Balancing an Inverted Pendulum

Kajal Damji Gada

ENPM808F Robot Learning
Final Project
12th December 2016

Contents

Abstract **3**

1 Introduction **3**

2 Related Work **4**

3 Approach **5**
3.1 Q-learning: Introduction 5
3.2 Q-learning: Exploration 5
3.3 Q-learning: Formula 5

4 Implementation **6**

5 Results **7**

6 Analysis **8**

7 Conclusion **8**

8 Future Work **8**

References **9**

Appendix **10**

Abstract

1 Introduction

An inverted pendulum is a touchstone which every Robotic student touches once [1]. Beginning from stabilization of unstable open-loop system to real-world application of Segway, it is a benchmark in Control Theory and Robotics. It is also a good application to aid in learning of any new algorithm, which in this scenario, is Q-learning. Thus, the goal of the project is to understand the working of Q-learning, a machine learning algorithm, by implementation for an inverted pendulum.

(a) (b)

Figure 1: (a)Segway [2] (b) Furuta Pendulum [3]

The inverted pendulum problem has many variations: Furuta Pendulum [3], Double Inverted Pendulum [4], etc. In this project, a case of inverted Pendulum on cart is considered. The system may appear simplistic in design. However, it is a non-linear system with a static stable (equilibrium) point at pending position (face-down) and dynamic equilibrium point at upright position.

This makes designing a control system for an inverted pendulum into a challenging problem. In the case of Q-learning, it is not needed to know the model. Q-learning is regarded as a model-free [5] reinforcement learning. However, it does come with its own set of challenges. One of the most important one being discretization of the model as Q-learning works for discrete system with an end-game reward.

Literature related to this project is discussed in section 2. Then in section 3, plan towards the project problem is charted out. Next in section 4 and 5, the actual implementation and results are shown. The results are analyzed in section 6 and concluded in section 7.

2 Related Work

The work by Lasse Scherffig [6] starts with explaination of Reinforcement Learning Theory and goes on to explain the difference between Supervised Learning and Reinforcement Learning. The main difference being Reinforcement Learning doesn't have a set of sample actions to be taken, it is infact learn by exploring and assessing the rewards.

The paper then discusses the Inverted Pendulum model, followed by the work done. The paper address 2 problems: balancing and full control. Balancing is about maintaining balance when in face-up position and Full control is about getting to face-up position from any position including face-down position. While the first problem is solved using Q-learning, the second part uses Artificial Neural Network (ANN) as the number of states are too large.

In a second paper, the author disusses use of resource-allocation network with Q-learning [7]. The paper starts with a discussion on use of supervised learning and memorization for balancing an inverted pendulum. The method essentially memorize each move using Gaussian signal. Then the disuccusion moves onto how the use Q-learning to solve the problem.

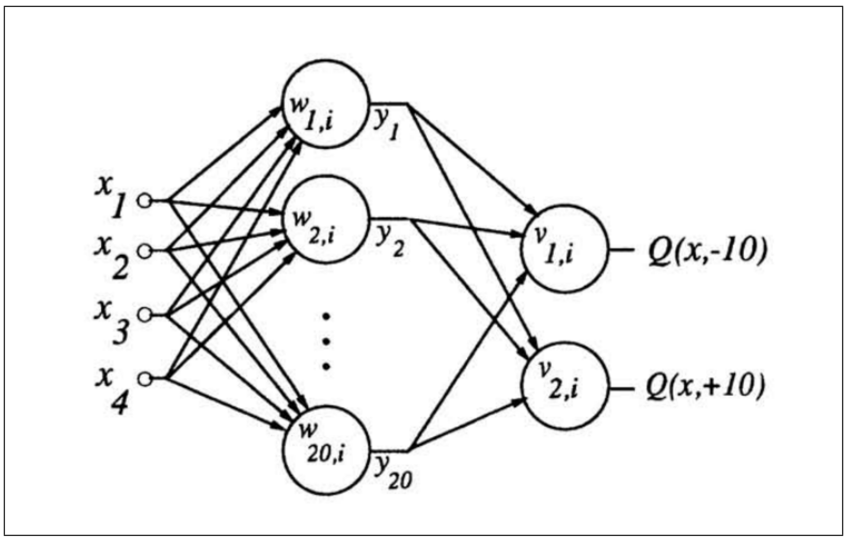

Figure 2: Q-learning network with Restart algorithm [7]

Instead of using a Q-table, the paper talks about use of Q-learning network as shown in Figure 2. The point is instead of storing each state-action pair and making it a large memorization table like supervised learning, use a network and *reallocate resources*. So everytime a new state-action is learnt, it is store at the unit that is least useful. This approach is called Restart Algorithm and gives results that work better than a combination of supervised learning and memorization.

3 Approach

3.1 Q-learning: Introduction

The task is defined as balancing an inverted pendulum on a cart in an upright position. The method chosen for this task is a machine learning algorithm: Q-learning. It is a method, that doesn't require knowledge of model for learning. It learns by experiencing the reward for taking a sequence of action [5].

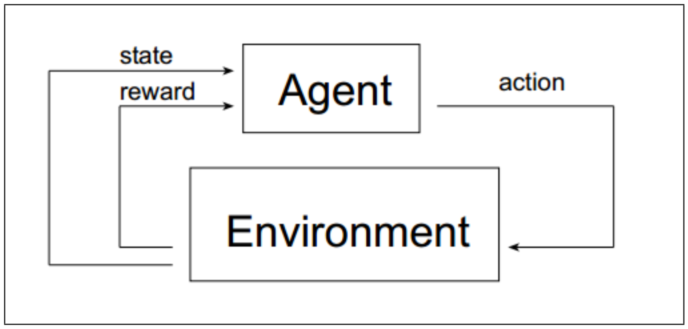

Figure 3: Interface between Agent and Environment in Q-learning [6]

In other words, the agent takes an action and observes the result in form of result from environment as shown in Figure 3. The reward is stored in a table, called Q-table, along with state. The next time, when the same state is encountered it decides to taken an action based on rewards learned last time.

3.2 Q-learning: Exploration

A good reward would lead to taking the action again. And a bad reward would lead to not taking the action again. But what if there was a better reward? Thus, there is a component of exploration. That is when deciding the next action, it takes an action not explored even when an existing action gives a good reward.

Based on the available combination of states-action pairs, the size of Q-table is decided. Also, it affects the number of iterations to be performed to obtain satisfactory results.

3.3 Q-learning: Formula

For each iteration, the current state (s) is observed. An action is chosen for execution based on equation (1) and then the Q-table is updates based on action chosen as mention in equaion (2):

$$\pi(s) = argmax_a Q(s,a) \tag{1}$$

$$\hat{Q}(s,a) = r + \gamma max_a \hat{Q}(s^{'}, a^{'}) \tag{2}$$

where $\pi(s)$ is policy for State s; a is action chosen; r is reward for action chosen; γ is delay reward factor and $s^{'}$ is the new state after action is executed [6].

4 Implementation

The program is implemented in Python 3. The code is written to build the Q-table over multiple iterations and store the best result. The best results can then be played in an animation using Penplot command from the plot.py file.

The program (*Inverted_pendulum_q_learning*) starts with an empty Q-table. The program iterates over multiple episodes and for each episode, the current state is randomized. A policy is calculated for the current state and all actions. An action is chosen based on the calculated policy and executed.

Based on the chosen action, a new state is calculated based on the system model. Based on this new state, a reward is calculated. The reward is based on position of cart and the angle of pendulum. The reward is used to updated the Q-table. If the pendulum is dropped, a new episode begins with new random start state.

Note that an inverted pendulum is a continuous system. Thus, each state is discretized for implementation.

The states chosen are:

- Position of cart (x)
- Linear Velocity of cart $\dot{x}$
- Angle of pendulum with cart (θ)
- Angular velocity of pendulum ($\dot{\theta}$)

Next, the actions set includes:

- Move left (-1)
- Move right (1)

Thus, the cart moves with a Force of F Newton on left or right based on action chosen. The F is set to $10N$ and can be changed. Other variables include:

- Magnitude of Force on cart (F) and Gravity constant (g)
- Mass of cart (m_c), Mass of pole (m_p) and Length of Pole (l_p)
- Reward delay factor (γ)
- Exploration factor (ϵ)

5 Results

The Figure 5 shows an example of results after 1000000 iterations. As seen, the pendulum is able to maintain itself in the upright position and eventually stops when it goes at the end of cart track (beyond 2.4 units).

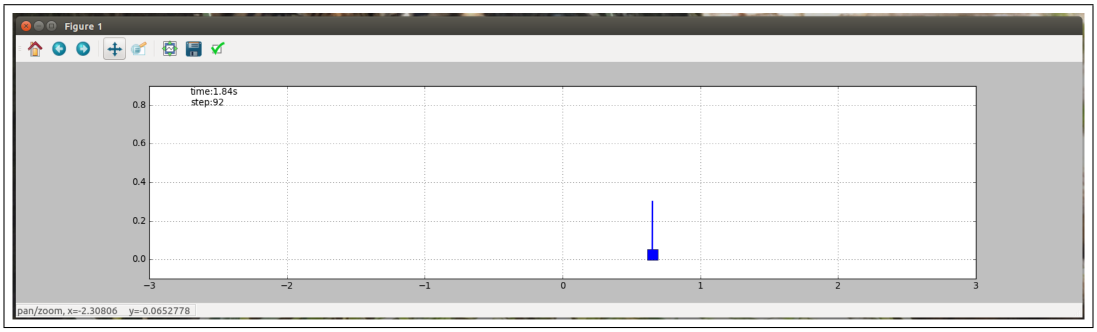

Figure 4: Snapshot of animation for Inverted Pendulum Balancing

It can be seen that as the reward is maximum at the top, it attempts to maintain the state. Note, that this system is dynamically system and thus must move continuously to be at the unstable equilibrium point.

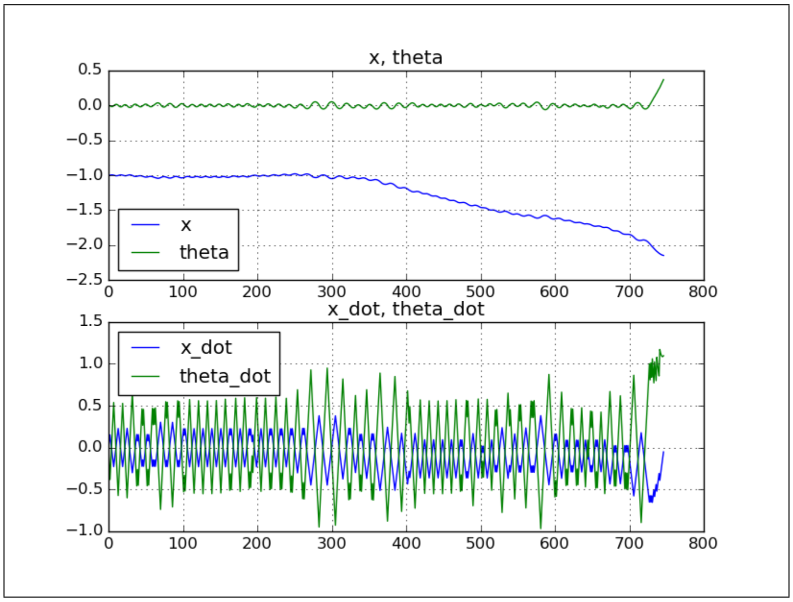

Figure 5: Results

6 Analysis

Based on results for various experimental runs, it was observed that system is able to identify the policy for maintaining the angle of pendulum between -1 to 1 degrees. To assist in learning, the initial few trials had the start state at $(x, \dot{x}, \theta, \dot{\theta}) = (0, 0, 0, 0)$. At later iterations (episodes), the system starts with initial state which is randomized. This helps learn better in fewer iterations.

To achieve better results, another method would be to create more discrete states. This also applies for the case when the algorithm wants to learn about bringing up the pendulum from face-down to face-up position. However, a Q-table would not be ideal for a high number of states. For such cases, Artifical Neural Networks (ANN) should be considered as shown in [6]

7 Conclusion

The project was concluded by implementing the Q-learning algorithm to balance an inverted pendulum in an upright position. It was also realized that it is difficult to implement a continuous system. It requires discretization of states which can prove challenging.

If the discretization is too little, the transition from one state to another is less accurate and with more states the Q-table becomes quite big. With lots of state, even more iterations are required to learn and build the Q-table. In such a case, other options such as Artifical Neural Networks should be explored.

8 Future Work

This project focused on balancing the pendulum, a natural extension would be to get the pendulum to come into an upright position from a face down position.

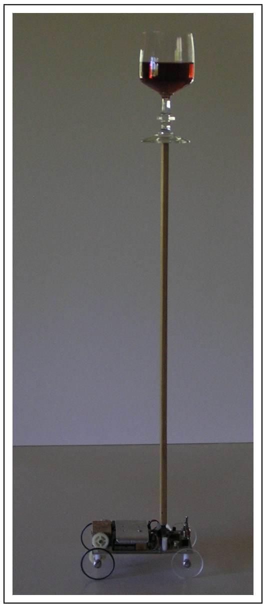

Figure 6: Balancing a glass of Wine

Though an interesting future work would be to learn to balance the inverted pendulum when moving in a particular direction. This could be seen applicable for a scenario when a mobile Robot would bring you a glass of wine while balancing it at the end of stick (an inverted pendulum) as shown in Figure 6.

References

[1] Boubaker Olfa, "The Inverted Pendulum: A fundamental Benchmark in Control Theory and Robotics", Education and e-learning Innovations (ICEELI), 2012.

[2] W. Younis, M. Abdelati, Design and implementation of an experimental segway model, AIP Conference Proceedings, vol. 1107, pp. 350-354, 2009

[3] J. . Acosta, Furuta's pendulum: A conservative nonlinear model for theory and practice, Mathematical Problems in Engineering, 2010.

[4] Henmi Tomohiro, Deng Mingcong, Inoue Akira, Ueki Nobuyuki and Hirashima Yoichi, "Swing-up Control of a Serial Double Inverted Pendulum", American Control Conference, 2004

[5] Watkins Christopher J.C.H, "Technical Note: Q-learning", Machine Learning, pp. 279-292, 1992.

[6] Scherffig Lasse, "Reinforcement Learning in Motor Control"

[7] Anderson, Charles W., "Q-learning with Hidden-Unit Restarting"

Appendix

Read Me

The program is coded in Python 3. To run the program:

python3 inverted_pendulum_q_learning.py

Ensure that both codes: (1) inverted_pendulum_q_learning.py and (2) plot.py are in the same folder. First compile and then run the code. In Ubuntu:

chmod +x inverted_pendulum_q_learning.py
chmod +x plot.py

To change values of parameters such as γ, ϵ, etc. change the value at start of function. To change display setting, use command:

Penplot(best_states, anime=True, fig=True)

where anime=True is for animation and fig=True is for graph.

Main Program (In python3)

```
#!/usr/bin/env python

import numpy as np
from plot import Penplot
import random
from math import degrees, sin, cos

# ——————————————————————————————#
#                    CONSTANT VALUES                          #
# ——————————————————————————————#

mass_pole = 0.1
mass_cart = 0.5
mass_total = mass_pole + mass_cart

length_pole = 0.3

force_magnitude = 2
constant_gravity = 9.8

tau = 0.02
alpha = 0.5
gamma = 0.5

global epsilon
epsilon = 0.2

# ——————————————————————————————#
#                    FUNCTIONS                                           #
# ——————————————————————————————#

def calculate_index(current_state):

        if current_state[0] < -0.8:
                x = 0
        elif current_state[0] < 0.8:
                x = 1
        else:
                x = 2

        if current_state[1] < -0.5:
                x_dot = 0
        elif current_state[1] < 0.5:
                x_dot = 1
        else:
                x_dot = 2

        if degrees(current_state[2]) < -12.0:
                theta = 0
        elif degrees(current_state[2]) < -6.0:
                theta = 1
        elif degrees(current_state[2]) < -1.0:
                theta = 2
```

```
        elif degrees(current_state[2]) < 0.0:
                theta = 3
        elif degrees(current_state[2]) < 1.0:
                theta = 4
        elif degrees(current_state[2]) < 6.0:
                theta = 5
        elif degrees(current_state[2]) < 12.0:
                theta = 6
        else:
                theta = 7

        if degrees(current_state[3]) < -50.0:
                theta_dot = 0
        elif degrees(current_state[3]) < -25.0:
                theta_dot = 1
        elif degrees(current_state[3]) < 25.0:
                theta_dot = 2
        elif degrees(current_state[3]) < 50.0:
                theta_dot = 3
        else:
                theta_dot = 4

        return x, x_dot, theta, theta_dot

def calculate_prob(current_state, Q_table):

        policy = []

        x, x_dot, theta, theta_dot = calculate_index(current_state)

        value = [Q_table[action, x, x_dot, theta, theta_dot] for action in
            range(2)]

        for action__ in value:
            if action__ == max(value):
                policy.append(1.0 - epsilon + epsilon / 2)
            else:
                policy.append(epsilon / 2)

        if sum(policy) == 1.0:
            return policy
        else:
                policy = [0.5, 0.5]
                return policy

def choose_action(policy):

        prob_num = random.randrange(0,100)/100.0

        if prob_num <= policy[0]:
                action_choosen = 0
        else:
                action_choosen = 1

```

```
        return action_choosen

def update_state(current_state, action_choosen):

        x_cur, x_dot_cur, theta_cur, theta_dot_cur = current_state

        if action_choosen == 0:                         # action 0 is left
                force_value = - force_magnitude
        else:                                           # action 1 is right
                force_value = force_magnitude

        temp = (force_value + (mass_pole*length_pole) * theta_dot_cur**2 * sin(theta_cur)) / mass_total

        theta_acc = (constant_gravity * sin(theta_cur) - cos(theta_cur) * temp) / \
                        (length_pole * ((4.0/3.0) - mass_pole * cos(theta_cur)**2 / mass_total))

        x_acc = temp - (mass_pole*length_pole) * theta_acc * cos(theta_cur) / mass_total

        x_new = x_cur + (tau * x_dot_cur)
        x_dot_new = x_dot_cur + (tau * x_acc)
        theta_new = theta_cur + (tau * theta_dot_cur)
        theta_dot_new = theta_dot_cur + (tau * theta_acc)

        return x_new, x_dot_new, theta_new, theta_dot_new

def update_Qtable(current_state, action_choosen, new_state, reward, Q_table):

        x, x_dot, theta, theta_dot = calculate_index(new_state)
        Q_max = max(Q_table[0, x, x_dot, theta, theta_dot], Q_table[1, x, x_dot, theta, theta_dot])

        x, x_dot, theta, theta_dot = calculate_index(current_state)
        Q_cur = Q_table[action_choosen, x, x_dot, theta, theta_dot]

        Q_table[action_choosen, x, x_dot, theta, theta_dot] = Q_cur + alpha * (reward + (gamma*Q_max) - Q_cur)

        return Q_table

def take_action(current_state, Q_table):

        policy = calculate_prob(current_state, Q_table)
        action_choosen = choose_action(policy)
        new_state = update_state(current_state, action_choosen)

        reward = 0

        if abs(new_state[0]) < 2.4:
```

```
                    if abs(degrees(new_state[2])) < 1.0:
                            reward = 10
                    elif abs(degrees(new_state[2])) < 3.0:
                            reward = 5
                    elif abs(degrees(new_state[2])) < 6.0:
                            reward = 2
                    elif abs(degrees(new_state[2])) < 20.0:
                            reward = 1

            Q_table = update_Qtable(current_state, action_choosen, new_state,
                reward, Q_table)

            return reward, new_state, Q_table

# ------------------------------------------------------------------------#
#                   MAIN PROGRAM                                          #
# ------------------------------------------------------------------------#

Q_table = np.zeros([2, 3, 3, 8, 5])                             # action (2) *
    state_x (3) * state_x_dot (3) * state_theta (6) * state_theta_dot (3)

max_steps = 0
best_states = []

max_episodes = 1000000
# max_episodes = 10000

for episode in range(1,max_episodes+1):

        states = []

        if episode < 10000:
                current_state = (0,0,random.randrange(-1,1),0)
                                                    # start state = 0
        elif episode < 20000:
                current_state = (0.1*random.randrange(-5,5),0,random.randrange
                    (-3,3),0)
        elif episode < 30000:
                current_state = (0.1*random.randrange(-8,8),0,random.randrange
                    (-5,5),0)
        elif episode < 50000:
                current_state = (0.1*random.randrange(-15,15),0,random.
                    randrange(-12,12),0)
        else:
                current_state = (0.1*random.randrange(-20,20),0,random.
                    randrange(-15,15),0)

        states.append(current_state)

        for step in range(1,1000):

                reward, new_state, Q_table = take_action(current_state,
                    Q_table)
                current_state = new_state
```

```
        states.append(current_state)

        if reward < 1:                                        # Pendulum dropped

            if step > max_steps:
                best_states = states
                max_steps = step

            if (episode % 10000) == 0:
                print ('After',episode,'episode')
                print ('Max steps: ',max_steps)
                print ('--------------------------')

                # Penplot(best_states, anime=True, fig=False)

                epsilon -= 0.002

                if epsilon < 0:
                    epsilon = 0

            break

Penplot(best_states, anime=True, fig=True)

# ------------------------------------------------------------ #
```

Program for animation (In python3)

```
#!/usr/bin/env python

import math
import matplotlib
matplotlib.use('Qt5Agg')
import matplotlib.pyplot as plt
# import matplotlib.pyplot as plt
import matplotlib.animation as animation

class Penplot(object):
    def __init__(self, states, anime=False, fig=False):
        self.anime = anime
        self.fig = fig
        self.x = [state[0] for state in states]
        self.x_dot = [state[1] for state in states]
        self.theta = [state[2] for state in states]
        self.theta_dot = [state[3] for state in states]
        self._process()

    def _plot(self, data):
        x, theta, frame = data
        self.time_text.set_text("time:%.2fs\nstep:%d" % (frame*0.02, frame))

        y = 0.05
        theta_x = x + math.sin(theta) * 0.25
        theta_y = y + math.cos(theta) * 0.25

        self.car.set_data(x, y / 2.0)
        self.line.set_data((x, theta_x), (y, theta_y))

    def _gen(self):
        for frame in range(len(self.x)):
            yield self.x[frame], self.theta[frame], frame

    def _process(self):
        if self.anime:
            fig = plt.figure(figsize=(20, 4.5))
            ax = fig.add_subplot(1, 1, 1)
            ax.set_xlim(-3.0, 3.0)
            ax.set_ylim(-0.1, 0.9)
            ax.grid()

            self.time_text = ax.text(0.05, 0.9, "", transform=ax.transAxes)
            self.car, = ax.plot([], [], "s", ms=15)
            self.line, = ax.plot([], [], "b-", lw=2)

            ani = animation.FuncAnimation(fig, self._plot, self._gen, interval
                =1, repeat_delay=3000, repeat=True)

            plt.show()

            if self.fig:
                steps = range(len(self.x))
```

```
        # plt.figure

        plt.subplot(2, 1, 1)
        plt.title("x, theta")
        plt.plot(steps, self.x, label="x")
        plt.plot(steps, self.theta, label="theta")
        plt.legend(loc="best")
        plt.grid()

        plt.subplot(2, 1, 2)
        plt.title("x_dot, theta_dot")
        plt.plot(steps, self.x_dot, label="x_dot")
        plt.plot(steps, self.theta_dot, label="theta_dot")
        plt.legend(loc="best")
        plt.grid()
        plt.show()
        plt.close()
```