Project proposal

This document details the proposal for the RLDMU project, 2023.

Organisation

The project is done in python and the work is shared across the 2 team members. The project’s source code can always be monitored on github: <https://github.com/maximewel/rldmuu_project>

The work is done in an iterative way. Firstly, a working discrete environment is created and tested. Then the environment can be rendered with a 2D GUI. Lastly, the environment can be passed to continuous observations in order to test the differences between Q-table based methods and Deep learning ones.

Environment overview

Arrived on the moon with the Lunar Lander, it is now time to explore. For this purpose, the Lunar Explorer was developed. Its objective is to get to a specific location. On the way there, there are potential interesting ores that could be a source of yet unknown metals (green cell). The Lunar Explorer aims to collect as many other precious metals as possible (blue cell). However, the Lunar Explorer has a limited drilling capacity (e.g. 3 drillings). It will therefore not be able to drill all the available resources but will have to select the most profitable ones. Furthermore, its landing point has been precisely defined (black cell). Overall, the Lunar Explorer should explore and discover the optimal exploitation of the current map.

However, no gas station is yet available on the moon. The Lunar Explorer can only use the initially available fuel and avoid useless trips - translating to a penalty of “” for each iteration. The lunar surface is also not very friendly. This is why, depending on the field, the Lunar Explorer can suffer from unexpected behaviours and must adapt its movements.

The goal of this environment is to have an agent balance between the two main mechanics – minerals and terrain type – in order to exploit the map.

Example of a standard map, no movement tile is reflected:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **…** | **…** | **…** | **…** | **…** | **…** | **…** | **…** | **…** | **n** |
| **1** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **…** |  |  |  |  |  |  |  |  |  |  |  |
| **n** |  |  |  |  |  |  |  |  |  |  |  |

TABLE 1 Example of the lunar explorer environment

*Legend*

* • Start location
* • Potentially interesting mineral
* • End location, stops simulation

Discrete environment

The first iteration aims to create a basic environment, with discrete observations and discrete states. This is typically a problem that can be solved by a Qtable-based algorithm.

Observation space

The observation space has 4 components, each of them having a limited range of valid values.

State space: , with

* being the horizontal position
* being the vertical position
* being the horizontal speed,
* being the vertical speed,

The tile type is not part of the observations as this is useless for Q-tables based algorithms: They will learn a map by hearth, and not learn to recognize a tile type and adapt to it. Furthermore, as only the current tile is returned as the observation, no generalization capabilities are expected to be developed by algorithm running on this env.

More explanation about the speed mechanism is detailed on the next section.

A little verification/analysis of the Q-table resulting of that:

|S| = map dimensions \* possible speeds \* possible types = [x \* y] \* [3 \* 3]

* For a 30\*30 map, this represents 8100 possible states.

Action space: , |A| = 6

* On the 30\*30 map, this represents a Qtable of size 48600.

Movements / Speed

The agent can move around the map by going in either direction or stopping. The speed can be seen as a 2D vector of two integer values (Vx, Vy). In discrete mode, the speed can be either “moving” (-1, 1) or “not moving” (0). Only one speed component can be activated at a time (no diagonal movements). The speed behaves as follow:

* The speed encodes the last movement performed. Going right (or left) sets to 1 (resp -1) the horizontal speed, down / up set to 1 (resp -1) the vertical speed for the next state.
* Doing nothing means the explorer stops and resets the speed to 0.
* No cost in turning: When going from horizontal to vertical or opposite, the absolute value of the speed is carried, only the orientation changes. This is required to keep the speed consistent across all manhattan paths and avoid shortcurts on slow tiles.
  + This means that going right then up results in the Vx, Vy speed to go from (0,0) -> (1, 0) -> (0, -1)
* Going into an opposite direction resets the speed to 0 (left to right, top to bottom, and inverse)

Surface tile types

The surface of the moon is not uniform. The lunar explorer has many challenges on its path.

The Next state is relative to the absolute value of the speed of the rover, the current and next tile type, and the chosen action:

1. Speed type “slow”: This terrain is frail and can break under the explorer’s weight: advance carefully.
   * If the speed is 0 when exiting the tile, the explorer can safely travel. If the speed is not 0, then the rover has a chance to fall and the simulation stops. This means that the explorer has to stop before exiting every slow tile it wants to go through.
2. Speed type “random”: This terrain is covered in craters, don’t get lost ! There is a chance to be moved to adjacent tiles when exiting it, regardless of the direction taken.
3. Speed type “normal”: Regular moon terrain. The agent will always move to the juxtaposed state, according to the action.
4. Speed type “fast”: This road-like terrain is particularly adapted to the explorer’s locomotion. When exiting with |speed| > 0, The agent has probability to move to the juxtaposed state and probability to move two states away from the current state, according to the action.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Current state with speed type** | | **Absolute speed** | **Action (Left, right, top, down)** | **New state coordinates** | **Prob.** |
|  |  | 0 |  |  |  |
|  |  | 1 |  |  |  |
|  |  | 1 |  |  |  |
|  |  | any |  |  |  |
|  |  | any |  |  |  |
|  |  | any |  |  |  |
|  |  | 0 |  |  |  |
|  |  | 1 |  |  |  |
|  |  | 1 |  |  |  |

Solution proposal

* Which algos do we want to try and why?

We will use a plethora of different algorithms developed during the course and lab of the RDLMU course. Namely, we have Q-table based algorihtms:

* Qlearning
* Sarsa
* DynaQ
* Eventually: Eligibility traces

And deep-learning based ones:

* DQN in pytorch, soft update
* DQN in tensorflow, full update every N epochs

The test process can be: Decade an epsilon value and a decaying methodology, and explore different algorithms: see how they react on the same map, how fast / stable they are in learning, their maximum performance, etc.

From discrete to continuous

If time allows it, the ‘position’ and ‘speed’ observations can be moved to continuous. In this case, similar to the lunar lender, the actions increment the speed by a set amount of acceleration each time they are called to remain discrete actions. The changes would then be:

* Some tiles can slightly change behavior (ex: speed tile ‘propulsion’ you by adding speed, only activated above a set amount of speed; craters having moving probabilities of being lost depending on speed value, etc…). The speed would stay limiter to the bound [-1; 1] in order to have a single tile detection in the board management.

In order to fit the continuous space and to allow for the testing of generalisation, the observation space would be:

State space: with

* ‘Activation grid’: Simple array given in observation with the same dimensions as the grid. The value of each tile is 1 of the player is in the tile, 0 if the player has never been in the tile, and a decreasing value between 0 and -1 for each tile previously passed through. This allows the agent to easily identify its position and path to reach it.
* Fov: tile type of close tiles In order to be able to work on new maps and not only train on one. The agents have to be able to react to their surroundings - for deep agents, not Q-table based ones. To avoid giving the whole grid at each observation, the FOV gives the tile type of tiles that are close to the player in an N\*N surrounding.
* being the horizontal speed,
* being the vertical speed,

In this form, the seed can be changed during training to teach the agent to react to the observation and not only find the best path for a given, single map.