Baby Tetris First Part Report

ERGIN Seçkin Yağmur and GROUSSON Dylan

November 29, 2025

https://github.com/diaarca/baby-tetris/

1 Question 1: Discounted MDP For Player

The objective of this part is to choose the optimal policy that maximizes the expected cumulative discounted reward over time for the "player".

States
The state is represented by the current configuration of the grid, and the next piece to be placed.
$S :=$ (current grid, next piece to place)

Actions
The actions correspond to a position and orientation to place the piece on the grid.
$A :=$ (position, orientation)

Transition Function
The transition function is defined as $P(s'|s, a) := 1/2$
There are two possible pieces (I or L) as the next piece to place, and considering our state representation, and our assumption of uniform distribution between the pieces, each transition has an equal probability of 1/2.

Reward Function and Discount Factor
The reward function is computed based on the number of lines cleared after placing the piece, and a discount factor λ is used to weigh future rewards.
$R(s, a, s') :=$ number of lines cleared and $\lambda \in [0, 1]$

1.1 Finding Optimal Policy

We use the value iteration method to find the optimal policy that maximizes the expected rewards, iteratively updating the value function for each state based on the Bellman equation until it converges to the optimal value (margin ϵ).

$$V(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s, a) \cdot (R(s, a, s') + \lambda V(s'))$$

Value iteration is better suited for maximizing the expected discounted reward since it computes the optimal value then extracts the policy and easier to implement

1.2 Implementation

1.2.1 Main Architecture

Based on the our design choices from Section 1, we decided to use an object oriented architecture (C++) for our implementation.

For our structure we have declared 7 classes:

1. *Point*: classical point (x,y) class used for grid coordinates
2. *Field*: the class that contain a boolean grid s.t. grid[i][j] = 1 if the corresponding cell is full in the field
3. *Tromino*: the class that correspond to the baby Tetris pieces (I Piece or L Piece)
4. *State*: the class that contain both a field grid and an incoming Tromino
5. *Action*: that represent an action choosable on a given state (combination for placement and rotation of an incoming Tromino)
6. *Game*: the class that contain the configuration (completed lines scores), the current state and score of a baby Tetris game (the configuration is loaded from a file : *config.txt*)
7. *MDP*: the class that all the necessary methods in order to compute the value iteration over a game

Now that we overviewed the global implementation architecture, we will describe our implementation choices.

1.2.2 Implementation Choices

- **Piece teleported to their final position**: in order to make the number of intermediate states when choosing an action, we compute on a state all available actions on this state, then according to the policy we make a choice over these actions and we teleport the incoming piece in the current state directly to its final position (with the right rotation)
- **Reward Function**: The reward function as defined in Section 1 take both the current state s and an available action a_s. Here since we can calculate the available actions, we can know in which state s' we're going to arrive and we can compute directly in s' the number of completed lines before remove them right to the game. Then we abstract the reward function by counting the number of complete lines in the resulting state
- **Game Ending**: According to the subject, the game should end whenever a piece exceed the height limit of the grid. Again there, we can compute all available actions on a given state, since we cannot place a Tromino outside the grid, such an action will not appear in the set of available actions. Then the game end when there is no available action for the current game state
- **Max Iteration**: Since the value iteration isn't an exact method, if we fix an ϵ which is too small, we may iterate for a very long moment. For this reason we fix a maximum number of iteration for the expected value improvement, then we always a value iteration which compute in a reasonable time even if the expected value vector is less qualitative than the ϵ margin we fixed

1.3 Results

Here, we will present multiple examples of execution of our implementation. In these executions we will use fixed values for: width = 4, $\epsilon = 0.00000001$, maxIteration = 100, probaIPiece = 0.5 and maxGameAction = 10000. Note that we fixed our ϵ that low to be as close to the optimal policy as possible and that our algorithm is in any case bounded in time by the *maxIteration*. In contrary, we will make evolve: the height $\in \{4, 5\}$ and the discount factor $\lambda \in [0, 1]$.

For each execution we will observe:

1. the iterations of the value iteration algorithm
2. the average expected gain of the optimal policy
3. the final score either at the end of the game or if we reach the maximum number of actions
4. the execution time of the said execution

1.3.1 Executions

```
./bin/tetris
Loaded config: [1, 3, 6]

All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.1

Iteration i = 0 and delta = 6
Iteration i = 1 and delta = 0.6
Iteration i = 2 and delta = 0.06
Iteration i = 3 and delta =
0.00525
Iteration i = 4 and delta =
0.00045
Iteration i = 5 and delta = 0

average over final V 0.412926
....
....
....
....

Game Over! Global score: 7861 in
10000 actions
```

Listing 1: height: 4 and λ: 0.1

```
./bin/tetris
Loaded config: [1, 3, 6]

All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.3

Iteration i = 0 and delta = 6
Iteration i = 1 and delta = 1.8
Iteration i = 2 and delta = 0.54
Iteration i = 3 and delta =
0.14175
Iteration i = 4 and delta =
0.03645
Iteration i = 5 and delta = 0

average over final V 0.452512
***.
**..
*.**
*.**

Game Over! Global score: 1500 in
1897 actions
```

Listing 1: height: 4 and λ: 0.3

```
./bin/tetris
Loaded config: [1, 3, 6]

All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.5

Iteration i = 0 and delta = 6
Iteration i = 1 and delta = 3
Iteration i = 2 and delta = 1.5
Iteration i = 3 and delta =
0.65625
Iteration i = 4 and delta =
0.28125
Iteration i = 5 and delta = 0

average over final V 0.495819
***.
..**
***.
***.

Game Over! Global score: 1406 in
1665 actions
```

Listing 2: height: 4 and λ: 0.5

```
./bin/tetris
Loaded config: [1, 3, 6]

All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.7

Iteration i = 0 and delta = 6
Iteration i = 1 and delta = 4.2
Iteration i = 2 and delta = 2.94
Iteration i = 3 and delta =
1.80075
Iteration i = 4 and delta =
1.08045
Iteration i = 5 and delta = 0

average over final V 0.544145
***.
.***
***.
***.

Game Over! Global score: 373 in
444 actions
```

Listing 2: height: 4 and λ: 0.7

1.4 Incoming Improvement

For now we compute naively the value iteration algorithm on all state combinations, which represents

$$\begin{aligned} &= 2^{\text{gridHeight}*\text{gridWidth}} * \text{nbOfTrominoTypes} \\ &= 2^{4*4} * 2 \\ &= 2^{17} \\ &= 131072 \end{aligned}$$

states in the smallest (4×4) version of the game. In the near future we want to considerably reduce this number by computing the value iteration only an states that are reachable from a given starting state s_0. For this purpose, we will make use of a Breadth First Search algorithm which will explore iteratively all reachable states from the current one.