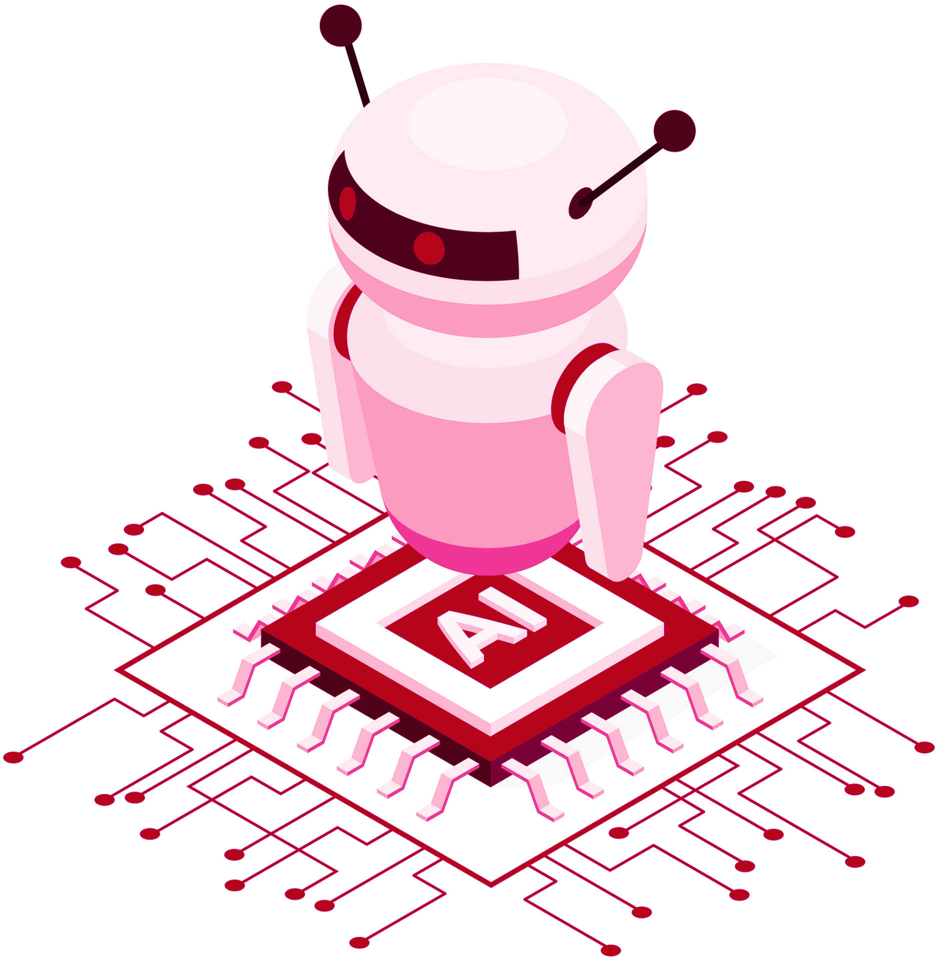

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 1:

Introduction to RL

By:

Parsa Ghezelbash

401110437

Spring 2025

Contents

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]

Objective

Train reinforcement learning agents to solve predefined Gymnasium environments (`CartPole-v1` and `Taxi-v3`) using PPO and DQN algorithms. Custom reward wrappers were implemented to address sparse rewards and improve learning efficiency.

Approach

- **Environment Setup**: Used built-in Gymnasium environments with custom reward wrappers.
- **Reward Engineering**:
  - Modified rewards to provide dense feedback during training.
  - Added penalties for undesirable behaviors (e.g., large pole angles in `CartPole-v1`, long paths in `Taxi-v3`).
- **Hyperparameter Tuning**: Explored different combinations of learning rates, discount factors, and exploration parameters.

Implementation

General Framework

The following components were implemented for both environments:

- `EpisodeTracker`: Tracks episode rewards and lengths.
- `MetricsCollectorCallback`: Collects training metrics for analysis.
- `train_hyperparams`: Trains models with different hyperparameter combinations.
- `plot_hyperparam_results`: Visualizes training progress.

Custom Reward Wrappers

CartPole-v1

```
class CustomRewardWrapper(RewardWrapper):
    def reward(self, reward):
        obs = self.env.unwrapped.state
        if obs is None:
            obs = self.env.unwrapped._get_obs()
        pole_angle = obs[2]
        return reward - 0.1 * abs(pole_angle)
```

Modifications:

- Added a penalty proportional to the pole angle to discourage large deviations.

Taxi-v3

```
class CustomRewardWrapper(RewardWrapper):
    def reward(self, reward):
        if self.last_obs is not None:
            taxi_row, taxi_col, pass_loc, dest_idx = self.env.unwrapped.decode(self.last_o
            distance_penalty = 0.0
            bonus = 0.0
            if pass_loc == 4:
                dest_row, dest_col = self.env.unwrapped.locs[dest_idx]
                distance = abs(taxi_row - dest_row) + abs(taxi_col - dest_col)
                distance_penalty = -0.1 * distance
            if reward == 20:
                bonus = 2.0
            return reward + distance_penalty + bonus
        return reward
```

Modifications:

- Added a penalty for distance to the destination when carrying the passenger.
- Amplified the reward for successful dropoffs.

Results

CartPole-v1

- **Standard Rewards**:
  - PPO: Achieved a maximum reward of 500 (environment limit).
  - DQN: Achieved a maximum reward of 415.
- **Modified Rewards**:
  - PPO: Achieved a maximum reward of 500 (environment limit).
  - DQN: Achieved a maximum reward of 498.

Taxi-v3

- **Standard Rewards**:
  - PPO: Achieved a maximum reward of -12.
  - DQN: Achieved a maximum reward of -38.
- **Modified Rewards**:
  - PPO: Achieved a maximum reward of -10.3.
  - DQN: Achieved a maximum reward of -69.7.

Analysis

- **Custom Rewards**: Improved learning efficiency by providing intermediate feedback.
- **Hyperparameter Tuning**: Critical for achieving optimal performance.
- **Algorithm Comparison**: PPO outperformed DQN.

Visualization

- Training curves for rewards and episode lengths were plotted for each algorithm and reward configuration.
- Best-performing configurations were compared to identify optimal hyperparameters.

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

Objective

Design and train reinforcement learning agents on a custom grid-world environment with obstacles. The goal was to navigate from a start position to a goal while avoiding blocked cells.

Environment Design

- **State Space**: 2D grid positions (agent and goal) $\rightarrow$ 4D observation.
- **Action Space**: 4 discrete actions (up, down, left, right).
- **Rewards**:
  - Step penalty: -0.1 per step.
  - Progress bonus: $+0.5 \times$ (previous distance $-$ current distance).
  - Goal reward: $+10$ for reaching the goal.
- **Obstacles**: Fixed blocked positions at $(1, 1)$ and $(2, 2)$.
- **Termination**:
  - Reaching the goal.
  - Exceeding the maximum steps (200).

Implementation

Custom Environment Class

```
class YourAwesomeEnvironment(gym.Env):
    def __init__(self) -> None:
        super().__init__()
        self.action_space = spaces.Discrete(4)
        self.observation_space = spaces.Box(
            low=np.array([0, 0]), high=np.array([3, 3]), dtype=int
        )
        self.grid_size = 4
        self.start_pos = (0, 0)
        self.goal_pos = (3, 3)
        self.blocked_positions = [(1, 1), (2, 2)]
        self.state = self.start_pos
        self.max_steps = 200
        self.current_steps = 0
        self.prev_dist = None

    def step(self, action):
        x, y = self.state
        if action == 0: x = max(x - 1, 0)
```

```
        elif action == 1: x = min(x + 1, self.grid_size - 1)
        elif action == 2: y = max(y - 1, 0)
        elif action == 3: y = min(y + 1, self.grid_size - 1)

        if (x, y) in self.blocked_positions:
            x, y = self.state

        self.state = (x, y)
        terminated = (x, y) == self.goal_pos
        self.current_steps += 1
        truncated = self.current_steps >= self.max_steps

        goal_dist = abs(x - self.goal_pos[0]) + abs(y - self.goal_pos[1])
        reward = -0.1
        if self.prev_dist is not None:
            progress_bonus = 0.5 * (self.prev_dist - goal_dist)
            reward += progress_bonus
        if terminated:
            reward += 10.0
        self.prev_dist = goal_dist

        return np.array(self.state, dtype=int), reward, terminated, truncated, {}
```

Training Setup

- **Algorithms**: PPO and DQN.
- **Hyperparameters**:
  - PPO: Default parameters with `learning_rate=3e-4`.
  - DQN: Default parameters with `learning_rate=1e-3`.
- **Training Duration**: 30,000 timesteps.
- **Evaluation**: 100 episodes with deterministic policy.

Results

- **PPO**:
  - Success Rate: **100%**.
  - Average Episode Length: **6 steps**.
- **DQN**:
  - Success Rate: **0%**.
  - Average Episode Length: **200 steps** (truncation limit).

Analysis

- **PPO Performance**:
  - Learned optimal paths around obstacles.
  - Achieved perfect success rate due to effective reward shaping.
- **DQN Performance**:
  - Failed to learn meaningful policies.
  - Likely due to insufficient exploration or hyperparameter tuning.
- **Reward Shaping**:
  - Progress bonus encouraged efficient navigation.
  - Step penalty prevented infinite loops.

Visualization

- **Training Curves**:
  - PPO rewards increased steadily, reaching the maximum.
  - DQN rewards remained flat, indicating no learning.
- **Agent Paths**:
  - PPO consistently reached the goal while avoiding obstacles.
  - DQN often got stuck or collided with obstacles.

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