Control of 2-degree of freedom robot using Advantage-Actor-Critic method

Nazerke Sandibay
Department of Robotics and Mechatronics
Nazarbayev University
nazerke.sandibay@nu.edu.kz

Abstract

The project work aims to find the optimal trajectory of 2-degree of freedom robot in a space with obstacles using Advantage-Actor-Critic Algorithm. The learning environment of the robot was constructed and the performance of a reinforcement learning algorithm concluded to be safe and optima for a robot but too cautious.

1 Problem formulation

The purpose of the project is to find the best trajectory between two positions of 2 degrees of freedom robot similar to a SCARA robot in the environment with obstacles. It can be done using motion planning algorithms such as A* and Reinforcement learning techniques.

The latter may be faster and universal compared to the former. Therefore, Advantage-Actor-Critic Reinforcement learning algorithm was written and tested ina simple environment with obstacles with a constant position. For the sake of convenience Open-Ai gym environment of the robot was created to work with RL baselines. Later it can be applied to a changing environment.

Figure 1. SCARA robot.

1.1 Task description

Our goal is to pick up an object in space at a specified position and place it to another position using an end-effector electromagnet that can be activated or deactivated at will. The robot's vertical movement can occur at any time without affecting obstacle collisions: as a result, the vertical movement is planned independently and we only need to determine the motion of the manipulator in the horizontal plane.[1] The robot can thus be modeled as a simple 2R planar manipulator. The robot base (i.e., where link 1 is fixed to the ground) is at (x,y) = (0,0). The links have lengths `1 = 0.5 m and `2 = 0.4 m, respectively. The first obstacle to be avoided is a wall, which runs parallel to the x-axis, keeping a distance of 0 m from it. Also, there are two other obstacles: these have a fixed position and have been conservatively represented by two circles, both with radius B = 0.2 m,and with center at (xc1,yc1) = (−0.6,0.7) mand (xc2,yc2) = (0.6,0.7) m, respectively.

The thickness of the links can be neglected, as the sizes of all obstacles have been already augmented to account for the robot link thickness as well. The angular motion of link 1 is only limited by the presence of the wall (so no additional constraints have to be inserted), while link 2 can only move within a range of ±90◦ with respect to the configuration in which it is perfectly aligned with link 1 (i.e., θ2 ∈ [−π/2,π/2]). Our task is to plan a motion from any given initial configuration (where the object is picked) to any final configuration (where the object is placed), chosen in the free space, avoiding any collision during the robot motion.

2 Background

2.1 Heuristic function

Reward functions play a crucial role in reinforcement learning. In my case reward was chosen to be proportional to negative of heuristic. Heuristic function approximates the distance between two objects. It was taken as 20 plus negative heuristic. It means that the reward of the states that are close to the goal is higher. For example, the heuristic of the far element is 15, while the heuristic of the closer element is 10. Consequently, their reward will be 5 (or 20-15=5) and 10(or 20-10=10). A reward of the closest element is higher, therefore the algorithm will try to move closer to the goal to maximize reward.

3 Environment

Two degrees of freedom of the robot corresponds to two agents. So, the environment is multiagent with agents that depend on each other. Therefore, outputs of the algorithm should be two angles (theta 1 and theta 2) corresponding to the angles of the arms with respect to the neutral axis.

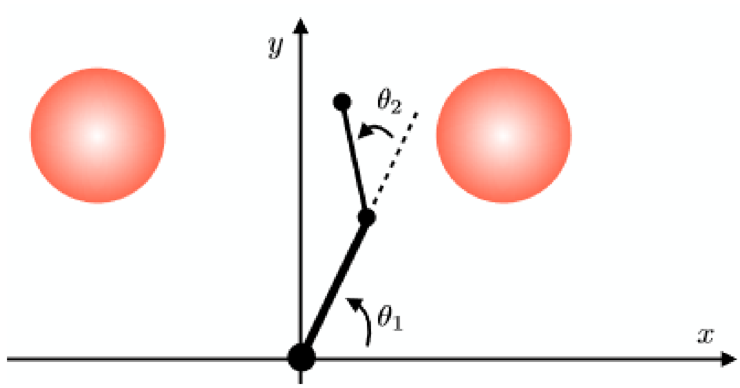

Figure 2. 2DOF robot (view from top)

3.1 Free space and obstacle space

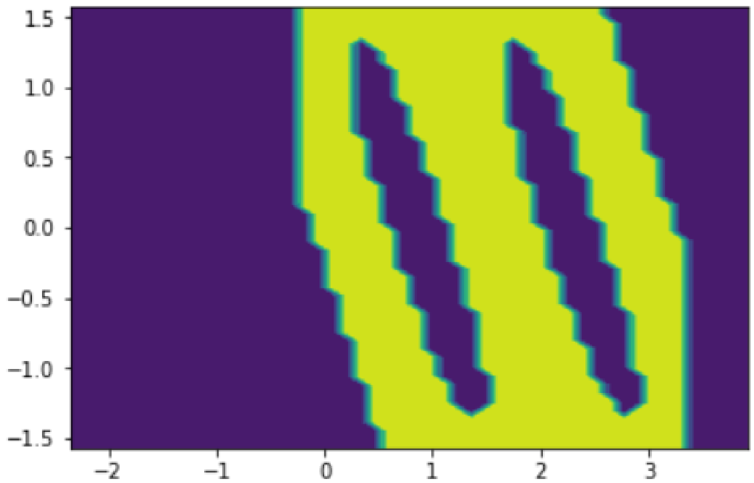

Figure 3. Free&Obstacle space of the environment

First of all, we have to represent the robot configuration space, the configuration being q = (θ1,θ2). A grid of points has to be defined on both angles in a range of 2π. It is better to choose the intervals for the two angles such that the free space is connected: for example, rather than representing the range of both angles from 0 to 2π, one could do it between−3π/4 and 5π/4. From a visual inspection, we notice that link 1 can collide with the wall, but not with any of the circular obstacles: as a consequence, there is no need to define spheres around link 1.

3.2 Create a gym environment

Gym environment with properties and functions similar to the OpenAI gym environment was created.

```
class GridEnvironment(gym.Env):
    metadata = { 'render.modes': [] }

    def __init__(self,D,x,y, agent, goal):
        self.x=x
        self.y=y
        self.bool=D
        self.low = np.array([-3 * math.pi / 4, -math.pi / 2])
        self.high = np.array([5 * math.pi / 4, math.pi / 2])

        self.observation_space = spaces.Box(self.low, self.hig
h, dtype=np.float32)
        self.action_space = spaces.Discrete(4)
        self.max_timesteps = 25001
        self.agent=self.cord(agent)
        self.goal=self.cord(goal)

    def reset(self):
        self.timestep = 0
        self.agent_pos = self.agent
        self.goal_pos = self.goal
        self.state = np.zeros((50,50))
```

```
        self.state[tuple(self.agent_pos)] = 1
        self.state[tuple(self.goal_pos)] = 0.5

        return self.agent_pos

    def cord(self,pos):
        x=getcordinates(self.x, self.y, pos)
        return np.array(x)
    def obs(self):
        observation = self.state.flatten()
        return observation

    def step(self, action):
        # 0 - down
        # 1 - up
        # 2 - right
        # 3 - left

        s=False

        if action == 0:
            if D[self.agent_pos[1],self.agent_pos[0]+1] and self.agent_pos[0]<48:

                         new=[self.agent_pos[0].copy() + 1,self.agent_pos[1]]

                         s=True

        if action == 1:
            if D[self.agent_pos[1],self.agent_pos[0]-1] and self.agent_pos[0]>1:

                         new=[self.agent_pos[0].copy() - 1,self.agent_pos[1]]

                         s=True

        if action == 2:
            if D[self.agent_pos[1]+1,self.agent_pos[0]] and self.agent_pos[1]<48:

                         new= [self.agent_pos[0],self.agent_pos[1].copy() + 1]
                         s=True

        if action == 3:
            if D[self.agent_pos[1]-1,self.agent_pos[0]] and self.agent_pos[1]>1:

                         new=[self.agent_pos[0],self.agent_pos[1].copy() - 1]
                         s=True
        if s:
```

```
            new=np.array(new)
        else:
            new=self.agent_pos
        r =  heuristic(new[0],new[1],self.goal_pos[0],self.goal_pos[1])
        if r<1:
            if r==0:
                reward=np.array([1000.0])
            else:
              reward=np.array([500.0+1/r])
        else:
              reward=np.array([20.0-r])

        self.agent_pos=new.copy()
        done = True if self.timestep >= self.max_timesteps else False


        self.timestep += 1

        info = {}
        if not s:
          reward= np.array([0.0])
        self.state[tuple(new)] = 1

        return new, reward, done, info

    def render(self):
        plt.imshow(self.state)
    def pos(self,cor):
        self.agent_pos=cor
    def agent(self):
        return  self.agent_pos
```

3.2.1 Calculation of reward

Choice of reward function is very important since the performance of the algorithm will depend on it.

The reward function is equal to twenty subtracted to the approximated distance. As the agent gets closer to the goal it starts to increase.

In this case, when the agent reaches the goal(distance is 0) it gets 1000 reward. If the distance is smaller than 1, it can get a reward between 500 and 500+1/(closest distance). Performance still can be improved by changing the reward.

3.2.2 Step function

Our environment contains obstacles. Therefore, an agent has to make sure that the next state is “safe”. It is done via the configuration space matrix described above. It makes a decision based on the value of free space matrix on a given position.

4 Algorithms

4.1 Actor-Critic method

A synchronous, deterministic variant of Asynchronous Advantage Actor-Critic (A3C) algorithm from the library of Stable Baseline was used.

4.2 Results

Trajectory of the agent moving from start point = ([0, 0]) to goal point = ([0, 1 rad]) . There is no obstacle between two points. Therefore, the trajectory looks like a straight line.

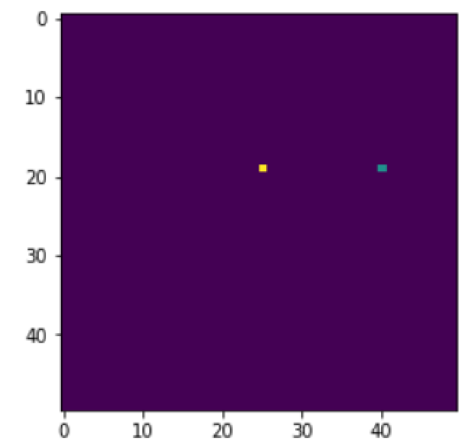

Figure 4. Grid before learning. Agent position and goal position.

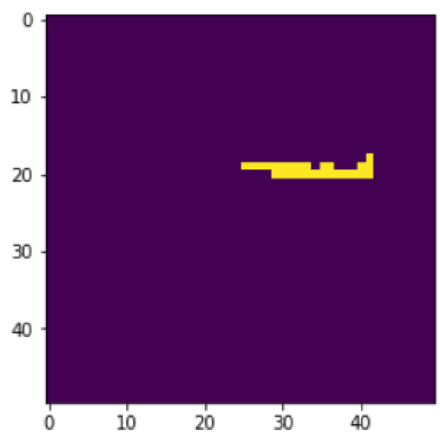

Figure 5. Grid after learning. Trajectory

Trajectory of the agent moving from start point = ([2, 0.5]) to goal point = ([3, -0.5]) was estimated. There is an obstacle between two points. Therefore, the trajectory is more complex. Learning was done for the environment with one obstacle with the following configuration space

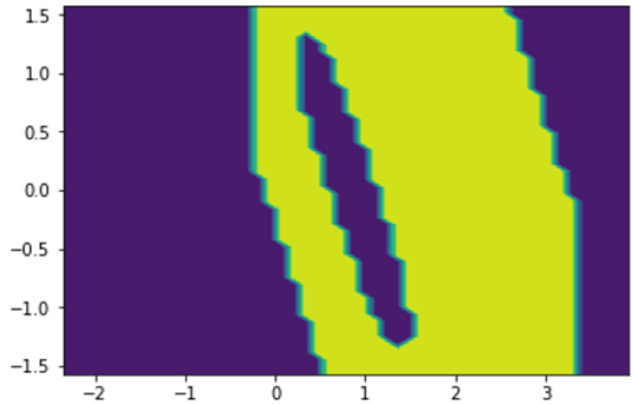

Figure 6. New free&obstacle space

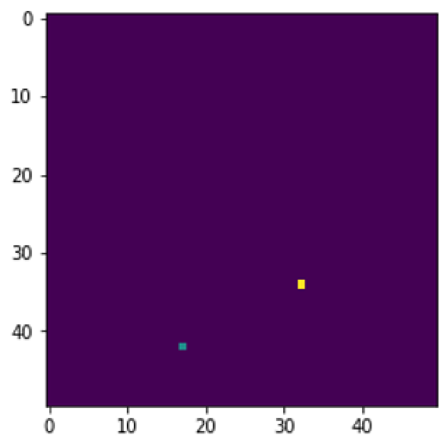

Figure 7. Grid before learning.

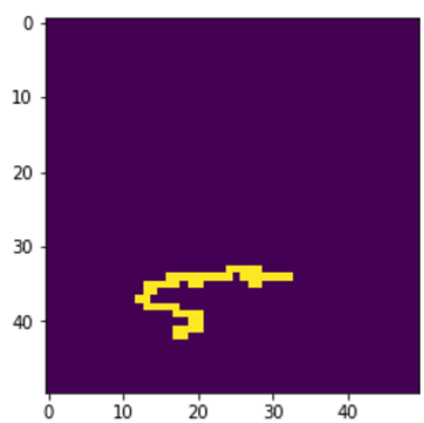

Figure 8. Grid after learning.Trajectory

4.3 Visualization

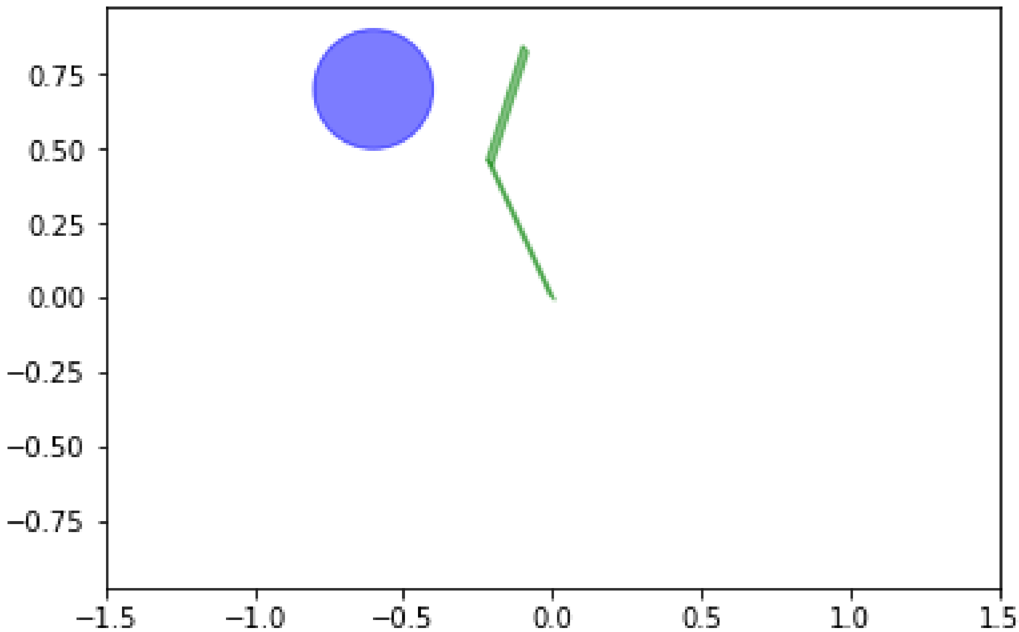

Figure 9.The initial position of a robot

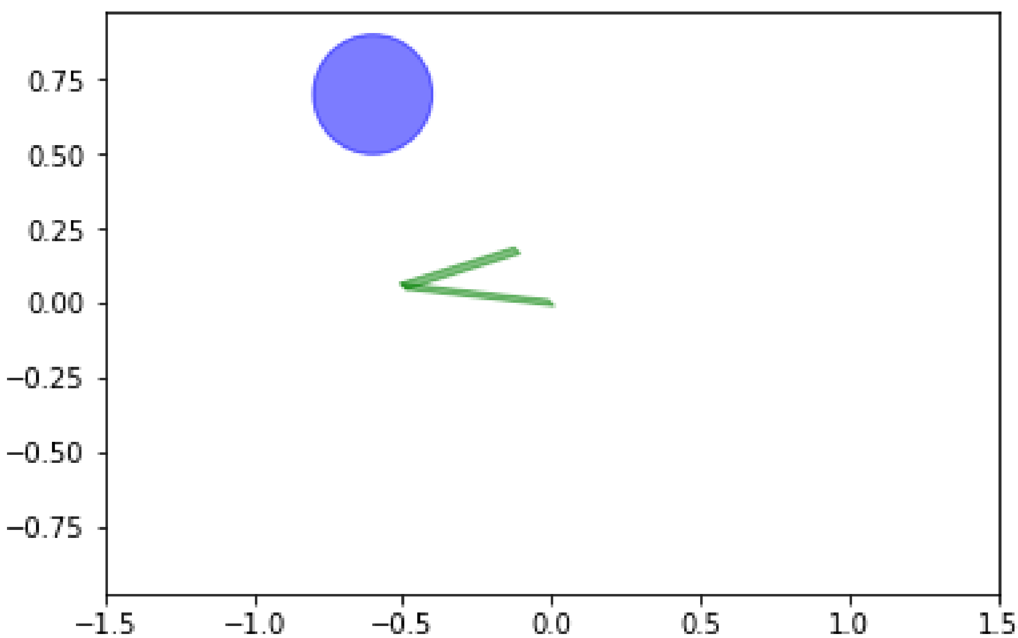

Figure 10. Final Position of a robot

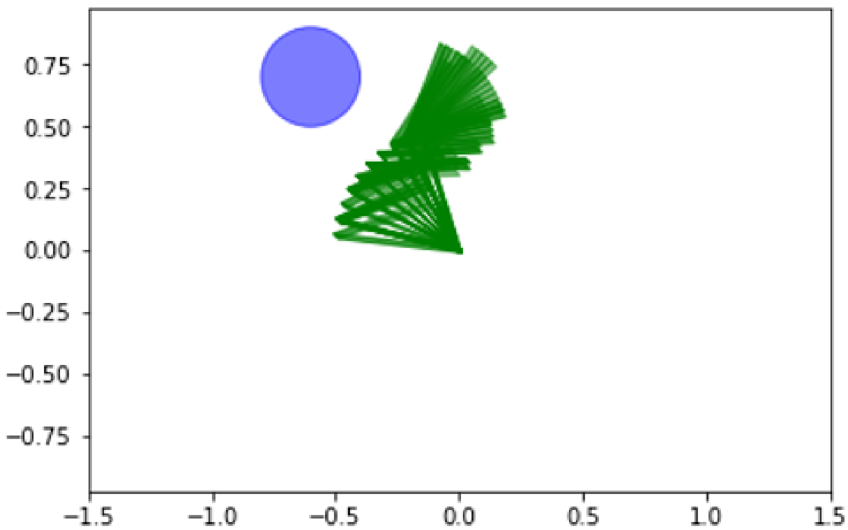

Figure 11. Whole trajectory

5 Improvement

The algorithm does not always generate an optimal trajectory. It might be caused by a complex environment or reward values. However, the agent often tries to move toward the goal. Performance of the algorithm can be improved by choosing another reward function(such as another type of heuristic function)

References

[1] https://www.fanuc.eu/de/en/robots/robot-filter-page/scara-series/selection-support

[2] Volodymyr Mnih et.al (2016) *Asynchronous Methods for Deep Reinforcement Learning* https://stable-baselines.readthedocs.io/en/master/modules/a2c.html