Alma Mater Studiorum · Università di Bologna

Dipartimento di Informatica - Scienza e Ingegneria
Artificial Intelligence

Flatland Challenge

Project Presentation

Professor
Andrea Asperti

Students
Alessandro Lombardi
Fiorenzo Parascandolo

Academic Year 2019/2020

Contents

1 Flatland **1**

1.1 The classes RailAgentStatus and EnvAgent 1

1.2 The class RailEnv 2

2 Multi Agent Reinforcement Learning (MARL) **5**

1 Flatland

The analyzed version is the 2.1.10

1.1 The classes RailAgentStatus and EnvAgent

RailAgentStatus extends Python IntEnum and assumes the following values:

- READY_TO_DEPART (0) the agent is not in the grid yet (position is None), the prediction is to stay at the starting position. If a MOVE_* action is performed during this state it becomes ACTIVE.
- ACTIVE (1) the agent is in the grid (position is not None) and hasn't reached the target yet, the prediction is the remaining path.
- DONE (2) the agent is still in the grid (position is not None) but has already reached the target, the prediction is to stay at the target forever.
- DONE_REMOVED (3) the agent has reached the target and it's removed from the grid.

Grid4TransitionsEnum extends Python IntEnum and assumes the following values: NORTH (0), EAST (1), SOUTH (2), WEST (3). Grid4TransitionsEnum is used to indicate absolute directions, related to the environment, like a compass. Possible usages are storing where the agent is facing or computing legal actions, for example including as observation a one hot encoding of the directions where the agent can move.
EnvAgent class models the agent and encapsulates in its internal state the following attributes:

- initial_position: Tuple[int, int], initial coordinate.
- initial_direction: Grid4TransitionsEnum, the initial agent facing direction.
- direction: Grid4TransitionsEnum, the current facing direction.
- target: Tuple[int, int], the final coordinate.
- moving: bool, True if the agent is in a moving state.
- speed_data: dictionary, TODO
- malfunction_data: dictionary, TODO
- status: RailAgentStatus, the current agent status.

The speed of an agent contains the keys 'position_fraction' used as a counter of the percentage of completion of an movement from a cell to another, 'speed' the value between 0 and 1 used to increment the 'position_fraction' and 'transition_action_on_cellexit' which contains the action to perform on the next cell (if it completes the one in the current step).

The malfunction of an agent contains the keys 'malfunction' which contains how many steps are necessary to fix the agent, 'malfunction_rate', the mean rate (average number of events in an interval) of the Poisson distribution, 'next_malfunction' the number of steps the next malfunction will occur and 'nr_malfunctions' the number of previous malfunctions.

1.2 The class RailEnv

From the documentation

> RailEnv is an environment inspired by a (simplified version of) a rail network, in which agents (trains) have to navigate to their target locations in the shortest time possible, while at the same time cooperating to avoid bottlenecks.

In the *step* function the number of steps is updated and if the overall task is still uncompleted, for each agent the associated reward is initially put to zero, a malfunction is tried to be induced and the specific step is performed. The info of the agent are prepared and finally the end malfunctions are repaired. Agents are handled in the order in which are passed.

Environment Actions

The avaiable actions are:

- DO_NOTHING (0) Default action if None has been provided or the value is not within this list. If agent.moving is True then the agent will MOVE_FORWARD.
- MOVE_LEFT (1) If agent.moving is False then becomes True. If it's possible turn the agent left, changing its direction, otherwise if agent.moving is True tries the action MOVE_FORWARD.
- MOVE_FORWARD (2) If agent.moving is False then becomes True. It updates the direction of the agent and if the new cell is a dead-end the new direction is the opposite of the current.
- MOVE_RIGHT (3) If agent.moving is False then becomes True. If it's possible turn the agent right, changing its direction, otherwise if agent.moving is True tries the action MOVE_FORWARD.

- STOP_MOVING (4) If agent.moving is True then becomes False. A penalty will be added. Stop the agent in the current occupied cell.

```
if agent is in DONE or in DONE_REMOVED (1th case) then
    no reward is computed
    return
if agent is in READY_TO_DEPART (2th case) then
    if the provided action is a MOVE_* type and the initial cell is free the
      agent become ACTIVE and is initialized
    reward is computed
    return
if agent is in malfunction ((3th case)) then
    reward is computed
    return
if agent is at the beginning of a cell then
    update agent.moving considering the observations above depending on
      the different action types.
    if agent.moving then
        the wanted action validity is first checked and if it is valid
          (considering also the possibility to backup from an invalid
          MOVE_RIGHT or MOVE_LEFT to a valid MOVE_FORWARD)
          action is stored otherwise agent.moving becomes False and
          penalties are added, in this process agent.moving and
          agent.speed_data['transition_action_on_cellexit'] are updated
if agent.moving (4th case) then
    Updates the percentage of completion then if it is completely arrived
      on the next cell, before updating the position, the direction and clears
      the completion percentage it checks whether the new cell is free.
      Until the cell remains occupied in the future executions the agent will
      repeat this process.
    reward is computed
else
    reward is computed
end
```

Algorithm 1: The *_step_agent* algorithm

Some useful questions:

- An agent can stop during an action between two cells? Absolutely no, it's like any other action.
- Requested actions during a malfunction are ignored? Yes.

- Requested actions during a not completed movement are saved for after execution? No, because the condition at line 11 is not executed and the conditions in line 16 check whether the cell is free and possibly complete agent data. Actions are not allowed to change within the cell, each agent can only chose an action to be taken when entering a cell. This action is then executed when a step to the next cell is valid.

- How is possible to understand if an agent is ready to perform an action? The entry info_dict["action_required"] returned by the function *step* of **RailEnv** contains True for the given agent. This doesn't mean that the action will be successfully executed due to the presence of malfunctions or blocking agents, in this case info_dict["action_required"] will remain True.

- When an agent reaches DONE is removed automatically the following step? TODO Yes.

- Does an agent automatically pass from READY_TO_DEPART to ACTIVE at the beginning? TODO No. A MOVE_* is necessary.

- Do collisions occur? No, agents check if the cell is free before moving. Deadlocks are possible when two agents are one in front the other without any chance to change path.

Malfunctions

The strategy depends on the passed *malfunction_generator_and_process_data*. TODO

- A malfunction can occur during the resolution of another? TODO

- An agent could have a malfunction during the completion of an action between two cells? TODO Yes.

Speed

The different speed profiles (speed is between 0 and 1) can be generated using the schedule_generator. Speed configurations can be build using **ScheduleGenerators**. TODO

Rewards

The rewards are based on the following values:

- invalid_action_penalty which is currently set to 0, penalty for requesting an invalid action

- **step_penalty** which is -1 * alpha, penalty for a time step.
- **global_reward** which is 1 * beta, a sort of default penalty.
- stop_penalty which is currently set to 0, penalty for stopping a moving agent
- start_penalty which is currently set to 0, penalty for starting a stopped agent

The full step penalty is computed as the product between step_penalty and agent.speed_data['speed']. There are different rewards for different situations:

- single agents that are in DONE or in DONE_REMOVED have zero reward (1^{th} *_step_agent* case).
- all agents that have finished in this episode (checked at the end of the *step*) or previously (checked at the beginning of the *step*), have reward equal to the global_reward (when in *step* all agents have reached their target)
- full step penalty is assigned when an agent is READY_TO_DEPART and in the current turn moves or stay there (2^{th} *_step_agent* case), or when is in malfunction (3^{th} *_step_agent* case).
- full step penalty plus the other penalties (invalid_action_penalty, stop_penalty and start_penalty) when the agent is finishing actions or start new ones (4^{th} *_step_agent* case). Currently the other penalties are all set to zero.

So each train starts counting rewards since the beginning, not since it becomes ACTIVE. Currently it is possible to say that rewards are always full step excluding the end of the episode and the single agents that have finished which have reward equal to 0.

2 Multi Agent Reinforcement Learning (MARL)

> *"Specifically, MARL addresses the sequential decision-making problem of multiple autonomous agents that operate in a common environment, each of which aims to optimize its own long-term return by interacting with the environment and other agents."[1]*

This section provides an overview of some useful works and theory behind MARL that we consider useful to suggest approaches or solutions to the Flatland Challenge. MARL algorithms can be divided into three groups [4]:

- **Fully cooperative**, where agents collaborate to optimize a common long-term return.

- **Fully competitive**, where the return of agents usually sum up to zero.
- **Mix of the two**, where both cooperative and competitive agents are involved.

We consider the Flatland Challenge a fully cooperative environment since each agent seems to compete to reach faster its destination but it should consider the common minimization goal. There exist two closely related theoretical frameworks for MARL [4]:

- **Markov/Stochastic Games** where all agents share the same state and differently from classical single agent's Markov Decision Process, the optimal performance of each agent is controlled not only by its own policy, but also the choices of all other players of the game. Usually agents share a common reward function but it is possible to have different functions like in the team-average reward setting. Markov Games can be extended to a partially observed environment called in multi-agent scenarios **Decentralized POMDP (Dec-POMDP)**. In Dec-POMDP each agent has its own local observation of the system state, follow that with no accessibility to other agents' observations, is not possible to maintain a global belief state. Some techniques have been proposed to overcome this problem, but in general Dec-POMDP approaches are usually considered difficult to solve.
- **Extensive-Form Game** inspired from computational game theory, it handles imperfect information. It is usually used in mixed or competitive environments.

Flatland environment allows to customize how agents perceive the world and how agents are coordinated.

The involvement of Deep Learning to tackle the problem of MARL defines a new specific subject called **Multi-agent Deep Reinforcement Learning (MADRL)**. [2] presents four categories of recent MADRL works:

- **Analysis of emergent behaviors**, in general, they do not propose learning algorithms, their main focus is to analyze and evaluate DRL algorithms in a multi-agent environment.
- **Learning communication**, they study communications techniques to share information.
- **Learning cooperation**, they directly explore approaches based on actions and observations to build multi-agent systems.

- **Agents modeling agents**, they study how agents reason about others to fulfill a task.

MARL frameworks inevitably adds many challenging problems on the single-agent scenario.

> *"Learning in multiagent settings is fundamentally more difficult than the single-agent case due to the presence of multiagent pathologies, e.g., the moving target problem (non-stationarity) [2, 5, 10], curse of dimensionality [2, 5], multiagent credit assignment [31, 32], global exploration [8], and relative overgeneralization [33, 34, 35]."[2]*

Some problems that may affect the Flatland Challenge:

- **Non-Stationarity** [3] [4]. In a single-agent environment, an agent is concerning only the outcome of its own actions. In a multi-agent scenario, an agent observes not only the outcomes of its own action but also the behavior of other agents. Agents may interact with each other and learn concurrently leading to a continuous reshape of the environment and to nonstationarity. The classical DQN does not provide working solutions, some derivations have been proposed to deal with this problem such as **Deep Repeated Update Q-network (DRUQN)**, **Deep Loosely Coupled Q-network (DLCQN)** and **Lenient-DQN (LDQN)**.

- **Partial observability** [3]. In the single-agent scenario this type of problem can be modelled as **partially observable Markov decision process (POMDP)**. An extension of the POMDP approach, **Deep Recurrent Q-network (DRQN)**, developed to solve the multi-agent case is called **Deep Distributed Recurrent Q-network (DDRQN)**. Another technique to deal with partial observability is **Deep Recurrent Policy Inference Q-network (DRPIQN)**.

- **Scalability** [4]. To handle non-stationarity, each individual agent may need to account for the joint action space, whose dimension increases exponentially with the number of agents, this is also referred to as the combinatorial nature of MARL. Many methods have been proposed to tackle this problem, one of them is the extension of **Curriculum Learning** to a multi-agent scenario.

- **Information Structures** and **Training Schemes** [3] [4]. In the single-agent case is easier to understand what information is visible to the agent. In Markov games is sufficient to observe the current state, while on extensive-form games agents may need to recall the history of past decisions. In addition agents struggle to fully access information like rewards and policies of other agents, increasing the non-stationarity viewed by by individual agents. These considerations led to the development of different training schemes such as **centralized-learning-decentralized-execution**, **fully decentralized** and **decentralized setting with networked agents**.

References

[1] Robert Babuska. A comprehensive survey of multiagent reinforcement learning. 38:156–172, 03 2008.

[2] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 33(6):750–797, Oct 2019.

[3] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications. *IEEE Transactions on Cybernetics*, page 1–14, 2020.

[4] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms, 2019.