ECE-GY 9243 / ME-GY 7973

Optimal and Learning Control for Robotics

Project Report

Simulation of 2-D Robot Arm

Yining Wen

yw3997

yw3997@nyu.edu

INTRODUCTION

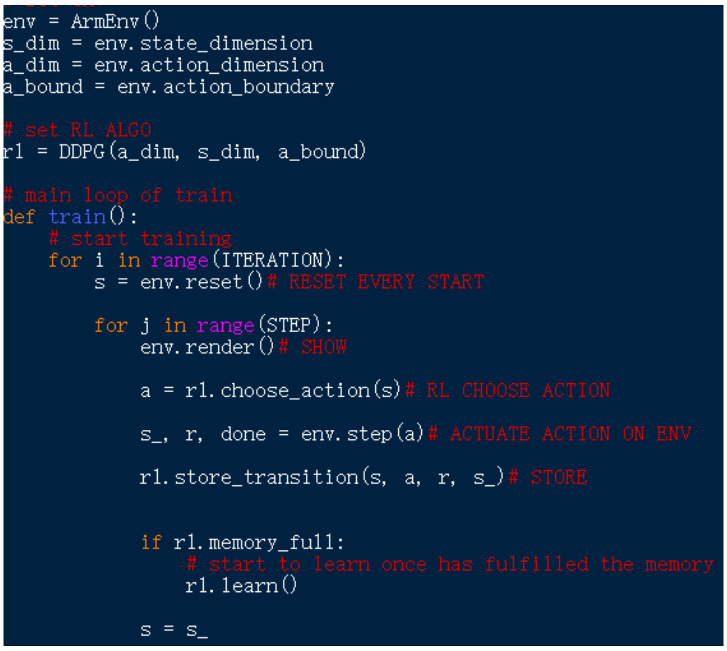
Nowadays, more and more robotic machines appear in human's daily life. No matter in the film or in the lab experiment, robotic arm is one of the hottest research area. Robotic arm consisted by arm and joints which generate dimensions of freedom. I start to learn robotic arm from 2-D simulation which is much easier than 3-D model, because each joint only generates 1 dimensions of freedom –angle of rotation. The purpose of the project is research using reinforcement learning method to train robotic arm to reach a certain preset point. Traditionally, we set a particular trajectory for a certain dynamic system to achieve the goal. However, it might be pretty complex for robotic arm with plenty of joints that each joint generates a bunch of dimensions to related with the whole system resulting in a mass of connected functions with numerous parameters. Therefore, we are pursuing for a method which could direct the robotic arm learn the environment and generate an optimal policy that solve the problem without any pre-settled trajectory or constraints.

It is obviously that the dynamic to be simulated is continuous in time. So I choose DDPG as the solution method. The algorithm of DDPG is actually an Actor Critic, and I use tensorflow to build a network.

Approach

1 construct main loop

Firstly, I need to settle some basic information for the simulation environment. It is obvious that the state dimension, action dimension and the step length are basic parameters for our robotic arm. Then send these basic parameters into DDPG to train. After that, build main loop. Firstly, set the iterations and steps for training. Secondly, reset the environment at every beginning of the iteration and then use render to show the animation at the beginning of every step. Thirdly, use RL method to choose an action and update the states with reward from the feedback from environment. Then store all the stuff above in the memory. (Fig. 1)



```
env = ArmEnv()
s_dim = env.state_dimension
a_dim = env.action_dimension
a_bound = env.action_boundary

# set RL ALGO
r1 = DDPG(a_dim, s_dim, a_bound)

# main loop of train
def train():
    # start training
    for i in range(ITERATION):
        s = env.reset()# RESET EVERY START

        for j in range(STEP):
            env.render()# SHOW

            a = r1.choose_action(s)# RL CHOOSE ACTION

            s_, r, done = env.step(a)# ACTUATE ACTION ON ENV

            r1.store_transition(s, a, r, s_)# STORE


            if r1.memory_full:
                # start to learn once has fulfilled the memory
                r1.learn()

            s = s_
```

Fig. 1

2 construct environment

In fact, env.py has an environment class that is not enough. This environment and the visual part are handled separately to better manage your code. So in env.py, in addition to environment, I also add class animation. The class handles the visual part separately. And environment handles the logic run. When no visualization is used, I don't need to call this animation class at all. So I need to visualize when I call the visual env.render().

In the class animation, add all the shape information to a batch, and then the entire batch will be refreshed when refreshing, saving time. v2f is the x, y information of each point, there are four points, so there are 8 numbers. Except v2f, there are other forms can find the corresponding description here. c3B represents the color of this object, each point has a color, and each color is represented by 3 primary colors, our object is solid color. So for each point, I use the same color, then *4 means that 4 vertices are blue. (Fig.2)(Fig.3)

Pyglet is a real-time refreshing animation module, so every time it is refreshed, a function is called. on_draw() is a function that pyglet needs when it refreshes. And render() is called in env.render(). Features. These two functions do not need to change when creating another environment, because this is a necessary function.

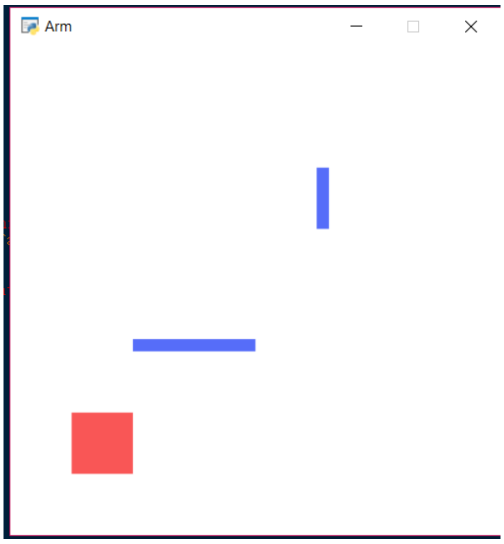
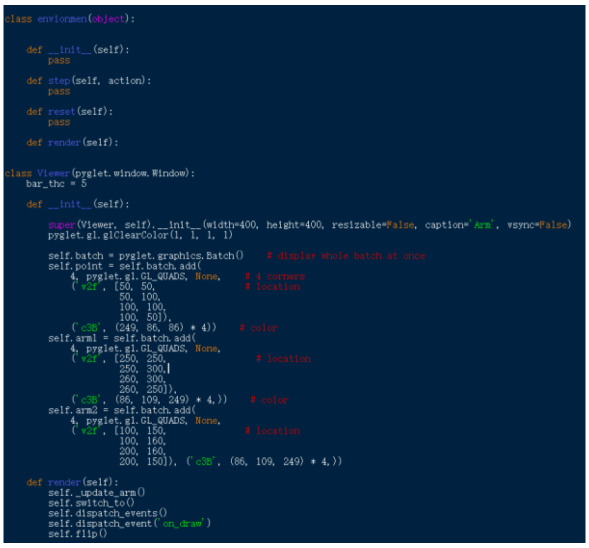


Fig. 2

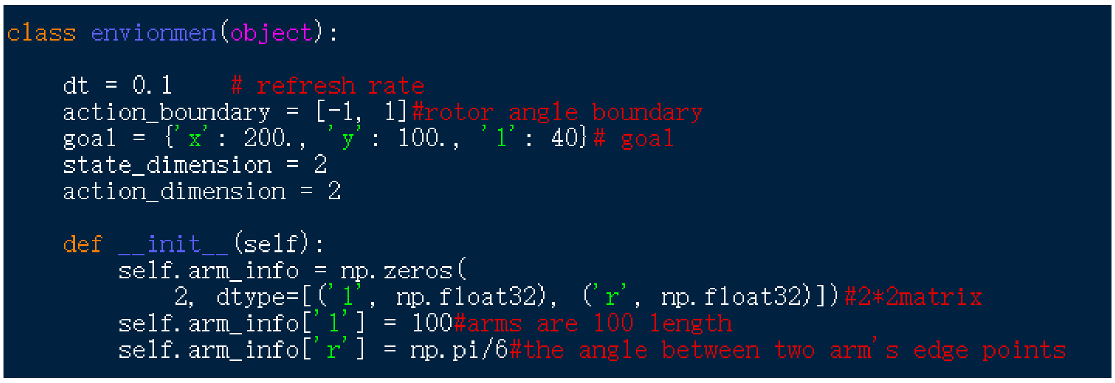


```
class envionmen(object):

    def __init__(self):
        pass

    def step(self, action):
        pass

    def reset(self):
        pass

    def render(self):

class Viewer(pyglet.window.Window):
    bar_thc = 5

    def __init__(self):

        super(Viewer, self).__init__(width=400, height=400, resizable=False, caption='Arm', vsync=False)
        pyglet.gl.glClearColor(1, 1, 1, 1)

        self.batch = pyglet.graphics.Batch()    # display whole batch at once
        self.point = self.batch.add(
            4, pyglet.gl.GL_QUADS, None,     # 4 corners
            ('v2f', [50, 50,                 # location
                     50, 100,
                     100, 100,
                     100, 50]),
            ('c3B', (249, 86, 86) * 4))    # color
        self.arml = self.batch.add(
            4, pyglet.gl.GL_QUADS, None,
            ('v2f', [250, 250,                 # location
                     250, 300,
                     260, 300,
                     260, 250]),
            ('c3B', (86, 109, 249) * 4,))    # color
        self.arm2 = self.batch.add(
            4, pyglet.gl.GL_QUADS, None,
            ('v2f', [100, 150,               # location
                     100, 160,
                     200, 160,
                     200, 150]), ('c3B', (86, 109, 249) * 4,))

    def render(self):
        self._update_arm()
        self.switch_to()
        self.dispatch_events()
        self.dispatch_event('on_draw')
        self.flip()
```

Fig. 3

After set a static state model, a dynamic state model will be built.

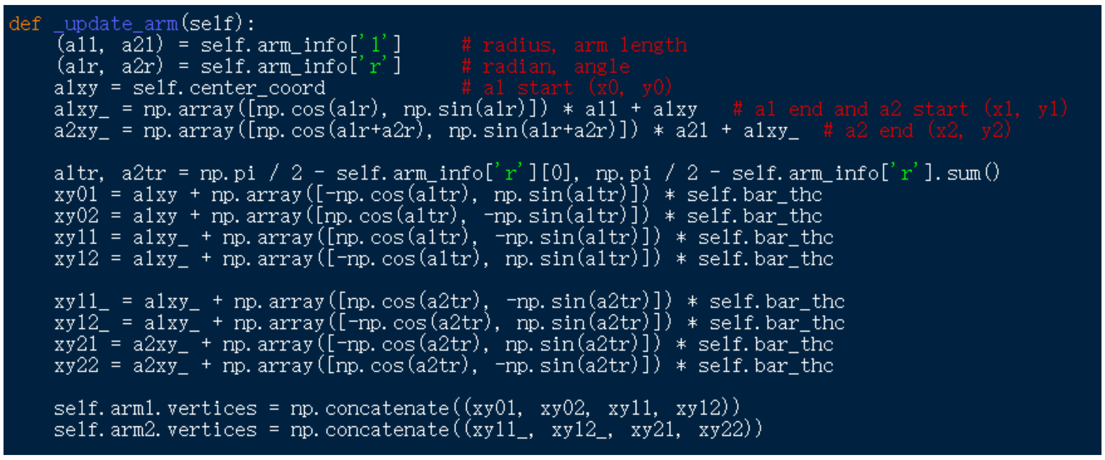
To define the dynamic arm, I need to use a table to store all the information about the arm. The information in this table is circulated, and can also pass this information to the class animation for visualization. So this table is very important. In class environment, in addition to some predefined environment properties, the table for defining the arm in __init__ is used. (Fig. 4)



```
class envionmen(object):

    dt = 0.1    # refresh rate
    action_boundary = [-1, 1]#rotor angle boundary
    goal = {'x': 200., 'y': 100., 'l': 40}# goal
    state_dimension = 2
    action_dimension = 2

    def __init__(self):
        self.arm_info = np.zeros(
            2, dtype=[('l', np.float32), ('r', np.float32)])#2*2matrix
        self.arm_info['l'] = 100#arms are 100 length
        self.arm_info['r'] = np.pi/6#the angle between two arm's edge points
```

Fig. 4

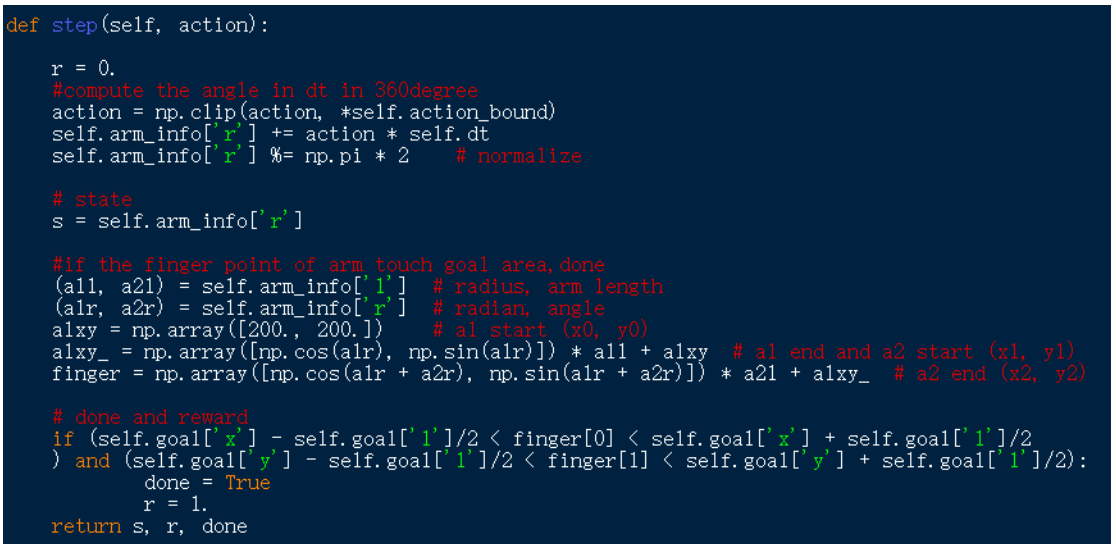
Self.arm_info is a 2x2 table, self.arm_info['l'] has two pieces of information, the length of the two arms, and self.arm_info['r'] records the rotation angle of the current two arms. After the arm is doing the action, it is only the change of the rotation angle. Then pass arm_info and goal in the Viewer and determine the initial center point of the arm. The arm will rotate around this center point. Here I redraw the goal position information according to the coordinates of the goal defined above. With the calculation of the trigonometric function, find the coordinates of the four endpoints of each arm, and then assign the coordinates to self.arm1.vertices and self.arm2.vertices. The arm can swing freely. (Fig.5)



```python
def _update_arm(self):
    (a1l, a2l) = self.arm_info['l']     # radius, arm length
    (a1r, a2r) = self.arm_info['r']     # radian, angle
    a1xy = self.center_coord            # a1 start (x0, y0)
    a1xy_ = np.array([np.cos(a1r), np.sin(a1r)]) * a1l + a1xy   # a1 end and a2 start (x1, y1)
    a2xy_ = np.array([np.cos(a1r+a2r), np.sin(a1r+a2r)]) * a2l + a1xy_  # a2 end (x2, y2)

    a1tr, a2tr = np.pi / 2 - self.arm_info['r'][0], np.pi / 2 - self.arm_info['r'].sum()
    xy01 = a1xy + np.array([-np.cos(a1tr), np.sin(a1tr)]) * self.bar_thc
    xy02 = a1xy + np.array([np.cos(a1tr), -np.sin(a1tr)]) * self.bar_thc
    xy11 = a1xy_ + np.array([np.cos(a1tr), -np.sin(a1tr)]) * self.bar_thc
    xy12 = a1xy_ + np.array([-np.cos(a1tr), np.sin(a1tr)]) * self.bar_thc

    xy11_ = a1xy_ + np.array([np.cos(a2tr), -np.sin(a2tr)]) * self.bar_thc
    xy12_ = a1xy_ + np.array([-np.cos(a2tr), np.sin(a2tr)]) * self.bar_thc
    xy21 = a2xy_ + np.array([-np.cos(a2tr), np.sin(a2tr)]) * self.bar_thc
    xy22 = a2xy_ + np.array([np.cos(a2tr), -np.sin(a2tr)]) * self.bar_thc

    self.arm1.vertices = np.concatenate((xy01, xy02, xy11, xy12))
    self.arm2.vertices = np.concatenate((xy11_, xy12_, xy21, xy22))
```

Fig.5

After defining the arm's update rule, continue to define the arm angle information update principle when we define the update rule. This update principle is reflected in class environment. Separated the arm movement from the arm drawing. So the part of the motion is written in class environment. the arm's self.arm_info information, plus an action passed in the step() function to update the arm's info information at the next moment.



```python
def step(self, action):

    r = 0.
    #compute the angle in dt in 360degree
    action = np.clip(action, *self.action_bound)
    self.arm_info['r'] += action * self.dt
    self.arm_info['r'] %= np.pi * 2    # normalize

    # state
    s = self.arm_info['r']

    #if the finger point of arm touch goal area,done
    (a1l, a2l) = self.arm_info['l']  # radius, arm length
    (a1r, a2r) = self.arm_info['r']  # radian, angle
    a1xy = np.array([200., 200.])    # a1 start (x0, y0)
    a1xy_ = np.array([np.cos(a1r), np.sin(a1r)]) * a1l + a1xy  # a1 end and a2 start (x1, y1)
    finger = np.array([np.cos(a1r + a2r), np.sin(a1r + a2r)]) * a2l + a1xy_  # a2 end (x2, y2)

    # done and reward
    if (self.goal['x'] - self.goal['l']/2 < finger[0] < self.goal['x'] + self.goal['l']/2
    ) and (self.goal['y'] - self.goal['l']/2 < finger[1] < self.goal['y'] + self.goal['l']/2):
            done = True
            r = 1.
    return s, r, done
```

Fig. 6

The environment generates some random actions to test the current environmental feasibility.

3 construct reinforcement learning

Using tensorflow to build neural network DDPG. DDPG using the Actor Critic structure, but the output is not the probability of behavior, but the specific behavior, used for the prediction of continuous action. DDPG combines the previously successful DQN structure to improve the stability and convergence of Actor Critic

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

Regarding the Actor part, his parameter update also involves Critic. The above is an update on the Actor parameter. The first part of the grad[Q] is from Critic. This is to say: How to move the Actor's action this time, in order to get a larger Q, the second half of grad[u] is from the Actor. This is to say: How does the Actor modify its own parameters, making the Actor more likely to do this action? So the two are together. Actor wants to modify the action parameters in a direction more likely to get a large Q.

Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

The above is about Critic's update. It draws on the DQN and Double Q learning methods. There are two neural networks that calculate Q. Q_target uses Actor to select actions according to the next state. At this time, the Actor is also an Actor_target (Has the parameters of the Actor long time ago. The Q_target obtained by this method can cut the correlation and improve the convergence like DQN.
use tensorboard to display the calculation flow of the entire DDPG(Fig. 7)

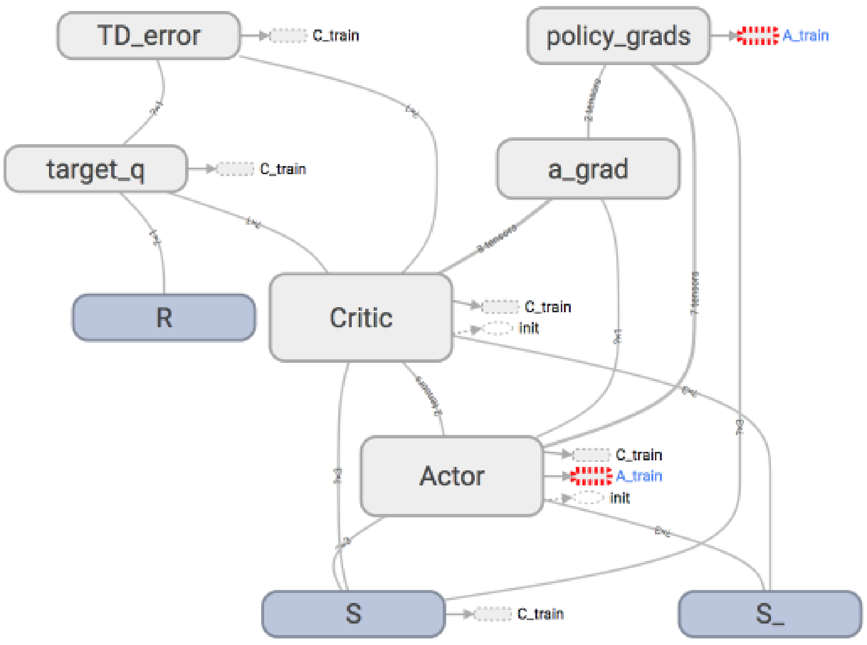


Fig. 7

Result

In the animation we can learn that the arms are learnt methods to touch a certain goal point with many different trajectories.