Creating a Deep Q-Learning Agent to Play Tetris

Christian Keaunui
ckeaunui@oxy.edu
Occidental College
May 2023

Abstract

In this paper, I propose using a Deep Q-Network (DQN) model for mastering the game of Tetris, a classic and challenging NP-complete problem. The primary objective of this model is to learn a policy which maximizes its' score before reaching the terminal condition of a column 21 blocks tall. The proposed model utilizes a neural network to predict a states Q-value after taking a given action. The model learns by exploring the Tetris environment, recording its observations into an experience replay, and training on minibatches of data after each game. The model learns by using the adam optimizer to minimize the mean squared error between the predicted and expected Q-values. After 2000 training episodes, the model learned a policy which on average is able to reach a score of 6.31 million, with a high score of 25 million. This greatly exceeds the human world record of 2.94 million, however the models success is partially based on the speed it functions at and not just its decision making abilities. Even at the fastest levels the model is able to move a tetrino (a single Tetris piece) into the correct horizontal position before the tetrino would have dropped by a single tile. The policy the model learned does not always result in the best piece placement, but the model is able to consistently recover from difficult positions, allowing it to reach very high scores.

1 Problem Context

The goal this project aims to reach is to train an AI model which is capable of outperforming a human on NES Tetris. Tetris was chosen as the environment for several reasons: It is the most downloaded game ever with over 500 million downloads across all platforms, so the reader of this paper will likely be familiar with how the game works and what the goals are. Tetris is also an NP-complete problem, which means an optimal solution has not been discovered. This gives the problem a high level of complexity which the computer will have to overcome. Finally, Tetris is a computationally simple game, making it easy to reproduce in Python.

One complexity in creating a Tetris AI is the breadth of unique states the model needs to explore. Each of the 200 tiles are either filled or empty, which is a theoretical 2^200 or $1.61e^{60}$ different states. This calculation does include states which are invalid, but accounting for this will not change the issue of the environments size. Thus, the model will need to learn a policy which can perform in states it has not previously observed.

Another problem for a Tetris AI is the delayed rewards the model receives. The most optimal way to play Tetris is to stack pieces up high and clear multiple rows at once, since each additional line cleared awards exponentially more points. However this strategy requires turning down an immediate reward for long periods of time in hopes of a much higher future reward, which is a policy the model may not discover.

2 Technical Background

To understand the methods used in this project, the reader should be familiar with the following terms:

Neural Network: A neural network is a computational system inspired by the structure and function of the human brain. It is composed of interconnected nodes called neurons which work together to approximate a function designed to complete a complex task. Each node has three attributes called a weight w, a bias $\boldsymbol{b}$, and an activation function $\boldsymbol{a(x)}$. The neural network takes a vector of values as an input, and propagates those values through its hidden layers. Each node $\boldsymbol{n}$ in a hidden layer $\boldsymbol{L}$ calculates a value based on the function

$$z_n = \sum_{i=1}^{m} w_{(n,i)}^{L}$$

where m is the size of the previous layer. The node then feeds this value into an activation function

$$a_n(z_n)$$

Once all nodes have calculated their activation value, they are fed forward as inputs to the next layer. The last layer, called the output layer, produces the final values which are returned to the user. A neural network trains by using an optimizer to mitigate a cost function, which is used to quantify

the difference between the neural networks expected and actual output values.

ReLU: A rectified linear unit (ReLU) is an activation function which is defined by:

$$a(z) = \begin{cases} z, & \text{if } x > 0. \\ 0, & \text{otherwise.} \end{cases}$$

for a given input ***z***. ReLU is currently the most used activation function for several reasons; the first is how computationally simple it is, so less time is needed to calculate each nodes' activation. It also solves the vanishing gradient problem which arises when gradients become very small as they are back-propagated through the layers, resulting in little to no learning. ReLU helps mitigate this problem because its gradient is either 0 or 1, reducing the likelihood of gradients becoming too small.

Linear Activation: The linear activation function is the simplest activation function, defined by:

$$a(z) = z$$

This activation function is usually used on the final layer to predict a quantitative value which can be positive or negative.

Mean Squared Error: Mean squared error ***mse*** is a cost function which quantifies the average of the squared differences between the predicted values $\hat{\boldsymbol{y}}$ and expected values ***y*** of a layer, where

$$MSE = (1/n) * \sum_{i=0}^{n} (y_i - \hat{y_i})^2$$

The cost function is used along with an optimizer to train a neural network by finding a policy which minimizes the final cost value.

Gradient: A gradient is a vector which represents the direction of greatest change in a multi variable scalar function at a given point. The gradient is obtained by taking the partial derivative of the given function with respect to each variable. For a function f(x,y,z) the gradient is defined by

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}$$

Stochastic Gradient Descent: Stochastic gradient descent is an optimizer used to find the parameters of a network which minimize the value of a cost function based on individual data points. Stochastic gradient descent works by creating a gradient of the cost function for each sample in a minibatch, averaging the results to find the mean gradient, and taking a step in the opposite direction. The step size is proportional to the value returned by the cost function, allowing for smaller refined steps to occur as the optimal policy is approached. This differs from standard gradient descent in that it does not calculate its gradient based on the entire batch. The gradient of the cost function is based on each of the networks individual weights and biases, and is defined by:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w} \\ \frac{\partial C}{\partial b} \end{bmatrix}$$

State: A state is an instance of an environment.

Q-value: A quality value or Q-value is a number which represents the expected reward an agent will receive from taking an action ***a*** in a given state. The Q-value is represented by the function:

$$Q(s, a) = r + \gamma max(Q(s', a'))$$

where γ is a discount rate to prioritize more immediate rewards, s' is the next state, and a' is the next action. The Q value is used to determine the best action to take at any given state.

Q-learning: Q learning is a learning algorithm used to discover an optimal policy for decision making in an environment. Q learning works by observing a state ***s***, taking an action ***a***, calculating the Q-value ***Q***, and storing the data in a Q-table as ((s,a),(Q)) pairs for all possible pairs. Then on future runs, the Q-table can be referenced to find the best action for any state.

Deep Q-Learning: Deep Q-learning is a variation of the q-learning model which trains a neural network to predict the Q-values of a state rather than relying on a q-table. This approach is used for problems where the number of (s,a) pairs are too large to explore, such as all possible chess positions, or in this case, all possible Tetris positions. The agent trains itself by predicting a Q-value for a given action in a state, performing that action to receive the real Q-value, and adjusting its parameters based on the difference. This will teach the agent to determine the Q-value of a state based on its attributes rather than relying on previous knowledge, allowing it to perform well on states not seen during the training phase.

Experience Replay: An experience replay is a buffer used to store ¡state, action, reward, next state¿ tuples as an agent explores its environment. It allows the agent to train itself in the future on minibatches rather than after every iteration.

Minibatch: A minibatch is a small randomized subset of the experience replay used for an agent to train on. Minibatches allow the agent to train faster and more accurately because computations rely on few data points and updates are made frequently. Minibatches also introduce some noise in the training data which can help the agent avoid getting trapped in local minima.

3 Prior Work

3.1 Playing Atari with Deep Reinforcement Learning

In 2013 Volodymyr Mnih, Koray Kavukcuoglu, and other researchers presented a novel approach to reinforcement learning by introducing the Deep Q-Network (DQN) algorithm. This model used a convolutional neural network to learn directly from high-dimensional sensory input, such as raw pixels from the game screen, without the need for hand-engineered features or game-specific knowledge. The training cycle the model followed was therefore generalized to function in a variety of environments. This gave it the ability to solve a diverse set of complex tasks without any restructuring. The model was trained on a variety of Atari 2600 games, and was able to obtain results comparable to better than humans. This project shows that reinforcement learning is capable of performing complex tasks, and could be a viable method for training to play Tetris. However, it should be noted that the Atari 2600 games have some key differences from Tetris. Most of the games tested were much more reliant on reaction times and piece movement, while Tetris is reliant on strategy and organizational abilities. This means the model suggested may need some restructuring to better fit the Tetris environment.

3.2 DQN Tetris by Sai Hemanth Beeraka

This is a project created by former NIT Trichy student Sai Beeraka which seeks to accomplish the same goal of creating a reinforcement learning agent to play Tetris. Sai proposed using the same deep q-learning model proposed by Mnih and his team to accomplish the task. The advantage behind this approach is that the computer can look at the board itself rather than a representation of it. This allows the model to determine which attributes and patterns it deems beneficial, but a problem arises with pattern locations. CNN's are designed to maintain spacial invariance, making it difficult to differentiate between the same pattern in two different locations. This is important for Tetris because the decision making process is largely based on the pieces absolute positions. For example, when the pieces are low it is best to stack them up and clear several lines at once, but when the stacks are tall it is best to clear lines as quickly as possible to prevent losing. This flaw can be seen in the Sai's results, where the model was only able to clear an average of 10 lines per episode. These results influenced my project to adapt the model to run on an artificial neural network with hand picked features instead.

4 Beating the world record in Tetris (GB) with genetics algorithm by Duc Anh Bui

This project explored using a genetic algorithm to play Tetris. This method trains similarly to Darwins' Theory of Evolution by evolving a policy over generations. A domain of model are initialized with random parameters, and left to perform in the environment. The ones which perform the best are allowed to breed, producing children with similar parameters and a few mutations. As the generations progress, the models become better fit to survive in its environment. Duc explored many different variables to best represent the environments state, and he found that the four most important attributes were the number of holes, the bumpiness of the board, the total height of all columns, and the number of rows that were just cleared. The project I am conducting seeks to determine if reinforcement learning is capable of achieving the same results. This paper was influential on my project for two reasons: It set a metric of success, and provided me with the best attributes for representing a boards state in an aratificial neural network.

5 Methods

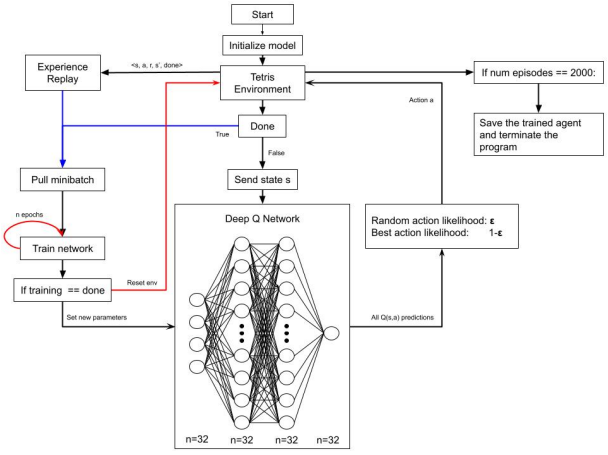

Figure 1: A flowchart of the models complete structure. Note: The different colored connections are to help with visibility, and do not represent any special functions or attributes

The environment the agent will train on is a version of NES Tetris made in Python. The agent will explore the environment and record its' observations at each timestep in an experience replay with the format ¡s, a, r, s', done¿. The first 1,000 actions the model performs are fully random so that the experience replay will initially be populated by a diverse data set. Afterwards, the model will predict the Q-

value or expected reward of each next state, and take the action which corresponds with the highest value.

When agents learn through self exploration they are susceptible to exclusively following the first decent path they discover, often resulting in better states never being explored. To help prevent this, a randomness factor called an exploration rate was included. This value determines the likelihood of the next action being performed randomly instead of what the agent predicts to be the best move. The exploration rate ϵ will start at 0.1 and steadily decay until reaching a value of $1e^{-3}$ after 1500 training episodes. This allows the earlier states to be thoroughly explored which prevents getting trapped in local minima, while also allowing the model to train through exploiting its policy once it has become refined.

The boards state will be defined by a vector with the attributes: number of rows cleared by the previous action, number of holes, bumpiness (sum of height differences between adjacent columns), and total height. It should be noted that the speed Tetrinos fall at increases with the level and therefore may seem important to include in the state, however the agent is able to place pieces faster than the highest gravity level, so its policy will not be dependent on the game speed.

The model will use a Deep Q-Network to train itself to predict the Q-values for each possible (s,a) pair in a given state. The neural network is comprised of an input layer, two fully connected hidden layers of size 32 with ReLU activation functions, and a fully connected output layer of size 1 with a linear activation function. The input layer accepts a vector of size 4 to represent the state, and propagates the values forward throughout the network. The input layer does not perform any computations on the input data. The output layer contains only one node, and uses the linear activation function to predict a states q-value. The agent will use the adam optimizer to train by minimizing a mean squared error cost function on the predicted and actual Q-values. The cost function will have a discount factor of 0.95 . The weights and biases will be tuned with a learning rate of $1e^{-3}$.

An alternative approach to solving this problem is to predict Q-values with a convolutional neural network (CNN), which takes a scaled down image of the board as its input. This has the advantage of not relying on hand picked data to represent the state, allowing it to find more subtle patterns during its training. It is also computationally fast, and excels at processing grid-like data. However as the prior work by Sai showed, these networks are not well suited to play Tetris due to spacial invariance. A CNN may be able to perform well in this environment, but it is not the best fit model for solving the given task

6 Evaluation Metrics

One metric I considered evaluating the model on was the number of rows it was able to clear in a single run. The number of rows are correlated to both the level and score reached, and is a good indicator of the models ability to compactly organize a random assortment of Tetrinos. However, one issue with this metric is it treats all line clears as equal. Clearing several lines at once awards bonus points, with 4 lines cleared in a single move (aka a Tetris) awarding the most. When humans play we have the added difficulty of reaction time, making the optimal strategy going for Tetris' exclusively. If the total rows cleared was the metric the model was evaluated on, it would have no incentive to learn a policy which postpones immediate reward in favor of a larger future reward, and therefore could not learn to play optimally.

Another metric that could judge the models quality is the level it reached. However, the level increases after every 10 lines cleared, making it a flawed metric for the same reasons as the total number of rows cleared.

The metric used to assess the models overall performance will be the scores it can reach. The way score is calculated can be seen in figure 2:

Level	Points for 1 line	Points for 2 lines	Points for 3 lines	Points for 4 lines
0	40	100	300	1200
1	80	200	600	2400
2	120	300	900	3600
...				
9	400	1000	3000	12000
n	40 * (n + 1)	100 * (n + 1)	300 * (n + 1)	1200 * (n + 1)

Figure 2: NES equation to calculate score based on the number of lines cleared and the current level

The score is an appropriate metric to use because maximizing it is the ultimate goal of Tetris. Furthermore this is the metric humans base their own performance on, making it easy to compare the model to humans. To get an accurate measurement of the models score, 50 games of Tetris will be ran with the results being displayed in a graph.

7 Results and Discussion

The results obtained from training were incredibly strong. After training the model over 2,000 episodes, a policy was reached which is able to survive for extended periods of time. After 50 test episodes, the model received

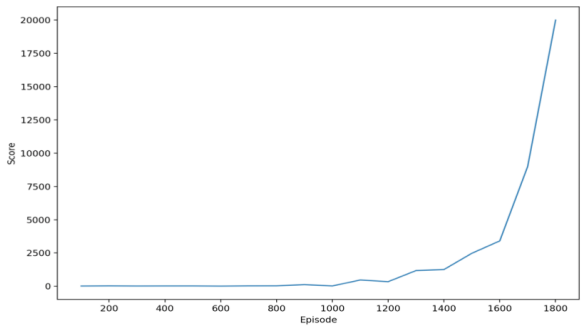

Figure 3: Maximum Score After Every 50 Training Episodes

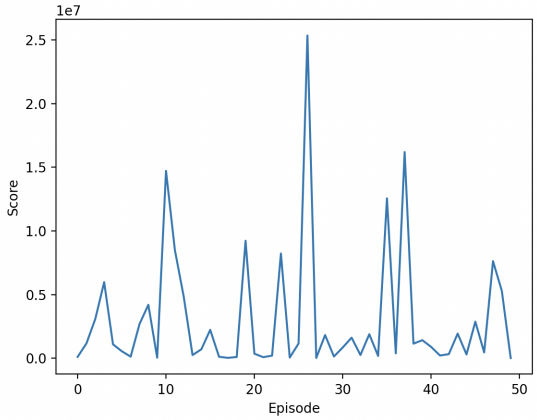

Figure 4: Scores obtained across 50 test episodes

an average score of 6.31 million with a high score of 25.64 million (See figure 4). For reference, the Tetris world record for a human is 2.94 million. These results greatly exceeded my expectations. The model makes many moves which are not great, yet manages to consistently recover from bad positions and survive well into the later levels.

However, it should be noted that a large part of the models success comes from the ability to place pieces immediately, allowing it to ignore the increasing rate at which blocks fall. At the quickest level(15) a tetrino will drop one tile every $7e^{-3}$ seconds, but the model can move a tetrino to the correct horizontal position in an average time of $3.5e^{-5}$ seconds. Once there the model pushes the piece down itself, so gravity will never affect the tetrinos. Gravity is the most difficult aspect of the game for humans to handle, yet the model does not find survival at top speeds any more difficult than during the early stages.

As figure 4 shows, the models performance is not the most consistent across episodes. Sometime runs such as test run 50 scored as low as 60,000, while other runs such as 27 were able to significantly surpass the best human players. This variance is likely a result of the randomness in the pieces the model receives. Although the model has discovered a solid policy, receiving the right or wrong piece in a critical position could drastically change the position of the board, making its performance somewhat reliant on luck.

One way I would improve this model in the future is implementing next piece information, which would allow the model to see several sequential pieces rather than just the active piece. This would increase the amount of (s,a) pairs for each state by a factor of 23 per piece, but greatly increase the computer's ability to organize Tetrinos.

Another adaptation I would make in the future is giving the model the ability to store pieces. Early iterations of the model ran slowly and had a difficult time training, so removing the ability to store pieces cut the number of (s,a) pairs in half. This helped improve performance initially, but after further optimization was implemented, the model ran fast enough to handle this added action. This would greatly improve the models performance, since it would give it twice as many options on where to place the next piece.

Another way I would improve upon the model is to use A* pathfinding to move the tetrinos into place. This would allow the computer to make more advanced moves than are currently possible. The current path finding algorithm is very elementary, and functions by moving the piece to the correct horizontal position before dropping it. With A* the agent would have greater control of how the tetrino is placed, allowing it to perform moves such as block shifting, T-spins, and wall kicks. Access to these additional actions should help increase the models ability to organize pieces.

8 Ethical Considerations

When creating a project it is important to explore any ethical concerns which may arise as a result. One concern with this project is the application it could have in online play. Tetris is a single player game, however there are versus modes online which put two players against each other. This model could easily be tuned to perform in this environment and give one player an unfair advantage over the other. Although many online platforms do have anti-cheat measurements enacted, they does not alleviate the moral issues of potentially causing harm.

Another ethical concern with this project is found in the model itself. Although the model is currently trained to play Tetris, the structure can easily be adapted to train in a different environment, as Mnih and his team discovered. This raises questions about the morality behind AI as a whole. Training algorithms on biased data sets could have serious real world repercussions, and this model is susceptible to the same flaw. This poses a moral question about publishing the model for easy use. It is a powerful tool which could do good, but is equally capable of causing harm through false or biased results.

9 Bibliography

- Human-level control through deep reinforcement learning
- 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.
- https://towardsdatascience.com/beating-the-world-record-in-tetris-gb-with-genetics-algorithm-6c0b2f5ace9b