University of Science and Technology of Hanoi

UNDERGRADUATE SCHOOL

Research and Development

BACHELOR THESIS

By

Hồ Minh Quang - BI12-370

Information and Communication Technology

Title

Reinforcement Learning in Tetris

Supervisor: Prof. Đoàn Nhật Quang

Lab name: USTH ICTLab

Hanoi, 2024

Declaration

I hereby, Ho Minh Quang, declare that my thesis represents my work after doing the internship at ICTLab, USTH.

I have read the University's internship guidelines. In the case of plagiarism appear in my thesis, I accept responsibility for the conduct of the procedures under the University's Committee.

Hanoi, July 2024

Signature

Ho Minh Quang

Acknowledgements

First and foremost, I extend my deepest gratitude to my supervisor, Professor Doan Nhat Quang. More than a mentor, Professor Quang has been an exceptional guide, providing unwavering support and insightful evaluations throughout this project. His guidance has been invaluable, and I am truly appreciative of the journey and the challenges we have navigated together.

Table of Content

1. Introduction

Reinforcement learning, a machine learning technique, focuses on allowing continuous learning for an agent. It does this by allowing the agent to recall its experiences, providing it with sufficient memory of environmental events, and assigning rewards or punishments to its actions according to a predefined reward policy. Nevertheless, a challenge with traditional reinforcement learning is that the agent's storage requirements and exploration time expand exponentially as the dimensions of the problem increase linearly.

Tetris, a game created in 1985 by Alexey Pajitnov, has become an important subject of research in mathematics and machine learning. Although seemingly simple, Tetris is proven to be NP-complete, posing a substantial challenge. This thesis focuses on applying reinforcement learning to Tetris, a non-trivial task due to the game's massive amount of possible states.

Our goal is to explore the capabilities of deep reinforcement learning within the context of the Tetris game. We aim to utilize Deep Q-Learning and Double Deep Q-Learning, a reinforcement learning variant, to train an agent for Tetris. This approach combines the strengths of deep neural networks with Q-Learning, enabling the agent to master complex strategies.

This research examined the effects of different hyperparameters, utilized a replay buffer for experience replay, implemented an epsilon-greedy strategy for action selection, and leveraged professional game replays for training purposes. Previous implementations of Deep Q-Learning in gaming have yielded promising outcomes, such as DeepMind's DQN agent reaching human-level proficiency in various Atari games. Nonetheless, applying Deep Q-Learning to Tetris introduces distinct challenges due to the game's inherent complexity.

In this thesis, we aim to:

- Develop a Deep Q-learning agent and Double Deep Q-learning agent for Tetris.
- Investigate the impact of different hyperparameters on the agent's performance.
- Evaluate the effectiveness of a replay buffer in experience replay.
- Implement an epsilon-greedy strategy for action selection.
- Use professional game replays for training the agent.

As a result, advancing our understanding of Deep Q-Learning and its application to complex game environments like Tetris.

Beginning with a basic algorithm that employs a quadratic reward function and a standard Q-learning framework, we experimented with incorporating additional

factors that might influence the agent's playstyle. By using Tetris's scoring system as the output of the reward function, the agent has learned to execute basic T-spins and achieve higher scores in less time than the previous protocol allowed.

With only a restricted training period of 6000 episodes with a batch size of 164, our final model managed to achieve an average score of 197137.5. The model executed

- 446 Single Line Clears
- 196 Double Lines Clear
- 49 Triple Line Clears
- 63 Quad Line Clears which is called Tetris.

The model has successfully executed 36 T-spins, which is impressively good. Our experiments indicate that deep reinforcement learning has significant potential for creating an exceptionally efficient Tetris player.

2. Background

This chapter provides a brief overview of Tetris, the challenges associated with applying Reinforcement Learning to the game, and an introduction to the methods we have explored.

2.1. Tetris

Tetris is a 30-year-old video game played on a 20x10 grid gameboard as shown in Figure 1.

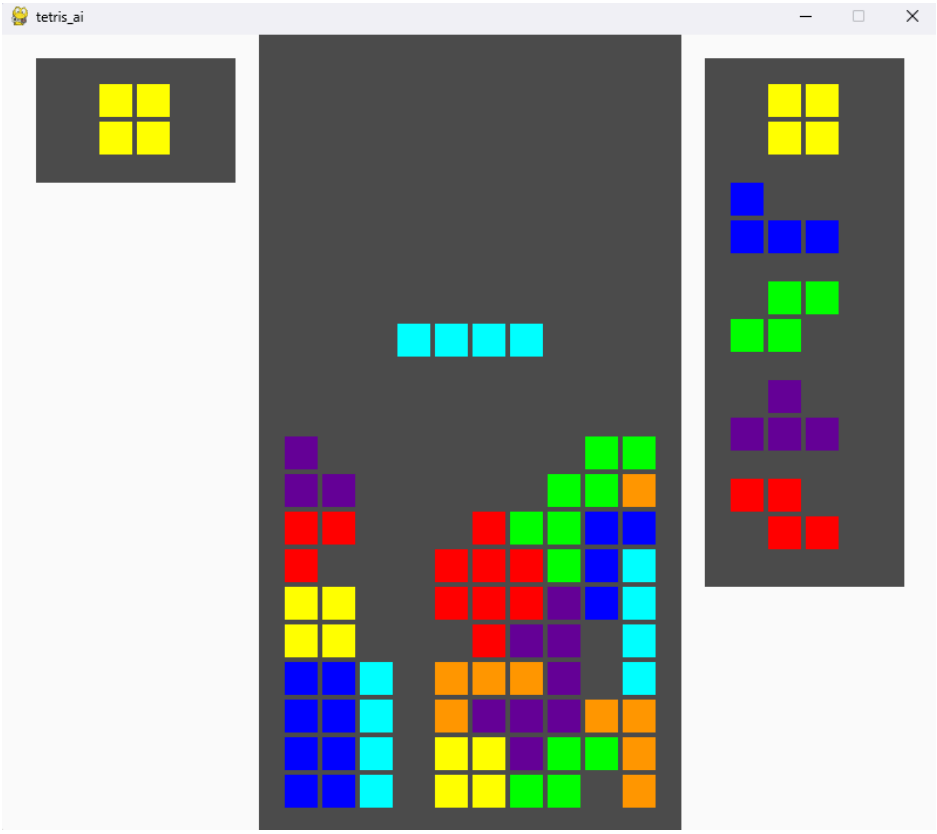


Figure 1: A Tetris board with 10 x 20 blocks

In each round of the game, a four-block piece known as a Tetromino appears at the top of the gameboard and slowly descends to the bottom. The current Tetromino is randomly chosen from a queue of seven pieces. This queue is replenished with seven new pieces whenever its length drops to two or fewer. Players must strategically position each Tetromino to clear lines by completely filling them, thus removing older pieces. There are seven distinct types of Tetrominoes, as indicated by Figure 2.

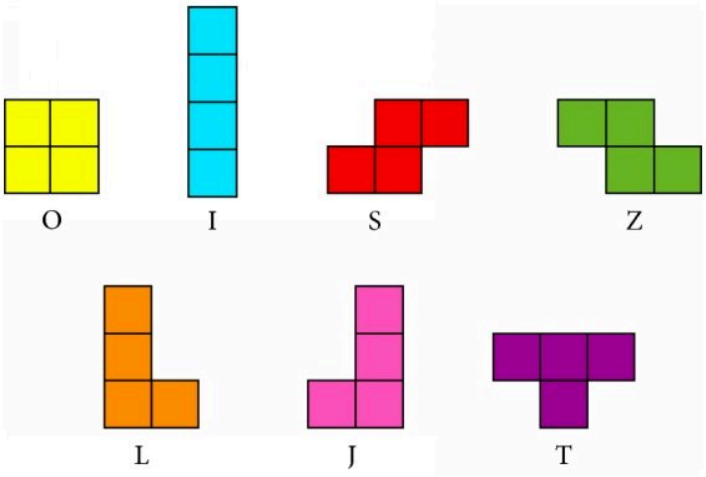


Figure 2: Seven types of Tetrominoes, usually called I, J, L, O, S, T, and Z piece based on their shape

Players can rotate the Tetromino or shift it to the left or right. Once a Tetromino hits the bottom of the gameboard or collides with another Tetromino that's already in a fixed position, it ceases to move and the players can no longer alter its state. The game concludes when a newly generated Tetromino reaches the top of the gameboard. To prolong the game, players must eliminate occupied spaces by filling a line. Once a line is fully occupied, it gets cleared from the gameboard and all the blocks positioned above it descend by one line.

The scoring mechanism, or even the objective of gameplay, can vary across different Tetris games. In some versions, the system awards a flat rate of 1 point for each line cleared, with the primary goal for players being survival to accumulate high scores. However, in most games, the system grants additional points for superior clears. For instance, players might earn 4 points by clearing 4 lines individually, or they could earn 16 points by clearing them all at once. The strategies for these two game types can be entirely different. One might maintain a low pile by clearing a single line at a time in the first type, but such a strategy would result in the lowest score in the second type. In the earliest game setup, the generation of Tetrominoes is entirely random. Players might have to wait for 40 pieces before receiving another 'I' piece, a situation referred to as a 'drought' by professional players. In contemporary Tetris games, however, the system generates a bag containing seven different pieces and randomly selects a piece from the bag each round. This setup can prevent extreme conditions like piece droughts and rain of 'Z', and 'S' pieces.

2.2. Reinforcement Learning

The goal of reinforcement learning is to create or train an agent by letting the agent interact with the environment. The agent learns from the consequences of its actions, rather than from being explicitly taught, and it selects its actions based on its past experiences (exploitation) and also by new choices (exploration).

The agent's observations from the environment are defined as a state, denoted as 's'. For instance, the layout of a Tetris board can be considered a state. In this context, 'S' represents the collection of all possible states of the gameboard. At every state 's', the agent either chooses the best action from the action space 'A' according to the current policy or opts for a random action in pursuit of a potentially superior solution. After completing the current action 'at' at each time step 't', the agent is rewarded with a numeric value 'rt' from the environment and observes a new state 's'. Our ultimate objective is to maximize the total of 'rt' before the agent reaches a termination state.

Q-learning, the heart of reinforcement learning, can directly approximate the optimal function. The inputs of Q-functions are the current state and action, and the agent will choose the state action pair that has the largest Q-value.

$$Q(s,a) = \mathbb{E}_{s'}[r + \max_{a'}(Q(s',a'))]$$

The update rule for Q-learning, where s and a are current state-action pairs and r is the reward gain from current action:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \max_{a'}(Q(s',a')) - Q(s,a))$$

The limitation of traditional Q-learning is the necessity to maintain a Q-value table for every possible state-action pair, which becomes impractical in environments with large state spaces like Tetris. To address this, Deep Q-Learning (DQN) is proposed. DQN employs a neural network to approximate the Q-function, bypassing the need for a Q-value table. This replacement enables the agent to manage large state spaces found in video games, where using a table would be impractical.

The Q-network accepts the state and action as inputs and outputs an estimated Q-value. Initially, a Feed Forward Neural Network is utilized to extract features from the game state and calculate the Q-values for each possible action. The agent then selects an action based on these Q-values. Subsequently, we explore Double Deep Q-Learning, which incorporates a Convolutional Neural Network (CNN) rather

than a simple Feed Forward Neural Network, significantly expanding the agent's capabilities.

2.3. Epsilon greedy

It's important to address how the agent decides on which action to take at each step. In both Q-learning and DQN, the agent needs to balance exploration (trying out new actions to discover their effects) and exploitation (choosing the best action based on current knowledge). The agent learns to predict the Q-values, which represent the expected future rewards for each action in each state. The action with the highest Q-value is typically chosen as the next action to perform. However, this strategy, known as greedy action selection, can lead to suboptimal learning. The agent may get stuck in a local optimum because it exploits its current knowledge without exploring potentially better actions.

To address this issue, we introduce an element of randomness in the action selection process using an epsilon-greedy strategy. In epsilon-greedy, the agent chooses a random action with a probability of epsilon and the action with the highest Q-value with a probability of 1 - epsilon. This approach balances exploration, the process of trying out new actions to improve knowledge, and exploitation, the process of using current knowledge to maximize rewards.

In our research, we used an epsilon value that starts at 0.5 and decays linearly to 0.08 for 3000 steps. This value was chosen because it provides a balance between exploration and exploitation. At the beginning of training, a higher epsilon value encourages more exploration (random action selection), and as training progresses, epsilon is reduced to encourage more exploitation (choosing the action with the highest predicted reward). The epsilon-greedy strategy with this epsilon value allowed the agent to explore the environment sufficiently while also exploiting its learned knowledge to achieve high rewards. We will go deeper into this in the next chapter

2.4. Experience Replay

It's important to introduce another crucial concept in reinforcement learning: Experience Replay. While the epsilon-greedy strategy is a method for exploration, deciding whether to take a random action or follow the policy, Experience Replay is a technique for learning from past experiences. In a typical reinforcement learning setup, an agent learns from the immediate reward of its actions. However, this approach can lead to a lack of diversity in the experiences, as the agent is only learning from consecutive experiences. Experience Replay mitigates this issue by maintaining a replay buffer of past experiences. The agent, instead of learning

from the latest experience only, randomly samples a batch of past experiences from the replay buffer for learning. This method provides a more diverse range of experiences for the agent to learn from, breaking the correlation between consecutive experiences and stabilizing the learning process. In the next section, we will delve deeper into the implementation and benefits of Experience Replay."

The main challenge in this project is the complexity of the Tetris game environment. The state space is high-dimensional, and the rewards for clearing lines are delayed, making it difficult for the agent to learn effective strategies. Additionally, the use of professional replays for training presents unique challenges, as it requires the agent to learn from both its own experiences and those of professional players.

3. Methodology

3.1. Environment

The agent is implemented in Python using the PyTorch library for deep learning. The game environment is a standard Tetris game with a 10x20 grid. The agent is trained on a machine with a CUDA-enabled GPU.

3.1.1. Game state

The state of the game is represented by a Board object, which is a 2D grid representing the game board. The state includes the current position of the tetromino, the type of the current tetromino, and the type of the next tetromino in the queue. We will return the current state of the game to the agent.

Besides all of the mentioned above, the game state also includes various statistics about the current game, such as the player’s current score; the number of moves the player has made; An array that tracks the number of single, double, triple, and quadruple line clears the player has made; the number of T-spin the agent has performed; the number of lines cleared in the last move; the score obtained in the last move; the highest back-to-back streak achieved; the highest combo achieved. This will be utilized in future development.

The action space consists of the possible moves that the agent can make. In Tetris, these are moving the current tetromino to the left, or right, rotating it, or dropping it. We have created the step method to take action and update the state of the game accordingly.

About the agent interacting with the environment, we have studied the MisaMino AI Tetris[1] created in 2013. The article stated that: “the AI generates a limited number of playfields for each level (considered next pieces) before going to the next level”. So we have created a method that generates all possible next states by placing the current piece at nearly all possible positions. Each possible next state corresponds to a different action that the agent could take. The method will return a tuple containing the next states, the scores for each state, the number of lines cleared by each state, and a boolean indicating whether the game is over for each state. Each possible next state corresponds to a different action that the agent could take, such as rotating the piece or moving it to a different position.

3.1.2. Reward function

The reward function is a rule that specifies the feedback the agent receives as a consequence of its actions in a given state. The agent’s goal is to learn a policy that maximizes the sum of these rewards over time. We have tested the agent to use a quadratic reward function (Reward function A, due to the original game using the square of lines cleared as scoring rules) and reward the agent for utilizing all of the widths of the board. The reason why in the first place we will try to use the widths of the board for the reward function is that the agent when start learning tends to just focus and drop all the pieces on each other and will not spread them out

[1]https://tetris.fandom.com/wiki/MisaMino

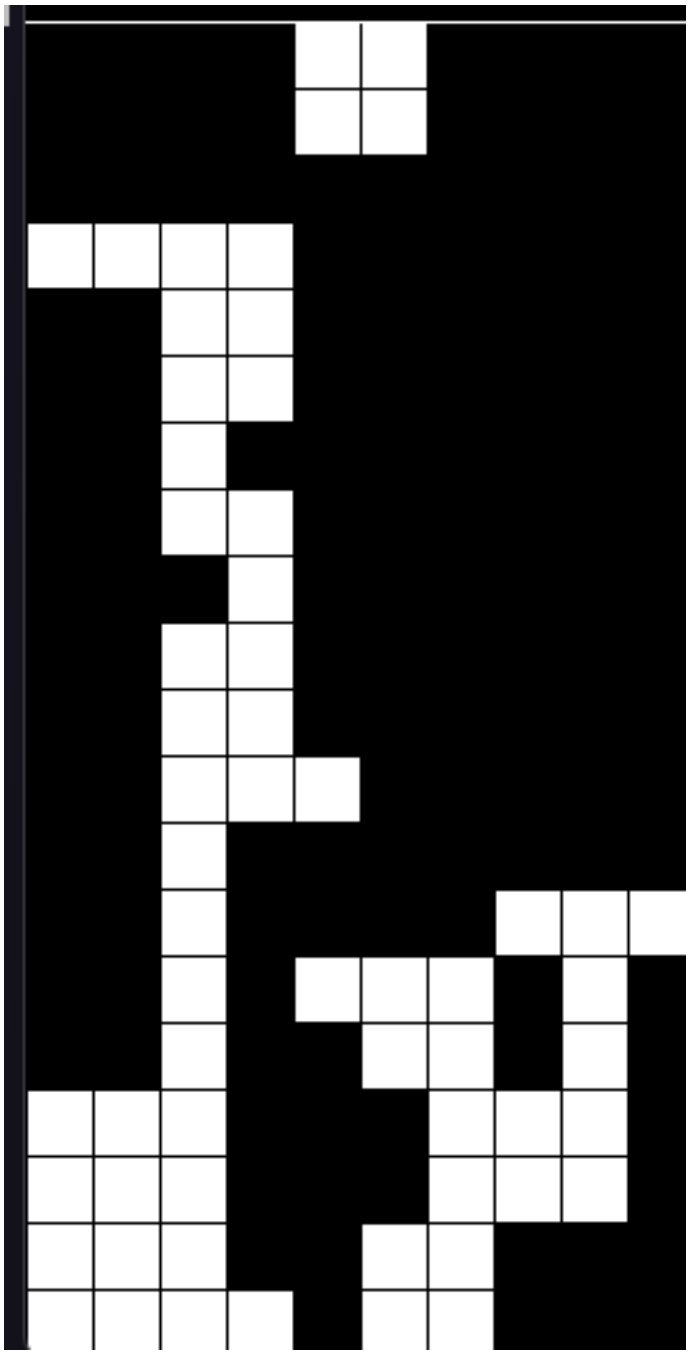


Figure 3: The agents continue stacking blocks on each other and lack spreading

After creating reward function A, the agent has played better and now spreads the pieces to the full board and does not just instantly drop on each other like before, and can survive for a long time. But the current agent does not know how to do T-spin or try to clear 4 lines at once and just keep clearing line by line. This is because of the reward function A itself and also because of the exploration-exploitation trade-off which we will cover in the next section. About the function, we have tried and tested a new reward function B. This new reward function will calculate the reward based on the number of cleared lines, whether a t-spin was performed, and the current combo and back-to-back streaks. The base score is determined

by the number of cleared lines and whether a t-spin was performed. Additional points are added if the agent has a combo or a back-to-back streak, or if it clears the entire board. The scoring system will be based on the official Tetris guideline scoring system. This will help the agent to try to learn to place pieces in a way that clears as many lines as possible and not just clear one line at a time. The reason why we don't attempt to train the agent to do an all-clear is that it is really difficult to set up and execute an all-clear, which will make the reward function too complex, making the learning process slower and more difficult, and even making the agent perform less efficient. We will cover the result of this reward function B in a different section.

3.1.3. Hold action and heuristics

For the environment to be as similar as possible to modern Tetris, we have created the hold method, which allows the agent to swap the current piece with a held piece - a feature in modern Tetris games. If there is no piece currently being held, the current piece is held and a new piece will be taken from the bag.

To train an agent efficiently, we will have to dig deeper into the game state. That is why we have created some features that follow the guidelines from Tetris AI – The (Near) Perfect Bot[2]. The research stated that: "The score for each move is computed by assessing the grid the move would result in. This assessment is based on four heuristics: **aggregate height**, **complete lines**, **holes**, and **bumpiness**, each of which the AI will try to either minimize or maximize.". So we have added some features to the game that are used to evaluate the current state of the game and make decisions about the next action to take:

- **Number of holes**: This calculates the number of holes in the game board. A hole is defined as an empty space that has a filled space somewhere above it in the same column.
- **Number of connected holes**: This calculates the number of connected holes in the game board. A connected hole is defined as an empty space that has a filled space both above and below it in the same column.
- **Column heights**: This calculates the height of each column, the maximum height, the minimum height, and the differences in height between adjacent columns. These features can be used to evaluate the "bumpiness" or unevenness of the game board.

[2]https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/

- **Wells**: This calculates the wells in the game board. A well is defined as an empty space that has filled spaces on both sides.
- **Number of occupied cells weighted**: This calculates a weighted sum of the filled spaces on the game board, where spaces higher up on the board are given more weight.

With the heuristics set up, we can make use of it in the future work

3.1.4. Kick mechanism

When the agent attempts to rotate a tetromino, but the position it would normally occupy after basic rotation is obstructed (either by a wall or floor the playfield, or by the stack), the game will attempt to "kick" the tetromino into an alternative position nearby. This mechanic will unlock much more options for the agent, which includes Tspin. We have learned and adapted from the SRS (Super Rotation System[3]) from the Tetris game to implement this mechanic. The article stated that:
"

- When a rotation is attempted, 5 positions are sequentially tested (inclusive of basic rotation); if none are available, the rotation fails.
- Which positions are tested is determined by the initial rotation state and the desired final rotation state. Because it is possible to rotate both clockwise and counter-clockwise, for each of the 4 initial states there are 2 final states. Therefore there are a total of 8 possible rotations for each tetromino and 8 sets of wall kick data need to be described.
- The positions are commonly described as a sequence of (x, y) kick values representing translations relative to basic rotation; a convention of positive x rightwards, positive y upwards is used, e.g. (-1,+2) would indicate a kick of 1 cell left and 2 cells up.
- The J, L, S, T, and Z tetrominoes all share the same kick values, the I tetromino has its own set of kick values, and the O tetromino does not kick.
- Several different conventions are commonly used for the naming of the rotation states. In this article, the following convention will be used:
  - 0 = spawn state
  - R = state resulting from a clockwise rotation ("right") from spawn
  - L = state resulting from a counter-clockwise ("left") rotation from spawn
  - 2 = state resulting from 2 successive rotations in either direction from spawn.

[3]https://tetris.wiki/Super_Rotation_System

	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)

Table 1: J, L, S, T, Z Tetromino Wall Kick Data

We have implemented this mechanic into the environment. This has opened up the ability to learn and execute the Tspin of the agent. This is the key difference between the environment that only allows the agent to clear line by line and an environment that enables the agent to fully rotate pieces into the gap and Tspin set-up place.

3.2. Deep Reinforcement Learning

We have made 2 experiments. One is with a DQN with a Feed Forward neural network and one is with Double DQN with CNN.

3.2.1. Model Architecture

3.2.1.1. Feed Forward neural network

Our first model consists of a feed-forward neural network, specifically designed as a Multilayer Perceptron (MLP) for approximating Q-values.

The network architecture comprises three linear layers: Input, hidden, and Output layer. The Input layer takes the states of the Tetris game as input. The state of the current configuration of Tetris blocks on the board. It has 4 input features and transforms it into a 64-dimensional space. The ReLU activation function introduces non-linearity, allowing the network to learn complex patterns. The hidden layer further processes the data from the input layer, again using 64 neurons and ReLU for non-linearity. It helps in capturing the relationships between different game states and the actions taken. The final layer reduces the dimensionality from 64 to 1, outputting a single value: the Q-value for the given state-action pair. This value represents the expected cumulative reward that can be achieved from the current state by following the policy learned by the network.

3.2.1.2. Double Deep Q-Network

The reason why we wanted to use Double DQN is that it has solved prolems that DQN can't do. Double DQN is an enhancement over the traditional Deep Q-Network to address the problem of overestimation of Q-values in the Q-learning algorithm. In standard DQN, the same network (the Q-network) is used to select and evaluate an action, which can lead to over-optimistic value estimates. Double DQN, introduced by Hado van Hasselt et al., separates these two steps by using two networks: a policy network for selecting the best action and a target network for evaluating that action's value.

Our model consists of 2 neural networks: the Q-network and the target Q-network. The Q-network will be used to estimate the Q-values representing the expected future rewards of taking each action in the current state. This network will be trained to minimize the difference between its predicted Q-values and the observed rewards plus the discounted estimated Q-values of the next state.

The Target Q-network will be used to estimate the Q-values of the next states. The target Q-network is a copy of the Q-network that is updated less frequently. This is done to prevent the target Q-values from changing too quickly, which can lead to instability and divergence in the learning process. By keeping the target Q-values relatively stable, we provide a fixed target for the Q-network to learn from during each update step.

The network consists of three convolutional layers, followed by a max pooling layer, a flattening layer, and three fully connected layers.

The first layer is a 2D convolutional layer with 32 output channels and a kernel size of 5. The number of output channels represents the number of filters that the layer will learn. Each filter is capable of recognizing a different feature in the input data. The kernel size is the size of the filter that is applied to the input data. A kernel size of 5 means that each filter will cover a 5x5 area of the input data. This size is chosen to capture larger patterns in the game state, such as the shape of a Tetris piece or a gap in the grid. The second and third convolutional layers have 64 output channels and a kernel size of 3. Increasing the number of output channels allows the network to learn more complex features. Reducing the kernel size allows the network to recognize smaller, more detailed features, such as the edges of a Tetris piece or a single block in the grid.

After the convolutional layers, there is a Max Pooling Layer with a kernel size of 2. pooling layer with a kernel size of 2. The max pooling layer reduces the spatial dimensions (width and height) of the input data by taking the maximum value over

the window defined by the kernel size. This helps to reduce the computational complexity of the network and also helps to make the network invariant to small translations in the input data. This could help the network to recognize the same patterns in different parts of the grid.

The max pooling layer is followed by a flattening layer, which reshapes the 2D feature maps into a 1D tensor that can be fed into the fully connected layers. The first two fully connected layers have 256 and 128 neurons respectively, and use the ReLU activation function. The ReLU function introduces non-linearity into the network, allowing it to learn more complex patterns. The function is defined as f(x) = max(0, x), which means that it sets all negative values in the feature map to zero. The last layer has 1 neuron and uses a linear activation function. This layer outputs the estimated Q-value of the action. The number of neurons in the fully connected layers is chosen to balance the complexity of the patterns that the network can learn and the computational cost of training the network.

DRAW THE NEURAL NETWORK ARCHITECT

3.2.2. Epsilon greedy

We have implemented an epsilon-greedy strategy in the learning and training process for both approaches. After many testing, we decided that the starting epsilon would be set at 0.5 because it would provide a balance between exploration and exploitation for the agent to learn effectively. We have tested different values of the starting epsilon and see that 0.5 is the optimal value. The reason why we don't put it as lower than 0.5 or just flat 0.1 is because this will make the agent heavily biased towards exploitation. It will mostly take the action that it currently believes to be the best, based on its limited initial knowledge. In the early stages of learning, this knowledge is likely to be poor or incorrect. This could lead to repetitive and suboptimal behavior. For example, in our protocol, we came across the starting epsilon value of 0.1. When we started the learning process, the agent always dropped pieces straight down without rotating them, or it might always place pieces in the same location. This could also lead to a quick game over, as the agent is not exploring different actions to learn about their consequences.

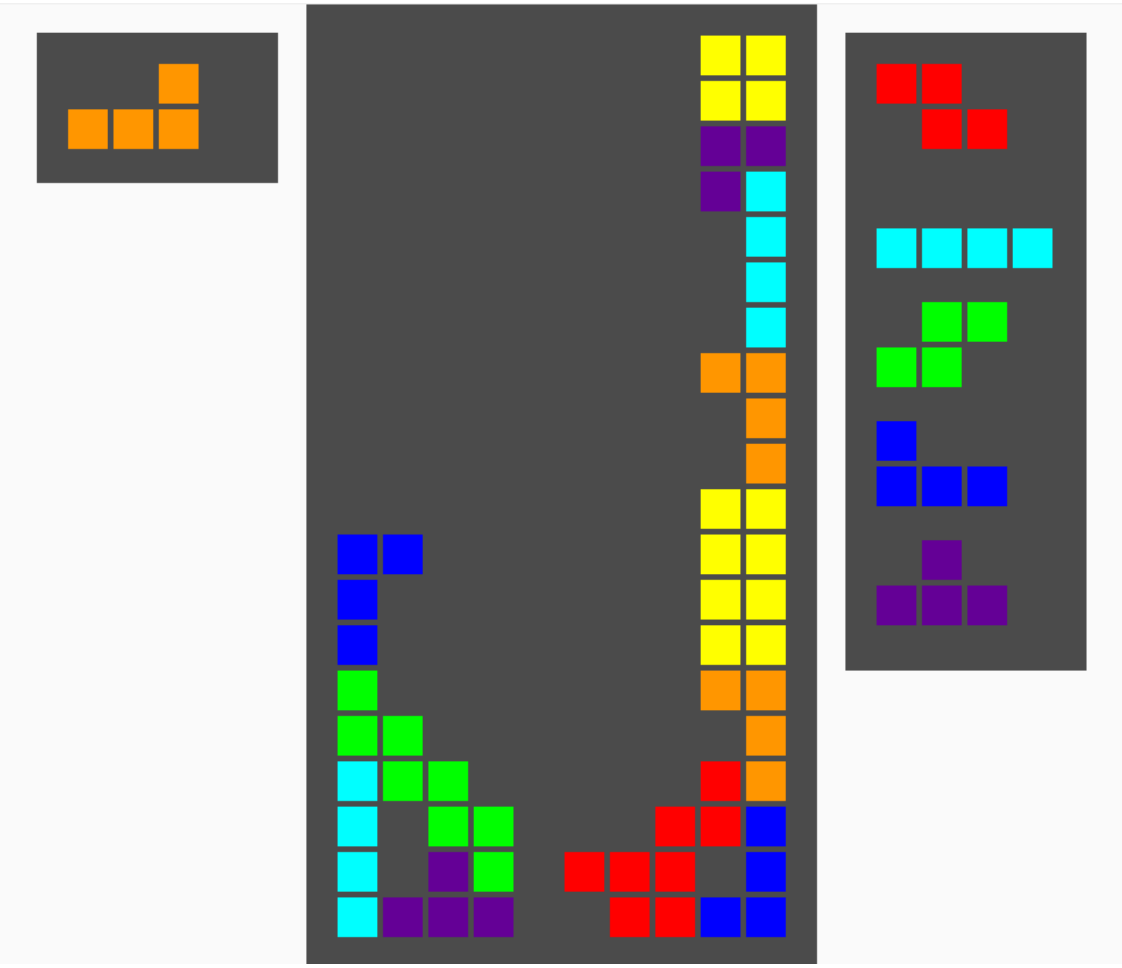


Figure 4: Agent when using epsilon = 0.1

We have also tested with a high starting epsilon such as 1.0. This can still work fine and usable but in some environments that have a different reward system, the agent will have erratic behavior.

3.2.3. Experience Replay

In the Double DQN approach, a replay buffer is implemented to store and reuse the agent's past experiences for study. This buffer retains the agent's experiences at each timestep, consisting of the state, action, reward, and subsequent state. The agent later revisits this data for experiential learning. This is especially beneficial in Tetris, where certain states and actions, though infrequent, such as T-spins and clearing 4 lines with a single piece, significantly influence the game's outcome.

The experience replay is encapsulated within a `replay_buffer` variable, a list holding tuples of the agent's experiences. Each tuple includes the game's current state, the subsequence post-action state, the earned score, and a boolean denoting the game's conclusion. The replay buffer is filled by playing more random games until it attains a minimum size, ensuring sufficient expertience for kicking off model training.

Maintaining the buffer at a minimum size allows for the gradual replacement of initial, less instructive experiences with more insightful ones as training continues and the buffer reaches full capacity. As a result, the model benefits from a broader spectrum of experiences. Throughout the training, learning occurs through the random sampling of experiences from the `replay_buffer`.

3.3. Training set up

3.3.1. Hyperparameters

The training process was controlled by a set of hyperparameters. These parameters were chosen based on our testing and reviews:

- Memory Size (`MEM_SIZE`): This parameter controls the maximum size of the replay buffer. It was set to 10,000. The replay buffer is used to store the agent's experiences (state, action, reward, next state), which are then randomly sampled during training to break the correlation between consecutive observations.
- Minimum Memory Size (`MIN_MEM_SIZE`): This is the minimum number of experiences in the replay buffer before the model starts learning. This is to ensure that the model has a diverse set of experiences to learn from when it starts.
- Discount Factor (`DISCOUNT_START`, `DISCOUNT_END`, `DISCOUNT_DURATION`): The discount factor, which determines the importance of future rewards, was initially set to 0.8 and gradually increased to 0.94 over 4,000 episodes. These parameters are related to the discount factor in the Q-learning algorithm. The discount factor determines how much future rewards are worth compared to immediate rewards. The values are set to gradually increase from 0.8 to 0.94 over 4000 steps to encourage the model to value future rewards more as it learns.
- Epsilon (`EPSILON_START`, `EPSILON_END`, `EPSILON_DURATION`): The epsilon parameter, which controls the trade-off between exploration and exploitation, was initially set to 0.5 and gradually decreased to 0.08 over 3,000 episodes.
- Learning Rate (`LEARNING_RATE_START`, `LEARNING_RATE_GAMMA`, LEARNING_RATE_STEP): The learning rate, which controls the rate at which the agent learns, was initially set to 3e-3. It was then reduced by a factor of 0.9 every 300 steps.
- Batch Size (`BATCH_SIZE`): The batch size, which is the number of experiences sampled from the replay buffer at each training step, was set to 164.
- Update Target Every (`UPDATE_TARGET_EVERY`): The target network was updated every 100 steps.

- Simulate Every (`SIMULATE_EVERY`): The agent played a game every 5 steps to generate new experiences for the replay buffer.
- Use Pro Play Chance (`USE_PRO_PLAY_CHANCE`): The agent had a 20% chance of using professional replays for training.
- Episodes (`EPISODES`): The total number of episodes for training was set to 6,000.

3.3.2. Model setup

We use a modified version of deep Q-learning to train our model. Instead of using state-action pairs, we only use the state as the network's input. This modification is based on a property of Tetris that the next state is deterministic given the current state and action.

We use a Convolutional Neural Network (CNN) for the network structure with three convolutional layers and three fully connected layers. The convolutional layers extract features from the input image (the game state), while the fully connected layers map these features to the Q-value. We adopt two common DQN optimizations. One is experience replay, where we store the agent's experiences (state, action, reward, next state) in a replay buffer and randomly sample a batch from it to train the network. We start training only when the buffer contains more than 1000 samples, and we delete the oldest sample when it exceeds the 10000 caps.

The other optimization is epsilon-greedy action selection, which is a strategy to balance exploration and exploitation in Q-learning. We start with a probability of 0.5 to perform a random action, and 1 - 0.5 to choose the action with the best Q-value. After each step, we linearly decay the epsilon value over 3000 steps until it reaches 0.08.

In addition to the regular replay buffer, we also use a separate replay buffer for "pro" replays. These are games played by a very good player or possibly another well-trained agent. There's a chance (0.2) in each training step that a batch from the pro replay buffer is used for training instead of a batch from the regular replay buffer. This approach is a form of expert imitation, where the agent not only learns from its own experiences but also from the experiences of an expert.

The agent also uses a discount factor to discount future rewards. The discount factor starts at 0.8 and linearly increases to 0.94 over 4000 steps. The Q-network's parameters are saved to a file every 100 episodes. This allows us to resume training from a saved state if necessary. The training process is monitored using

the Weights & Biases tool, which logs various metrics such as the game score and the training loss.

TRAINING PROCESS

1. Initialize the Tetris environment `env`, current state `s`, and the Deep Q-Network `model`.
2. Initialize the replay buffer
3. While `episode` is less than `EPISODES` do
   - If episode is a multiple of SIMULATE_EVERY, reset the env and play a new game:
     - While the current game does not end do:
       - Enumerate all possible next states `S` and their corresponding scores and end flags done
       - Choose the best next state or a random next state based on the epsion-greedy policy
       - Store the current state, choose the next state, the corresponding score, and end flag in replay_buffer.
       - If the chosen next state is an end state, record the game score and break the loop. Otherwise, step forward to the chosen next state.
     - If the size of replay_buffer exceeds MEM_SIZE, discard the oldest entries.
   - If use_pro_replays is true, sample a batch from the professional replay dataset. Otherwise, sample a batch from replay_buffer
   - Extract the current states and next states from the batch
   - Compute the Q-values of the current states and the expected Q-values of the next states.
   - Compute the target Q-values: if the next state is an end state, the target Q-value is the corresponding score; otherwise, it is the corresponding score + the discounted expected Q-value of the next state
   - Compute the target Q-values: if the next state is an end state, the target Q-value is the corresponding score; otherwise, it is the corresponding score plus the discounted expected Q-value of the next state.
   - Fit the Q-network by minimizing the mean squared error between the computed Q-values and the target Q-values.
   - Update the epsilon and discount factor.
   - Update the target network if necessary.
   - If episode is a multiple of 100, save the current state of the Q-network.
4. After all episodes, save the final state of the Q-network
5. Return the trained Q-network

4. Experiments and Results

We have done a lot of experiments to find out about the optimal parameter values. The discount factor gamma starting value is set to 0.8 and decays to 0.94 over 4000 episodes. The reason is that we wanted the agent to value the importance of future rewards rather than just taking the immediate rewards.

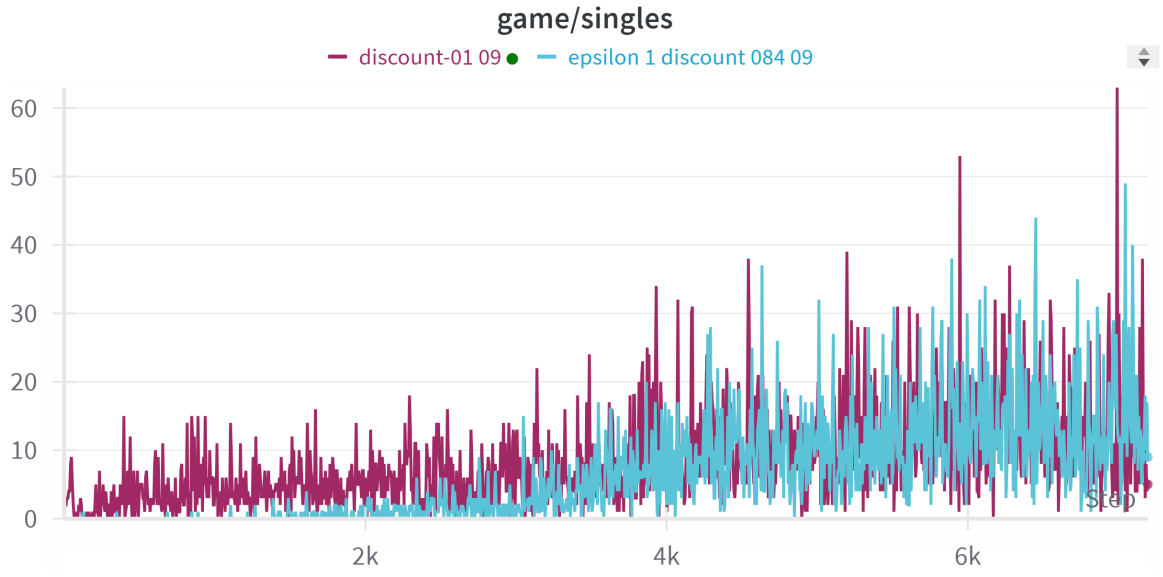


Figure 5: Comparison single clear between 2 gamma values

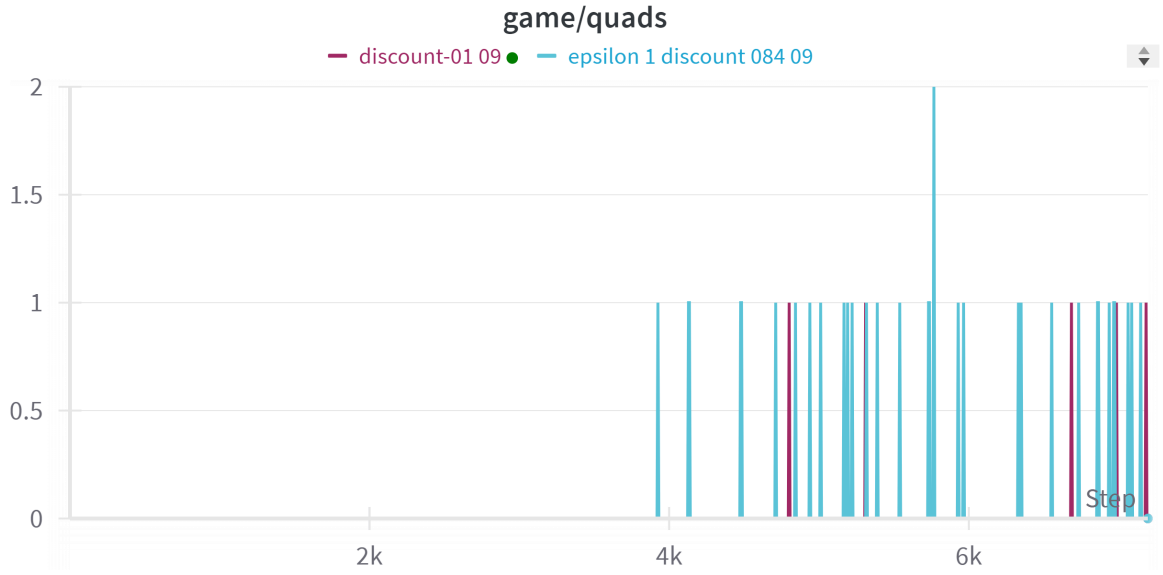


Figure 6: Comparison quad clear between 2 gamma values

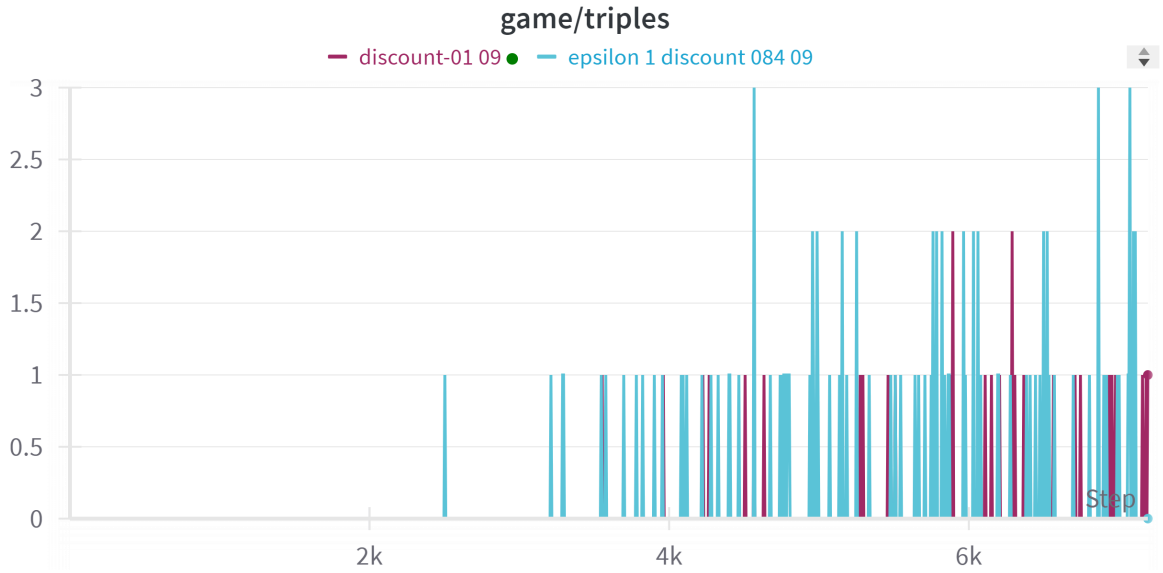


Figure 7: Comparison triple clear between 2 gamma values

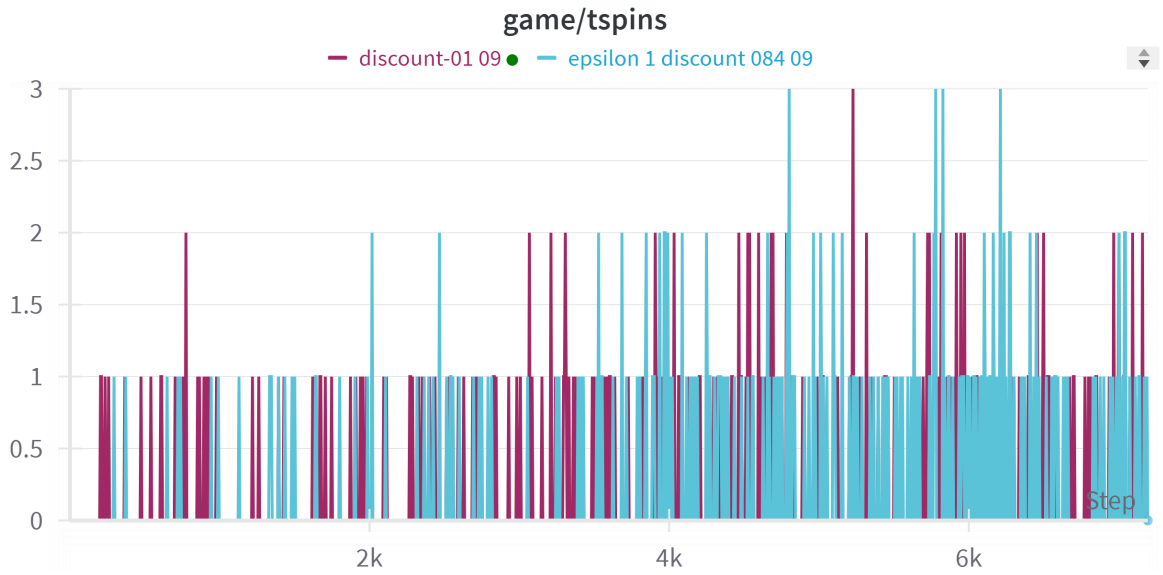
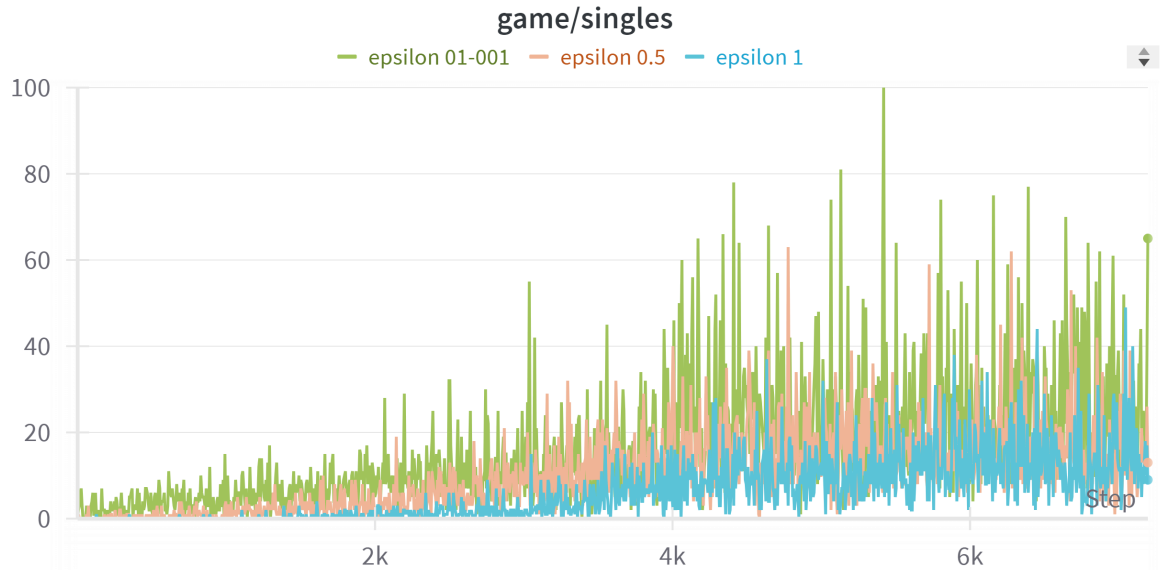
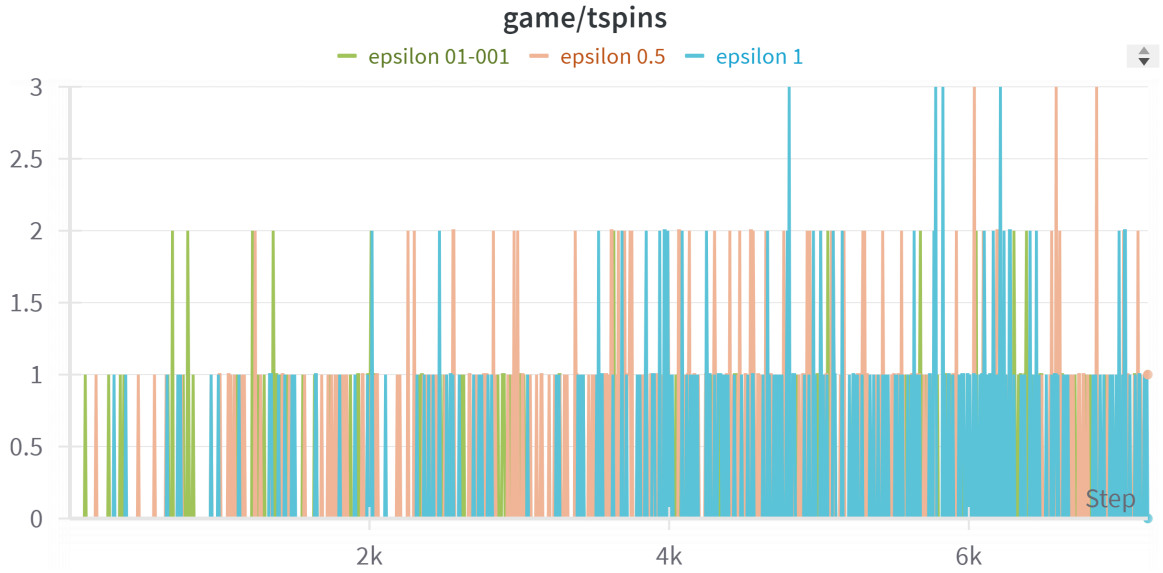


Figure 8: Comparison Tspin clear between 2 gamma values

As we can see in the experiments, a higher gamma value is likely to perform triple, quad, and Tspin clear Tetris because the agent is incentivized to set up the board for more complex maneuvers that yield higher future rewards. On the other side, the low gamma value makes the agent perform single clears more frequently because single clears are simpler, more immediate rewards that the agent is optimized to pursue.





Our focus remains on the exploration-exploitation trade-off after identifying a suitable gamma discount value. The epsilon parameter manages this trade-off. A significant difference in performance has been noted when comparing numerous epsilon decay strategies; however, we will discuss only two due to their representativeness. Initially, we set epsilon to start at 0.1 and decay to 0.08, as opposed to starting at 0.5 and decaying to 0.08 throughout 3000 episodes.

A starting epsilon of 0.1 indicates a lower level of exploration, implying that the agent is more inclined to adhere to its existing strategies for clearing lines. This is not ideal since we are training the agent from the beginning, and it may hinder

the agent's ability to learn complex strategies necessary for consistently setting up and clearing double or triple lines.

The low exploration rate does not sufficiently encourage the agent to experiment with the diverse placements required for these scenarios. Consequently, we have increased the starting epsilon to 0.5 and 1. This higher exploration rate prompted the agent to explore a broader range of block placements and strategies, aiding in the execution of higher-scoring moves.

We train our model for 6000 episodes and then let it play for 10 games. In the end, the model best run have scored 147000 , cleared 641 single line, 136 double lines , 29 triple lines , 26 quad lines and execute 9 T-spins and 0 all_clears. This result is significantly better than the first version of the agent when it only able to execute single and double lines clear

5. Pro Replay

To further improve the Agent's performance, we have intergrate a specialized dataset, referred as the ProReplayDataset. This dataset comprises of the gameplay replays from the top 20% player of tetr.io, stored in JSON format, which encapsulate a rich set of game states and corresponding player actions. The primary objective of leveraging this dataset is to expose the Agent to high-quality decision-making patterns, thereby enhancing its ability to generalize across a wide range of game scenarios. Each files represent a replay containing multiple game states and actions. These files are then parsed, and their contents are transformed into tensors suitable for neural network processing. During initialization, the dataset loads the entire collection of replays into memory. This approach significantly reduces I/O overhead during training, at the cost of increased memory usage. Each replay is decomposed into tuples of the current game state, the next game state, the score increment, and a boolean flag indicating whether the game has ended. These tuples are then stored in a buffer, ready to be fetched during the training process.