Approximate Dynamic Programming for Tetris

Wenkai Han, Zhongxiao Li

August 2019

1 Introduction

Tetris is a single player game whose the goal is to stack random falling pieces onto a fixed rectangular game board by rotating and translating them. The player wins score when one horizontal line is completed with pieces and cleared from the board, while game terminates when the maximum height exceeds the height of the red bar. The simple rules and non-trivial mathematical nature behind Tetris draw much attention from both human player side and artificial intelligence community. () proved that Tetris is NP-complete on its offline version, so it is impossible to exactly solve the problem efficiently. Besides, the state number is also huge. On a $m \times n$ game board with k pieces, the total number of states is $k \times 2^{m \times n}$. That huge state number encourages us to use the approximate dynamic programming strategies for large game boards and complex pieces.

In this project, our objective is to develop a utility-based agent to play Tetris. We first use an offline algorithm, value iteration(), on small game boards as a baseline. We then use the online algorithm, SARSA(), with both table-based representation and linear functional approximation of the state-action pair value function (Q function). Given the next piece and current state, the agent evaluates and assigns scores to any legal movement and take actions according to the score.

2 Method

In this section, we discuss the formulation of the Tetris game and how it is cast into the dynammic programming framework. We also describe both the table-based algorithms and the function approximation algorithms to solve the problem.

Figure 1: On the left hand side is the small game board (3×3) and with the pieces that are used. On the right is the larger game board (8×6) with more complex pieces.

In the game of Tetris, the agent is faced with placing a random falling piece in Fig. 1. The falling piece can be rotated and translated from left to right. We consider it as a discrete-time finite-state stochastic system. Let the set $\mathcal{S}$ be the state space of the game. For each $s \in \mathcal{S}$, it is uniquely determined

by the current game board configuration and the falling piece. The actions allowed for each state is denoted by the set $\mathcal{A}(s)$, which consists of allowed rotations and translations of the current piece. The state-action value function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ maps a state-action pair (s, a) to a real value that estimate the expected reward of s after taking action a. The stochastic matrix P^a_{ij} gives the probability of transitioning from state i to state j when the agent takes a as its action at current step. In the Tetris game, the randomness comes from the random choice of the next falling piece which is uniformly sampled from the set of pieces by the game.

As shown in Fig. 1, we are using two game boards in our experiment. The board capacity (the number of board cells beneath the red bar) of the smaller one is 3×3 and the larger one is of 4×6. Each of them has three candidate pieces but the small board's pieces are "easier" ones. For table-based algorithms, as they need to store the Q values of all states in the memory which grows exponentially as the board size increases, are only performed on the smaller ones. For function approximation algorithms, both of them are used.

2.1 Table-Based Algorithm: Value Iteration

Value iteration is an offline planning algorithm which assumes that the agent knows completely the dynamics of the game, *i.e.* the state transition probabilities and the probability distribution of reward. The algorithm computes the state-action value function $Q(s, a)$ by the following iteration:

$$Q^+(s, a) \leftarrow R^a_s + \gamma \sum_{s' \in \mathcal{S}} P^a_{ss'} \max_{a' \in \mathcal{A}(s')} Q(s', a') \tag{1}$$

where R^a_s is the expected reward of state s after taking action a. In the Tetris game, because the reward is simply the number of rows cleared (we also give a -1 penalty when the game ends prematurely by hitting the red bar), R^a_s can be easily determined.

2.2 Table-Based Algorithm: Sarsa

The state–action–reward–state–action (SARSA) algorithm [cite], is an online reinforcement learning algorithm. The Sarsa algorithm iteratively updates $\tilde{Q}(s, a)$, the estimation of $Q(s, a)$, using the temporal difference (TD) estimation of the right-hand side of Equation (1):

$$\tilde{Q}^+(s, a) \leftarrow \tilde{Q}(s, a) + \alpha \left[R + \gamma \tilde{Q}(s', a') - \tilde{Q}(s, a) \right] \tag{2}$$

where R is the observed reward after taking action a at state s. s' and a' are next state and next action after following the ϵ-greedy policy under the current $\tilde{Q}$:

$$\pi(s, a) = \begin{cases} \frac{\epsilon}{|\mathcal{A}(s)|} + 1 - \epsilon & \text{if } a = \arg\max_{a'} \tilde{Q}(s, a') \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases} \tag{3}$$

Features	Description
Max height of state s	This feature counts the max height of current state after making an action. The larger the max height is, the worse it is.
Min height of state s	This feature counts the min height of current state after making an action. The smaller the min height is, the better it is.
Reward after taking action a	This feature counts the number of rows cleared after taking an action. When a game terminates, set reward as -1. The greater the reward is, the better it is.
Number of holes after taking action a	A hole is defined as a empty position beneath a filled slot in a column, this feature counts total number of holes in the board. The smaller the number of holes is,the better it is.

Table 1: Features in linear approximation

where $\pi(s, a)$ is the probability of taking action at state s.

2.3 Function Approximation Algorithm: Sarsa with Linear Function Approximation

However, when the state space is large, we cannot enumerate all states, thus a parametric function that approximates $Q(s, a)$ is needed. Naturally, we can define some good heuristics as human players makes evaluations when placing a new piece, *e.g.*, the maximum height of current state. A move that results in a higher maximum height is worse than a move that has a lower max height. All features used in our work is shown in Table 1.

We consider the approximation function F_w as a combination of these feature functions using a weighted linear sum. If we let f_i and w_i, with $i = 1 \dots n$ denoting the n feature functions and their corresponding weights, the approximation function F_w is then defined as:

$$F_w(s, a) = \sum_{i=1}^{n} w_i f_i(s, a) \tag{4}$$

We also use the Sarsa algorithm to update the weight parameters of the linear

model. The update formula is:

$$w_i^+ \leftarrow w_i + \alpha \left[R + \gamma F_w(s', a') - F_w(s, a) \right] f_i(s, a) \tag{5}$$

Like before, s' and a' are also the are next state and next action after following the ϵ-greedy policy under the current F_w.