COMP3702

Tutorial 11 + A3 help

Semester 2, 2022

Aryaman Sharma (aryaman.sharma@uq.edu.au)

The University of Queensland
School of Information Technology and Electrical Engineering

Q-learning vs SARSA

On-policy learning

- Q-learning does off-policy learning: it learns the value of an optimal policy, no matter what it does.
- This could be bad if the exploration policy is dangerous.
- On-policy learning learns the value of the policy being followed. e.g., act greedily 80 percent of the time and act randomly 20 percent of the time
- Why? If the agent is actually going to explore, it may be better to optimize the actual policy it is going to do.
- SARSA uses the experience (s, a,r,s', a) to update Q(s, a).

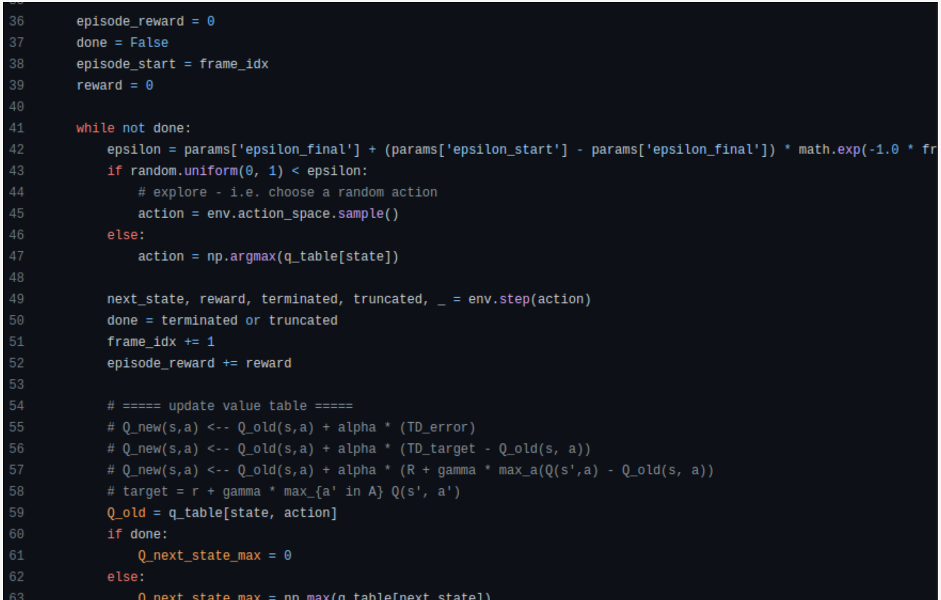
SARSA

```
initialize $\hat{Q}(s, a)$ arbitrarily
observe current state $s$
select an action $a$
**repeat** for each episode, until convergence:
    carry out an action $a$
    observe reward $r$ and state $s'$
    select an action $a'$ (in $s'$)
    $\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \left( r + \gamma \hat{Q}(s', a') - \hat{Q}(s, a) \right)$
    $s \leftarrow s'$
    $a \leftarrow a'$
```

Q-learning vs SARSA

Q-learning	**SARSA**
initialise $\hat{Q}(s, a)$ arbitrarily	initialise $\hat{Q}(s, a)$ arbitrarily
observe current state s	observe current state s
	select an action a
repeat for each episode, until convergence:	**repeat** for each episode, until convergence:
select and carry out an action a	carry out an action a
observe reward r and state s'	observe reward r and state s'
	select an action a'
$\hat{Q}(s, a) \leftarrow \hat{Q}(s, a)$	$\hat{Q}(s, a) \leftarrow \hat{Q}(s, a)$
$+\alpha \left(r + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a) \right)$	$+\alpha \left(r + \gamma \hat{Q}(s', a') - \hat{Q}(s, a) \right)$
$s \leftarrow s'$	$s \leftarrow s'$
	$a \leftarrow a'$

Q-learning implementation



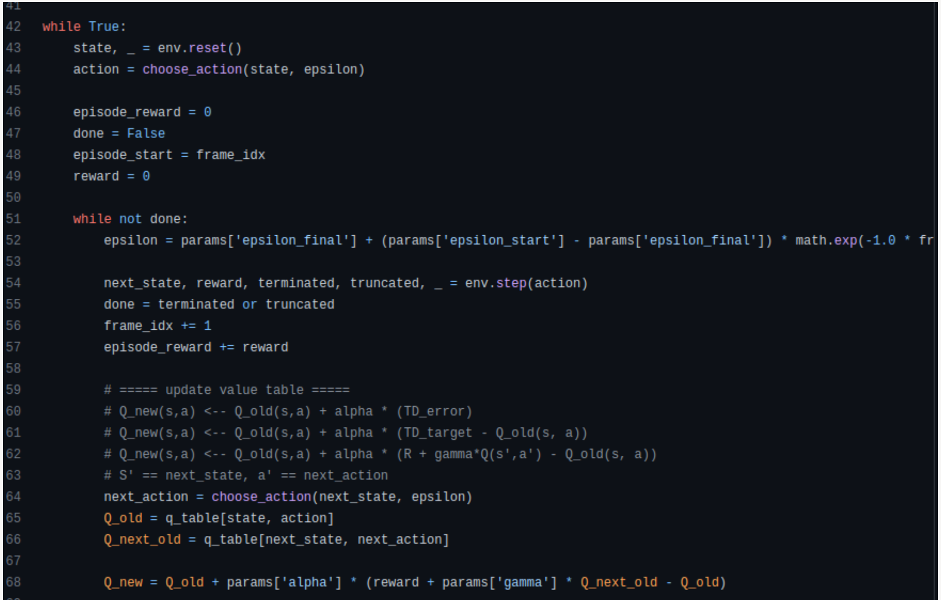
```
        episode_reward = 0
        done = False
        episode_start = frame_idx
        reward = 0

        while not done:
            epsilon = params['epsilon_final'] + (params['epsilon_start'] - params['epsilon_final']) * math.exp(-1.0 * fr
            if random.uniform(0, 1) < epsilon:
                # explore - i.e. choose a random action
                action = env.action_space.sample()
            else:
                action = np.argmax(q_table[state])

            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            frame_idx += 1
            episode_reward += reward

            # ===== update value table =====
            # Q_new(s,a) <-- Q_old(s,a) + alpha * (TD_error)
            # Q_new(s,a) <-- Q_old(s,a) + alpha * (TD_target - Q_old(s, a))
            # Q_new(s,a) <-- Q_old(s,a) + alpha * (R + gamma * max_a(Q(s',a) - Q_old(s, a))
            # target = r + gamma * max_{a' in A} Q(s', a')
            Q_old = q_table[state, action]
            if done:
                Q_next_state_max = 0
            else:
                Q_next_state_max = np.max(q_table[next_state])
```

SARSA implementation



```python
while True:
    state, _ = env.reset()
    action = choose_action(state, epsilon)

    episode_reward = 0
    done = False
    episode_start = frame_idx
    reward = 0

    while not done:
        epsilon = params['epsilon_final'] + (params['epsilon_start'] - params['epsilon_final']) * math.exp(-1.0 * fr

        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        frame_idx += 1
        episode_reward += reward

        # ===== update value table =====
        # Q_new(s,a) <-- Q_old(s,a) + alpha * (TD_error)
        # Q_new(s,a) <-- Q_old(s,a) + alpha * (TD_target - Q_old(s, a))
        # Q_new(s,a) <-- Q_old(s,a) + alpha * (R + gamma*Q(s',a') - Q_old(s, a))
        # S' == next_state, a' == next_action
        next_action = choose_action(next_state, epsilon)
        Q_old = q_table[state, action]
        Q_next_old = q_table[next_state, next_action]

        Q_new = Q_old + params['alpha'] * (reward + params['gamma'] * Q_next_old - Q_old)
```

https://github.com/comp3702/tutorial10

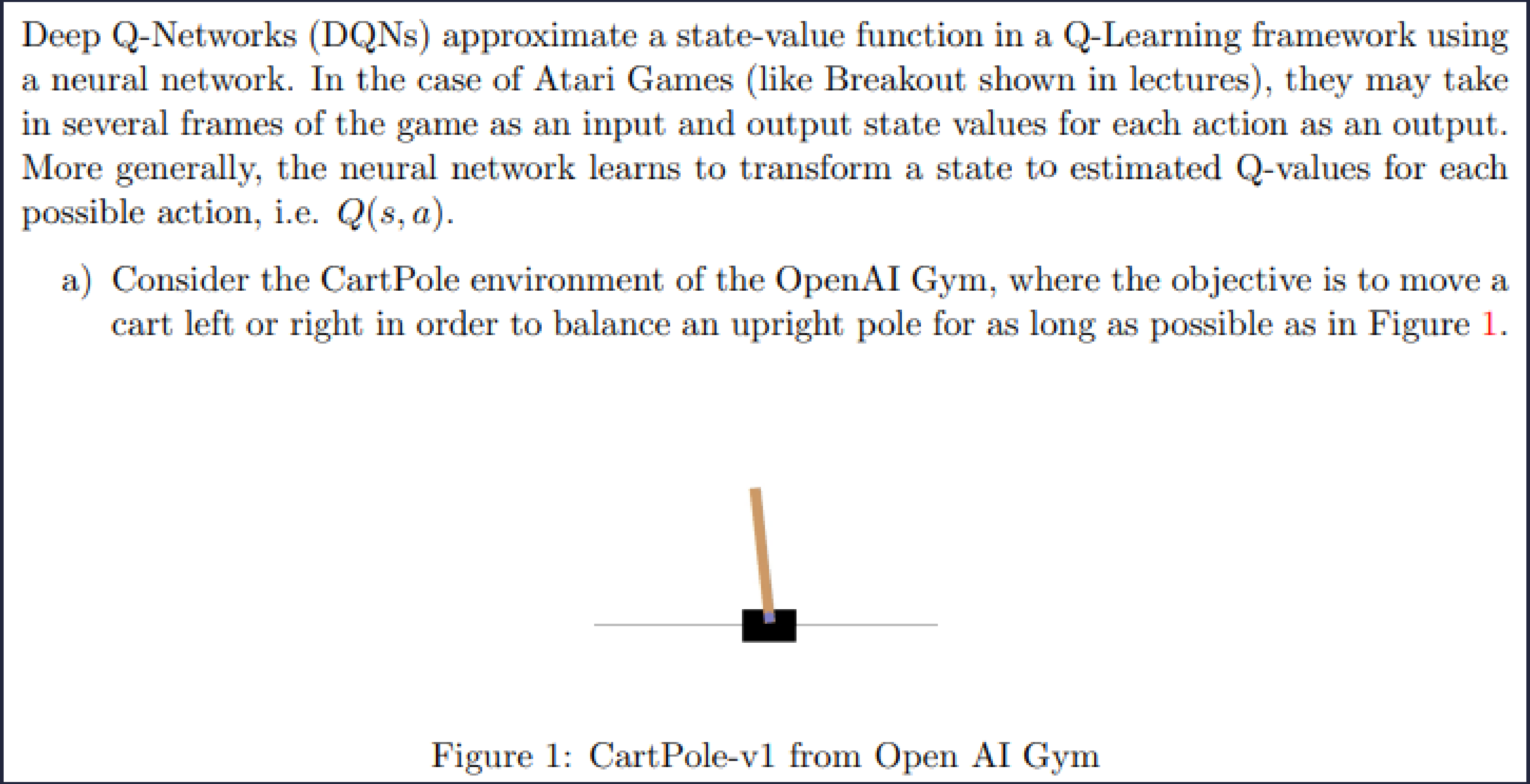
COMP3702 Tutorial 10

Matt Choy | matthew.choy@uq.edu.au

Deep Q-Networks

- If we don’t know the system dynamics, should we take exploratory actions (that give us more information about the environment) or exploit current knowledge to perform as best as we can?
- Using a greedy policy (exploiting current knowledge), bad initial estimates in the first few cases can drive policy into a sub-optimal region, and never explore further
- Instead of acting according to a greedy policy, we act according to a sampling strategy that will explore (state, action) pairs until we get a “good” estimate of the value function

Deep Q-Networks



Deep Q-Networks (DQNs) approximate a state-value function in a Q-Learning framework using a neural network. In the case of Atari Games (like Breakout shown in lectures), they may take in several frames of the game as an input and output state values for each action as an output. More generally, the neural network learns to transform a state to estimated Q-values for each possible action, i.e. $Q(s, a)$.

a) Consider the CartPole environment of the OpenAI Gym, where the objective is to move a cart left or right in order to balance an upright pole for as long as possible as in Figure 1.

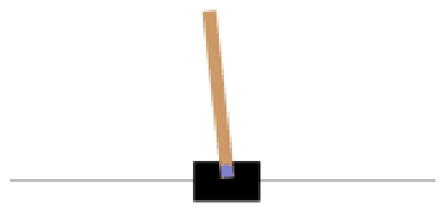
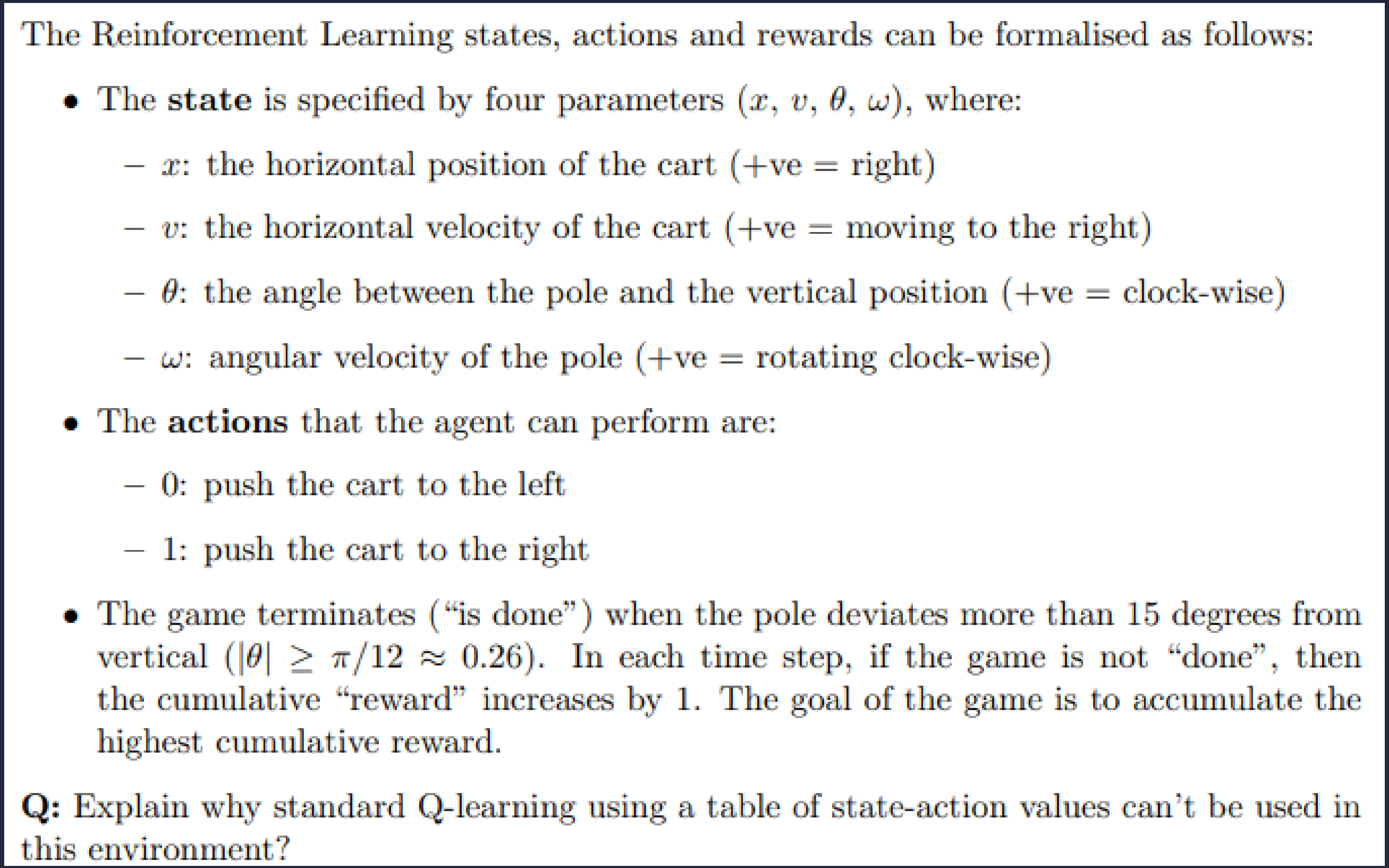


Figure 1: CartPole-v1 from Open AI Gym

Exercise 11.1

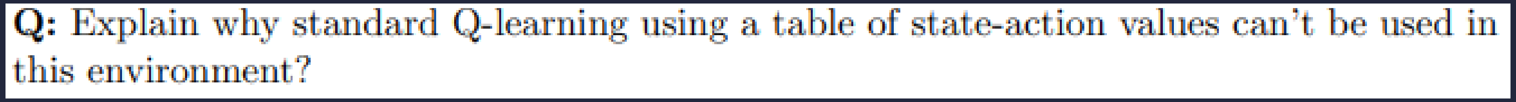


The Reinforcement Learning states, actions and rewards can be formalised as follows:

- The **state** is specified by four parameters (x, v, θ, ω), where:
  - x: the horizontal position of the cart (+ve = right)
  - v: the horizontal velocity of the cart (+ve = moving to the right)
  - θ: the angle between the pole and the vertical position (+ve = clock-wise)
  - ω: angular velocity of the pole (+ve = rotating clock-wise)
- The **actions** that the agent can perform are:
  - 0: push the cart to the left
  - 1: push the cart to the right
- The game terminates ("is done") when the pole deviates more than 15 degrees from vertical ($|\theta| \geq \pi/12 \approx 0.26$). In each time step, if the game is not "done", then the cumulative "reward" increases by 1. The goal of the game is to accumulate the highest cumulative reward.

Q: Explain why standard Q-learning using a table of state-action values can't be used in this environment?

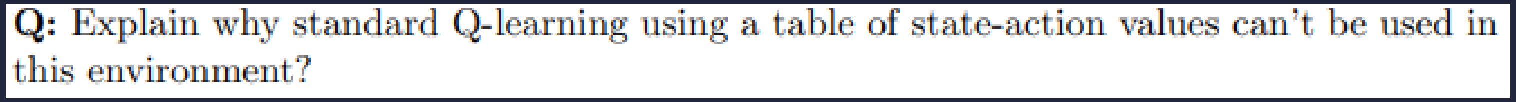
Exercise 11.1a



Q: Explain why standard Q-learning using a table of state-action values can't be used in this environment?

- The state in the cartpole environment is uniquely identified by four parameters:
$$(x, v, \theta, \omega)$$
- These four values are continuous in nature.
- To update the Q-Values in our table-based Q-Learning algorithm, we would need to discretise the values.

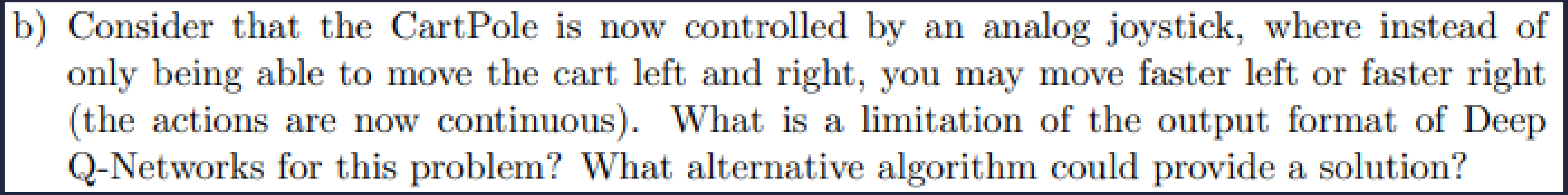
Exercise 11.1a


Q: Explain why standard Q-learning using a table of state-action values can't be used in this environment?

- The state in the cartpole environment is uniquely identified by four parameters:
$$(x, v, \theta, \omega)$$
- These four values are continuous in nature.
- To update the Q-Values in our table-based Q-Learning algorithm, we would need to discretise the values.

- Instead, the values can be input into a value function approximator, and performing function approximation (using a neural network), to estimate the value of each action for a given state.

Exercise 11.1b


b) Consider that the CartPole is now controlled by an analog joystick, where instead of only being able to move the cart left and right, you may move faster left or faster right (the actions are now continuous). What is a limitation of the output format of Deep Q-Networks for this problem? What alternative algorithm could provide a solution?

- Deep Q-Networks only output discrete values, and therefore cannot output analogue values.
- We could use Policy Gradients, which can output continuous probability distributions
- We could describe the output as how fast to move on the horizontal axis.