In the name of God

| | | |
|---|---|---|
| School of Electrical and Computer Engineering | Shiraz University | Spring 1405 |
| Reinforcement Learning | Due Date: Sep. 17, 23:59 | Final Project |

# Bricks Game

**Goal:** Design and implement a custom RL environment for the Bricks game, then train a RL agent to maximize received reward.

**Objective:** Students will learn to formalize a novel problem into an RL framework, and evaluate how environment design choices affect agent learning performance.

This project asks you to design, implement, and train an agent for a custom RL environment. Unlike Tetris, where new pieces drop from the top and are rotated/positioned before landing, in this game bricks are pushed up from the bottom of the board, and the player's only action is to slide an existing brick left or right. Gravity then pulls the brick down to fill any gap beneath it. The objective is to complete horizontal rows, which are cleared for points.

This project is intentionally open-ended in its lower-level design choices (network, state, and action representation) so that you get practice translating an informal game description into a formal MDP.

## Part 1: The Environment

### 1.1. Environment:

The game environment is a grid of size 6 × 5 (6 rows, 5 columns each), each row within the environment can contain several bricks of different sizes alongside with empty spaces. Bricks size refers to the length of the brick in the row, there are bricks of length 1, 2, 3, and 4 in this game. The only constraint is that the total length of all bricks and empty spaces should not exceed the column size.

### 1.2. Bricks respawning:

New bricks are introduced at the bottom of the board after an action is taken and the environment changes as the result of the action, pushing existing bricks upward by one row. Introduced bricks are chosen based on the respawn probability of each brick size.

Several bricks of a same size are allowed

Several bricks of different sizes are allowed

The new row must has at least one brick

### 1.3. Action Space:

The agent selects one brick already on the board and moves it to the left or right as much as the environment allows. For example, if a brick of size one is placed at the left of two empty spaces and at the right of another brick, it can only move 1 or 2 cells to the right. You should decide on the action space of this game.

There may be several invalid actions in the action space including:

- selecting an empty cell
- moving into an already-occupied cell
- moving off the edge of the board

you must decide how to handle these (action masking vs. a reward penalty) and justify your choice.

### 1.4. Transition Dynamics:

- **Move:** The selected brick shifts up to 4 cells in the chosen direction, if the destination cell is empty and within bounds.
- **Gravity:** After the move, for each brick, if the cells directly below it are empty, the brick falls straight down until it lands on another brick or reaches the floor.
- **Row-clear check:** Any row that becomes completely filled (no empty cells) is cleared. All bricks in that row are removed, and all rows above shift down by one. Please keep in mind that clearing a row may cause another brick to fall down again, and maybe as a result, make another complete row.
- If multiple rows are completed in a single resolution step, they are all cleared simultaneously and agent gets a bonus reward.

### 1.5. Reward:

A positive reward for each row cleared in a step, with a bonus multiplier for clearing multiple rows simultaneously in one step. To encourage the agent to play efficiently, a large negative reward for invalid actions (if you chose to use negative reward instead of invalid action masking) or reaching the terminal state.

### 1.6. Termination:

The episode ends when a brick's position reaches the topmost row of the board (i.e., no more room to accommodate a spawn).

## Part 2: DQN control method

To solve this problem, you must implement a Deep Q-Network (DQN), a reinforcement learning algorithm that combines Q-learning with function approximation using a neural network. DQN is used when the state space is too large for traditional Q-tables (for example more than a few thousands). Another advantages of DQN is that a neural network can generalize better across unseen or rarely visited states which makes it popular among the RL agent developers.

You should design the network yourself (just its architecture) but you have to justify your choice in the report.

**DQN Components**:

- Experience Replay Buffer:
  - Store transitions in the format: (state, action, reward, next_state, done)
  - Sample random mini-batches from this buffer to update the Q-network
  - Helps decorrelate the training data and stabilize learning
- Target Network
  - Use a second, frozen Q-network (the target network) to compute the target Q-values
  - Periodically copy weights from the main Q-network to the target network
  - Prevents harmful feedback loops during learning

**Some useful link for DQN**:

- https://medium.com/@samina.amin/deep-q-learning-dqn-71c109586bae
- https://docs.pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

## Parameters:

- $\gamma = 0.99$
- $\alpha = 0.0005$
- $reply\ buffer\ size = 100{,}000$
- $batch\ size = 32$
- $maximum\ episode\ length = at\ least\ 1000$
- $update\ frequency = every\ 1000\ steps$
- $number\ of\ trainign\ episode = at\ least\ 250$
- $\varepsilon = start\ with\ 1\ and\ decay\ gradually\ to\ 0.01$
- $respawn\ probability$:
  - $empty\ cell: 1/3$
  - $bricks\ with\ length\ of\ 1: 2/9$
  - $bricks\ with\ length\ of\ 2: 2/9$
  - $bricks\ with\ length\ of\ 3: 2/9$
- $one\ row\ clear\ reward: +1$
- $multiple\ row\ clear\ reward: n + 0.2 \times (n-1)^2$
- $reaching\ terminal\ state\ reward: -10$
- $taking\ invalid\ action\ reward: -10$

## Expected Deliverables:

- Your network architecture.
- Your choice on state space and action space
- Your action space masking mechanism, if any.
- Simulate 1000 different episodes and plot these:
  - Histogram of received reward
  - Histogram of episode length
  - Histogram of the percentage valid actions taken of each episode

- Trained network of part 2 and example code showing how to load it and run an independent simulation using the trained network.
- Plot these metrics on your train process:
    - Return per episode over time.
    - Episode length
    - Valid action taken percentage
    - Loss curve over training.
- A short report including:
    - The above-mentioned figures,
    - Each design choice you were made in this project and a short justification for that choice.
- Your codes in **THE CORRECT FILE TYPE**.

## Notes:

- Allowed programming languages: **MATLAB**, **Python**
- Any sign of cheating would result in a **zero** grade for this assignment.
- Your report should be in PDF format.
- You don't have to add your code to your report, unless it is necessary for making a point.
- Place your code in a directory named "codes" and your generated figures in a directory named "results". Create a .rar file containing both directories and your report, rename the created file as "FirstNameFamilyName-StudentNumber.rar" and upload it to the site.

  E.g.: AliSaber-40330249.rar

```
|------ codes
|       |------ code_1.py
|       `------ code_2.m
|------ results
|       |----- figure_1.tif
|       `----- figure_2.tif
`----- report.pdf
```