**Project Title:**

**Warehouse Environment + RL with Q-learning**

# **Team Members:**

|  |  |  |
| --- | --- | --- |
| **N** | **Student Name** | **Section Number** |
| **1** | **Youssef Mohammed AbdelFattah Ahmed** | **4** |
| **2** | **Ahmed Mohammed Qasiem AbdElAziz** | **1** |
| **3** | **Maha Mohammed Hamdy** | **4** |

Outlines:

1. Introduction
2. Phases of the Project

2.1 Problem Defining (Game)

2.2 Building Environment

2.3 Training

3.Pictorial Representation

1. Introduction

Reinforcement Learning is enabling agents to learn optimal behaviors through interaction with their environment.

Warehouse game is a grid-based RL game environment, implemented using the Gym library, which provides a rich platform for training and evaluating agents in a simulated world.

GridWorld RL games in general challenges agents to navigate through a dynamic grid environment, where they encounter obstacles and rewards.

Creating a custom environment in Gym involves defining the state space, action space, reward structure, and transition dynamics that govern agent-environment interaction.

By adhering to Gym's standardized interface, custom environments seamlessly integrate with existing RL algorithms and evaluation methodologies, enabling rigorous experimentation and comparison across different domains.

1. Phases of the Project

2.1 Problem Defining (Game)

* The problem is to optimize a robot that works in a Warehouse.
* The Warehouse is divided into a rectangular grid.
* A Target is randomly placed on the grid and the Robot's goal is to reach the Target.
* So, in order to reach the shelves, we need to define somethings first to establish the foundations. As an example, we need to:
  + Define Actions the Robot is capable of performing as going in a certain direction (Right, Left, Up and Down).
  + Initialize the grid size.
  + Identify important Imports from python Libraries and Modules.
* This module enables us to access the images we are using to build the game from files on the PC.

from os import path

* It is a Python module used for designing video games, by allowing computer graphics and sound libraries in order to develop high-quality and user interactive games.

import pygame

* This represents the action space of the robot or what it can do.

class RobotAction(Enum):

LEFT=0

DOWN=1

RIGHT=2

UP=3

* Using Enum module to enumerate the states of the Tiles either to state that it is empty or having the robot or having the target on the tile.

# print the layout of the grid

class GridTile(Enum):

    \_FLOOR=0 # neither target nor the robot

    ROBOT=1     # where the robot is

    TARGET=2   # where the target is

   # Return the first letter of tile name, for printing to the console.

    def \_\_str\_\_(self):

' ' '

overwriting string function to return the first letter of each variable

in the class GradeTile(\_,R,T)

' ' '

return self.name[:1]

* Connecting the grid and the robot actions to form the warehouse and study the actions to be taken.

class WarehouseRobot:

* Initialize the grid size by manually setting it as a value to the function. Pass in an integer seed to make randomness (Targets) repeatable.

def \_\_init\_\_(self, grid\_rows=4, grid\_cols=5, fps=1):

        self.grid\_rows = grid\_rows

        self.grid\_cols = grid\_cols

      self.reset()

* Reset() function make the robot at start position (0,0) whenever the game is started

def reset(self, seed=None):

        # Initialize Robot's starting position

        self.robot\_pos = [0,0]

        # Random generate Target position

# The random number generator needs a number to start with (a seed value), to be able to

# generate a random number.

      random.seed(seed)

* This method initializes all the necessary pygame modules

def \_init\_pygame(self):

pygame.init() # initialize pygame

* It is used help in controlling the frames per second for the rendering.

self.clock = pygame.time.Clock()

* Adjusting the scale of the images used to build the game with the frame(window\_size) size.

self.cell\_height = 64

self.cell\_width = 64

self.cell\_size = (self.cell\_width, self.cell\_height)

file\_name = path.join(path.dirname(\_\_file\_\_), "sprites/bot\_blue.png")

img = pygame.image.load(file\_name)

self.robot\_img = pygame.transform.scale(img, self.cell\_size)

file\_name = path.join(path.dirname(\_\_file\_\_), "sprites/floor.png")

img = pygame.image.load(file\_name)

self.floor\_img = pygame.transform.scale(img, self.cell\_size)

file\_name = path.join(path.dirname(\_\_file\_\_), "sprites/package.png")

img = pygame.image.load(file\_name)

self.goal\_img = pygame.transform.scale(img, self.cell\_size)

self.window\_size = (self.cell\_width \* self.grid\_cols, self.cell\_height \* self.grid\_rows + self.action\_info\_height)

* Display the game for the user

  self.window\_surface = pygame.display.set\_mode(self.window\_size)

* Randomly generate the position of the parcel according to the number of rows and columns passed values in the \_\_init\_\_ function

self.target\_pos = [

          random.randint(1, self.grid\_rows-1),

          random.randint(1, self.grid\_cols-1)

       ]

def \_\_init\_\_(self, grid\_rows=3, grid\_cols=3, fps=1):

def perform\_action(self, robot\_action:RobotAction) -> bool:

        self.last\_action = robot\_action

        # Move Robot to the next cell

        # make sure that the robot stay on the grid

        if robot\_action == RobotAction.LEFT:

            if self.robot\_pos[1]>0:

                self.robot\_pos[1]-=1

        elif robot\_action == RobotAction.RIGHT:

            if self.robot\_pos[1]<self.grid\_cols-1:

                self.robot\_pos[1]+=1

        elif robot\_action == RobotAction.UP:

            if self.robot\_pos[0]>0:

                self.robot\_pos[0]-=1

        elif robot\_action == RobotAction.DOWN:

            if self.robot\_pos[0]<self.grid\_rows-1:

                self.robot\_pos[0]+=1

        # Return true if Robot reaches Target

return self.robot\_pos == self.target\_pos and print("Target is found")

* This represents the motion of the row either left or right.

robot\_pos[1]-

* This represents the motion of the column either UP or Down.

robot\_pos[0]

* This ensures that the robot stays inside the grid frame.

if self.robot\_pos[1]<self.grid\_cols-1:

* If the position of the target is the same as the robot then the target is achieved and would print it in the terminal.

return self.robot\_pos == self.target\_pos and print("Target is found")

* This would print the position of the target and the robot relative to each other in the grid for observation.

for r in range(self.grid\_rows):

            for c in range(self.grid\_cols):

                if([r,c] == self.robot\_pos):

                    print(GridTile.ROBOT, end=' ')

                elif([r,c] == self.target\_pos):

                    print(GridTile.TARGET, end=' ')

                else:

                    print(GridTile.\_FLOOR, end=' ')

* This class returns the first letter of variables in the class by overwriting string function to visualize each step taken.

GridTile.ROBOT,

* It iterates over each cell in the grid using nested loops, where r iterates over rows and c iterates over columns.

For each cell, it draws the floor using the self.floor\_img image at the calculated position pos.

* If the current cell [r, c] matches the position of the target (self.target\_pos), it draws the target (self.goal\_img) at the same position pos.
* If the current cell [r, c] matches the position of the robot (self.robot\_pos), it draws the robot (self.robot\_img) at the same position pos.

It renders the text indicating the last action taken (self.last\_action) using a specified font (self.action\_font). The rendered text is then blitted onto the window surface at the bottom left corner of the window.

The blit method in the pygame library is used to draw one image onto another

for r in range(self.grid\_rows):

            for c in range(self.grid\_cols):

                # Draw floor

                pos = (c \* self.cell\_width, r \* self.cell\_height)

                self.window\_surface.blit(self.floor\_img, pos)

                if([r,c] == self.target\_pos):

                    # Draw target

                    self.window\_surface.blit(self.goal\_img, pos)

                if([r,c] == self.robot\_pos):

                    # Draw robot

                    self.window\_surface.blit(self.robot\_img, pos)

        text\_img = self.action\_font.render(f'Action: {self.last\_action}', True, (0,0,0), (255,255,255))

        text\_pos = (0, self.window\_size[1] - self.action\_info\_height)

        self.window\_surface.blit(text\_img, text\_pos)

* Allows to update portions of the screen. Passing no arguments, updates the entire display after changing the position of the robot.

pygame.display.update()

* pygame.QUIT, it means the user clicked on the X at the top right corner of the window, indicating the intention to close the window. In response, the program gracefully shuts down by calling pygame.quit() to uninitialize all pygame modules and sys.exit() to exit the program.

If the pressed key is the Escape key (pygame.K\_ESCAPE), the program also shuts down gracefully by calling pygame.quit() and sys.exit().

def \_process\_events(self):

        # Process user events, key presses

        for event in pygame.event.get():

            # User clicked on X at the top right corner of window

            if event.type == pygame.QUIT:

                pygame.quit()

                sys.exit()

            if(event.type == pygame.KEYDOWN):

                # User hit escape

                if(event.key == pygame.K\_ESCAPE):

                    pygame.quit()

                    sys.exit()

2. 2. Building Environment

* Once the game is constructed, the next phase focuses on adapting it into a reinforcement learning environment using the Gym library.
* This entails translating the game mechanics into a format compatible with Gym's standardized interface, defining the observation space, action space, and reward structure.
* Additionally, customizing the environment may involve fine-tuning parameters to optimize the learning process and balance exploration and exploitation.
* **Firstly, Import necessary modules from Gymnasium:**

import gymnasium as gym

from gymnasium import spaces

from gymnasium.envs.registration import register

from gymnasium.utils.env\_checker import check\_env

* These lines import essential modules from Gymnasium, a toolkit for developing and comparing reinforcement learning algorithms.
* **Importing Warehouse Robot Environment Implementation:**

import warehouseRobot as wr

import numpy as np

* This imports the implementation of the warehouse robot environment from the warehouseRobot module and the NumPy library for numerical operations.
* **Registering the Warehouse Robot Environment with Gymnasium:**

register(

    id='warehouse-robot-v0',

    entry\_point='warehouseRobotEnv:WarehouseRobotEnv',

)

* This code registers the WarehouseRobotEnv class as a Gym environment with the ID 'warehouse-robot-v0'. This allows it to be accessed using *gym.make('warehouse-robot-v0').*
* **Defining Warehouse Robot Environment Class:**

class WarehouseRobotEnv(gym.Env):

* + Defines a new class WarehouseRobotEnv which inherits from gym.Env, indicating that it's a Gym environment.
* **Initializing Warehouse Robot Environment:**

def \_\_init\_\_(self, grid\_rows=4, grid\_cols=5, render\_mode=None):

* + This method initializes the Warehouse Robot environment with default grid size of 4x5, and an optional rendering mode.
* **Resetting the Environment:**

def reset(self, seed=None, options=None):

* + This method resets the environment, optionally specifying a seed for reproducibility and additional options.
* **Taking a Step in the Environment:**

def step(self, action):

* + This method performs an action in the environment and returns the new observation, reward, termination status, and additional information.
* **Rendering the Environment:**

def render(self):

* + This method renders the Warehouse Robot environment, displaying the current state.
* **Creating an Instance of the Environment for Unit Testing:**

if \_\_name\_\_=="\_\_main\_\_":

* + This block of code creates an instance of the environment for unit testing purposes.
  + **Checking the Environment for Consistency:**

check\_env(env.unwrapped)

* + This line checks the custom environment for consistency using Gym's check\_env function.
* **Resetting the Environment and Taking Random Actions:**

obs = env.reset()[0]

* + This line resets the environment and retrieves the initial observation.

rand\_action = env.action\_space.sample()

obs, reward, terminated, \_, \_ = env.step(rand\_action)

* + These lines take random actions in the environment, and retrieve the new observation, reward, and termination status.

2. 3. Training the Model using Q-Learning

With the environment in place, the final phase revolves around training the reinforcement learning model using the Q-learning algorithm.

Q-learning is a model-free RL algorithm that learns the optimal action-value function by iteratively updating Q-values based on observed rewards and transitions.

During training, the agent interacts with the environment, selecting actions according to an exploration strategy (e.g., epsilon-greedy) and updating Q-values using the Bellman equation.

Through repeated episodes of exploration and learning, the agent gradually improves its policy, ultimately converging towards an optimal strategy for navigating the environment and maximizing cumulative rewards.

Firstly, we need to identify important Imports from python Libraries and Modules:

* Imports the gym library, which is used to create and manage the custom environment warehouse-robot-v0.

import gymnasium as gym

* Imports the numpy library for numerical operations, including the creation of the Q-table and calculations.

import numpy as np

* Imports matplotlib for plotting the average steps per episode over time, providing a visual representation of the agent's performance.

import matplotlib.pyplot as plt

* Imports the random module for generating random actions during training, specifically when using the epsilon-greedy strategy.

import random

* Imports the pickle module for saving and loading the Q-table to and from disk, allowing for persistence of the trained model.

import pickle

* Imports the custom environment class, even though it's not directly used in the script, to ensure the environment is registered.

import warehouseRobotEnv

* **Function:** run\_q()
* **Purpose:** Trains or tests an agent in the warehouse-robot-v0 environment using Q-Learning.
* **Parameters:**

episodes: Number of episodes to run.

is\_training: Boolean flag indicating whether to train or test the agent.

render: Boolean flag to control rendering of the environment during episodes.

* **Process:**
* Initializes the environment and creates a Q-table based on the environment's grid dimensions and action space.
* For each episode, the agent selects actions based on the epsilon-greedy strategy, performs actions, and updates the Q-table based on the rewards received.
* Tracks the number of steps taken per episode to evaluate the efficiency of the agent.
* After training, saves the Q-table to a file and plots the average steps per episode over time.

def run\_q(episodes, is\_training=True, render=False):

* Environment Creation: Creates an instance of the warehouse-robot-v0 environment using gym.make. The render\_mode is set to 'human' if render is True, otherwise it's None.

env = gym.make('warehouse-robot-v0', render\_mode='human' if render else None)

* Q-Table Initialization: If is\_training is True, initializes the Q-table as a 5D array with zeros. If is\_training is False, loads the Q-table from a file named 'v0\_warehouse\_solution.pkl'.

    if(is\_training):

        q = np.zeros((env.unwrapped.grid\_rows, env.unwrapped.grid\_cols, env.unwrapped.grid\_rows, env.unwrapped.grid\_cols, env.action\_space.n))

    else:

        f = open('v0\_warehouse\_solution.pkl', 'rb')

        q = pickle.load(f)

        f.close()

* Hyperparameters: Sets the learning rate (learning\_rate\_a), discount factor (discount\_factor\_g), and epsilon for the epsilon-greedy strategy.

learning\_rate\_a = 0.9

discount\_factor\_g = 0.9

epsilon = 1

* Steps Tracker: Initializes an array to track the number of steps taken per episode.

  steps\_per\_episode = np.zeros(episodes)

* Episode Loop: Begins a loop to run each episode.

step\_count=0

for i in range(episodes):

* Rendering: Prints the current episode number if render is True.

if(render):

            print(f'Episode {i}')

* State Reset: Resets the environment to get the initial state and sets a flag terminated to False.

state = env.reset()[0]

terminated = False

* Action Selection Loop: Enters a loop where actions are selected and performed until the episode ends.

while(not terminated):

* Action Selection: Uses the epsilon-greedy strategy to select an action. If in training mode and a random number is less than epsilon, a random action is chosen. Otherwise, the action with the highest Q-value for the current state is selected.

if is\_training and random.random() < epsilon:

                action = env.action\_space.sample()

else:

                q\_state\_idx = tuple(state)

action = np.argmax(q[q\_state\_idx])

* Perform Action: Performs the selected action and receives the new state, reward, and termination status.

new\_state,reward,terminated,\_,\_ = env.step(action)

* Indexing: Prepares indices for accessing and updating the Q-table.

q\_state\_action\_idx = tuple(state) + (action,)

q\_new\_state\_idx = tuple(new\_state)

* Q-Table Update: Updates the Q-value for the selected action based on the reward and the maximum Q-value for the new state.

if is\_training:

          q[q\_state\_action\_idx] = q[q\_state\_action\_idx] + learning\_rate\_a \* (

                       reward + discount\_factor\_g \* np.max(q[q\_new\_state\_idx]) - q[q\_state\_action\_idx] )

* State Update and Steps Recording: Updates the current state, increments the step count, and records the steps taken if the episode has ended.

state = new\_state

step\_count+=1

if terminated:

steps\_per\_episode[i] = step\_count

step\_count = 0

* Epsilon Decay: Decreases epsilon by a small amount after each episode to gradually shift from exploration to exploitation.

epsilon = max(epsilon - 1/episodes, 0)

* Environment Closure: Closes the environment after all episodes have been completed.

  env.close()

* Performance Evaluation: Calculates the average steps per episode over the last 100 episodes and plots them. Saves the plot as 'v0\_warehouse\_solution.png'.

    sum\_steps = np.zeros(episodes)

    for t in range(episodes):

        sum\_steps[t] = np.mean(steps\_per\_episode[max(0, t-100):(t+1)])

    plt.plot(sum\_steps)

    plt.savefig('v0\_warehouse\_solution.png')

* Q-Table Saving: If in training mode, saves the final Q-table to a file named 'v0\_warehouse\_solution.pkl'.

    if is\_training:

        f = open("v0\_warehouse\_solution.pkl","wb")

        pickle.dump(q, f)

        f.close()

* **Main Execution Block**

**Purpose:** Executes the run\_q function for both training and testing phases.

**Training Phase:** Runs the agent through 1000 episodes to train it using Q- Learning.

**Testing Phase:** Runs the agent through a single episode to test its performance after training.

if \_\_name\_\_ == '\_\_main\_\_':

* Training Phase: Calls the run\_q function with 1000 episodes, setting is\_training to True to indicate that the agent should be trained. The render parameter is set to False, meaning the environment will not be rendered during training.

run\_q(1000, is\_training=True, render=False)

* Testing Phase: Calls the run\_q function with 1 episode, setting is\_training to False to indicate that the agent should be tested. The render parameter is set to True, meaning the environment will be rendered during testing to visually observe the agent's performance.

run\_q(1, is\_training=False, render=True)

3.Pictorial Representation

Target:

A box with a blue tape

Description automatically generated

Robot:



Floor to build the grid:



Robot trying to find the target:

A screenshot of a computer

Description automatically generated

A computer screen shot of a game

Description automatically generated

Robot found the target:

A screenshot of a video game

Description automatically generated

[*See Source Code On GitHub*](https://github.com/Youssef-Mohammed72/Warehouse-Robot)