Warehouse Optimization - Reinforcement Learning

Nikunj Gupta

May 16, 2018

Introduction

Reinforcement Learning (RL) is basically mapping situations to actions in order to maximize the rewards (numerical). The RL agent attempts to discover the actions in respective situations which yield maximum reward by trying them, unlike other machine learning algorithms where the agent is told what actions to take. It is not necessary that only immediate rewards are taken into consideration, the agent might as well look for delayed rewards (maximizing the total reward in the long run). The basic idea is that the agent must be able to interact with the environment, sense the state and take actions that affect that state of the environment. The actions taken must bring the agent closer to achieve a particular goal or goals. [1]

Problem Statement

There is a warehouse with grid markers to specify locations.
1. one input bay at location (Ix, Iy)
2. one output bay at location (Ox, Oy)
3. 12 racks in a 4x3 rectangle (with locations such that there are corridors between two racks)

There are 4 robots, each one is capable of:

- Moving (from a point A to a point B)
- Picking an object from a rack/bay
- Dropping an object on a rack/bay
- Getting itself recharged by going to the charging station and requesting for recharge

There is a sequence of external requests at times $t_1 < t_2 < t_3 < ...$
Each request is of type:

- Store o1:T1, o2:T2,.. where object o1 is of type T1, and so on.
- Get n1:T1, n2:T2,.. where n1 objects of type T1, and so on.

When there is a **store** request, one or more robots are allocated to take the objects from the input bay to the free racks. When there is a **get** request, again one or more robots are allocated to pick up the required objects from the racks and drop them on the output bays. All the operations have some costs e.g. cost of moving an object from A to B depends upon the distance and the type of object. This determines the cost for Store and Get. The **problem** we are trying to address is, given the warehouse description above and a cost C, we would like to design an RL based system to ensure that the cost of Get requests are less than C.
Notes:
1. When there are more Get requests of certain type of objects, these objects could be arranged to be near the output bay. If there are no free racks near the output bay, then objects can be swapped to make space. Such swapping can be done by the robots in the spare time. This intuition should be validated by RL.
2. Swapping also incurs cost. There should be a generalized notion of cost to capture the efficiency of the warehouse. For any finite sequence S_k of requests, let the no. of objects to be accessed (Get) is N_k and the total cost incurred by the sequence is C_k. One definition of efficiency could be C_k/N_k: the lesser the value, the more efficient the warehouse is.
Figure 1 shows a look of the warehouse with 4 robots.

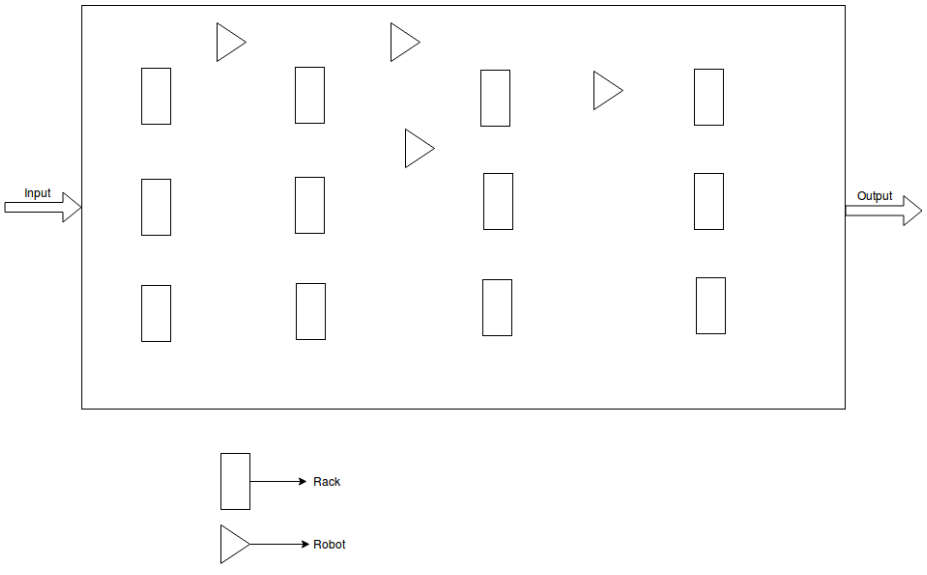

Figure 1: Warehouse (Top View)

OpenAI Gym

OpenAI gym can be used to make our own environment. Gym is a toolkit for developing and comparing reinforcement learning algorithms which also supports teaching agents everything from walking to playing games like Pong or Pinball. It is used for reinforcement learning research and includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms. [2]

Making our own Environment

OpenAI Gym also supports making our own environments in it and run appropriate RL algorithms on it for testing purposes.

Procedure to make your own environment

- Write your environment in an existing collection or a new collection. All collections are subfolders of '/gym/envs'.
- Import your environment into the __init__.py file of the collection. This file will be located at /gym/envs/my_collection/__init__.py. Add

  from gym.envs.my_collection.myenv import MyEnv

  to this file.
- Register your env in /gym/envs/__init__.py:

  register(
  id='MyEnv-v0',
  entry_point='gym.envs.my_collection:MyEnv',
)
- Add your environment to the scoreboard in /gym/scoreboard/__init__.py:

  add_task(
  id='MyEnv-v0',
  summary="My new environment",
  group='my_collection',
)

Taxi-v2

Taxi-v2 task was introduced in [3] to illustrate some issues in hierarchical reinforcement learning. There are 4 locations (labeled by different letters) and your job is to pick up the passenger at one location and drop him off in another. You receive +20 points for a successful dropoff, and lose 1 point for every timestep

it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions. Figure 2 shows a snapshot of this environment.

Figure 2: Taxi-v2 OpenAI Gym Environment

Warehouse-v0

The above environment was quite close to our problem statement and so it is used. The updated **taxi** environment, lets call it **warehouse-v0** environment, is shown in figure 3. The environment has 12 racks: A to L, input gate X, output gate Y and 4 robots as described earlier.

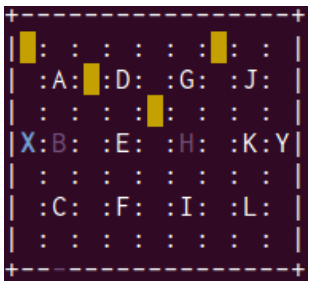

Figure 3: Warehouse Environment built using OpenAI Gym

Simplistic Version

Some assumptions were made, stated as follows:
Assumption 1: The robots do not need recharging.
Assumption 2: There is only 1 robot.
Assumption 3: Any number of items can be kept in a rack.
Assumption 4: Only one type of item can be kept in one rack. But a particular type of item can be found in multiple racks.
Assumption 5: The cost of transporting any type of item is equal, irrespective

of its size or quantity.
Assumption 6: The robot (agent) is capable of doing certain tasks like moving from locatiion A to B, picking up an object, and dropping it off at a given location. This problem statement only refers to the optimization of GET requests, and consequently STORE requests, in order to operate within a minimum cost, say C.

Figure 4 shows a simplistic version of the warehouse environment incorporating the assumptions made above.

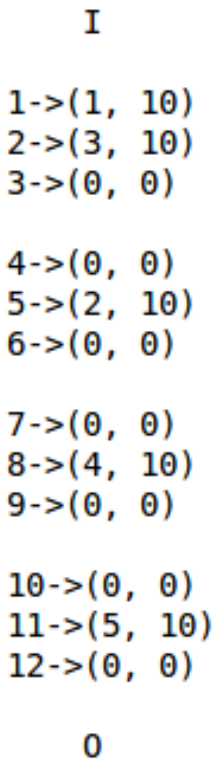

Figure 4: Simplistic Environment

The format is

rack_number −> (Item_type, Item_quantity)

Here the racks 1, 2, 3 are the closest to the input gate I and the racks 10, 11, 12 are the closest to the output O gate. (This is a 90 degrees rotated view of the warehouse when compared to figure 3).

The keys to a reinforcement learning problem are as follows:

State s

The snapshot of the warehouse at any given time is the state of the environment at that time. Figure 4 shows the state of the environment at some time t.

Rewards r

The rewards signal is attached to the GET requests. Thus, whenever an item_type is requested for, a relevant reward is given to the agent. (Note that, here r is the immediate reward being referred to and not any kind of delayed or future rewards. It is the reward received by the agent immediately after performing an action on a state of the environment).
The rewards are designed as follows:

if rack in range(1,3): reward 50
elif rack in range(4,6): reward 200
elif rack in range(7,9): reward 350
elif rack in range(10,13): reward 500

If the item is lying farther from the output gate O, a lower reward is given, whereas a higher reward is given when the item is found near the gate O. The intuition behind this is that the agent must start learning that if an item is requested for (GET request) more number of times than other items, then the agent must begin storing (on encountering that item's STORE request) those particular items at locations closer to the gate O.

Actions

The action to be performed is the rack number where an item must be stored, so as to minimize the cost incurred in getting that item in future. So the action space ranges from 1 to 12 (the available rack numbers).

Reinforcement Learning Strategies

Random Strategy

One approach for exploring the state space is to generate actions randomly with uniform probability. [4] On receiving a store request, the rack number alloted to store an item is randomly generated and the item is stored in that rack. The agent, in this strategy, keeps randomly storing the items in the warehouse.

Epsilon Greedy approach

One of the challenges in reinforcement learning is balancing the use of exploration and exploitation which has a great bias on learning time and the quality

of learned policies. Too much explorartion can lead to increased time to learn a policy because the selected actions might yield negative rewards from the environment and too much exploitation is not useful becuase the learned policy may or may not be optimal. This is popularly known as *the dilemma of exploration and exploitation.* [1]

One of the widely used method for balancing exploration and exploitation is the ϵ-greedy method. At each timestep, the agent selects an action with a fixed probability instead of greedily selecting an action suggested by another RL algorithm (for instance, Q-Learning).

$$\pi(s) = \begin{cases} \text{random action } a_r, & \text{if } \eta \geq \epsilon \\ \text{action from other strategy } a_o, & \text{otherwise} \end{cases}$$

where η is a uniform random number drawn at each timestep ranging from 0 to 1.
The other strategy used here is a function defined manually. It calculates and then uses the probability distribution of the items from a given list of requests that has been executed till now. Using this distribution, the item which has the highest probability of being requested in the near future, the action for storing that particular item is set as 'nearest to the gate O'. Other RL algorithms can also be used for example Q Learning, which is described in the next subsection.

Q Learning

Q-learning is a simple way for agents to learn how to act optimally in controlled Markovian domains. It amounts to an incremental method for dynamic programming which imposes limited computational demands. It works by successively improving its evaluations of the quality of particular actions at particular states. [5]

We can express Q(s, a), recursively, as follows:

$$\text{Q(s, a)} = \text{r} + \gamma \; max_{a'}(\text{Q(s', a')})$$

where, s' and a' are the next state and action. This equation, known as **the Bellman equation**, shows that the maximum future reward is the immediate reward the agent received for entering the current state s plus the maximum future reward for the next state s'.

We have used the combination of ϵ-greedy and Q-learning. The exploration is handled using a random strategy and the exploitation is done using Q learning. The parameters used are as follows:

$$\epsilon = 0.6$$
$$\gamma = 0.8$$
$$\text{number of episodes} = 10000$$
$$\text{probability of item\_type 5 is kept high (intentionally)}$$

Results

To evaluate the algorithm, cumulative rewards were recorded for all the episodes. Figure 5 shows the plot obtained for number of episodes v/s the cumulative reward earned till then.

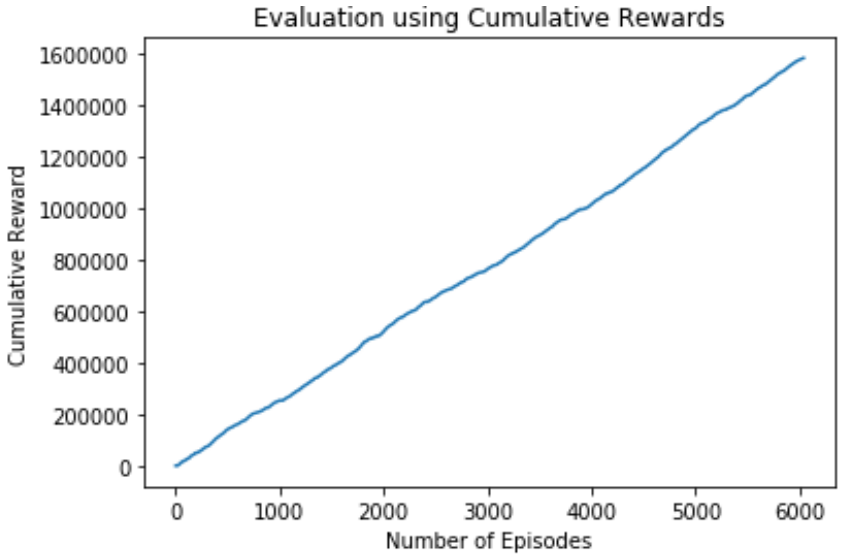

Figure 5: Result

Note here that the x-axis shows only till 6000 episodes, because the model is learnt only on GET requests and not on STORE requests. The input file for requests has 6000 episodes for GET and 4000 for STORE.

References

[1] Montague, P. Read. "Reinforcement Learning: An Introduction, by Sutton, RS and Barto, AG." Trends in Cognitive Sciences 3.9 (1999): 360.

[2] Brockman, Greg, et al. "Openai gym." arXiv preprint arXiv:1606.01540 (2016).

[3] T Erez, Y Tassa, E Todorov, "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition", 2011.

[4] McFarlane, Roger. "A Survey of Exploration Strategies in Reinforcement Learning."

[5] Watkins, Christopher JCH, and Peter Dayan. "Q-learning." Machine learning 8.3-4 (1992): 279-292.