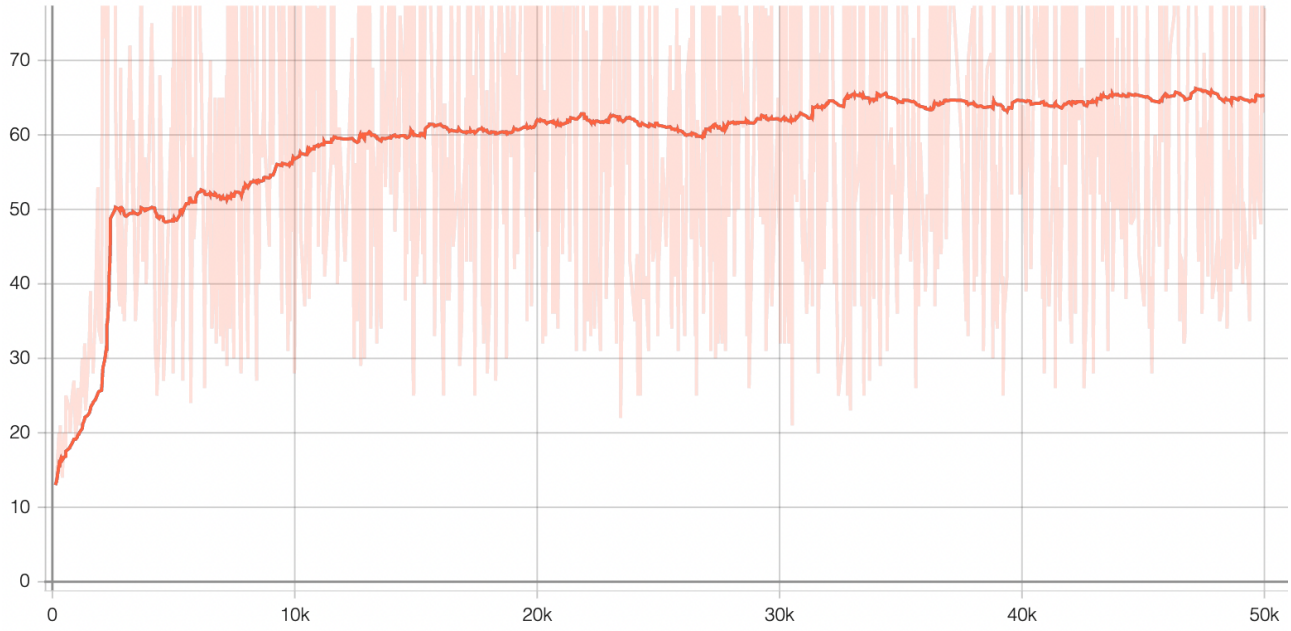

Fig. 4. Number of Tetrominoes

5.3 Experiments

After the multi-agent model was trained, We also trained the single-agent model on a single-player game. There is no opponent and the goal is to last as long as possible. Figure 5 compares the training score of the single-agent model in blue and the multi-agent model in red.

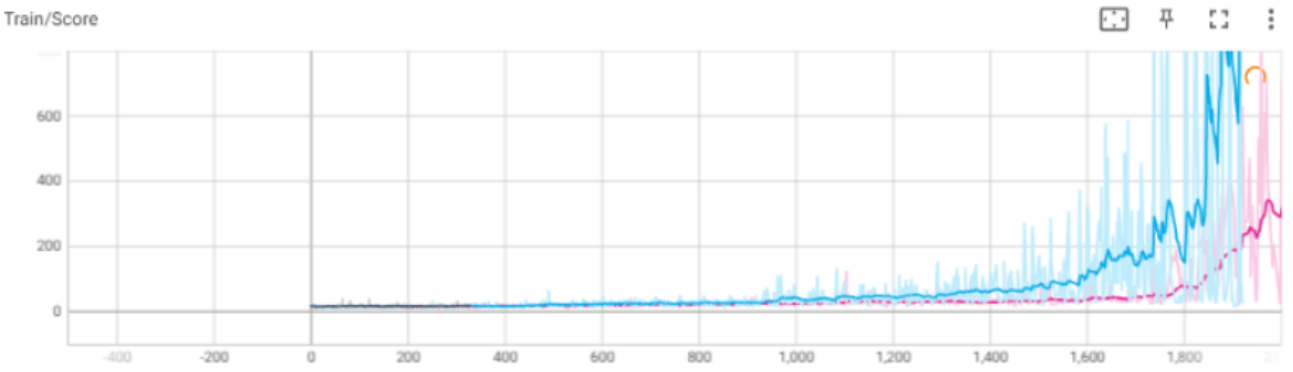

Fig. 5. Mulit-agent model vs Single-agent model score performance

We have also set up an environment where the single-agent model and the multi-agent model are pitted against each other. This means that the multi-agent model will have insight into the state of the single-agent model, but not vice versa. The multi-Agent model beats the single-agent model 87.2% in 3000 games and table 1 shows the average statistics.

	Multi Agent-Model	Single-Agent Model
Average Score	730.41	613.93
Average Lines Cleared	35.90	29.03

Table 1. Experiment results

5.4 Analysis

Despite the single-agent model having a higher score and the ability to play over 100,100 tetrominoes in the single-player mode, the multi-agent model was able to consistently beat it in multiplayer mode. This shows that taking into account the opponent's board state has a significant competitive advantage. The strategy of the single-player mode seems to favor clearing lines as often as it can, while the multi-agent model takes advantage of the garbage sent by clearing multiple lines. The multi-agent model would often save up lines in a "well" and clear all 4 with a long tetromino, while the single-agent model would prefer to keep the height low and clear 1 line whenever possible.

6 CONCLUSION

In conclusion, our project has successfully demonstrated the effectiveness of using deep Q-learning for MARL in a multiplayer Tetris environment. By including the opponent board in the game state, our multiplayer agent was able to outperform the single player agent consistently. This result indicates that the multiplayer agent was able to learn strategies that took into account the actions of the opponent and made more optimal decisions as a result.

While the single player agent was able to survive longer in a single player environment, the multiplayer agent's ability to consider the actions of the opponent allowed it to achieve better overall performance. This suggests that including the opponent's board in the game state can be a valuable approach for improving the performance of reinforcement learning agents in multiplayer environments.

Overall, this project has provided valuable insights into the use of deep Q-learning for MARL, and has potential implications for the application of this approach in other multi-agent scenarios. It also highlights the importance of considering the states of opponents in multiplayer environments, and the potential benefits of incorporating this information into the game state for RL agents.

6.1 Future Work

In the future, there are several extensions and improvements we could possibly make to improve the performance of our agents as well as make the agents more independent.

First, there are modifications we could make to our state and action spaces. Currently, we feed a 4-dimensional board-state embedding to represent the game board, but to allow the agent to learn more features about the game, we could pass in the raw grid values as input instead. We could implement a convolutional neural network (CNN) to create a state embedding for the grid data to reduce the

state space, or we could use a CNN in our DQN approximator to handle the massive state space itself, which would be $2^{width \cdot height}$ large. Additional things such as the upcoming piece, held piece, and upcoming bag could also be added to the state to allow for more agent planning.

We could also expand our action space to allow for more advanced multiplayer techniques. Modern Tetris games use the super rotation system (SRS) to allow for t-spins and other piece spins, but we did not have the time to implement this and add spins to our action space. We also didn't have time to add tucks, or placing a piece below impeding blocks by sliding it from the side, to our action space, which we could have with more time. Lastly, we could add "holding" a piece for future use in our action space, which is implemented in most Tetris games.

Next, there are alternatives to our training itself. We currently use semi-gradient TD(0), but a separate target network utilizing the full gradient could be used to improve training and training stability. Prioritized sweeping could also be implemented for sampling from the experience replay instead of simple random sampling based on TD error, which would stabilize and speed up convergence.

ACKNOWLEDGMENTS

To Dr. Wang and the rest of the CS 4501 instructor team, for the sandwiches, guidance, and teaching this semester.

REFERENCES

[1] Max Bergmark. 2015. Tetris: a heuristic study: using height-based weighing functions and breadth-first search heuristics for playing tetris.
[2] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. 2010. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1* (2010), 183–221.
[3] Donald Carr. 2005. Applying reinforcement learning to Tetris. *Department of Computer Science Rhodes University* (2005).
[4] Ziao Chen. 2021. *Playing Tetris with deep reinforcement learning.* Ph. D. Dissertation.
[5] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. 2020. A theoretical analysis of deep Q-learning. In *Learning for Dynamics and Control*. PMLR, 486–489.
[6] Alexander Groß, Jan Friedland, and Friedhelm Schwenker. 2008. Learning to play Tetris applying reinforcement learning methods.. In *ESANN*. 131–136.
[7] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. 2014. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. *Advances in neural information processing systems* 27 (2014).
[8] Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. 2020. Towards continual reinforcement learning: A review and perspectives. *arXiv preprint arXiv:2012.13490* (2020).
[9] Sascha Lange, Thomas Gabel, and Martin Riedmiller. 2012. Batch reinforcement learning. In *Reinforcement learning*. Springer, 45–73.
[10] Michael L Littman. 1994. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*. Elsevier, 157–163.
[11] Nicholas Lundgaard and Brian Mckee. 2007. Reinforcement Learning and Neural Networks for Tetris.
[12] David Henry Mguni, Taher Jafferjee, Jianhong Wang, Nicolas Perez-Nieves, Oliver Slumbers, Feifei Tong, Yang Li, Jiangcheng Zhu, Yaodong Yang, and Jun Wang. 2021. LIGS: Learnable Intrinsic-Reward Generation Selection for Multi-Agent Learning. *arXiv preprint arXiv:2112.02618* (2021).
[13] Matt Stevens and Sabeek Pradhan. 2016. Playing tetris with deep reinforcement learning. *Convolutional Neural Networks for Visual Recognition CS23, Stanford Univ., Stanford, CA, USA, Tech. Rep* (2016).
[14] DiJia Su, Jason D Lee, John M Mulvey, and H Vincent Poor. 2022. Competitive Multi-Agent Reinforcement Learning with Self-Supervised Representation. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 4098–4102.
[15] István Szita and András Lörincz. 2006. Learning Tetris using the noisy cross-entropy method. *Neural computation* 18, 12 (2006), 2936–2941.
[16] Zhenheng Tang, Shaohuai Shi, and Xiaowen Chu. 2020. Communication-efficient decentralized learning with sparsification and adaptive peer selection. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1207–1208.
[17] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 242–264.
[18] uvipen. 2019. Deep Q-learning for playing Tetris. https://github.com/uvipen/Tetris-deep-Q-learning-pytorch.
[19] Dujia Yang, Xiaowei Qin, Xiaodong Xu, Chensheng Li, and Guo Wei. 2020. Sample efficient reinforcement learning method via high efficient episodic memory. *IEEE Access* 8 (2020), 129274–129284.
[20] Dongbin Zhao, Haitao Wang, Kun Shao, and Yuanheng Zhu. 2016. Deep reinforcement learning with experience replay based on SARSA. In *2016 IEEE symposium series on computational intelligence (SSCI)*. IEEE, 1–6.

Received 15 December 2022