METU EE 449

Computational Intelligence

From Mazes to Moon Landings: A Journey through Reinforcement Learning

Homework 2 - Reinforcement Learning

Due: 23:55, 04/05/2025

Late submissions are welcome, but penalized according to the following policy:

- 1 day late submission: HW will be evaluated out of 70.
- 2 days late submission: HW will be evaluated out of 40.
- 3 or more days late submission: HW will not be evaluated.

You should prepare your homework by yourself alone and you should not share it with other students, otherwise you will be penalized.

Introduction

In this homework, you will explore reinforcement learning through two major tasks. First, you will model a maze environment and implement Temporal Difference (TD(0)) Learning to solve the navigation problem. Second, you will apply Deep Q-Networks (DQN) to solve the LunarLander-v3 environment from Gymnasium.

Your implementations must be in Python, using libraries such as NumPy, Matplotlib, Gymnasium and PyTorch. Starter code is given in this document and utility functions are provided in the *HW2* folder on *ODTUClass*.

Homework Task and Deliverables

In the scope of this homework, you will explore the application of Reinforcement Learning in two distinct problems. First, you will address a **maze navigation problem** by employing **Temporal Difference (TD) Learning**. Second, you will apply **Deep Q-Learning (DQN)** to solve a control task in the **LunarLander-v3** environment.

For the maze task, you will begin by modeling the maze, defining its states, actions, transition probabilities, and a strategic reward function. Subsequently, you will implement TD(0) Learning to estimate utility value functions and utilize Q Learning to directly learn an optimal navigation policy. Throughout the assignment, you are required to provide various plots such as heatmaps of utility values and learning curves to visualize the progression and effectiveness of the algorithms.

For the Deep Q-Learning task, you will formulate the LunarLander problem, implement a DQN agent in PyTorch, complete missing code parts (such as the ε-greedy policy and training loop), and experiment with different hyperparameters and network architectures. You will also analyze the training process with performance plots and saved experimental data.

You should submit a single report in which your answers to the questions, the required experimental results (performance curve plots, visualizations etc.) and your deductions are presented for each part of the homework. Moreover, you should append your Python codes to the end of the report for each part to generate the results and the visualizations of the experiments. Namely, all the required tasks for a part can be performed by running the related code file. The codes should be well structured and **well commented**. The non-text submissions (e.g. image) or the submissions lacking comments will not be evaluated. Similarly answers/results/conclusions written in code as a comment will not be graded.

The report should be in portable document format (pdf) and named as *hw2_name_surname_eXXXXXX* where *name*, *surname* and *X*s are to be replaced by your name, surname and digits of your user ID, respectively. You do not need to send any code files to your course assistant(s), since everything should be in your single pdf file.

Do not include the codes in *utils.py* to the end of your pdf file.

1 Temporal Difference (TD) Learning

1.1 Implementing the Maze

In this part, you will implement the maze in Figure 1 in Python, using NumPy and basic mathematical operations. Starting point is shown with blue cell, traps with red and the goal point with green. Use 0 for free space, 1 for obstacle, 2 for traps and 3 for goal on your maze in Python. Define states (each cell of the maze), possible actions (move up, down, left, right), and rewards (negative for each state, large positive for the desired state (goal), large negative for undesired final state (trap)).

The environment is stochastic, the probabilities for the results of the chosen actions are as follows:

- Probability of going for the chosen direction: 0.75
- Probability of going opposite of the chosen direction: 0.05
- Probability of going each of perpendicular routes of the chosen direction: 0.10

If the agent hits the wall or goes out of the bounds as the result of the action, it goes back to the current state and the episode continues with the next action.

Firstly you need to call required libraries and functions. You also need to call functions from `utils.py`.

```
import numpy as np
from matplotlib import pyplot as plt
```

Then, you will define an MazeEnvironment class. You can use the following code directly, but you need to complete missing parts denoted with FILL HERE.

```
class MazeEnvironment:
    def __init__(self):
        # Define the maze layout, rewards, action space (up, down, left, right)

        self.start_pos = (0,0) # Start position of the agent
        self.current_pos = self.start_pos

        self.state_penalty = -1
        self.trap_penalty = -100
        self.goal_reward = 100

        self.actions = 0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0, 1)

    def reset(self):
        self.current_pos = #FILL HERE
        return #FILL HERE
    def step(self, action):
        #FILL HERE
```

Make sure the functions of the MazeEnvironment class works properly, then you may go on with the next part.

1.2 Implementing the Algorithm

In this part, **TD(0) algorithm** is implemented to update the utility value estimate after every action taken in the environment. This means adjusting the estimated utility value of the current state based on the reward received and the estimated utility value of the next state. For that, ε-greedy strategy can be used, where the agent randomly chooses an action with probability ε (exploration), or chooses the best known action based on the current utility values (exploitation).

```
class MazeTD0(MazeEnvironment): # Inherited from MazeEnvironment
    def __init__(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000):
```

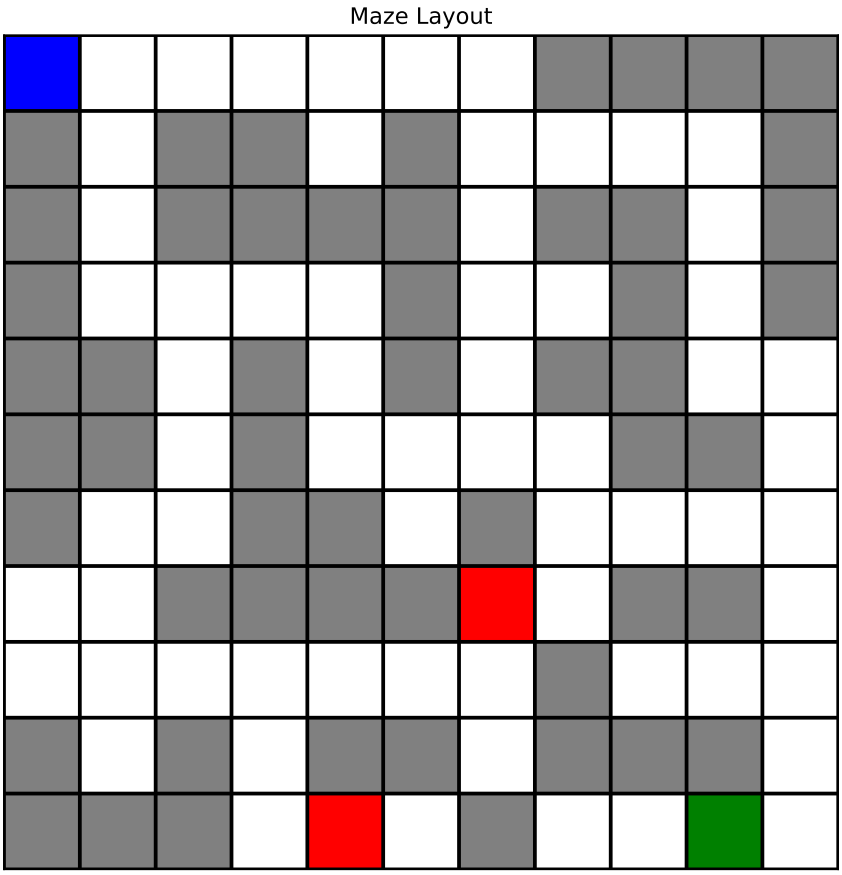

Figure 1: The maze.

```
    super().__init__()

    self.maze = maze
    self.alpha = alpha #Learning Rate
    self.gamma = gamma #Discount factor
    self.epsilon = epsilon #Exploration Rate
    self.episodes = episodes
    self.utility = #FILL HERE, Encourage exploration



def choose_action(self, state):
    #Explore and Exploit: Choose the best action based on current utility values
    #Discourage invalid moves
    #FILL HERE
def update_utility_value(self, current_state, reward, new_state):
    current_value = #FILL HERE
    new_value = #FILL HERE

    # TD(0) update formula
```

Table 1: Parameter configurations to be experimented.

α	0.001	0.01	**0.1**	0.5	1.0
γ	0.10	0.25	0.50	0.75	**0.95**
ε	0	**0.2**	0.5	0.8	1.0

```
        self.utility[current_state] = #FILL HERE

    def run_episodes(self):
        #FILL HERE
        return self.utility
```

Then, the environment and the agent should be ready for the learning.

```
# Create an instance of the Maze with TD(0) and run multiple episodes
    maze = #FILL HERE
    maze_td0 = MazeTD0(self, maze, alpha=0.1, gamma=0.95, epsilon=0.2, episodes=10000)
    final_values = maze_td0.run_episodes()
```

1.3 Experimental Work

You will experiment TD(0) Learning on several different hyperparameters. Namely,

- Learning rate (α)
- Discount factor (γ)
- Initial exploration rate (ε)

To see the effect of each hyperparameter, run the algorithm for each value in a row given in Table 1 while using default values for the other hyperparameters. The default values are bolded. Use <num_episodes>= 10000.

For each experiment, you need to include:

- Utility value function heatmap for episodes 1, 50, 100, 1000, 5000, 10000 (use the function from `utils.py`)
- Policy for episodes 1, 50, 100, 1000, 5000, 10000 (use the function from `utils.py`)
- Convergence plot. A line graph showing the change in the value function from one iteration to the next over time, to assess convergence. Plotting the sum of absolute differences between the value functions of successive iterations against the number of episodes can illustrate this.

1.4 Discussions

Answer the following questions:

1. Discuss the rationale behind your transition probabilities and reward function. How do they influence the agent's behavior?
2. How did the utility value functions evolve over time? You may explain via your heatmap plots at different stages.
3. Did the utility value function converge? If so, how many episodes were required for convergence?
4. How sensitive is the TD(0) Learning process to parameters like learning rate and discount factor? What effects did changes in these parameters have on the learning outcomes?
5. What challenges did you encounter while implementing TD(0) Learning, and how did you address them?

6. Discuss the impact of exploration strategies (e.g., ε-greedy) on the learning process. How did changes in ε affect performance over time?
7. What modifications to the maze design or problem setup could potentially improve the learning process or make it more challenging?
8. How could the algorithms be adjusted or combined with other techniques to enhance their performance or reliability?

2 Deep Q-Learning

In this part, you will apply Deep Q-Networks (DQN) to solve the Gymnasium LunarLander-v3 environment. You will: (1) formulate the reinforcement learning problem (state space, action space, rewards, etc.), (2) understand a provided PyTorch-based DQN implementation, (3) implement missing parts of the code (e.g. the ε-greedy policy and training loop components), and (4) experiment with different hyperparameters, documenting their impact on learning performance.

The LunarLander-v3 environment involves controlling a rocket lander to safely touch down on a landing pad, as illustrated in Figure 2. The lander starts at the top of the screen and is influenced by gravity. The agent can fire a main engine or side engines to adjust the lander's position and orientation. The goal is to land softly on the pad without crashing.

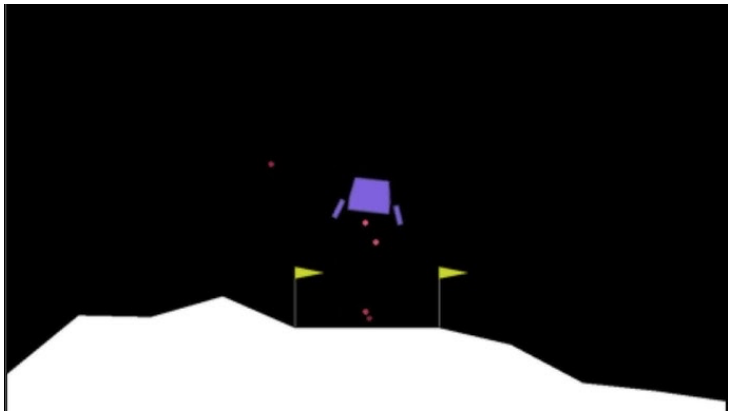

Figure 2: The Lunar Lander environment.

2.1 Problem Description

LunarLander-v3 is a classic continuous-state control problem. At each time step, the agent observes the state of the lander and must choose one of the discrete engine actions. A well-shaped reward function provides feedback for the agent to learn an optimal landing policy. We will train a DQN agent to learn this task through trial and error.

Formulate the LunarLander task as a reinforcement learning problem by identifying the state space, action space, and reward structure, and defining episode termination conditions. Note that, these are already defined in LunarLander-v3 environment and you don't need to define manually:

- **State Space:** The state is an 8-dimensional continuous vector describing the lander's pose and velocity. It includes the lander's position (x, y), velocity (v_x, v_y), angle, angular velocity, and two booleans indicating whether the left or right landing leg is in contact with the ground.
- **Action Space:** The environment has four discrete actions: do nothing, fire left orientation engine, fire main engine, or fire right orientation engine.
- **Reward Function:** The agent receives positive reward for moving closer to the landing pad and coming to rest, and negative reward for moving away. Each leg contact yields a bonus of +10.

Firing the main engine costs -0.3 per frame (side engine -0.03). Landing safely gives an additional +100 reward, while crashing gives -100.)

- **Episode Termination:** The episode ends when the lander crashes or lands successfully, or if it flies off screen. There may also be an upper limit on the number of time steps per episode to prevent infinite loops.
- **Solved Criterion:** LunarLander-v3 is considered "solved" when the agent achieves an average reward of $\geq$200 over 100 consecutive episodes.

Given this formulation, can we use standard tabular RL methods to solve it? Considering the size and continuity of the state space, which is continuous 8-dimensional – making tabular methods infeasible. Instead, you will use function approximation (Deep RL) to generalize across states.

2.2 Algorithm Implementation

Firstly you need to call required libraries and functions.

```
import gymnasium as gym
import numpy as np
import torch
from collections import deque
import random
```

Then, define a simple deep neural network for Q-value approximation, like the ones in Homework 1:

```
class QNetwork(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim1=128, hidden_dim2=128):
        super(QNetwork, self).__init__()
        #FILL HERE
    def forward(self, x):
        #FILL HERE
```

Define a replay memory to store experiences:

```
class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0
    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append((state, action, reward, next_state, done))
        else:
            self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity
    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)
    def __len__(self):
        return len(self.buffer)
```

Define the DQN Agent encapsulating the networks, memory, and training procedure:

```
class DQNAgent:
    def __init__(self, state_dim, action_dim,
                 memory_size=50000, batch_size=64,
                 gamma=0.99, alpha=1e-3,
                 epsilon_start=1.0, epsilon_min=0.01, epsilon_decay=0.995,
                 target_update_freq=10):
        self.state_dim = state_dim
        self.action_dim = action_dim
```

```
        self.gamma = gamma
        self.batch_size = batch_size
        self.epsilon = epsilon_start
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.memory = ReplayMemory(memory_size)
        self.policy_net = QNetwork(state_dim, action_dim)
        self.target_net = QNetwork(state_dim, action_dim)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()
        self.optimizer = torch.optim.Adam(self.policy_net.parameters(), lr=alpha)
        self.target_update_freq = target_update_freq
        self.solved_score = 200.0    #target average reward for 'solved'
        self.solved_window = 100     #number of episodes to average for solved check
    def get_action(self, state):
        # Select an action for the given state using epsilon-greedy policy.
        # FILL HERE
    def train_step(self):
        #Perform one training step.
        if len(self.memory) < self.batch_size:
            return  # Not enough samples to train yet.
        batch = self.memory.sample(self.batch_size)
        states = torch.tensor(np.array([exp[0] for exp in batch]),dtype=torch.float32)
        actions = torch.tensor(np.array([exp[1] for exp in batch]),dtype=torch.int64).unsqueeze(1)
        rewards = torch.tensor(np.array([exp[2] for exp in batch]),dtype=torch.float32).unsqueeze(1)
        next_states = torch.tensor(np.array([exp[3] for exp in batch]),dtype=torch.float32)
        dones = torch.tensor(np.array([exp[4] for exp in batch]),dtype=torch.float32).unsqueeze(1)
        curr_Q = self.policy_net(states).gather(1, actions)
        next_Q = self.target_net(next_states).max(1)[0].unsqueeze(1)

        #FILL HERE: Compute target Q
        #FILL HERE: Compute MSE loss between current Q and target Q
        #FILL HERE: Backpropagate

    def update_target(self):
        #FILL HERE: Update target network to match policy network.
    def decay_epsilon(self):
        #FILL HERE: Decay exploration rate after each episode.
```

Then, the environment and the agent should be ready for the learning.

```
env = gym.make("LunarLander-v3")
state_dim = env.observation_space.shape[0]   # should be 8
action_dim = env.action_space.n             # should be 4
agent = DQNAgent(state_dim, action_dim)
num_episodes = 10000
for episode in range(1, num_episodes+1):
    state, _ = env.reset()
    for t in range(1000): # limit max steps per episode to avoid long runs
        # FILL HERE: Get action, step it into environment, push to the memory, train agent
        for one step, update state and accumulate reward. If done, break from the for loop.
    # FILL HERE: Decay epsilon.
    if episode % agent.target_update_freq == 0:
        # FILL HERE: Update target network.
env.close()
```

Table 2: Parameter configurations to be experimented.

α	1e-4	**1e-3**	5e-3		
γ	0.98	**0.99**	0.999		
ε-decay	0.98	0.99	**0.995**		
f	1	**10**	50		
Net*	FC-128, ReLU	FC-64, ReLU, FC-64, ReLU	**FC-128, ReLU, FC-128, ReLU**	FC-128, ReLU, FC-128, ReLU, FC-128, ReLU	FC-256, ReLU, FC-256, ReLU

*Note that all Q-Network architectures include a Q-value prediction layer at the end, whose size matches the action dimension and which uses no nonlinear activation.

2.3 Experimental Work

You will experiment with Deep Q-Learning using various hyperparameters and network architectures. Specifically:

- Learning rate (α)
- Discount factor (γ)
- Exploration decay rate (ε-decay)
- Target network update frequency (f)
- Q-Network architecture (Net)

To observe the effect of each hyperparameter, run the algorithm for every value listed in the corresponding row of Table 2, while keeping the other hyperparameters at their default values. The default values are indicated in **bold**. Use <num_episodes>= 5000.

During training, you need to save the following items in separate JSON files for each experiment:

- `episode_rewards`: A list of the total reward obtained in each episode (length = number of episodes run).
- `average_scores`: A list of the moving average of episode rewards (using a 100-episode window) at each episode. This smooths out variability and is used to assess whether the environment is solved.
- `hyperparameters`: A dictionary of hyperparameters used in that training run—e.g., learning rate, discount factor (γ), epsilon decay schedule, target network update frequency, and network architecture details (such as hidden layer sizes). This helps track the experiment settings.
- `solved_episode`: The episode number at which the environment was considered solved, if applicable. As explained earlier, "solved" typically means achieving an average reward ≥ 200 over 100 consecutive episodes. If the agent never meets this criterion during training, `solved_episode` will be `null` in the JSON file or `None` in Python.

When running multiple experiments, make sure to save the results of each run under a unique filename (e.g., `results_lr_1e-3.json`, `results_lr_5e-4.json`, etc.) to avoid overwriting previous data and to enable easy comparison later.

You need to include the following plots:

- Learning curve plots that include both raw episode rewards and average scores, shown as subplots in a single figure. Plot exactly one figure per parameter type (e.g., for learning rate: plots for 1e-4, 1e-3, 5e-3 on the same figure). Use the function provided in `utils.py` by passing a list of JSON paths you want to plot.
- A bar chart showing the solved episodes for all experimental results in one figure. Use the corresponding function from `utils.py` and pass a list of all relevant JSON paths.

2.4 Discussions

Answer the following questions:

1. How did the three learning rates impact the agent's learning speed and stability? For example, did a higher learning rate converge faster or overshoot (causing instability)? Which learning rate achieved the best final performance, and what trade-offs did you notice (e.g. overshooting vs. slow learning)?
2. Compare the results for different γ values. How did a lower γ (favoring short-term rewards) differ from a higher γ (favoring long-term rewards) in terms of convergence and final scores? Which γ led to solving the environment fastest, and why do you think that is?
3. Examine the curves for the different exploration (ε) decay schemes. How did the rate at which exploration was reduced affect the agent's ability to find a good policy? Did a slower decay (exploring for longer) result in better final performance, or was a faster decay sufficient? Discuss the balance between exploration and exploitation you observed.
4. What differences do you see between frequent vs. infrequent target network updates? Did updating the target network more often lead to more stable learning or quicker convergence, or did it cause more oscillation? Explain how the update frequency might be affecting the stability of Q-value estimates.
5. How did the various neural network architectures (different hidden layer sizes/counts) perform relative to each other? Did a larger network (more neurons or layers) learn faster or achieve higher rewards than a smaller network? Consider the potential reasons, such as function approximation capability vs. overfitting or learning complexity. Which architecture seems to hit the solve criteria fastest, and which struggled?
6. Among all the hyperparameters you experimented with, which one had the most pronounced effect on the agent's performance? Provide examples from your results (refer to the plots or solved episode counts) to support your reasoning. Also, discuss any interactions you suspect between hyperparameters (for instance, could a certain learning rate work better with a certain epsilon schedule? though you tested them independently, think about possible combinations).
7. Look at the shape of the learning curves (the reward trajectories over time). Do some configurations have more volatile learning (lots of spikes and dips) while others are smoother? What might cause those differences in training dynamics? Relate this to the hyperparameter settings (e.g., a high learning rate or infrequent target updates might cause more volatility).