Tetris Mastery: Enhancing Gameplay with Double Deep Q-Networks

Seyed Mohsen Sadeghi* Mehrab Kalantari†

January 2024

Abstract

This study explores the potential of a Double Deep Q-Network (DQN) in enhancing Tetris gameplay through reinforcement learning. Utilizing a convolutional neural network to estimate the Q function, our approach incorporates the DQN framework to guide decision-making at various game states. While computational constraints prevented the testing of our approach, the theoretical foundation and insights gained from the integration of the Double DQN provide a promising direction for future investigations.

1 Introduction

Tetris is one of the most popular and challenging games ever created, a game that requires players to stack pieces of different shapes on a rectangular grid. The goal is to fill in the gaps and avoid creating gaps that cannot be filled. To succeed in this game, players need to have good visual skills and strategic thinking. These are also the skills that convolutional neural networks (CNNs) can learn through reinforcement learning (RL). In this study, we use a convolutional Deep Q network to create an agent that can play Tetris at a high level, using the raw pixels of the game screen and a little bit of inductive bias for faster convergence as input. We believe that this is a natural and effective way to tackle this problem, as CNNs are well-suited for visual pattern recognition and object detection.

2 Methods

Q-Learning is a model free reinforcement learning algorithm that learns the optimal policy for an agent by estimating the action-value function, also known

*Department of Data and Computer Science, Shahid Beheshti University, Tehran, Iran. Email: seyedm.sadeghi@mail.sbu.ac.ir, Student ID: 99222059

†Department of Data and Computer Science, Shahid Beheshti University, Tehran, Iran. Email: me.kalantari@mail.sbu.ac.ir, Student ID: 99222088

as the Q-function. The Q-function maps a state-action pair to the expected future reward of taking that action in that state and following the optimal policy thereafter. The Q-function is learned iteratively by applying the Bellman equation, which states that the optimal Q-value for a state-action pair is equal to the immediate reward plus the discounted Q-value for the next state-action pair:

$$Q^*(s,a) = r + \gamma \max_{a'} Q^*(s',a')$$

where s and a are the current state and action, r is the reward, γ is the discount factor, s' and a' are the next state and action, and $\max_{a'}$ denotes the maximum over all possible actions.

Double Q-learning is a variant of Q-learning that addresses the overestimation bias of Q-learning, which occurs when the maximum Q-value is consistently overestimated due to noise or approximation errors. This can lead to suboptimal policies and poor performance. Double Q-learning mitigates this problem by using two Q-functions, Q_A and Q_B, and updating them alternately with different experiences. The key idea is to use one Q-function to select the best action and the other Q-function to evaluate that action, breaking the positive feedback loop that causes overestimation. The update rule for Double Q-learning is:

$$Q_A(s,a) \leftarrow Q_A(s,a) + \alpha \left[r + \gamma Q_B(s', \arg\max_{a'} Q_A(s',a')) - Q_A(s,a) \right]$$

and vice versa for Q_B, where α is the learning rate and $\arg\max_{a'}$ denotes the action that maximizes the Q-value.

Double DQN is an adaptation of Double Q-learning to the deep Q-learning framework. It uses the same idea of decoupling action selection and action evaluation, but instead of using two separate Q-networks, it uses the main network and the target network. The target Q-values are computed as follows:

$$y = r + \gamma Q_{target}(s', \arg\max_{a'} Q_{main}(s',a'))$$

where Q_{main} and Q_{target} are the main network and the target network, respectively. The main network is then trained to minimize the mean-squared error between the predicted Q-values and the target Q-values.

3 Network Architecture

The network architecture consists of three convolutional layers, followed by three fully connected layers. The convolutional layers are used to extract features from the board state, which is represented as a one-channel image. Each convolutional layer has a kernel size of three and is followed by a ReLU activation function. The number of filters in the convolutional layers are 8, 16, and 32, respectively. The output of the third convolutional layer is flattened and fed into the first fully connected layer, which has 512 units and a ReLU activation function. The second fully connected layer has 64 units and a ReLU activation function. The third fully connected layer has five units, corresponding to the number

of possible actions. The network also has an embedding layer that maps six features extracted from the board manually (they will be discussed later) to a scalar value, which is multiplied by the output of the second fully connected layer. This allows the network to learn different policies for different situations. The embedding layer uses a sigmoid activation function. The network is trained using the Adam optimizer with a learning rate of 3×10^{-4} and the mean-squared error loss function.

4 The Environment

Our version of Tetris has seven types of blocks that fall from the sky. The player can choose one of five actions: do nothing, move left, move right, rotate, or push down. The player can perform three actions before the block drops one row. The player also knows the next block in advance. The agent's observation space includes the board and six features: the current and next block indices, the current block orientation (in radians), the number of holes, the total height, and the bumpiness. A hole is an empty cell with a tile above it. Holes are hard to clear, so we want to avoid them. A well is a column that is lower than its neighbors. Wells prevent easy line clears, so we want to reduce them. Bumpiness measures the variation of column heights by summing the absolute differences between adjacent columns. Height measures the sum of column heights. We want to minimize both bumpiness and height.

5 Reward Function

The reward module is designed to encourage the agent to clear as many rows as possible, while avoiding holes, wells, and high columns. The reward function is composed of several terms, each reflecting a different aspect of the game. The first term is the ratio of the board area to the number of holes, which penalizes the agent for creating holes

6 Challenges and limitation

One of the main challenges we faced in our project was the slow convergence rate of the epsilon greedy approach. This approach is based on balancing exploration and exploitation, where the agent chooses a random action with a probability epsilon and the best action with a probability 1-epsilon. The epsilon value is gradually decreased over time, so that the agent explores more in the beginning and exploits more in the end. However, this process requires a large number of iterations for the epsilon to reach a sufficiently low value, which makes the training process very time-consuming. As a result, we could not fully test our method and evaluate the agent's performance in different scenarios. Although we observed some improvement in the agent's behavior over time, we do not know how well the agent would perform in the long run and what is the optimal

value of epsilon for our problem. Therefore, we suggest that future work should test our method with more advanced tools and techniques, such as parallel computing, hyperparameter tuning, or alternative exploration strategies, to speed up the convergence and improve the performance of the agent.

7 Conclusion

In this report, we presented a deep Q-network based reinforcement learning approach for playing Tetris. We designed a network architecture that consists of three convolutional layers, three fully connected layers, and an embedding layer that maps the inductive bias to a scalar value. We also designed a reward function that encourages the agent to clear rows, avoid holes, and reduce height and bumpiness. We implemented our method using PyTorch and tested it on a custom Tetris environment.

Our method shows promising results in learning to play Tetris, as the agent improves its performance over time and achieves high scores. Our method also demonstrates the effectiveness of using an embedding layer to incorporate the inductive bias into the network. Our method can be applied to other games or problems that require learning from complex and high-dimensional state spaces.