Deep Q Learning to Play Tetris

Henry Bender, jhb332

Emily Reynolds, emr249

CS4701

github.com/henrybender/4701_tetris

Tetris Gameplay and Implementation

I. Introduction

The following project is an effort to build a Deep Learning Tetris AI to play on a constructed GUI. The entirety of this project is written in Python, and includes the use of the following libraries : PyGame, PyTorch & NumPy. Our implementation of Tetris is based on the guidelines and rules described on the Tetris Wiki. This project incorporates most of the defined indispensable rules while omitting some that were deemed unnecessary. Features omitted include lock down time, piece preview, hold queue, ghost piece, and T-spin detection, in which their complexity outweighed its role in the AI implementation.

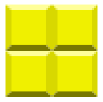
Initial effort was aimed at implementing a simplified Tetris game for one player. Once starting the game, the player aims to fit the randomly generated tetromino piece into the existing stack using the keys on their keyboard. Once the stack reaches above the 20 allotted squares in any column, the player finishes with the game over. As an evaluation metric, the player is able to view the number of pieces successfully placed on the stack and the score of their game. This approach allowed successful implementation of the core features of the game and thorough debugging before incorporating the complex AI component. We initially based our approach off of Timur Bakibayev's implementation of the game, cited below, but made adjustments to better suit the learning component.

Following a finalized version of the one player game, we focused on incorporating the AI aspect to replace the role of the single player. This aspect of the game aimed to maximize the score while determining the optimal drop location for each tetromino piece. In other words, the possible rotations and locations open were compared using their respective impact on the optimization metric. Specifically, the optimal placement of a piece was that which improved the score, preserved low height variability of the stack across columns, minimized the holes between pieces in the stack, and if possible, cleared a row by filling the remaining empty squares. Logically, our aim was to maximize the score in placing any piece, while also penalizing holes and variability in height in order to better situate the stack for placement of later pieces.

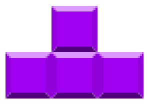
II. Figure Class

The figure class models the pieces in the Tetris game, defining possible shapes of pieces and functionality for rotating a piece. Each shape is associated with a respective 2-dimensional list as follows :

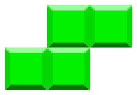
O : [[1 , 1] ,
[1 , 1]]



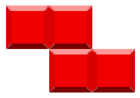
T : [[0 , 2 , 0] ,
[2 , 2 , 2]]



S : [[0 , 3 , 3] ,
[3 , 3 , 0]]



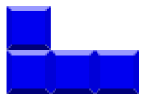
Z : [[4 , 4 , 0] ,
[0 , 4 , 4]]



I : [[5 , 5 , 5 , 5]]

L : [[0 , 0 , 6] ,
[6 , 6 , 6]]

J : [[7 , 0 , 0] ,
[7 , 7 , 7]]

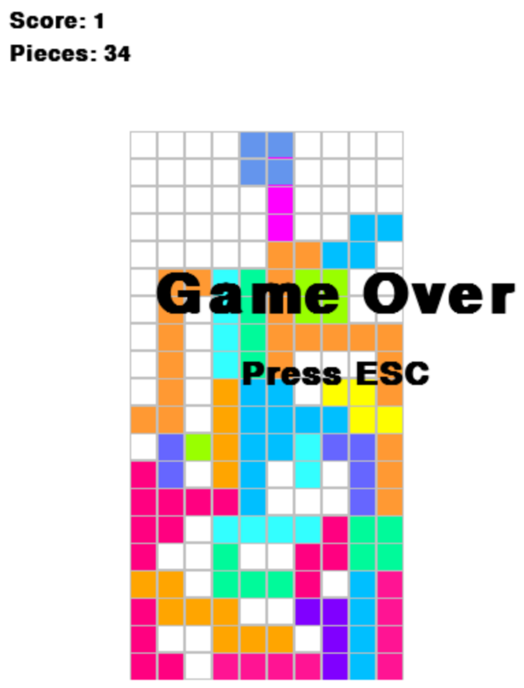


In our implementation, pieces are ordered at random, using a random number generator to index from the list of figures as well as from a list of RGB colors. In addition, rotation is implemented by updating the 2-dimensional list of a piece with the respective list transformed by 90 degrees.

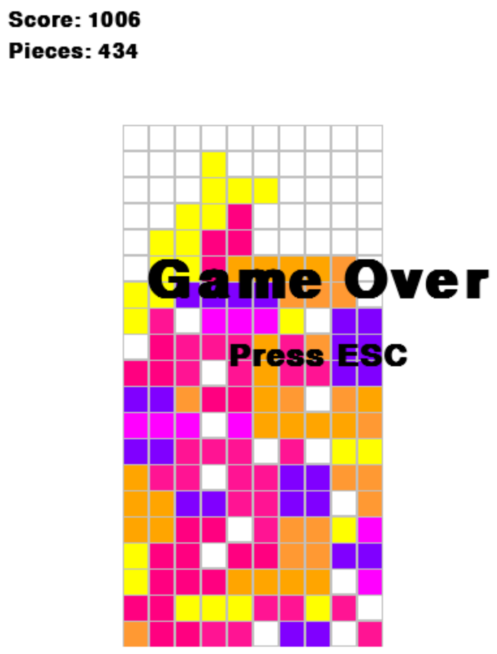
III. Tetris Class

The Tetris class models the components and conditions of the game. It initializes and maintains the GUI in addition to the state of the game. Once starting a game, it manages the random generation of each additional piece appearing in the first row, and its visual descension towards the stack, alongside visual rotations of the piece or horizontal movement, through the step and draw functions. In other words, it handles the keyboard input from a player as well as the AI imitation. Similarly, intersections are handled here, allowing pieces to stack on top one another instead of overlaying or stacked floating on top of empty squares. Score is updated in correspondence, incrementing with each successful placement of a piece, as well as with any cleared rows. Another critical component of this class is checking the gameover condition, immediately ending the game in the event that a piece stacks above the allotted height.

Furthermore, this class includes the computations of the optimization metrics for the AI component. As noted above, the score is updated in this class, as well as calculation of height variability and holes in the stack. Additionally, this class includes the functionality for checking if any rows are entirely filled, and if so, clearing the row and increasing the score. These metrics are vital for the AI implementation and must be accurately computed for each potential placement of a piece for optimal performance.



1 Player

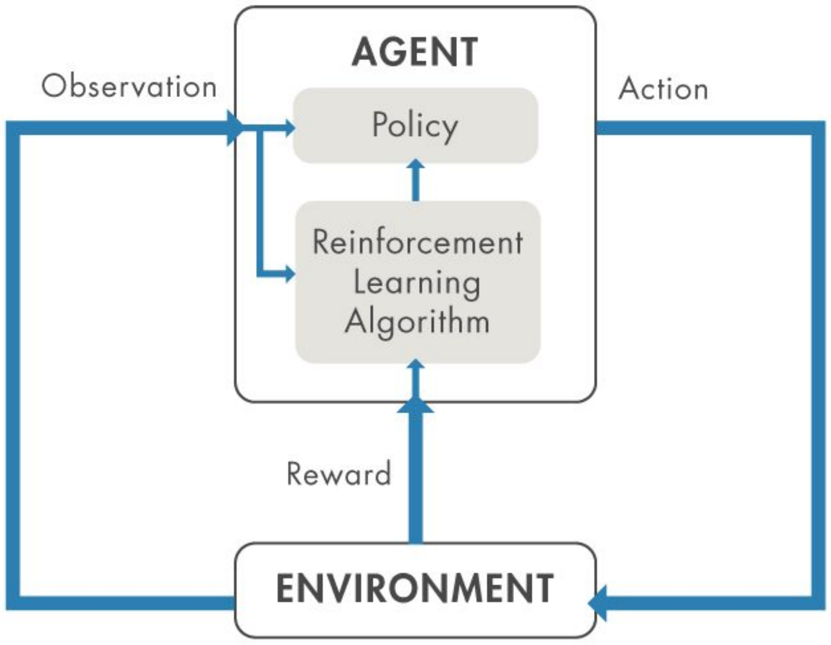


AI Player

Deep Q Learning

I. Overview of the Deep Q Learning Network

Deep Q Learning is a type of reinforcement learning with an added neural network. The main difference between a reinforcement learning task like Q learning from a supervised or unsupervised learning model is the presence of an agent and an environment, as opposed to a large dataset and labels. The agent is tasked with acting rationally in the environment, and transitioning to better states as it learns. In our case, we attempt to train the agent to play tetris (environment). We can see this feedback loop in the diagram below.



The agent learns by playing the game, observing state transitions and rewards, and will eventually learn to take actions that maximize rewards.

In traditional Q Learning, the objective is to create a Q-table to map states and actions to a Q-value, which is an estimated future value of the state. The agent's main goal is to learn to predict accurate Q-values from an action A in a state S. In our deep Q network, however, we utilize a neural network to approximate a Q function to predict these values from states and actions. The network then uses the Bellman equation to update network weights:

$$Q(S_t, A_t) = (1 - \alpha) * Q(S_t, A_t) + \alpha * (R_t + \lambda * max_a(Q(S_{t+1}, a))$$

In this equation, we represent the state as S, the agent's action as A, the reward for that action as R, the time step t, learning rate α, and discount factor λ. The network also uses a replay memory, a varied storage of state, action, reward tuples that gives our network a chance to learn batches of random game transitions as opposed to a narrow set from certain games. After training epochs, the network is able to select actions that it predicts will yield the highest future Q-value.

II. Components and Implementation

A. Agent - The "agent" in this case is the Deep Q Network making predictions in a tetris game loop. At each iteration until the loss condition, we feed the network an array of pytorch tensors containing the state information of all of the next possible moves.
B. States, Actions, Rewards - Our states for the model are a dictionary mapping x-coordinate and rotation of piece to a 4-tuple of lines cleared, holes in the board, height variability (average difference in columns), and total height of all columns on the board.
C. Neural Network - The network is a 3 layer sequential neural network built using the pytorch nn library. We initialize weights using the xavier uniform, and use the rectified linear unit function as an activation function.
D. Replay Memory - We implemented the Replay Memory as its own class, but under the hood it is a python Collections deque with basic push, length, and

random sample functionality so that our network can acquire random batches of transitions to learn.

E. Loss and Optimizer - We use the Adam Optimizer for stochastic optimization, and mean squared error as our loss criterion. We had used these optimizers and loss functions in previous courses and found that they worked well, but did not experiment much with other alternatives.

III. Training

We trained our model for 2500 epochs, with each epoch being a full game of tetris played by the agent. We store each transition we see in the games in our replay memory, and at every 3000th move we optimize the model. We do so by taking a random sample of 512 transitions (Batch Size) stored in the replay memory, and predict their Q-values, comparing them with the Q-values (reward) actually generated by those transitions. We backpropagate this loss throughout the network, and proceed.

Initially, to keep the model from favoring certain moves or sides of the board, we add an element of randomness to the move choice. We have a select_action function, which chooses a move for our model during training. At each move, we either select a random move or let the model choose, and the proportion of moves that are selected at random decays by a factor of 1/2000 per move selected. The intention behind these random actions is to provide the model with more opportunities to explore states, especially early on when it has not learned ideal transitions yet.

IV. Hyperparameters

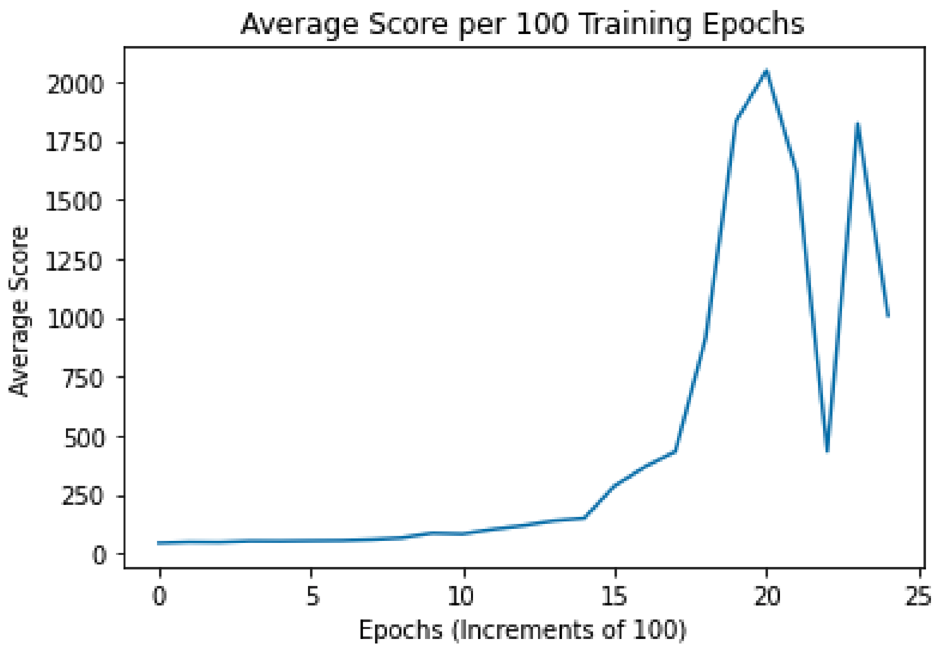
A. Batch Size - The amount of state transitions our model will sample when optimizing. We chose a value of 512. These are sampled randomly so that the model sees uncorrelated transitions, instead of all of them from the same game.

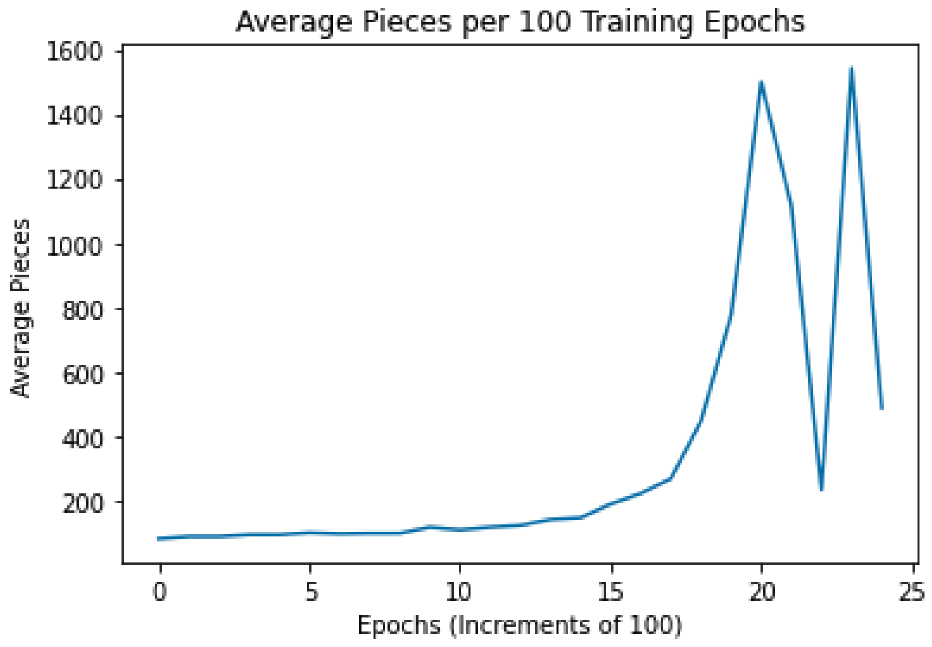
B. Gamma - The discount factor placed on rewards for transitions. This factor, between 0 and 1, applies a penalty on rewards seen in the future, Instead prioritizing immediate rewards. We chose a value of 0.99.
C. Epsilon Decay - This is the rate at which our proportion of randomly selected moves will decay. With a higher rate, our model will begin selecting moves more often and will explore the total state space left. We found that a rate of 2000 allows for deep enough exploration, and does not leave the model stuck in choosing a small range of moves.
D. Memory Size - The memory size determines how many transitions our replay memory will store. We chose to use 30,000, as each game of tetris that our model plays can have anywhere between 50-2000 moves, so on average we are storing about 15-600 games worth of moves to sample from.
E. Learning Rate - The learning rate is the pace at which weights in our network are updated by our optimizer. We chose a standard learning rate of 0.001, often used as a baseline in deep learning tutorials. With much larger learning rates (0.01), training took nearly 3x as long and sometimes would hang indefinitely.

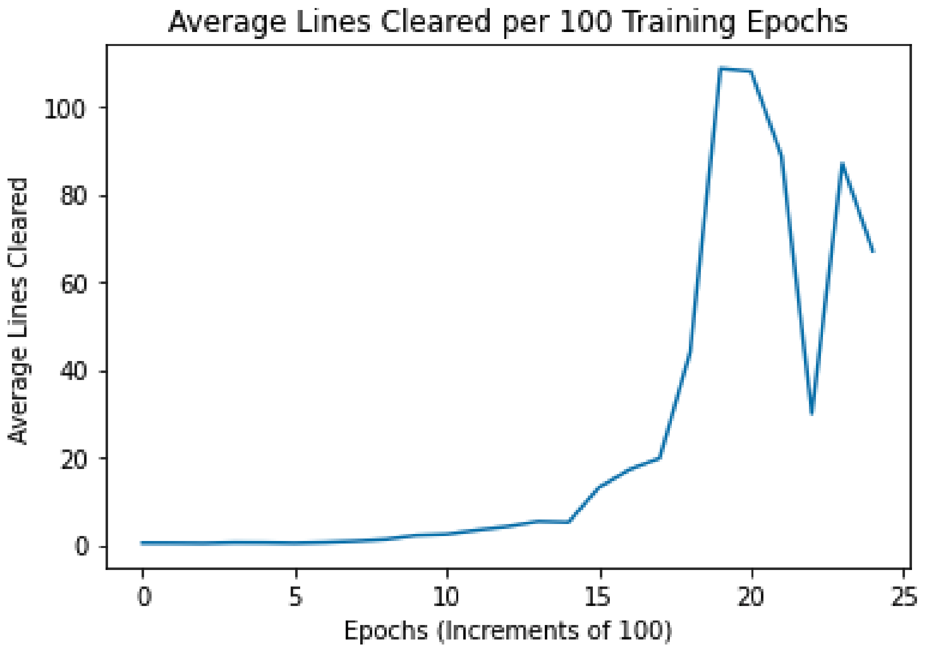
Results and Discussion

After training, our model was able to consistently play tetris games well beyond the level of average to good players. The world record for lines cleared in tetris is 207, and on average over 100 games our fully trained model could clear over 100 lines. Below are some results of the training represented graphically. We plotted the average lines cleared, pieces, and score across 100 training games over the course of 2500 games.

We see that the learning starts off very slow (learning rate = 0.001 in these plots), but increases almost exponentially around the 1500-2000th game played. This is to be expected, as the agent will often select random moves in the beginning of training to explore states, and these moves often result in less than ideal board positioning. As training progresses, we would rely more on the model's predicted best moves, which extends the duration of the game as we see in the average number of pieces played. This also gives our model many more transitions to sample from in replay memory, which gives it more transition data to work with.




Average Pieces per 100 Training Epochs
Average Pieces
Epochs (Increments of 100)
1600
1400
1200
1000
800
600
400
200
0
5
10
15
20
25


Average Lines Cleared per 100 Training Epochs
Average Lines Cleared
Epochs (Increments of 100)
100
80
60
40
20
0
0
5
10
15
20
25

In experimenting a bit with hyperparameter tuning of the model, we did not notice much change when altering the size of the layers of the network, or batch size. One of the largest differences we saw in performance and training speed was changing the learning rate of the optimizer. As expected, when we increased the learning rate to 0.005 from 0.001, the model quickly began clearing many more lines and achieving a much higher score in about half as many epochs. However, training time was significantly increased (and in some cases would freeze in terminal) due to the model playing many more games for a longer duration. The comparison of the two learning rates can be seen below in the table. We see that the 0.005 learning rate model begins the exponential growth in learning much earlier than the 0.001 learning rate model. This is due to the larger updates of the model's weights in each optimization run.

Average Pieces Cleared per 100 epochs

Epoch Interval (100s)	Learning Rate 0.001	Learning Rate 0.005
1	81.54	83.82
2	88.3	95.14
3	88.34	98.56
4	93.82	102.12

5	93.9	106.22
6	100.0	122.66
7	96.99	125.06
8	98.3	195.56
9	98.28	504.14
10	117.2	717.16

References

Paszke, A. “REINFORCEMENT LEARNING (DQN) TUTORIAL”. PyTorch Tutorials https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

M. Stevens, S. Pradhan. “Playing Tetris with Deep Reinforcement Learning”. Stanford University, 2016, http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf

T. Bakibayev. “How to write Tetris in Python”, May 2020, https://levelup.gitconnected.com/writing-tetris-in-python-2a16bddb5318

Tzorakoleftherakis, Emmanouil. “Reinforcement Learning: A Brief Guide.” *MATLAB & Simulink*, MathWorks, www.mathworks.com/company/newsletters/articles/reinforcement-learning-a-brief-guide.html.