Inverted Pendulum Study

Pierce Rotman
CAP 6673: Dr. Pashaie
December 13 2023

1 Introduction

An inverted pendulum is a system in which a massless rod is positioned such that a body of some mass m is above its pivot point. The problem of the inverted pendulum is to maintain a vertical position of $\theta = \pi$ (i.e. the pendulum is upright). The inverted pendulum is placed on what is known as a cart-pole apparatus, in which a cart can move along a horizontal track along the interval $[-x, x]$. The dynamics of this system are modeled in the following equation:

$$\frac{d^2\theta}{dt^2} + k_v\frac{d\theta}{dt} + \frac{g}{l}\sin\theta + \frac{1}{l}\frac{d^2x}{dt^2}\cos\theta = 0 \tag{1}$$

The angle θ is the angle from the downward vertical position. k_v is the friction coefficient.

$$\frac{d^2x}{dt^2} = \frac{1}{\tau}(\frac{dx_c}{dt} - \frac{dx}{dt}) - f_c * sign(\frac{dx}{dt}) - f_d \tag{2}$$

The control velocity is as follows:

$$\frac{dx_c}{dt} = k_U U \tag{3}$$

We will attempt to solve the problem using both Q-learning and Deep Q-Learning. These methods attempt to train the system using a reward function. There are significant problems with standard Q-learning resulting from the large size of the Q-table. These problems are largely alleviated by the Deep Q-Learning network.

2 Q Learning

2.1 Setup

This system utilizes a Markov Decision Process. We have a state $s = (\theta, \frac{d\theta}{dt}, x, \frac{dx}{dt})$ and we have three possible actions: $a \in \{-12, 0, 12\}$. Our function Q is estimated by

$$Q(s_i, a_i) \approx r_i + \gamma Q(s_{i+1}, a_{i+1}) \tag{4}$$

We can then update Q by setting

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha(r_i + \gamma \max_{a'} Q(s_{i+1}, a') - Q(s_i, a_i)) \tag{5}$$

In our we define the reward function to be:

$$r(\theta, x) = (1/2)(1 - \cos\theta) - (x/x_0)^2 \tag{6}$$

The idea behind the Q-Learning (QL) process is to approximate the nonlinear Q function by use of a Q-table. Since we are working with continuous variables (angle, position, etc.), we need to split the space into bins. As in Israilov, we use 50 bins for $\sin\theta$, $\cos\theta$, and $\frac{d\theta}{dt}$, and 10 bins for x and $\frac{dx}{dt}$.

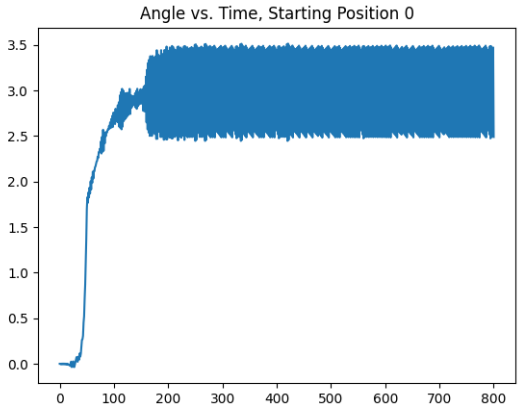
2.2 Pseudocode Algorithm

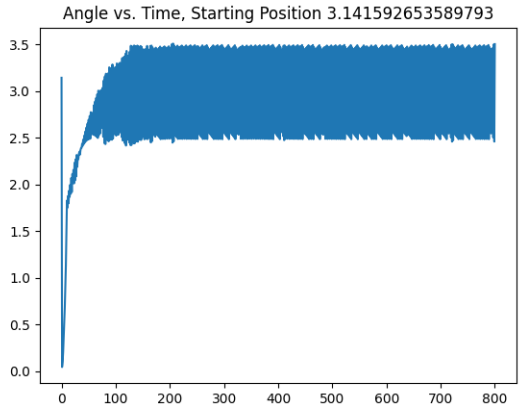
(Full code and results are in Appendix Section A)

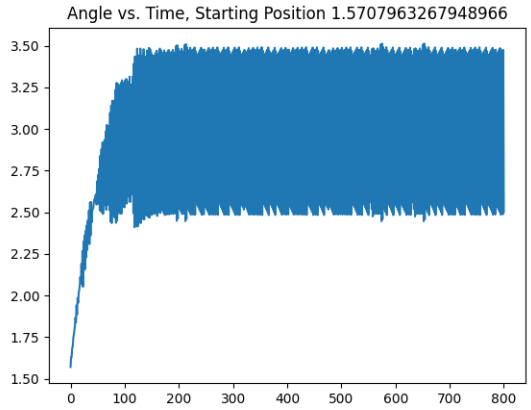
- For event:
  - Initialize state to $\theta = 0 \pm 0.05$, $\frac{d\theta}{dt}, x, \frac{dx}{dt} = 0$
  - Get binned state
  - While exit conditions (outside x-bounds, time = 40 minutes) are not met:
    - If random number ¡ epsilon: select random action from Q[state]
    - Else: Choose argmax(Q[binned state]) as action
    - Take action and update state using dynamics, get new binned state
    - Reward $= 0.5 * (1 - \cos\theta_{i+1}) - x_{i+1}^2$
    - If abs(x) greater than 0.35 or time = 40 minutes: set done = True
    - Update Q: Q[binned state, action] += 0.01 * (reward + 0.99*np.max(Q[new binned state]) - Q[binned state, action])
    - Set state = new state, binned state = new binned state

2.3 Results

Our Q-learning algorithm achieves rather disappointing results, but this is not unexpected. After running the algorithm for 10000 iterations, a limit imposed only for time constraints and lack of computing power, we test the algorithm with three starting angles: $\theta = 0, \theta = \pi, \theta = \frac{\pi}{2}$. These results use the same friction parameters and voltage [7.1V] as Israilov.







We see that the algorithm oscillates rapidly around $\theta = \pi$. The system has clearly learned to oscillate around the desired value, but it cannot effectively remain at $\theta = \pi$ because of the discretization of the variables and the lack of exploration (perhaps if we had time to train for, say 10^7 iterations, we would achieve more desirable results).

Still, though, we achieve relatively promising results. The pendulum is oscillating around our desired equilibrium point, and we are able to remain on within our cart limits. The pendulum also reaches its oscillation phase quickly and remains there until the time runs out.

2.4 Comparison With Israilov

Our methodology differs slightly from that used in Israilov. They receive very different results because they set a maximum angular velocity of 14 rad/s for the pendulum, while we do not. Because of this, their system repeatedly reaches $\theta = \pi$ but quickly falls back down to $\theta = 0$. Our allowance for infinite angular velocity means that the system can rapidly oscillate around our desired position.

Israilov's method is more effective than ours, likely because they are able to run the the learning for many more iterations. Even with their increased resources, though, their model is only able to achieve $\theta = \pi$ for very short periods of time before the pendulum falls back to its start position. Since both our model and that in Israilov are ineffective at solving the inverted pendulum problem, we turn to deep learning.

3 Deep Learning

We now utilize a Deep Q Network to overcome the pitfalls of our simple Q-Learning algorithm. Instead of using a table to estimate the Q function, we are able to use a neural network. This solves our two major problems: the continuous variables and the cost of exploring all states in the Q table.

3.1 Setup

We have the same dynamics as in our Q-Learning set up, but without binned states. In our deep learning setup we have 2 inner layers with 256 nodes in each layer. The input layer is a 5x1 state vector consisting of $(\sin theta, \cos\theta, \frac{d\theta}{dt}, x, \frac{dx}{dt})$ and our output layer is a 3x1 vector of Q values corresponding to actions. The hidden layers utilize the Relu function. Unlike in Israilov, in which a memory replay was used to update the weights of the neural network, we update the weights online, using a steepest descent and the Huber loss function.

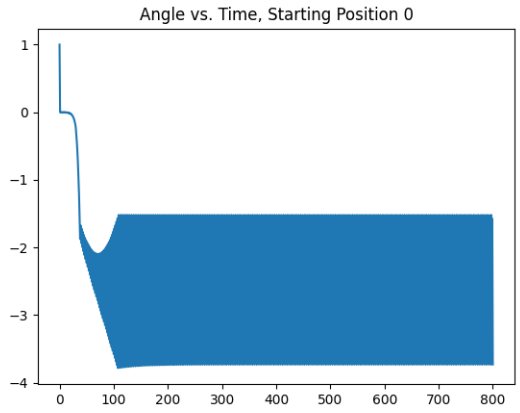
3.2 Pseudocode Algorithm

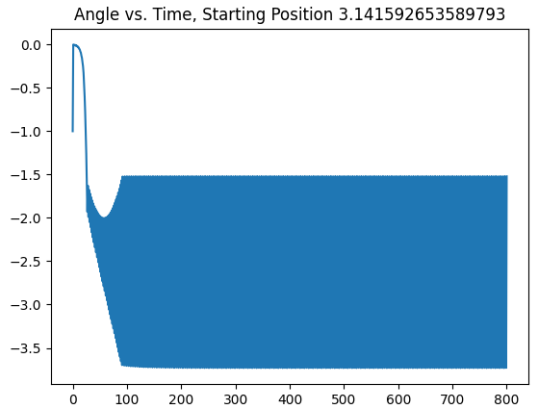
(Full code and results are in Appendix Section B)

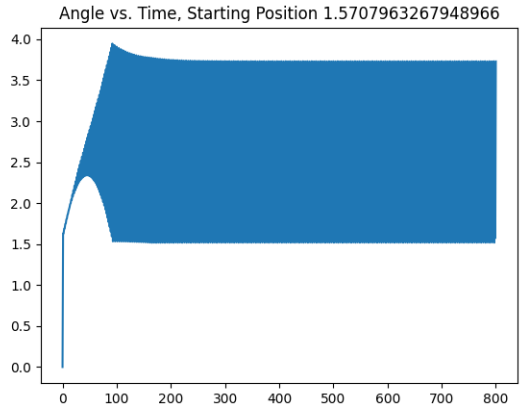
- For event:
  - Initialize state to $\theta = 0 \pm 0.25$, $\frac{d\theta}{dt}, \frac{dx}{dt} = 0$, $x = \pm 0.25$
  - While exit conditions (outside x-bounds, time = 40 minutes) are not met:
    - If random number ¡ epsilon: select random action
    - Else: Choose max(neural-network(state)) as action
    - Take action and update state using dynamics
    - Reward $= 0.5 * (1 - \cos\theta_{i+1}) - x_{i+1}^2$
    - If abs(x) greater than 0.35 or time = 40 minutes: set done = True
    - Update neural network parameters using Huber loss and Adam optimization
    - Set state = new state

3.3 Results

First we present the test results of our neural network using online updates to the weights matrices.







We get similar results as with the standard Q-learning approach. Again these results are not good, but we expected to receive relatively poor results due to the use of using online updates. Still, though, our system stabilizes in a rapid oscillation around $\theta = \pm\pi$, which is close to what our desired result it. The system is certainly more stable than the standard Q-learning, as we have constant oscillations that do not vary in amplitude, as we do with standard Q-learning.

3.4 Comparison With Israilov

Israilov gets much better results for their DQN. They utilized a memory replay and batch updates for their neural network. They are also able to train the network for longer with more computing power, and are able to solve the system, maintaining the equilibrium position. Clearly, batch updates to the network using randomly sampled memory replay is a more effective way to train the network.

4 Effects of Parameters

In this section, we discuss the effects of varying some of the parameters of the dynamics as detailed by Israilov. We rely on Israilov for this section because the amount of time required to explore different parameters is not feasible given our technological limitations.

4.1 Voltage, Acceleration, Velocity

Israilov found that low voltages are not suitable for solving this problem. Voltages below 4.7V do not yield good cumulative rewards for an episode of learning, while higher voltages, such as the 7.1V used in the model, yield more robust results. According to Israilov, this 7.1V is ideal for the system, and as such that is what we used. This voltage allows the pendulum to swing up to a higher position much faster than lower voltages. However, once there, the system suffers because any movement results in a large shift in angle, as seen above in our results. Therefore, it would seem that we might want to have less acceleration and velocity in order to prevent the large swings that we see in our data. However, this

could be due to a lack of learning time (i.e. with more time, the system would better learn the states near $\theta = \pi$ and not fluctuate so wildly). More exploration is needed in this area.

4.2 Friction

According to Israilov, the friction coefficient can have a huge impact on learning, even causing the system to fail to learn. For static friction (f_c) values between 0 an the experimental value of 1.166, the system has no significant difference in learning and is able to effectively perform the task. However, if we increase the static friction to 11.66, the system is unable to learn and fails to stabilize. The effect of the viscous friction coefficient (k_v) is identical: for values between 0 and the experimental value of 0.07, the network is able to learn, but when the value is increased to 0.7, learning is severely negatively affected.

5 Conclusion

Deep learning is not strictly needed to solve the inverted pendulum problem. However it is a better method than standard Q-learning. In the standard Q-learning approach, our results are fairly good - we reach our oscillation state relatively quickly and remain there for the duration of the episode. However, the oscillations are very wide and not particularly stable. Our DQN approach is similarly not particularly stable, also because of computational limitations and the use of on-line updates. However, the Israilov's DQN approach is very effective and stable, and shows that a DQN is the best and most efficient RL method for solving the inverted pendulum problem. Therefore, because DQN allows us to use continuous variables and solves the inefficiency problem of using and exploring a large Q table of states, DQN is a better method for solving the inverted pendulum problem than standard Q-learning.

6 Appendix

6.1 A. Q-Learning Code

```
import numpy as np
from time import sleep
import matplotlib.pyplot as plt
x_max = 0.35
g = 9.8
actions = [-7.1, 0, 7.1]

dt = 0.05

#Q initialization and binning
Q = np.random.random(size=(50, 50, 50, 10, 10, 3))
sin_bins = np.linspace(-1, 1, 50)
cos_bins = np.linspace(-1, 1, 50)
av_bins = np.linspace(-20, 20, 50)
x_bins = np.linspace(-0.35, 0.35, 10)
v_bins = np.linspace(-5, 5, 10)
bins = [sin_bins, cos_bins, av_bins, x_bins, v_bins]


#Q learning
for event in range(10000):
    #Initialize State
    done = False
    theta_init = 0 + np.random.uniform(-1, 1)

    x_init = np.random.uniform(-0.2, 0.2)
    state = np.array([np.sin(theta_init), np.cos(theta_init), 0, x_init, 0])
    binned_state = [np.digitize(state[i], bins[i])-1 for i in range(5)]
    n = 0
    cum_reward = 0
    #Iterate
    while not done:
        n += 1
        epsilon = max(0.1, min(1, 1-np.log10((n+1)/(10**3 - 10**6)))) #get epsilon
        if np.random.rand() < epsilon:
            action_index = np.random.randint(0, 3)
        else:
            action_index = np.argmax(Q[binned_state])
        action = actions[action_index]

        #Update state
        cv = 0.051*action
        a = (1/0.0482) * (cv-state[4]) - 1.166*np.sign(state[4]) + 0.097
        aa = -0.07*state[2] + (4.882**2)*state[0] + (4.882**2)*a*state[1]/g

        v_new = state[4] + a * dt
        x_new = state[3] + v_new * dt
```

```
        av_new = state[2] + aa * dt
        theta_new = np.arcsin(state[0]) + av_new * dt
        sin_new = np.sin(theta_new)
        cos_new = np.cos(theta_new)
        state_new = np.array([sin_new, cos_new, av_new, x_new, v_new])
        binned_state_new = [np.digitize(state_new[i], bins[i])-1 for i in range(5)]
        reward = 0.5*(1-state_new[1]) - (state_new[3]/1)**2

        #Check exit conditions
        if np.abs(state_new[3]) > 0.35:
            reward -= 400
            done = True
        if n == 800:
            done = True

        cum_reward += reward

        #Update Q
        i1, i2, i3, i4, i5 = binned_state
        n1, n2, n3, n4, n5 = binned_state_new
        current_Q = Q[i1, i2, i3, i4, i5, action_index]

        Q[i1, i2, i3, i4, i5, action_index] += 0.01 * (reward + 0.99*np.max(Q[n1, n2,
n3, n4, n5, :]) - current_Q)

        #Assign new states to state
        state = state_new
        binned_state = binned_state_new

#Testing
for i in range(3):
    #Initialization
    done = False
    thetas = [0, np.pi, np.pi / 2]
    state = np.array([np.sin(thetas[i]), np.cos(thetas[i]), 0, 0, 0])
    binned_state = [np.digitize(state[i], bins[i])-1 for i in range(5)]

    n = 0
    cum_reward = 0
    position_array = [0]
    angle_array = [thetas[i]]
    v_array = [0]
    #Iteration
    while not done:
        n += 1
        b1, b2, b3, b4, b5 = binned_state
        action_index = np.argmax(Q[b1, b2, b3, b4, b5])
        action = actions[action_index]
```

```
        #Update state
        cv = 0.051*action
        a = (1/0.0482) * (cv-state[4]) - 1.166*np.sign(state[4]) + 0.097
        aa = -0.07*state[2] + (4.882**2)*state[0] + (4.882**2)*a*state[1]/g

        v_new = state[4] + a * dt
        x_new = state[3] + v_new * dt
        v_array.append(v_new)
        av_new = state[2] + aa * dt
        theta_new = np.arcsin(state[0]) + av_new * dt
        sin_new = np.sin(theta_new)
        cos_new = np.cos(theta_new)
        state_new = np.array([sin_new, cos_new, av_new, x_new, v_new])
        binned_state_new = [np.digitize(state_new[i], bins[i])-1 for i in range(5)]
        reward = 0.5*(1-state_new[1]) - (state_new[3]/1)**2

        #Check exit conditions
        if np.abs(state_new[3]) > 0.35:
            cum_reward -= 400
            done = True
        if n == 800:
            done = True

        #Update state variables
        state = state_new
        binned_state = binned_state_new
        #Get graph points
        position_array.append(x_new)
        angle_array.append(theta_new)

    #Plot data
    plt.plot(angle_array)
    plt.title(f"Angle vs. Time, Starting Position {thetas[i]}")
    plt.show()
```

6.2 B. DQN Code

```
import numpy as np
from time import sleep
import matplotlib.pyplot as plt
import tensorflow as tf

actions = [-7.1, 0, 7.1]

neural_network = tf.keras.Sequential([
    tf.keras.layers.Dense(256, activation='relu', input_shape=(5,)),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(3, activation='linear')  # Adjust the output size
])

optimizer = tf.keras.optimizers.Adam(learning_rate=0.0003)


neural_network.compile(optimizer=optimizer, loss='huber')
dt = 0.05

#Training
for event in range(25):
    #Initialize
    done = False
    theta_init = 0
    x_init = 0
    state = np.array([[np.sin(theta_init), np.cos(theta_init), 0, x_init, 0]])
    t = 0
    cum_reward = 0
    #Iterate
    while not done:
        t += 1
        Qs = neural_network.predict(state, verbose = False)[0]
        if np.random.rand() < 0.178:
            action_index = np.random.randint(3)
        else:
            action_index = np.argmax(Qs)

        action = actions[action_index]
        state = state[0]
        cv = 0.051 * action
        a = (1/0.0482) * (cv - state[4]) - 1.166 * np.sign(state[4]) + 0.097
        aa = -0.07 * state[2] + (4.882**2) * state[0] + (4.882**2) * a * state[1] /
9.81

        v_new = state[4] + a * dt
        x_new = state[3] + v_new * dt

        av_new = state[2] + aa * dt
```

```
        theta_new = np.arcsin(state[0]) + av_new * dt
        sin_new = np.sin(theta_new)
        cos_new = np.cos(theta_new)
        state_new = np.array([[sin_new, cos_new, av_new, x_new, v_new]])
        state = np.array([state])
        reward = 0.5 * (1 - state_new[:,1]) - (state_new[:,3]/1)**2


        if np.abs(state_new[0,3]) > 0.35:
            cum_reward -= 400
            done = True

        if t >= 800:
            done = True

        cum_reward += reward

        Q_stars = neural_network.predict(state_new, verbose = False)[0]
        Q_star = np.max(Q_stars)

        target = reward + 0.995* Q_star

        #Update Q
        with tf.GradientTape() as tape:
            q_values = neural_network(state, training=True)
            loss = tf.keras.losses.huber(Qs, q_values)

        gradients = tape.gradient(loss, neural_network.trainable_variables)
        optimizer.apply_gradients(zip(gradients, neural_network.trainable_variables))

        state = state_new

    print(f"Episode: {event}, Cum Reward: {cum_reward/t}")

#Testing
theta_init = [0, np.pi, np.pi/2]
for i in range(3):
    done = False
    state = np.array([[np.sin(theta_init[i]), np.cos(theta_init[i]), 0, 0, 0]])
    t = 0
    cum_reward = 0
    ts = [0]
    thetas = [np.cos(theta_init[i])]
    while not done:
        t += 1
        Qs = neural_network.predict(state, verbose = False)[0]
        action_index = np.argmax(Qs)
```

```
        action = actions[action_index]
        state = state[0]
        cv = 0.051 * action
        a = (1/0.0482) * (cv - state[4]) - 1.166 * np.sign(state[4]) + 0.097
        aa = -0.07 * state[2] + (4.882**2) * state[0] + (4.882**2) * a * state[1] /
9.81

        v_new = state[4] + a * dt
        x_new = state[3] + v_new * dt

        av_new = state[2] + aa * dt
        theta_new = np.arcsin(state[0]) + av_new * dt
        sin_new = np.sin(theta_new)
        cos_new = np.cos(theta_new)
        state_new = np.array([[sin_new, cos_new, av_new, x_new, v_new]])
        state = np.array([state])
        reward = 0.5 * (1 - state_new[:,1]) - (state_new[:,3]/1)**2

        ts.append(t)
        thetas.append(theta_new)

        if np.abs(state_new[:,3]) > 0.35:
            reward -= 400
            done = True

        if t >= 800:
            done = True

        cum_reward += reward
        state = state_new
    plt.plot(thetas)
    plt.title(f"Angle vs. Time, Starting Position {theta_init[i]}")
    plt.show()
```

7 Citations

Israilov S, Fu L, Sánchez-Rodríguez J, Fusco F, Allibert G, Raufaste C, et al. (2023) Reinforcement learning approach to control an inverted pendulum: A general framework for educational purposes. PLoS ONE 18(2): e0280071. https://doi.org/10.1371/journal.pone.0280071