Reinforcement Learning

Presented by: Almiqdad Elzein

Instructor: Dr. Yasser Alginahi

Date: November 17th, 2023

University of Windsor

Reinforcement VS Supervised and Unsupervised Learning

- So Far, we have discussed input-to-variable mapping tasks
- Given an input, we predict the value of a discrete or continuous variable
  - Discrete variables: Clustering, Classification
  - Continuous Variables: Regression
- What about the following game-like scenario:
  - An agent exists in some environment with some goal
  - The agent's goal could be to maximize profit or win a board game
  - **Key Question: What actions should the agent take in this environment?**

Game-Like scenarios

- The actions the agent should take at any point in time depends on:
  - The agent's goal – e.g, winning a chess game
  - The state of the environment – e.g, the state of the chess board
  - How the agent's actions affect the environment

- Main Challenge: There are around 10^{43} possible states in a chess board [1]
  - It's impossible to manually outline the state-action mapping
  - Can the mapping be learnt?

- Reinforcement Learning (RL) solution: **play and learn!**

Rewards – The Driver of Learning

- In RL, the agent gets ***rewards*** for its actions
- Large rewards imply good actions while small rewards imply worse actions
- The reward signal allow the agent to learn the right actions to take given the current state
  - An agent maintains a ***policy*** telling it what action to take based on the state of the environment
- An agent in RL adjusts its policy based on the ***rewards*** it received

Reinforcement Learning Definition

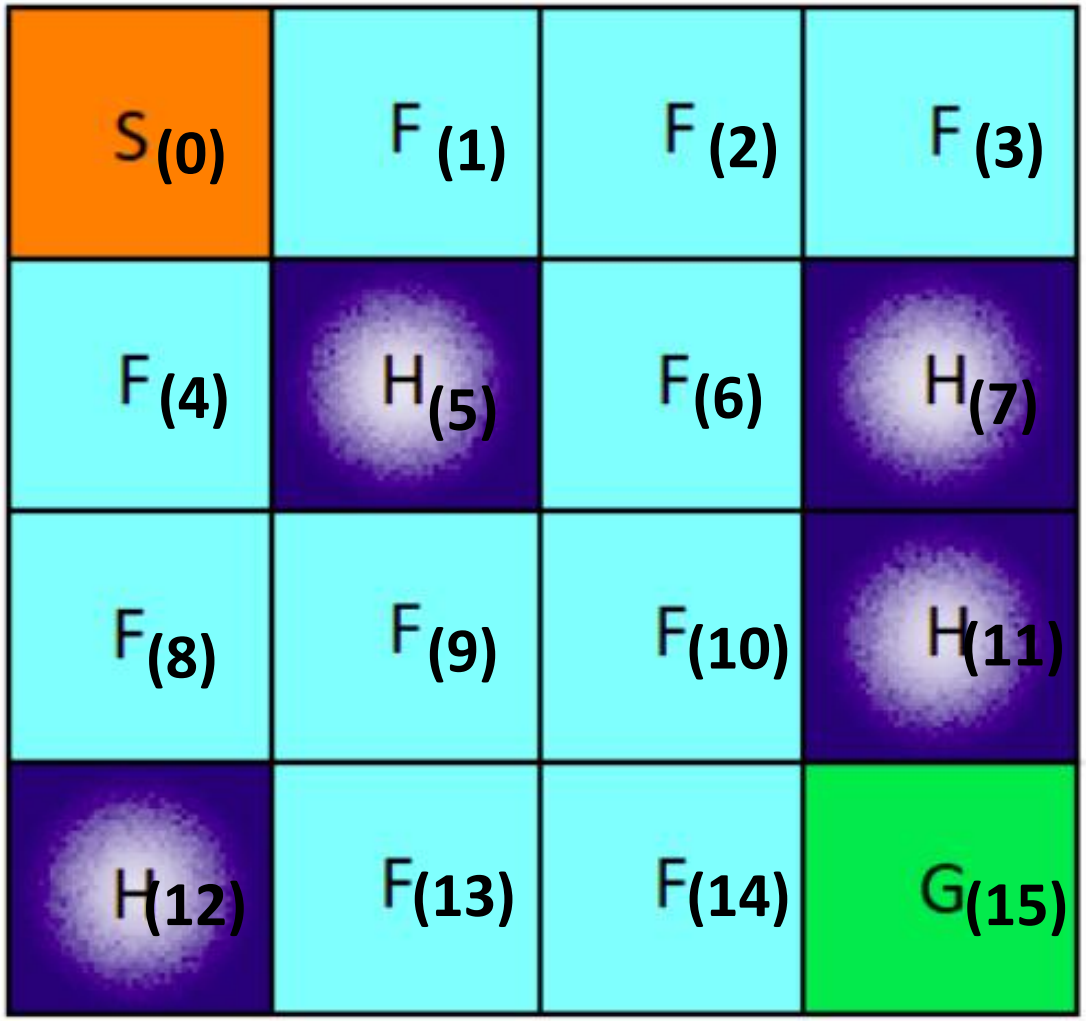
- RL is an online learning paradigm for decision-making [2] where an agent:
  1. takes ***actions*** in the ***environment*** based on its ***policy***
  2. Gets ***rewards*** for its ***actions***
  3. uses those rewards to update its ***policy***
  4. Go back to step 1
- This process is known as a Markov Decision Process [3]
  - MDPs assume that only previous actions effect the state
- An agent may learn a sequence of actions to reach its goal – an eye for the future
  - Supervised and Unsupervised Learning have no consideration of future "decisions"

Why RL As a Learning Paradigm?

- Assume that we are able to implement a RL agent to act in some environment, what would we achieve?
    - We would be able to estimate the state-action mappings in evolving environments

- Is this "learning"?
    - Yes, we learn a good policy by repeatedly taking actions and updating our policy based on the rewards
    - This is similar to updating weights to minimize loss in Deep Learning

Example – Frozen Lake



source: https://tinyurl.com/24zruwt3

- Goal: Maximize Reward
- ***States:*** 16 possible states depending on agent's location (unchanging environment)
- ***Actions:*** Left, Right, Up, Down
- ***Rewards:*** +1 for **F**, -10 for **H**, +100 for **G**
- Designing rewards is key in RL

What should Our Policy in Frozen Lake be?

- The agent's goal is to find a good policy to play the game

- What is a policy? A state-action mapping (wrong)
    - The best action to take given each of the possible states

- This approach doesn't allow us to update our policy in a way that incorporates past information
    - The best action to take given a state is a categorical value

What should Our Policy in Frozen Lake be?

- Remember our goal: To maximize reward
- The reward to maximize is the cumulative reward (until the game is shut-down)
- Given our goal, what action should we take?
  - The action that will maximize the cumulative reward given the current state
- Approach: Given the current state, take the action that maximizes the cumulative reward

Q-Learning

- Our policy is a table that stores the expected cumulative reward for each state-action pair

- The expected cumulative reward of a state-action pair is the ***Q-value*** of that state-action pair

- Q-Learning is an approach where a Q-table is maintained and updated for all possible state-action pairs

The Q-table in Frozen Lake

- Below is a sub-set of the Q-table (our policy) from the Frozen Lake game where all values are initialized to 0

- $Q(s, a)$ is the expected future return that results from being in state s and taking action a

	LEFT	**Right**	**UP**	**Down**
State 0	0	0	0	0
State 1	0	0	0	0
State 2	0	0	0	0
State 3	0	0	0	0

Our First move in Frozen Lake (Take an Action)

- We start at state 0
- Since all actions have an equal Q-value at state 0, we choose a random action
- Assume we go RIGHT
- We observe a reward of +1
- Next: Update Q-table

	LEFT	**Right**	**UP**	**Down**
State 0	0	0	0	0
State 1	0	0	0	0
State 2	0	0	0	0
State 3	0	0	0	0

Our First move in Frozen Lake (Update the Q-table)

- The goal is to combine old information (current Q-value) with new information (new Q-value)

- We know the current $Q(0,\ RIGHT)$, what should be its new value?
  - We want to compute the new estimate of the cumulative return based on the new observed reward
  - The new estimate for the cumulative return is the reward + best cumulative return of the next state - State 1 (since we moved right from State 0)

- New estimate for $Q(0,\ RIGHT)$: $+1 + \max_{a'}(Q(1,\ a')) = +1 + 0 = 1$

Our First move in Frozen Lake (Update the Q-table)

- Principle of learning: *combine* new information with old information

$$Q^{old}(0, RIGHT) = 0,\ Q^{new}(0, RIGHT) = 1$$

- The updated $Q(0, RIGHT)$ is obtained from the above values as:

$$Q(0, RIGHT) \ = \ (1 - \alpha)Q^{old} + \alpha Q^{new}$$

- $0 \leq \alpha \leq 1$ and is known as a *learning rate*

Our First move in Frozen Lake (Update the Q-table)

- Since future rewards are uncertain, we discount them when updating Q-values
- We redefine $Q^{new} = R + \gamma \max_{a'}(Q(1, a'))$, where $0 \leq \gamma \leq 1$
- Introducing this discount also prevents Q-values from growing out of control
- In Summary, taking action a in state s and observing a reward R and s' as the reward and next state, we update $Q(s, a)$ as follows:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(R + \gamma \max_{a'}(Q(s', a'))) \ (1)$$

Our First move in Frozen Lake (Update the Q-table)

- Assume $\alpha = 0.5$ and $\gamma = 0.95$, and since:

$$Q^{old}(0, RIGHT) = 0,\ Q^{new}(0, RIGHT) = 1$$

- Then the updated Q-value for $(0, RIGHT)$ becomes:

$$Q(0, RIGHT) = (1 - \alpha)Q(0, RIGHT) + \alpha(R + \gamma \max_{a'}(Q(1, a')))$$

- Where $Q(0, RIGHT) = 0$ and $\max_{a'}(Q(1, a')) = 0$ (from Q-table)

- Thus, $Q(0, RIGHT) = (1 - 0.5)(0) + 0.5(1 + 0) = 0.5$

Our First move in Frozen Lake (Update the Q-table)

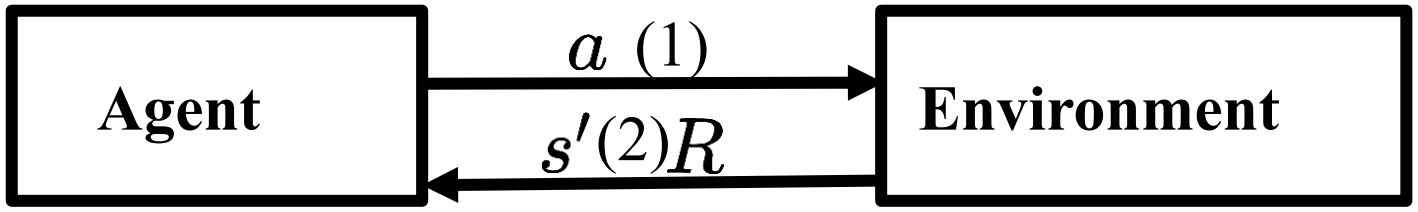
- The new Q-table will become as follows:

	LEFT	**Right**	**UP**	**Down**
State 0	0	0.5	0	0
State 1	0	0	0	0
State 2	0	0	0	0
State 3	0	0	0	0

- When another action is taken from state 1, the same updating process for the Q-table will take place

The Q-Learning Loop

- Q-learning is a potentially-infinite process where the agent
    1. Given s, takes an action a such that $a = \arg\max_{a'}(Q(s, a'))$
    2. Observes reward R and the new state s'
    3. Updates $Q(s, a)$ based on (1) in slide 15
    4. Go back to step 1



Counteracting a Greedy Strategy

- With time, some paths will be favored by the agent since they have high Q-values
- This will prevent the agent from *exploring* new territory in the state-action space
- Following the Q-table is *exploiting* learnt information, can we encourage *exploring* other states?
- With some probability ϵ, take a random action – ***epsilon greedy strategy***
  - One smart approach is to decay ϵ with time (exploit more with more info)

Why does Q-learning work?

- Recall the Q-value updating function:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(R + \gamma \max_{a'}(Q(s',a')))$$

- Through iterative updating, $Q(s,a)$ approaches $R + \gamma \max\left(Q(s',a')\right)_{a'}$
  - The value of $R + \gamma \max\left(Q(s',a')\right)_{a'}$ will be different each time, so the Q-value approaches the true average of $R + \gamma \max\left(Q(s',a')\right)_{a'}$
  - When $Q(s,a)$ reaches this true average, it will continue to be oscillating close to this average; learning will be completed

Updating Q-values based on a Trajectory (Episodic Rewards)

- Assume that a trajectory of state-action pairs lead to some reward R
  - E,g: $s_0 \Rightarrow a_0 \Rightarrow s_1 \Rightarrow a_1 \Rightarrow \ldots \Rightarrow a_{T-1} \Rightarrow s_T$
  - A chess move doesn't give an immediate reward
  - ***Episode:*** a trajectory where the final state is some pre-defined s_T

- Our goal is to update $Q(s_0, a_0), \ldots, Q(s_{T-1}, a_{T-1})$ based on R

- **C*redit Assignment Problem:*** The problem of finding which action(s) resulted in the reward

Solving the Credit Assignment Problem

- Approach: $Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha\left(\frac{T-t}{T}R\right)$
  - $\frac{T-t}{T}R$ is the estimated cumulative reward of (s_t, a_t)
  - We are assuming all actions are equally-responsible for the reward
  - This approach is named Monte Carlo Learning [5]

- Approach 2: re-distribute the rewards to the state-action pairs (reshaping) [6]
  - Exponential decay: Reward at time t is given by $r_t = Re^{-\beta(t-T)}$
  - Linear decay: $r_t = R(1-\beta t)$
  - Extrapolation: $r_t = \frac{R}{t}$
  - After all $r_t, t \in [T-1]$ are computed, the Q-table can be updated via (1)

Very Large Q-Tables

- In RL, S is the set of all states and A is the set of all possible actions
  - The Q-table has $|S| \times |A|$ values
  - In chess $|S| = 10^{43}$
- In other scenarios S and A may be *infinite* sets (continuous variables)
  - A self-driving car whose action space is the speed and steering angle of a wheel (infinitely-many possible actions)
  - The joint angle of a robotic arm (infinitely-many possible states)

- With large state and action sets, we represent $Q(s, a)$ as an adjustable function with 2 inputs through a Neural Network – ***Deep Q-Learning (DQL)***

Neural Networks as functions

- A neural network with parameters θ (weights and biases) transforms an X input Y into an output vector
- A Q-table transforms $X = \binom{s}{a}$ into $Q(s, a)$, the cumulative return of (s, a)
- Thus, the Q-function can be implemented through a Neural Net (a Q-network)
- One condition: Can we identify the "ideal" output for any input $X = \binom{s}{a}$?

Loss For a Q-Network

- Recall (1), the updating rule of the Q-value:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(R + \gamma \max_{a'}(s',a'))$$

- Clearly, the target value for $Q(s,a)$ is $Q^*(s,a) = R + \gamma \max_{a'}(Q(s',a'))$

- We compute $\max\left(Q(s',a')\right)_{a'}$ by passing $\binom{s'}{a'}$ to the network for each $a' \in A$ and finding the maximum Q-value

- The Q-network is updated based on the loss between $Q(s,a)$ and $Q^*(s,a)$

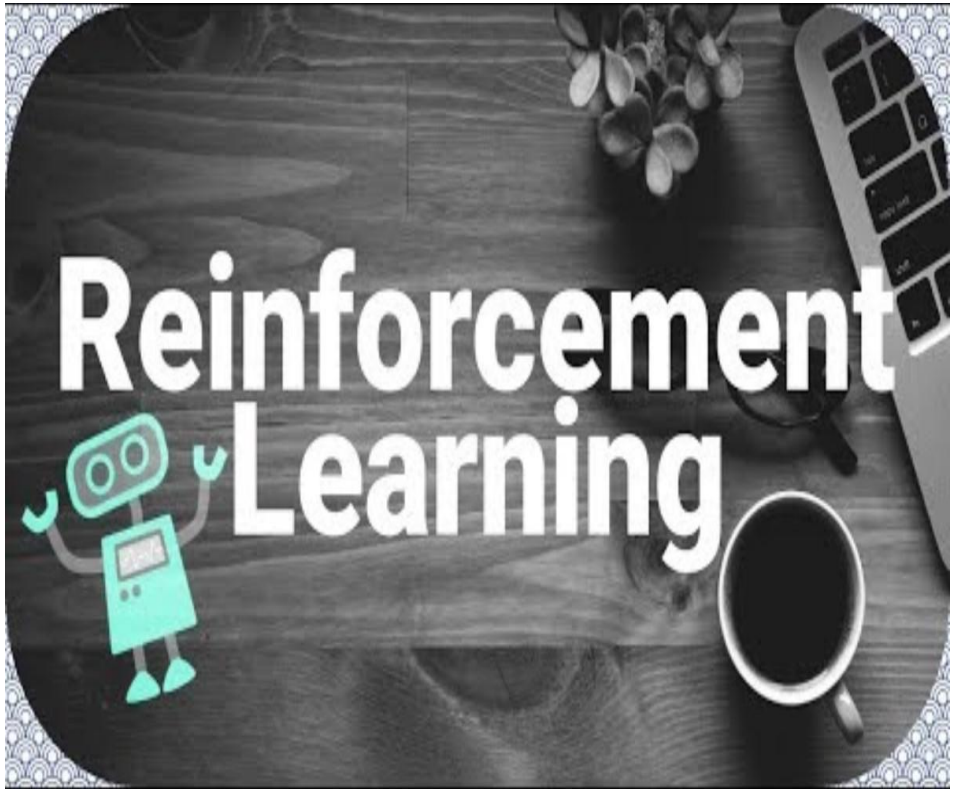
Loss For a Q-Network with an infinite actions space

- We compute $\max\left(Q(s', a')\right)_{a'}$ by passing $\binom{s'}{a'}$ to the network for each $a' \in A$ and finding the maximum Q-value
- If $|A| = \infty$, then we use gradient-based optimization: $\frac{\partial Q(s',a)}{\partial a} = 0$ to find a^*
  - Recall that a Neural Network computes gradients
- Then, $Q^*(s, a) = Q(s', a^*)$
- Summary: (1) take action a based on s (2) get R and s' (3) get $Q^*(s, a)$ (4) update network and state (5) repeat

Conclusion

- Reinforcement Learning is a branch of Machine Learning where the goal is to find the best action policy of an agent existing in some environment
- Q-Learning is a method of RL where the agent updates a table of Q-values of all possible state-action pairs
- The Q-value of a state-action pair is the expected cumulative return resulting from being in some state and taking some action
- Q-Learning is used in Robotics, Games, and development of self-driving cars

Video for Q-Learning



https://youtu.be/qhRNvCVVJaA?si=D--w6AU8LIUfRky1

Implementing Q-Learning – Essential Libraries

```
import numpy as np
from tqdm import tqdm
import random
import time
import pygame
import sys
```

Implementing Q-Learning – Defining state-transition function

- Given a state and action, return the next state and reward

```
def get_next_state(s, a, action_list):
    wall_reward = -1
    F_reward = 1
    G_reward = 100
    H_reward = -10

    reward = None
    done = False
    if action_list[a] == "UP":
        next_state = s - 4
        if next_state < 0:
            next_state = s
            reward = wall_reward

    elif action_list[a] == "LEFT":
        if s in [0, 4, 8, 12]:
            next_state = s
            reward = wall_reward
        else:
            next_state = s - 1
    elif action_list[a] == "DOWN":
        if s in [12, 13, 14]:
            next_state = s
            reward = wall_reward
        else:
            next_state = s + 4
    elif action_list[a] == "RIGHT":
        if s in [3, 7, 11]:
            next_state = s
            reward = wall_reward
        else:
            next_state = s + 1


    if reward == None:
        if next_state == 15:
            reward = G_reward
            done = True
        elif next_state in [5, 7, 11, 12]:
            reward = H_reward
        else:
            reward = 0
    return next_state, reward, done
```

Implementing Q-Learning – initialize Q-table and Define hyper-parameters

```
action_list = ["UP", "LEFT", "DOWN", "RIGHT"]

# Initialize Q-table with zeros
state_space_size = 16
action_space_size = 4

Q = np.zeros((state_space_size, action_space_size))

# Set hyperparameters
learning_rate = 0.8
discount_factor = 0.95
num_episodes = 5000

# Exploration-exploitation trade-off parameters
epsilon = 1.0  # Exploration probability
min_epsilon = 0.01
max_epsilon = 1.0
decay_rate = 0.005
```

Implementing Q-Learning – Train RL Agent

```
# Q-learning algorithm
for episode in tqdm(range(num_episodes)):
    state = 0
    done = False
    while not done:
        # Exploration-exploitation trade-off
        if np.random.uniform(0, 1) < epsilon:
            action = random.randint(0,3)  # Explore
        else:
            action = np.argmax(Q[state, :])  # Exploit
        # Take the selected action and observe the next state and reward
        next_state, reward, done = get_next_state(state, action, action_list)
        # Update Q-value using the Bellman equation
        Q[state, action] = (1 - learning_rate) * Q[state, action] + learning_rate * (reward + discount_factor * np.max(Q[next_st
ate, :]))

        # Move to the next state
        state = next_state

    # Decay exploration probability
    epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay_rate * episode)

print("finished training")
```

Implementing Q-Learning – Use the Trained Agent in Pygame

```
# Initialize Pygame
pygame.init()

# Set up the display
width, height = 400, 400
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Frozen Lake Agent Visualization")

# Set up colors
white = (255, 255, 255)
black = (0, 0, 0)
light_blue = (135, 206, 235)
dark_blue = (25,25,112)
orange = (255, 165, 0)
green = (124,252,0)

s0_x = 0
s1_x = int(width / 4)
s2_x = 2*int(width / 4)
s3_x = 3*int(width / 4)
s4_x = 0
s5_x = int(width / 4)
s6_x = 2*int(width / 4)
s7_x = 3*int(width / 4)
s8_x = 0
s9_x = int(width / 4)
s10_x = 2*int(width / 4)
```

Implementing Q-Learning – Use the Trained Agent in Pygame (cont)

```
s11_x = 3*int(width / 4)
s12_x = 0
s13_x = int(width / 4)
s14_x = 2*int(width / 4)
s15_x = 3*int(width / 4)

s0_y = 0
s1_y = 0
s2_y = 0
s3_y = 0
s4_y = int(height / 4)
s5_y = int(height / 4)
s6_y = int(height / 4)
s7_y = int(height / 4)
s8_y = 2* int(height / 4)
s9_y = 2* int(height / 4)
s10_y = 2* int(height / 4)
s11_y = 2* int(height / 4)
s12_y = 3*int(height / 4)
s13_y = 3*int(height / 4)
s14_y = 3*int(height / 4)
s15_y = 3*int(height / 4)

agent_x, agent_y = s0_x, s0_y

# Set up agent variables

w_cell, h_cell = int(width / 4), int(height / 4)

agent_size = 100
```

Implementing Q-Learning – Use the Trained Agent in Pygame (cont)

```
# Main game loop
First = True
state = 0
done = False
while not done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # Clear the screen
    screen.fill(white)
    if not First:
        a = np.argmax(Q[state, :])
        print("action:",action_list[a])
        next_state, reward, done = get_next_state(state, a, action_list)
        state = next_state
        if next_state == 0:
            agent_x, agent_y = s0_x, s0_y
        if next_state == 1:
            agent_x, agent_y = s1_x, s1_y
        if next_state == 2:
            agent_x, agent_y = s2_x, s2_y
        if next_state == 3:
            agent_x, agent_y = s3_x, s3_y
        if next_state == 4:
            agent_x, agent_y = s4_x, s4_y
        if next_state == 5:
            agent_x, agent_y = s5_x, s5_y
        if next_state == 6:
```

Implementing Q-Learning – Use the Trained Agent in Pygame (cont.)

```
        if next_state == 7:
            agent_x, agent_y = s7_x, s7_y
        if next_state == 8:
            agent_x, agent_y = s8_x, s8_y
        if next_state == 9:
            agent_x, agent_y = s9_x, s9_y
        if next_state == 10:
            agent_x, agent_y = s10_x, s10_y
        if next_state == 11:
            agent_x, agent_y = s11_x, s11_y
        if next_state == 12:
            agent_x, agent_y = s12_x, s12_y
        if next_state == 13:
            agent_x, agent_y = s13_x, s13_y
        if next_state == 14:
            agent_x, agent_y = s14_x, s14_y
        if next_state == 15:
            agent_x, agent_y = s15_x, s15_y

    First = False

    # Draw the agent

    pygame.draw.rect(screen, black, (agent_x, agent_y, agent_size, agent_size))

    # draw the states
    if agent_x != s0_x or agent_y != s0_y:
        pygame.draw.rect(screen, orange, (s0_x, s0_y, w_cell, h_cell))
    if agent_x != s1_x or agent_y != s1_y:
        pygame.draw.rect(screen, light_blue, (s1_x, s1_y, w_cell, h_cell))
```

Implementing Q-Learning – Use the Trained Agent in Pygame (cont.)

```
    if agent_x != s2_x or agent_y != s2_y:
        pygame.draw.rect(screen, light_blue, (s2_x, s2_y, w_cell, h_cell))
    if agent_x != s3_x or agent_y != s3_y:
        pygame.draw.rect(screen, light_blue, (s3_x, s3_y, w_cell, h_cell))
    if agent_x != s4_x or agent_y != s4_y:
        pygame.draw.rect(screen, light_blue, (s4_x, s4_y, w_cell, h_cell))
    if agent_x != s5_x or agent_y != s5_y:
        pygame.draw.rect(screen, dark_blue, (s5_x, s5_y, w_cell, h_cell))
    if agent_x != s6_x or agent_y != s6_y:
        pygame.draw.rect(screen, light_blue, (s6_x, s6_y, w_cell, h_cell))
    if agent_x != s7_x or agent_y != s7_y:
        pygame.draw.rect(screen, dark_blue, (s7_x, s7_y, w_cell, h_cell))
    if agent_x != s8_x or agent_y != s8_y:
        pygame.draw.rect(screen, light_blue, (s8_x, s8_y, w_cell, h_cell))
    if agent_x != s9_x or agent_y != s9_y:
        pygame.draw.rect(screen, light_blue, (s9_x, s9_y, w_cell, h_cell))
    if agent_x != s10_x or agent_y != s10_y:
        pygame.draw.rect(screen, light_blue, (s10_x, s10_y, w_cell, h_cell))
    if agent_x != s11_x or agent_y != s11_y:
        pygame.draw.rect(screen, dark_blue, (s11_x, s11_y, w_cell, h_cell))
    if agent_x != s12_x or agent_y != s12_y:
        pygame.draw.rect(screen, dark_blue, (s12_x, s12_y, w_cell, h_cell))
    if agent_x != s13_x or agent_y != s13_y:
        pygame.draw.rect(screen, light_blue, (s13_x, s13_y, w_cell, h_cell))
    if agent_x != s14_x or agent_y != s14_y:
        pygame.draw.rect(screen, light_blue, (s14_x, s14_y, w_cell, h_cell))
    if agent_x != s15_x or agent_y != s15_y:
        pygame.draw.rect(screen, green, (s15_x, s15_y, w_cell, h_cell))

    # draw horizontal lines
```

Implementing Q-Learning – Use the Trained Agent in Pygame (cont)

```
    pygame.draw.line(screen, black, (0,0), (400,0), width=3)
    pygame.draw.line(screen, black, (0,100), (400,100), width=3)
    pygame.draw.line(screen, black, (0,200), (400,200), width=3)
    pygame.draw.line(screen, black, (0,300), (400,300), width=3)
    pygame.draw.line(screen, black, (0,400), (400,400), width=3)

    # draw vertical lines
    pygame.draw.line(screen, black, (0,0), (0,400), width=3)
    pygame.draw.line(screen, black, (100,0), (100,400), width=3)
    pygame.draw.line(screen, black, (200,0), (200,400), width=3)
    pygame.draw.line(screen, black, (300,0), (300,400), width=3)
    pygame.draw.line(screen, black, (400,0), (400,400), width=3)

    # Update the display
    pygame.display.flip()
    time.sleep(1.5)
```

References

- [1] S. Steinerberger, “On the number of positions in chess without promotion,” *International Journal of Game Theory*, vol. 44, no. 3, pp. 761–767, 2014.
- [2] W. Qiang and Z. Zhongli, “Reinforcement learning model, algorithms and its Application,” *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*, 2011.
- [3] D. J. White, *Markov Decision Processes*. Chichester England: John Wiley & Sons, 1993.
- [4] S. J. Majeed and M. Hutter, “On Q-learning convergence for non-markov decision processes,” *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, 2018.
- [5] M. I. Jordan, Y. LeCun, and S. A. Solla, “Reinforcement Learning in Continuous Action Spaces through Sequential Monte Carlo Methods,” in *Advances in neural information processing systems*, Cambridge, MA: MIT Press, 2001
- [6] W. C. J. C. Hellaby, “Learning from delayed rewards,” thesis, University of Cambridge, 1989