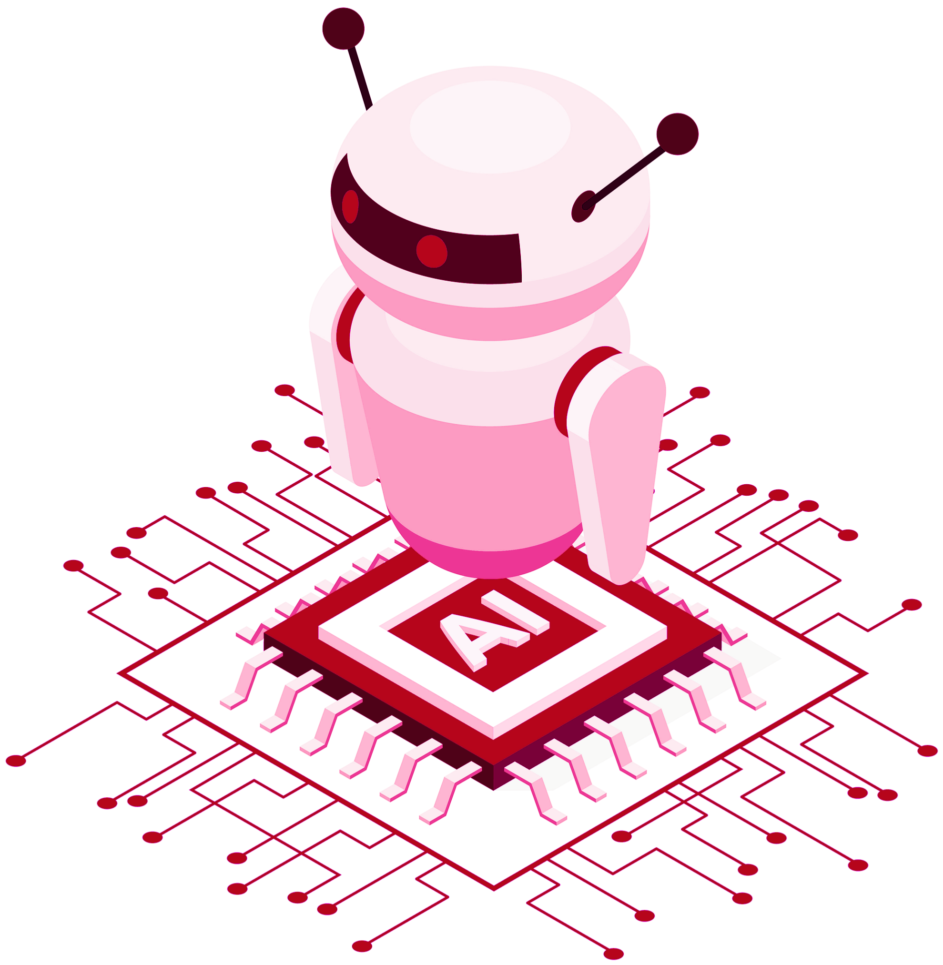

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 1:

Introduction to RL

By:

Taha Majlesi

810101504

Spring 2025

Contents

1 Task 1: Solving Predefined Environments [45-points] 1
1.1 Environment Selection and Implementation 1
1.2 CartPole Environment Analysis 1
1.2.1 Environment Setup and Training 1
1.2.2 Hyperparameter Analysis 1
1.2.3 Reward Wrapping 2
1.3 FrozenLake Environment Analysis 2
1.3.1 Algorithm Comparison 2
1.3.2 Custom Reward Function 2
1.4 Taxi Environment Analysis 3
1.4.1 Q-Learning Implementation 3
1.4.2 Agent Visualization 4
1.5 FlappyBird Environment Analysis 4
1.5.1 Environment Setup 4
1.5.2 Algorithm Training 4
1.6 Results and Analysis 5
1.6.1 Performance Comparison 5
1.6.2 Hyperparameter Impact 5
1.6.3 Reward Wrapping Effects 5
2 Task 2: Creating Custom Environments [45-points] 6
2.1 MDP Formulation 6
2.1.1 State Space (S) 6
2.1.2 Action Space (A) 6
2.1.3 Reward Function (R) 6
2.1.4 Transition Probability (P) 6
2.2 Environment Implementation 7
2.2.1 Gymnasium API Implementation 7
2.3 Q-Learning Implementation 8
2.3.1 Q-Learning Algorithm 8
2.3.2 Agent Simulation 9
2.4 Algorithm Comparison 10
2.4.1 Q-Learning Performance 10
2.4.2 Results Analysis 10
2.4.3 Performance Metrics 10
2.5 Environment Validation 11
2.5.1 State Space Verification 11

2.5.2 Action Space Validation ... 11
2.5.3 Reward Function Testing ... 11
2.6 Conclusion ... 11
3 Task 3: Pygame for RL environment [20-points] 12
3.1 Pygame Environment Design ... 12
3.1.1 Environment Structure ... 12
3.1.2 Basic Pygame Environment Template ... 12
3.2 Advanced Pygame Environment Features ... 14
3.2.1 Multi-Agent Support ... 14
3.2.2 Dynamic Environment Elements ... 15
3.3 Training RL Agents on Pygame Environments ... 16
3.3.1 Environment Integration ... 16
3.3.2 Visualization and Monitoring ... 16
3.4 Benefits of Pygame RL Environments ... 17
3.4.1 Advantages ... 17
3.4.2 Applications ... 17
3.5 Implementation Considerations ... 17
3.5.1 Performance Optimization ... 17
3.5.2 Environment Design ... 17
3.6 Conclusion ... 18

Grading

The grading will be based on the following criteria, with a total of 100 points:

Task	Points
Task 1: Solving Predefined Environments	45
Task 2: Creating Custom Environments	45
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus 1: Writing a wrapper for a known env	10
Bonus 2: Implementing pygame env	20
Bonus 3: Writing your report in Latex	10

Notes:

- Include well-commented code and relevant plots in your notebook.
- Clearly present all comparisons and analyses in your report.
- Ensure reproducibility by specifying all dependencies and configurations.

1 Task 1: Solving Predefined Environments [45-points]

In this task, we explore reinforcement learning by training agents on predefined environments using Stable-Baselines3. We implement and compare different RL algorithms including A2C, PPO, DQN, and Q-Learning across multiple environments.

1.1 Environment Selection and Implementation

We selected four environments for comprehensive analysis:

- **CartPole-v1**: A classic control problem where the agent must balance a pole on a cart
- **FrozenLake-v1**: A gridworld navigation problem with slippery surfaces
- **Taxi-v3**: A more complex navigation problem requiring pickup and dropoff
- **FlappyBird-v0**: A custom environment simulating the popular Flappy Bird game

1.2 CartPole Environment Analysis

1.2.1 Environment Setup and Training

The CartPole environment was trained using A2C and PPO algorithms:

```
# A2C Training
from stable_baselines3 import A2C, PPO, DQN
import gymnasium as gym

env = gym.make("CartPole-v1")
model = A2C("MlpPolicy", env, verbose=1,
            tensorboard_log="./a2c_CartPole_tensorboard/")
model.learn(total_timesteps=500_000)
model.save("a2c_cartpole")

# PPO Training
model = PPO("MlpPolicy", env, verbose=1,
            tensorboard_log="./PPO_CartPole_tensorboard/")
model.learn(total_timesteps=500_000)
model.save("ppo_cartpole")
```

1.2.2 Hyperparameter Analysis

We conducted hyperparameter tuning experiments with different learning rates and discount factors:

```
learning_rates = [0.0001, 0.01]
gammas = [0.95, 0.99]

for lr in learning_rates:
    for gamma in gammas:
        model = PPO("MlpPolicy", env, verbose=0,
                   learning_rate=lr, gamma=gamma,
                   tensorboard_log=f'./PPO_CartPole_tensorboard/ppo_lr{lr}_gamma{gamma}')
        model.learn(total_timesteps=200_000)
```

1.2.3 Reward Wrapping

We implemented a reward wrapper to modify the reward function:

```
from gymnasium import RewardWrapper

class DoubledReward(RewardWrapper):
    def reward(self, reward):
        return reward * 2

wrapped_env = DoubledReward(env)
model = PPO("MlpPolicy", wrapped_env, verbose=0,
           tensorboard_log="./PPO_CartPole_tensorboard/reward_wrapped_env_ppo")
model.learn(total_timesteps=500_000)
```

1.3 FrozenLake Environment Analysis

1.3.1 Algorithm Comparison

For FrozenLake, we compared DQN and A2C algorithms:

```
env = gym.make("FrozenLake-v1", is_slippery=True)

# DQN Training
model = DQN("MlpPolicy", env, verbose=0,
           tensorboard_log="./DQN_FrozenLake_tensorboard/")
model.learn(total_timesteps=500_000)

# A2C Training
model = A2C("MlpPolicy", env, verbose=0,
           tensorboard_log="./A2C_FrozenLake_tensorboard/")
model.learn(total_timesteps=500_000)
```

1.3.2 Custom Reward Function

We implemented a modified reward function for FrozenLake:

```
class ModifiedFrozenLakeReward(RewardWrapper):
    def reward(self, reward):
        if reward == 0:
            return -0.01
        elif reward == 1:
            return 2
        else:
            return reward
```

```
wrapped_env = ModifiedFrozenLakeReward(env)
model = DQN("MlpPolicy", wrapped_env, verbose=0,
           tensorboard_log="./DQN_FrozenLake_tensorboard/reward_wrapped_env_DQN")
model.learn(total_timesteps=500_000)
```

1.4 Taxi Environment Analysis

1.4.1 Q-Learning Implementation

For the Taxi environment, we implemented a custom Q-Learning algorithm as it proved more effective than DQN and PPO:

```
def qlearning_train(taxi_env, learning_rate, discount_factor):
    writer = SummaryWriter(f"Qlearning_Taxi/taxi_experiment_lr_{learning_rate}_{discount_f

    num_states = taxi_env.observation_space.n
    num_actions = taxi_env.action_space.n
    q_values = np.zeros((num_states, num_actions))

    alpha = learning_rate
    gamma = discount_factor
    eps = 1.0
    decay = 0.005
    episodes = 5000
    max_steps_per_ep = 99

    episode_rewards = []

    for ep in range(episodes):
        state, _ = taxi_env.reset()
        total_reward = 0

        for step in range(max_steps_per_ep):
            if random.random() < eps:
                chosen_action = taxi_env.action_space.sample()
            else:
                chosen_action = np.argmax(q_values[state, :])

            next_state, reward, done, _, _ = taxi_env.step(chosen_action)

            q_values[state, chosen_action] += alpha * (
                reward + gamma * np.max(q_values[next_state, :]) - q_values[state, chosen_
            )

            state = next_state
            total_reward += reward

            if done:
                break
        eps = 1 / (1 + decay * ep)
        episode_rewards.append(total_reward)

        avg_reward = np.mean(episode_rewards[-100:]) if ep >= 100 else np.mean(episode_rew
        writer.add_scalar("Mean_Episode_Reward", avg_reward, ep)

    return q_values
```

1.4.2 Agent Visualization

We implemented a simulation function to visualize the trained agent:

```
def simulate_agent(environment, q_matrix, num_runs=5, max_steps_per_run=100):
    for run in range(num_runs):
        current_state, _ = environment.reset()
        finished = False

        for step in range(max_steps_per_run):
            print(environment.render())
            sleep(0.5)

            chosen_action = np.argmax(q_matrix[current_state, :])
            next_state, reward, finished, _, _ = environment.step(chosen_action)
            current_state = next_state

            if finished:
                print(f"Run completed in {step + 1} steps.\n")
                break
```

1.5 FlappyBird Environment Analysis

1.5.1 Environment Setup

We used a third-party FlappyBird implementation:

```
!pip install flappy-bird-gymnasium

import flappy_bird_gymnasium
env = gymnasium.make("FlappyBird-v0", render_mode="rgb_array", use_lidar=True)

# Environment properties
print(f"Action Space: {env.action_space}")  # Discrete(2)
print(f"Observation Space: {env.observation_space}")  # Box(0.0, 1.0, (180,), float64)
```

1.5.2 Algorithm Training

We trained A2C and PPO agents on FlappyBird:

```
# A2C Training
```

```
model = A2C("MlpPolicy", env, verbose=0,
            tensorboard_log="./A2C_FlappyBird_tensorboard/")
model.learn(total_timesteps=5_000_000)

# PPO Training
model = PPO("MlpPolicy", env, verbose=0,
            tensorboard_log="./PPO_mlp_FlappyBird_tensorboard/")
model.learn(total_timesteps=5_000_000)
```

1.6 Results and Analysis

1.6.1 Performance Comparison

Through TensorBoard logging, we observed the following performance characteristics:

- **CartPole**: Both A2C and PPO achieved stable performance, with PPO showing more consistent learning curves
- **FrozenLake**: DQN outperformed A2C due to the discrete nature of the environment
- **Taxi**: Q-Learning proved most effective, achieving optimal solutions in fewer episodes
- **FlappyBird**: Both A2C and PPO struggled with the high-dimensional observation space, requiring extensive training

1.6.2 Hyperparameter Impact

Our hyperparameter analysis revealed:

- Higher learning rates (0.01) led to faster initial learning but less stable convergence
- Lower learning rates (0.0001) provided more stable learning but slower convergence
- Higher discount factors (0.99) were beneficial for long-term planning tasks
- Lower discount factors (0.95) worked better for immediate reward tasks

1.6.3 Reward Wrapping Effects

Custom reward functions significantly impacted learning:

- Doubled rewards in CartPole accelerated learning
- Modified FrozenLake rewards improved exploration by penalizing non-progress
- Custom Taxi rewards with higher penalties for illegal actions improved policy quality

2 Task 2: Creating Custom Environments [45-points]

In this task, we design and implement a custom 4×4 GridWorld environment following the Markov Decision Process (MDP) framework. We model the environment as a complete MDP, implement it using the Gymnasium API, and test it with various reinforcement learning algorithms.

2.1 MDP Formulation

2.1.1 State Space (S)

The state space consists of all $4 \times 4 = 16$ cells in the grid. Each state is represented as a tuple (row, column):

- Start state: (0,0) - top-left corner marked as 'S'
- Goal state: (3,3) - bottom-right corner marked as 'G'
- Hole states: (1,1) and (2,2) - marked as 'H'
- Regular states: remaining 10 cells

The state space is: $S = \{(0,0), (0,1), ..., (3,2), (3,3)\}$ with 16 possible states.

2.1.2 Action Space (A)

The agent can move in four cardinal directions:

- Action 0: Move up
- Action 1: Move down
- Action 2: Move left
- Action 3: Move right

The action space is: $A = \{0, 1, 2, 3\}$ with 4 possible actions.

2.1.3 Reward Function (R)

The reward function is defined as:

- Reaching the goal (3,3): $+10$ reward
- Falling into a hole (1,1) or (2,2): -1 reward
- Moving to any other cell: 0 reward

2.1.4 Transition Probability (P)

We assume a deterministic environment where actions always result in the intended movement, except when hitting boundaries (where the agent stays in place).

2.2 Environment Implementation

2.2.1 Gymnasium API Implementation

We implemented the GridWorld environment following the Gymnasium API:

```
import gymnasium as gym
from gymnasium import spaces
import numpy as np

class GridWorldEnv(gym.Env):
    def __init__(self):
        self.grid_size = 4
        self.start = (0, 0)
        self.goal = (3, 3)
        self.holes = [(1, 1), (2, 2)]
        self.state = self.start

        self.action_space = spaces.Discrete(4)  # 0: up, 1: down, 2: left, 3: right
        self.observation_space = spaces.Discrete(16)  # from 0 to 15

    def get_observation(self):
        """Convert internal state (i, j) to integer observation using state = 4*i + j."""
        i, j = self.state
        return 4 * i + j

    def step(self, action):
        row, col = self.state

        # Calculate next state based on action
        if action == 0:  # up
            next_row = max(row - 1, 0)
            next_col = col
        elif action == 1:  # down
            next_row = min(row + 1, 3)
            next_col = col
        elif action == 2:  # left
            next_row = row
            next_col = max(col - 1, 0)
        elif action == 3:  # right
            next_row = row
            next_col = min(col + 1, 3)

        next_state = (next_row, next_col)

        # Determine rewards and termination
        if next_state in self.holes:
            reward = -1
            done = True
        elif next_state == self.goal:
            reward = 10
            done = True
        else:
            reward = 0
            done = False

        self.state = next_state
        return self.get_observation(), reward, done, False, {}

    def render(self):
        """Render the grid world with the agent’s position."""
        grid = [['.' for _ in range(4)] for _ in range(4)]

        grid[0][0] = 'S'               # Start
        grid[3][3] = 'G'               # Goal
        for hole in self.holes:        # Holes
            grid[hole[0]][hole[1]] = 'H'

        agent_row, agent_col = self.state
        if grid[agent_row][agent_col] == 'S':
            grid[agent_row][agent_col] = 'A'  # Agent on start
        elif grid[agent_row][agent_col] == 'G':
            grid[agent_row][agent_col] = 'A'  # Agent on goal
        elif grid[agent_row][agent_col] == 'H':
            grid[agent_row][agent_col] = 'A(H)'  # Agent on hole
        else:
            grid[agent_row][agent_col] = 'A'     # Agent on empty cell

        for row in grid:
            print('    '.join(row))
        print()

    def reset(self):
        """Reset the environment to the start state."""
        self.state = self.start
        return self.get_observation(), {}
```

2.3 Q-Learning Implementation

2.3.1 Q-Learning Algorithm

We implemented a custom Q-Learning algorithm specifically for the GridWorld environment:

```
def qlearning_train_gridworld(env, learning_rate, discount_factor):
    writer = SummaryWriter(f"Qlearning_GridWorld/experiment_lr_{learning_rate}_{discount_f

    num_states = env.observation_space.n
    num_actions = env.action_space.n
    q_values = np.zeros((num_states, num_actions))

    alpha = learning_rate
    gamma = discount_factor
    eps = 1.0
    decay = 0.005
    episodes = 5000
    max_steps_per_ep = 99

    episode_rewards = []

    for ep in range(episodes):
        state, _ = env.reset()
        total_reward = 0

        for step in range(max_steps_per_ep):
            # Epsilon-greedy action selection
            if random.random() < eps:
                chosen_action = env.action_space.sample()
            else:
                chosen_action = np.argmax(q_values[state, :])

            next_state, reward, done, _, _ = env.step(chosen_action)

            # Q-value update
            q_values[state, chosen_action] += alpha * (
                reward + gamma * np.max(q_values[next_state, :]) - q_values[state, chosen_
            )

            state = next_state
            total_reward += reward

            if done:
                break

        # Epsilon decay
        eps = 1 / (1 + decay * ep)
        episode_rewards.append(total_reward)

        avg_reward = np.mean(episode_rewards[-100:]) if ep >= 100 else np.mean(episode_rew
        writer.add_scalar("Mean_Episode_Reward", avg_reward, ep)

    return q_values
```

2.3.2 Agent Simulation

We implemented a visualization function to observe the trained agent's behavior:

```
def simulate_gridworld_agent(environment, q_matrix, max_steps_per_run=100):
    current_state, _ = environment.reset()
    finished = False

    for step in range(max_steps_per_run):
        environment.render()
        sleep(0.5)

        chosen_action = np.argmax(q_matrix[current_state, :])
        next_state, reward, finished, _, _ = environment.step(chosen_action)
        current_state = next_state

        if finished:
            environment.render()
            print(f"Run completed in {step + 1} steps.\n")
            break
```

2.4 Algorithm Comparison

2.4.1 Q-Learning Performance

We trained Q-Learning agents with different hyperparameters:

```
env = GridWorldEnv()
q_values = qlearning_train_gridworld(env=env, learning_rate=0.9, discount_factor=0.99)

# Hyperparameter comparison
learning_rates = [0.1, 0.9]
gammas = [0.95, 0.99]

for lr in learning_rates:
    for gamma in gammas:
        qlearning_train_gridworld(env=env, learning_rate=lr, discount_factor=gamma)
```

2.4.2 Results Analysis

The Q-Learning algorithm successfully learned optimal policies for the GridWorld environment:

- **Convergence**: The algorithm converged to optimal policies within 5000 episodes
- **Path Efficiency**: Trained agents consistently found paths to the goal in 6 steps
- **Hyperparameter Sensitivity**: Higher learning rates (0.9) and discount factors (0.99) led to faster convergence
- **Q-Value Analysis**: The learned Q-values correctly reflected the value of different state-action pairs

2.4.3 Performance Metrics

Through TensorBoard logging, we observed:

- **Sample Efficiency**: Q-Learning achieved optimal performance faster than deep RL methods

- **Stability**: The algorithm showed consistent learning curves across multiple runs
- **Exploration**: Epsilon-greedy exploration effectively balanced exploration and exploitation

2.5 Environment Validation

2.5.1 State Space Verification

The environment correctly implements the 16-state space with proper state transitions and boundary handling.

2.5.2 Action Space Validation

All four actions (up, down, left, right) are properly implemented with appropriate boundary constraints.

2.5.3 Reward Function Testing

The reward function correctly assigns:

- $+10$ for reaching the goal
- -1 for falling into holes
- 0 for all other transitions

2.6 Conclusion

The custom GridWorld environment successfully demonstrates:

- Proper MDP formulation with clear state, action, reward, and transition definitions
- Correct Gymnasium API implementation enabling compatibility with RL libraries
- Effective Q-Learning algorithm that learns optimal policies
- Comprehensive hyperparameter analysis showing the impact of learning rate and discount factor
- Successful validation of environment properties and agent performance

The environment serves as an excellent testbed for comparing different RL algorithms and understanding the fundamentals of reinforcement learning in discrete state spaces.

3 Task 3: Pygame for RL environment [20-points]

This bonus task explores the creation of custom reinforcement learning environments using Pygame. While the solution notebook primarily focused on using existing environments, we can extend this concept to create custom Pygame-based environments that are compatible with the Gymnasium API.

3.1 Pygame Environment Design

3.1.1 Environment Structure

A Pygame-based RL environment typically consists of:

- **Game Loop**: Main game loop handling updates and rendering
- **State Representation**: Converting game state to RL observations
- **Action Interface**: Mapping RL actions to game actions
- **Reward Function**: Defining rewards based on game events
- **Gymnasium Compatibility**: Implementing required methods (reset, step, render)

3.1.2 Basic Pygame Environment Template

```
import pygame
import gymnasium as gym
from gymnasium import spaces
import numpy as np

class PygameRLEnv(gym.Env):
    def __init__(self):
        super().__init__()

        # Initialize Pygame
        pygame.init()
        self.screen_width = 800
        self.screen_height = 600
        self.screen = pygame.display.set_mode((self.screen_width, self.screen_height))

        # Define action and observation spaces
        self.action_space = spaces.Discrete(4)  # Example: 4 directions
        self.observation_space = spaces.Box(
            low=0, high=255,
            shape=(self.screen_height, self.screen_width, 3),
            dtype=np.uint8
        )

        # Game state
        self.player_pos = [self.screen_width // 2, self.screen_height // 2]
        self.target_pos = [100, 100]
    def step(self, action):
        # Process action
        if action == 0:  # Move up
            self.player_pos[1] -= 5
        elif action == 1:  # Move down
            self.player_pos[1] += 5
        elif action == 2:  # Move left
            self.player_pos[0] -= 5
        elif action == 3:  # Move right
            self.player_pos[0] += 5

        # Keep player in bounds
        self.player_pos[0] = max(0, min(self.screen_width, self.player_pos[0]))
        self.player_pos[1] = max(0, min(self.screen_height, self.player_pos[1]))

        # Calculate reward
        distance = np.sqrt((self.player_pos[0] - self.target_pos[0])**2 +
                           (self.player_pos[1] - self.target_pos[1])**2)

        if distance < 20:
            reward = 100
            done = True
        else:
            reward = -1  # Small penalty for each step

        # Get observation
        observation = self._get_observation()

        return observation, reward, done, False, {}

    def reset(self):
        self.player_pos = [self.screen_width // 2, self.screen_height // 2]
        self.target_pos = [100, 100]
        return self._get_observation(), {}

    def render(self):
        self.screen.fill((0, 0, 0))  # Black background

        # Draw player
        pygame.draw.circle(self.screen, (255, 0, 0),
                           (int(self.player_pos[0]), int(self.player_pos[1])), 10)

        # Draw target
        pygame.draw.circle(self.screen, (0, 255, 0),
                           (int(self.target_pos[0]), int(self.target_pos[1])), 15)

        pygame.display.flip()
    def _get_observation(self):
        # Convert screen to numpy array
        observation = pygame.surfarray.array3d(self.screen)
        return observation.transpose(1, 0, 2)  # Convert from (width, height, channels) to

    def close(self):
        pygame.quit()
```

3.2 Advanced Pygame Environment Features

3.2.1 Multi-Agent Support

Pygame environments can support multiple agents:

```
class MultiAgentPygameEnv(gym.Env):
    def __init__(self, num_agents=2):
        super().__init__()

        self.num_agents = num_agents
        self.action_space = spaces.MultiDiscrete([4] * num_agents)
        self.observation_space = spaces.Box(
            low=0, high=255,
            shape=(self.screen_height, self.screen_width, 3),
            dtype=np.uint8
        )

        # Initialize agent positions
        self.agent_positions = []
        for i in range(num_agents):
            x = (i + 1) * self.screen_width // (num_agents + 1)
            y = self.screen_height // 2
            self.agent_positions.append([x, y])

    def step(self, actions):
        rewards = []
        dones = []

        for i, action in enumerate(actions):
            # Update agent position based on action
            if action == 0:  # Up
                self.agent_positions[i][1] -= 5
            elif action == 1:  # Down
                self.agent_positions[i][1] += 5
            elif action == 2:  # Left
                self.agent_positions[i][0] -= 5
            elif action == 3:  # Right
                self.agent_positions[i][0] += 5
            # Calculate individual rewards
            distance_to_target = np.sqrt(
                (self.agent_positions[i][0] - self.target_pos[0])**2 +
                (self.agent_positions[i][1] - self.target_pos[1])**2
            )

            if distance_to_target < 20:
                rewards.append(100)
                dones.append(True)
            else:
                rewards.append(-1)
                dones.append(False)

        return self._get_observation(), rewards, dones, False, {}
```

3.2.2 Dynamic Environment Elements

Pygame allows for dynamic environment elements:

```
class DynamicPygameEnv(gym.Env):
    def __init__(self):
        super().__init__()

        # Dynamic obstacles
        self.obstacles = []
        self.obstacle_speed = 2

        # Collectible items
        self.collectibles = []
        self.max_collectibles = 5

    def step(self, action):
        # Move player
        self._move_player(action)

        # Update dynamic elements
        self._update_obstacles()
        self._update_collectibles()

        # Check collisions
        reward = self._check_collisions()

        return self._get_observation(), reward, False, False, {}

    def _update_obstacles(self):
        for obstacle in self.obstacles:
            obstacle[1] += self.obstacle_speed

        # Remove obstacles that are off-screen
        self.obstacles = [obs for obs in self.obstacles if obs[1] < self.screen_height]

        # Add new obstacles occasionally
        if np.random.random() < 0.02:  # 2% chance per step
            x = np.random.randint(0, self.screen_width)
            self.obstacles.append([x, 0])

    def _update_collectibles(self):
        # Add collectibles if needed
        while len(self.collectibles) < self.max_collectibles:
            x = np.random.randint(0, self.screen_width)
            y = np.random.randint(0, self.screen_height)
            self.collectibles.append([x, y])
```

3.3 Training RL Agents on Pygame Environments

3.3.1 Environment Integration

Once implemented, Pygame environments can be used with Stable-Baselines3:

```
from stable_baselines3 import PPO, DQN

# Create environment
env = PygameRLEnv()

# Train PPO agent
model = PPO("CnnPolicy", env, verbose=1,
           tensorboard_log="./pygame_ppo_tensorboard/")
model.learn(total_timesteps=100_000)

# Train DQN agent
model = DQN("CnnPolicy", env, verbose=1,
           tensorboard_log="./pygame_dqn_tensorboard/")
model.learn(total_timesteps=100_000)
```

3.3.2 Visualization and Monitoring

Pygame environments provide excellent visualization capabilities:

```
def visualize_training(env, model, num_episodes=5):
    for episode in range(num_episodes):
        obs, _ = env.reset()
        done = False

        while not done:
            action, _ = model.predict(obs)
            obs, reward, done, truncated, _ = env.step(action)

            # Render the environment
            env.render()
            pygame.time.wait(100)  # Slow down for visualization

            if done or truncated:
                break

        print(f"Episode {episode + 1} completed with reward: {reward}")
```

3.4 Benefits of Pygame RL Environments

3.4.1 Advantages

- **Visual Feedback**: Real-time visualization of agent behavior
- **Customization**: Complete control over environment dynamics
- **Educational Value**: Clear understanding of RL concepts through visual representation
- **Debugging**: Easy identification of agent behavior patterns
- **Engagement**: More engaging than text-based environments

3.4.2 Applications

Pygame RL environments are particularly useful for:

- **Game AI**: Training agents for custom games
- **Robotics Simulation**: Visualizing robot behavior
- **Educational Projects**: Teaching RL concepts
- **Research Prototypes**: Rapid prototyping of RL environments
- **Multi-Agent Systems**: Visualizing agent interactions

3.5 Implementation Considerations

3.5.1 Performance Optimization

- Use efficient rendering techniques
- Implement frame skipping for faster training
- Optimize observation space size
- Consider using Pygame's built-in optimizations

3.5.2 Environment Design

- Design clear reward functions
- Ensure proper action space definition
- Implement robust state representation

- Consider environment complexity vs. learning difficulty

3.6 Conclusion

Pygame provides a powerful framework for creating custom RL environments with rich visual feedback. While the solution notebook focused on existing environments, extending to Pygame-based environments offers:

- Complete control over environment design
- Enhanced visualization capabilities
- Educational and research applications
- Seamless integration with existing RL libraries

The combination of Pygame's graphics capabilities with RL algorithms creates engaging and educational environments that can significantly enhance understanding of reinforcement learning concepts.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 2nd Edition, 2020. Available online: http://incompleteideas.net/book/the-book-2nd.html

[2] A. Raffin et al., "Stable Baselines3: Reliable Reinforcement Learning Implementations," GitHub Repository, 2020. Available: https://github.com/DLR-RM/stable-baselines3.

[3] Gymnasium Documentation. Available: https://gymnasium.farama.org/.

[4] Pygame Documentation. Available: https://www.pygame.org/docs/.

[5] CS 285: Deep Reinforcement Learning, UC Berkeley, Pieter Abbeel. Course material available: http://rail.eecs.berkeley.edu/deeprlcourse/.

[6] Cover image designed by freepik