Playing Tetris with Deep Q Learning

ckeaunui

December 2022

1 Problem Context

With over 500 million downloads since its release in 1984, Tetris has become the most purchased video game of all time. Some professional players have dedicated decades towards mastering their craft and have even physically broken the game by achieving scores higher than the theoretical max (999,999). This left me wondering, is it possible for a computer to accomplish the same feat? Tetris has some key attributes which make it a perfect environment for utilizing reinforcement learning. First, Tetris is an NP-complete problem meaning no efficient solution algorithm has been found. This suggests the difficulty in training a functioning agent is suitable for a comprehensive coding project. Creating a Tetris environment is also computationally cheap and the rules are extremely straightforward. It is for these reasons that I decided to create a RL-Tetris agent

2 Technical Background

To start, I created a version of NES Tetris in Python utilizing Numpy arrays. This environment has some slight alterations to the true game. First off, one point is awarded for each piece that is placed. In the real game the number of points awarded is $20 - placement_height$ per piece. This change is meant to make training more consistent across episodes. Furthermore during training the actions which the computer takes are not the same as in game moves. To help focus the computers training on organizing pieces rather than learning the way they move, I set the actions to be all possible final locations of the piece represented as (y, x) coordinates.

To understand how this project functions we need to understand how basic q-learning works. Q-learning is an approach where an agent explores an environment by taking various actions, and records the results in a table as (state, value) pairs. When the table is completed the agent will know which rewards any state award, and can then optimize its play with a given policy. In theory this method would work very well for Tetris, but there are technical issues which prevent this. Primarily, filling a database of all Tetris positions would result in roughly $2^200/2$ different entries, which is far too many.

To get around this issue, a deep q-learning algorithm will be used instead. The main difference in methods is that deep q learning works by creating a function which takes in states and outputs a predicted q value for that state. This bypasses the need for a large database as the agent will be able to accurately guess what a states q value should be. To create such a function, a neural network will be used which takes in state values and outputs the expected q-value. Initially the network will not be able to predict moves very accurately, but will learn to by minimizing its own loss received from the mean squared error loss function. As the agent plays more this loss value should converge on 0, and once the agent learns a decent policy its score should increase exponentially. This is because in theory making better moves produces better data and will in turn make the agent able to predict better moves.

3 Prior Work

3.1 Neuro-dynamic Programming by Dimitri P. Bertsekas and John Tsitsiklis

This is a book which I read to learn about how neural networks function. Prior to reading this I had no experience with artificial intelligence, and this was the main source I referenced to gain an understanding of how to create a deep Q learning algorithm.

3.2 Deep Q-Network (DQN)-II Experience Replay and Target Networks

This source describes the functionality of the experience replay and how it is used to improve the training of deep q networks. Furthermore it goes into depth on how deep q-learning works and how it differs from standard q-learning, and this is a source I read several times to understand the premise of a DQN. By predicting a states Q value rather than exploring and remembering it, an agent can be trained to perform at an extremely high level even in positions it has not yet seen, and this is extremely powerful in games with large state domains such as Tetris.

3.3 Github: DQN Tetris by Sai Hemanth Beeraka

This is a project created by a former student at NIT Trichy which seeks to accomplish the same goal as me. In this repository, Sai outlines his approach to creating a DQN to play Tetris. The method he used was similar to mine and I based some of my hyper-parameters off of values proposed by him, but he utilized a convolutional neural network rather than an artificial neural network. In theory this would be better since the computer can look at the board itself rather than a representation of it, but a problem arises with pattern locations. Convolutional neural networks are not optimized for assessing a pattern based

on its absolute position, and as a result has a hard time distinguishing between stacks that are high and low. The results Sai got were not very good with the agent clearing an average of 10 lines per episode, and I believe this is due to the use of a convolutional neural network.

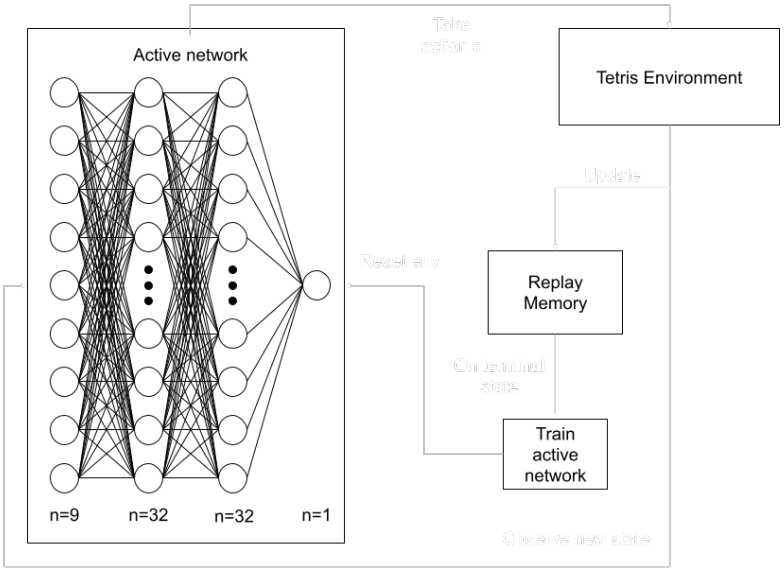
3.4 Playing Atari with Deep Reinforcement Learning

This was the single most impactful source on my project. Many of the algorithms I used in my project were taken directly from this team, and early iterations of my agent utilized the same structure this team used. I wanted to create an agent which plays Tetris from high level inputs just like this team did for Atari games, but I quickly realized this was not realistic for my project. This is because Atari games are mostly reaction based, while Tetris relies on planning and organizing. In Tetris a players reaction time is not typically considered but rather the time it takes them to decide where to place a piece, and for a computer this is a negligible amount. This made the suggested network ill suited for Tetris, however the methods used for the rest of the agent were directly applied to my agent.

3.5 Divergence in Deep Q-Learning: Tips and Tricks

This team expands upon the previous cited related work by discussing their future adaptations, and described in detail ways to improve the performance of a given DQN. I used this source to improve my agent after early training stages returned poor results. The algorithms I used in my agent were taken from this team, and from the previous Atari team.

4 Methods



The agent consists of a neural network which takes in 4 values to represent a given state as its input, and outputs a q value for that state. The four values are number of lines cleared, total bumpiness aka sum of height differences between adjacent columns, the summed height of all columns, and the number of holes present. These values were chosen because they proved to be the most impactful on training the model. initially 9 values were used which included each columns height, active piece, and other information, but much of it proved to be redundant and could be simplified down to these four. Another method would be to take a screenshot of the board or feed the board into a convolutional neural network as a 2D array, but this presented issues with pattern locations. Convolutional neural networks are able to detect patterns very well but don't usually differentiate the patterns location, which means a flat surface at the bottom and top of the board would be treated as the same. This is not true for Tetris since the higher the stack gets the more dangerous it is and would require a more greedy policy to survive. The network consists of two hidden layers of size 32 with relu activation functions fully connected with an output layer of size 1 with a linear activation function. The network will begin by predicting a value for a given state, and then learns by minimizing the difference between its own prediction and the actual value observed. The adam optimizer was used for this network with a mean squared error loss function.

When agents learn through self exploration, they tend to get stuck following the first decent path they discover and often times causes better states to never

be explored. To help resolve this issue, a randomness factor called an exploration rate was included. This is a value to determines the frequency which random moves are played instead of what the agent predicts to be the best move. Over time this value decays, ending at 0 after 1500 training episodes. This allows the earlier stages to be more thoroughly explored while also allowing the agent to abuse the policy it learned in later episodes.

The policy used to play the game was a = a where $Q = MAX(Q(s(t), a(t)) + Q(s(t+1), a(t+1)$, where α is the discount factor. This is meant to slightly prioritize rewards that can be obtained sooner while still allowing the agent to reject a reward in exchange for a greater future payout.

A learning rate of 1e-3 was used to update the weights of the network. This is the rate at which the networks policy changes for each epoch.

Finally, an experience replay was added to assist in the training. This is a replay memory of the last 20,000 moves the computer has made, and is saved in the format ¡state, action, next state, reward, done¿ By randomly sampling from the memory, the agent can learn from its past experiences, leading to better decision making.

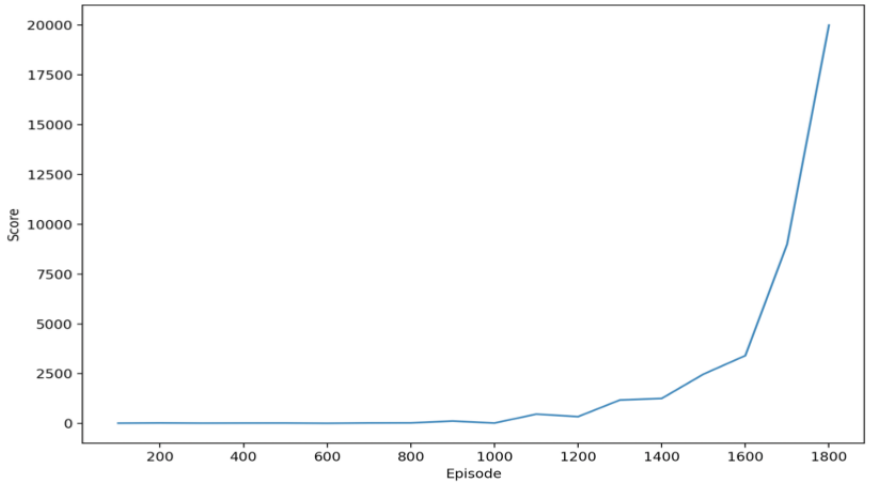
The main advantage of experience replay is that it helps the agent break the temporal correlations between consecutive experiences. This means that the agent is able to generalize its learning across different situations which is very helpful when dealing with the randomness in Tetris. Otherwise, it would learn policies which rely on identical sequences of active pieces which are not consistent.

5 Evaluation Metrics

During the training phase, I took the maximum score achieved every 50 episodes to see how the results improved over time. the results are shown below:

After the model was trained, I ran it for 20 games on a new environment with an exploration rate of 0 so it would rely only on its own policy. I then based the results off of the high score achieved during this test phase. Before starting the project my goal was to have the agent clear 100 lines, which would equate to a score of roughly 4,000 during the training phase. Later I switched to evaluating performance just based on the score, since this incentivizes clearing several lines at a time.

6 Results and Discussion



The results obtained from training were very positive. After training for around 1000 episodes the agent started learning a policy which was capable of leading to lines being cleared. Around episode 1400 it refined this policy to make line clearing much more consistent, and at episode 1727 it optimized its policy to reach the terminal score of 25000. At this point, further training was not improving the model and each iteration was taking increasingly long since the agent would not lose the game. Its own reaction time was negligible until the end game, so further training would not improve this result without modifying the states structure to include gravity. After 20 test runs, the trained agent was able to achieve a high score of 218,740. Compared to the average human score of 100,000 the agent was able to perform very well. However there was certainly room for improvement. When watching the agent play I realized it usually cleared 1-2 lines at a time. This is because the policy it learned rarely decided against an immediate reward in hopes of a greater future reward.

To further improve this agent I would try implementing next piece information, which would allow the computer to see several sequential pieces rather than just the active piece. This would increase the number of states for each board by a factor of 23, but greatly increase the computer's ability to setup Tetris'.

Another way I would improve upon the model is to use A* pathfinding rather than raw (x, y) coordinates. This would allow the computer to make more advanced moves than are currently possible. At the moment the pathfinding algorithm is very elementary and functions by moving the piece in position horizontally then dropping it. With A* the piece would be able to move in all legal directions at each timestep which would allow for moves such as block shifting and T-spins. This would broaden the computers abilities and should lead to improved performance. Another alteration would be to allow the computer to store pieces. This would double the number of states to assess at each step, but would unlock new possibilities for the computer to perform better.

7 Ethical Considerations

After careful considerations, there were not many ethical concerns with my project since it does not involve humans in any meaningful way. The data it gathers is unbiased since it is within its own training environment and has no ways of giving any person an advantage over another. The main ethical consideration I could pin on my project was the use of a computer cluster for training which would require a relatively large amount of power, but even so the model only trained for around 7 hours before completing its task. This is much less than many other projects which can train for days to weeks before reaching an optimal policy.

8 Bibliography

- Aflak, Omar. "Neural Network from Scratch in Python." Medium, 24 May 2021, towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65. Accessed 15 Dec. 2022.
- Algorta, Simón, and Özgür Şimşek. "The Game of Tetris in Machine Learning." ArXiv:1905.01652 [Cs], 10 May 2019, arxiv.org/abs/1905.01652. Accessed 15 Dec. 2022.
- Bertsekas, Dimitri P. "Textbook: Neuro-Dynamic Programming." Athenasc.com, athenasc.com/ndpbook.html. Accessed 15 Dec. 2022.
- "Divergence in Deep Q-Learning: Tips and Tricks." Aman, 19 Aug. 2020, amanhussain.com/post/divergence-deep-q-learning/. Accessed 15 Dec. 2022.
- Doshi, Ketan. "Reinforcement Learning Explained Visually (Part 6): Policy Gradients, Step-By-Step." Medium, 24 Apr. 2021, towardsdatascience.com/reinforcement-learning-explained-visually-part-6-policy-gradients-step-by-step-f9f448e73754.
- "Q Learning Intro/Table − Reinforcement Learning P.1." Www.youtube.com, www.youtube.com/watch?v=yMkXtIEzH8list=PLQVvvaa0QuDezJFIOU5wDdfy4e9vdnx−7. Accessed 15 Dec. 2022.