Picker Route Model Documentation (PoC)

Table of Contents

1.Introduction

Reinforcement Learning (RL), a machine learning technique along with Streamlit, a web application framework, is used to create a Picker route model for Warehouse Management. The model accepts the input parameters from the user to create a dynamic warehouse layout and calculates the optimal distance coordinates that cover all the items in the order list.

2.Getting Started

For this project we are developing a custom environment RL model with PPO policy along with its actions and rewards. We make an instance of this environment, pass our input parameters and display the output via localhost.

3.Prerequisites

3.1 Hardware Requirements:

OS: Windows

RAM: 4GB

3.2 Software Requirements:

IDE: VSCode/Any

Application:

Anaconda (optional)

4. Installation

4.1 Conda Virtual Environment/ Python Virtual Environment

Anaconda Installation:

Install Anaconda. Create a Conda Virtual Environment using a Terminal or Anaconda Prompt. Give a name for your environment in ‘myenv’.

-> conda create –n myenv python=3.9

Activate the conda environment in VSCode using:

-> conda activate myenv

Python Venv Installation:

Create a virtual environment in python to install the required packages.

-> python -m venv /path/to/new/virtual/environment

4.2 Packages

Download the required packages either by creating a requirements.txt file or by using the pip command separately. Packages:

→ pip install pandas
→ pip install numpy
→ pip insall streamlit
→ pip install gym #For creating Custom RL environment
→ pip install stable-baselines3 #Reinforcement Learning Package
→ pip install matplotlib

5. Model Code Logic Overview

The article: https://towardsdatascience.com/reinforcement-learning-with-python-part-1-creating-the-environment-dad6e0237d2d ,provides a comprehensive undestanding of the working of the

RL model with PPO (Proximal Policy Optimization). PPO assigns a reward (1) for the optimal coordinates the model picks each time. There are three components in the code:

1. Create a custom Gym Environment for our model (Warehouse Layout Creation-Dynamically)
2. Training the Model with necessary parameters after creating an environment instance
3. Using Matplotlib to plot the Warehouse Layout and Optimal Path
4. Using Streamlit to display the Warehouse Layout and the Optimal Path

5.1 Creating Custom Gym Environment

The article [1] gives a comprehensive understanding of creating a custom gym environment. Through following the steps provided in the article, we will:

→ Define a custom warehouse code by passing the paramaters: ‘size’, ‘num_racks’, ‘rack_width’, to create the warehouse layout dynamically for each instance.

Code:

```
class CustomWarehouseEnv(gym.Env):
  def __init__(self, size, num_racks, rack_width):
    self.size = size                                   # Dynamically accept the dimensions of the Warehouse
    self.num_racks = num_racks                         # Dynamically accept the no.of.racks
    self.rack_width = rack_width                       # Dynamically accept the rack width
    self.grid = np.zeros((self.size, self.size), dtype=int) # Numpy array of zeros for the warehouse dimensions
    self.picker_position = (0, 0)
    self.action_space = gym.spaces.Discrete(4)         # Four types of action are possible by the picker
    self.observation_space = gym.spaces.Box(
      low=0, high=2, shape=(self.size, self.size, 1), dtype=np.uint8       #Obervations in the grid will be numbered
as 0,1,2 (racks and aisles)
    )
    self.target_items = []                             # Initialize the list of items to be picked
```

→ Define the walking actions that can happen inside a warehouse (Left ,Right ,Up ,Down)

Code:

```
  def step(self, action):                              # Defines the possible movements of the picker
    dx, dy = 0, 0
    if action == 0:  # left
      dx -= 1
    elif action == 1:  # right
      dx += 1
    elif action == 2:  # up
      dy -= 1
    elif action == 3:  # down
      dy += 1
```

```
    new_x = min(max(0, self.picker_position[0] + dx), self.size - 1)           #Position within warehouse bounds
    new_y = min(max(0, self.picker_position[1] + dy), self.size - 1)


    if self.grid[new_x, new_y] != 1:                # Check if the new position is in a rack area, and if so, don't move
      self.picker_position = (new_x, new_y)


    self.update_observation()                   #Update the position of picker at that instance

    reward = 1 if self.picker_position in self.target_items else 0      #Assign the reward if the picker picks the item
    done = len(self.target_items) == 0                     #True if all items were picked
    # Return info dict
    info = {'episode': {}}                                 #Empty dict to avoid parameters missing error
    return self._observe(), reward, done, info

  def reset(self):                                         #Reset env to its initial state
    self.grid = np.zeros((self.size, self.size), dtype=int)
    self.picker_position = (0, 0)
    return self._observe()

  def _observe(self):                                      #Creates current obs of the env, assign 2 to picker position
    obs = np.copy(self.grid)
    obs[self.picker_position[0], self.picker_position[1]] = 2
    return obs.reshape((self.size, self.size, 1))


  def update_observation(self):                            #Update the obs, assign 2, assign 3 to items to be picked
    obs = np.copy(self.grid)
    obs[self.picker_position[0], self.picker_position[1]] = 2
    for item in self.target_items:
      obs[item[0], item[1]] = 3
    return obs


  def render(self, mode='human'):                                          #Used for rendering the environment
    if mode == 'rgb_array':
      return self.grid
```

→ Define a Path Finding Algorithm for finding the optimal coordinates. A* heuristics algorithm is used here

Code:

```
 def calculate_optimal_route_from_position(self, start_position): #Use a suitable pathfinding algorithm
    def astar(grid, start, targets):                                          #A* Algorithm is used for this project
      def heuristic(node):                        #Calculate the heuristics cost from picker position to all the target items
```

```
    return min(abs(node[0] - target[0]) + abs(node[1] - target[1]) for target in targets)


def reconstruct_path(came_from, current):
    path = []
    while current in came_from:
        path.insert(0, current)
        current = came_from[current]
    return path


open_set = PriorityQueue()                    #Use a priority queue to store the info of visited nodes
open_set.put((0, start))
came_from = {}
g_score = {(x, y): float("inf") for x in range(len(grid))
           for y in range(len(grid[0]))}


g_score[start] = 0


while not open_set.empty():
    _, current = open_set.get()


    if current in targets:
        return reconstruct_path(came_from, current)


    for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:                  #Explore all possible actions to cover the target
items
        new_x, new_y = current[0] + dx, current[1] + dy


        if (
            0 <= new_x < len(grid)
            and 0 <= new_y < len(grid[0])
            and grid[new_x][new_y] != 1
        ):
            tentative_g_score = g_score[current] + 1


            if tentative_g_score < g_score[(new_x, new_y)]:
                came_from[(new_x, new_y)] = current
                g_score[(new_x, new_y)] = tentative_g_score
                f_score = tentative_g_score + \
                    heuristic((new_x, new_y))
                open_set.put((f_score, (new_x, new_y)))

return []                                        # If no valid path is found, return an empty list
```

```
grid_copy = np.copy(self.grid)                                          # Make a copy of the grid

for item in self.target_items:                          # Set the positions of items to be picked as aisles (2)
    grid_copy[item[0], item[1]] = 2
optimal_route = []                     # Calculate the optimal route from the specified picker's position to items
current_position = start_position


while self.target_items:
    path = astar(grid_copy, current_position, self.target_items)       #Loop over the A* Search for target items
    if not path:
        print("No path found to remaining items.")                   # Handle the case where no path was found
        break

    next_position = path[1] if len(path) > 1 else path[0]               # Ensure there is a valid next position
    optimal_route.extend(path[:-1])


    if next_position in self.target_items:
        self.target_items.remove(next_position)


    current_position = next_position


# Calculate the optimal distance
optimal_distance = len(optimal_route)


return optimal_route, optimal_distance
```

5.2 Training the Model

→ Accept the Grid parameters dynamically from the user
→ Create an instance of Custom Gym Environment

Code:
```
                                                                       # Streamlit UI
st.title("Warehouse Picker Simulation")                                # Dynamic Input Assignment
size = st.number_input(f"Warehouse Dimensions", value=19, min_value=1)
num_racks = st.number_input(f"No. of Racks", value=6, min_value=1)
rack_width = st.number_input(f"Rack Width", value=2, min_value=1)
env = CustomWarehouseEnv(size, num_racks, rack_width)
```

→ Train the model with your desired timesteps

Code:
```
# Initialize the model with MlpPolicy
model = PPO("MlpPolicy", env, verbose=0)
model.learn(total_timesteps=100)                              # You can adjust the number of timesteps
```

```
grid = np.zeros((size, size), dtype=int)                                   # Create an empty grid

rack_spacing = (size - (num_racks * rack_width) - 2) // (num_racks - 1)  # Place the racks and aisles in the grid
bottom_padding = 1                                            # Number of grids to pad at the bottom
top_padding = 1                                               # Number of grids to pad at the top


rack_labels = iter("ABCDEFGHIJKLMNOPQRSTUVWXYZ")                  # Use alphabet labels for Racks
rack_label_dict = {}                                          # Create a dictionary to store rack labels


for i in range(num_racks):                                    # Start the rack after the first aisle space
    rack_x = i * (rack_width + rack_spacing) + 1
    grid[bottom_padding:size - top_padding, rack_x: rack_x + rack_width] = 1

    rack_label = next(rack_labels)                            # Assign the next alphabet label to the rack

    rack_label_dict[rack_label] = (                             # Update the grid with the rack label
        bottom_padding, size - top_padding, rack_x, rack_x + rack_width)
    grid[bottom_padding:size - top_padding, rack_x: rack_x +
         rack_width] = 1  # Indicate the rack with 1

picker_x, picker_y = size - 1, 0                                  # Initialize the picker's position
```

5.3 Using Matplotlib and Streamlit

By using Matplotlib package, the Warehouse grid will be plotted. Refer the official Streamlit documentation [2] for a comprehensive understanding.

→ Create a colormap and legends for the warehouse layout
→ Display the plotted figure

Code:

```
                              # Create legends for aisles, racks, picker, items, and optimal route
legends = {
    0: "Aisles",
    1: "Racks",
    2: "Picker",
    3: "Items",  # Items to be picked
    4: "Optimal Route from Current Position",
}

                                              # Display the environment in Streamlit
st.title("Custom Warehouse Environment")


                                  # Allow the user to dynamically specify the picker's location
st.sidebar.header("Picker's Position")
user_picker_x = st.sidebar.selectbox(
    "X-coordinate (0-18)", list(range(size)))
user_picker_y = st.sidebar.selectbox(
```

```
    "Y-coordinate (0-18)", list(range(size)))

                                              # Check if the user-selected picker's position is valid (not in a rack)
if grid[user_picker_x, user_picker_y] != 1:
    # Update the picker's position
    picker_x, picker_y = user_picker_x, user_picker_y
    grid[picker_x, picker_y] = 2


                                                            # Allow the user to input the orders in this format
st.sidebar.header("Input Orders")
order_json = st.sidebar.text_area(
    "Enter orders in this format", '[ [3, 4], [6, 7] ]')
                                                                   # Parse the input to get the list of orders
try:
    orders = json.loads(order_json)
except json.JSONDecodeError:
    orders = []

                                                                          # Clear previously picked items
env.target_items = []



                                                     # Validate and display the items to be picked on the grid
for order in orders:
    if isinstance(order, list) and len(order) == 2:
        item_x, item_y = order
        if 0 <= item_x < size and 0 <= item_y < size and grid[item_x, item_y] == 1:
            env.target_items.append((item_x, item_y))
            grid[item_x, item_y] = 3  # Display items on the grid

                                                                   # Calculate the optimal picking route
if env.target_items:
    optimal_route, optimal_distance = env.calculate_optimal_route_from_position(
        (picker_x, picker_y))

                                                     # Mark the optimal route coordinates in yellow on the grid
    for coord in optimal_route:
        grid[coord[0], coord[1]] = 4  # Yellow color for optimal route

                                                                   # Display the optimal route and distance
st.write(f"Optimal Picking Route: {optimal_route}")
st.write(f"Optimal Distance from Picker: {optimal_distance}")

                    # Create a colormap to display aisles, racks, picker, items, and optimal route in different colors
cmap = plt.cm.colors.ListedColormap(
    ['white', 'gray', 'green', 'blue', 'yellow'])

fig, ax = plt.subplots(figsize=(8, 8))
ax.imshow(grid, cmap=cmap, extent=[
    0, size, size, 0], origin="upper")
ax.set_xticks(np.arange(0, size, 1))
ax.set_yticks(np.arange(0, size, 1))
ax.grid(color='r', linewidth=2)
ax.set_aspect('equal')
                                                                          # Add rack labels to the grid
```

```
for label, (y1, y2, x1, x2) in rack_label_dict.items():
    ax.text((x1 + x2) / 2, (y1 + y2) / 2, label,
            ha='center', va='center', fontsize=12, fontweight='bold', color='black')

                                                        # Create legends as small squares
legend_elements = [
    Patch(facecolor='white', edgecolor='black', label=legends[0]),
    Patch(facecolor='gray', edgecolor='black', label=legends[1]),
    Patch(facecolor='green', edgecolor='black', label=legends[2]),
    Patch(facecolor='blue', edgecolor='black', label=legends[3]),
    Patch(facecolor='yellow', edgecolor='black',label=legends[4]),
]
                                                              # Display legends
ax.legend(handles=legend_elements, loc='upper right', fontsize='medium')
st.pyplot(fig)
```

6. Running the Code

To run the python script:

→ Activate your virtual environment
→ Run: streamlit run model_filename.py

7. References:

1. https://towardsdatascience.com/reinforcement-learning-with-python-part-1-creating-the-environment-dad6e0237d2d
2. https://docs.streamlit.io/