CS-866 Deep Reinforcement Learning

MDP Solutions - I
Model-Based Learning

Nazar Khan
Department of Computer Science
University of the Punjab

MDP Solution Methods: Big Picture

- We now move from *formulation* of MDPs to their *solution*.
- Goal: find the optimal policy $\pi^\star$ that maximizes expected return.
- Solution methods typically rely on:
  - *Recursion*: breaking problems into smaller subproblems.
  - *Dynamic Programming (DP)*: systematic recursion + memoization.
  - *Value Iteration (VI)*: an iterative DP algorithm to solve Bellman equations.

Recursion Intuition

- The Bellman equation is inherently *recursive*:

$$V(s) = \max_a \mathbb{E}[r + \gamma V(s')]$$

- The value of a state depends on the values of its successor states.
- Just like recursion in programming: a function calls itself with smaller inputs.
- Eventually we reach terminal states, where values are known.

Droste effect: recursion in pictures

Dynamic Programming: Divide and Conquer

- DP applies recursion systematically across the entire state space.
- Principle: *divide and conquer*.
  1. Start from a root state whose value we want.
  2. Recursively compute values of sub-states closer to terminal states.
  3. At terminals: rewards are known.
  4. Propagate values back up: combine child values into parent values.
  5. Eventually arrive at the root value.
- This mirrors recursive algorithms in computer science.

Value Iteration: The Idea

- A basic DP algorithm to compute optimal values.
- Initialize value function $V(s)$ arbitrarily (e.g., random or zeros).
- Repeatedly update values using Bellman optimality equation:

$$V_{k+1}(s) = \max_{a \in A} \left[R(s,a) + \gamma \sum_{s'} T(s,a,s') V_k(s') \right]$$

- Keep iterating until values converge (stop changing much).
- Once $V(s)$ is known, extract the optimal policy:

$$\pi^\star(s) = \arg\max_a \; Q^\star(s,a)$$

Value Iteration Pseudocode

```
def value_iteration():
 initialize(V)
 while not convergence(V):
     for s in range(S):
         for a in range(A):
             for s‘ in range(S):
                 Q[s,a] = Q[s,a] + T_a(s,s‘)(R_a(s,s‘) +
                     gamma * V[s‘])
         V[s] = max_a(Q[s,a])
 return V
```

Discussion: Pros and Cons of Value Iteration

Strengths

- Simple, elegant, and guaranteed to converge to the optimal value function.
- Works with any finite MDP (finite S, A).
- Provides both $V^\star(s)$ and $\pi^\star(s)$.

Weaknesses

- Computationally expensive:
  - Triply nested loop over states, actions, and next states.
  - Repeated full sweeps of the state space.
- Convergence can be slow.

OpenAI Gym: Introduction

- **Gym** is a Python suite of *environments* for reinforcement learning.
- Created by OpenAI, it has become the **de facto standard**.
- *However, OpenAI Gym was discontinued in 2022*.
- **Gymnasium** is a community maintained fork of OpenAI's Gym library.
- Available at `https://gymnasium.farama.org/`
- Runs on Linux, macOS, and Windows.
- Large and active community: new environments are continuously added.

OpenAI Gym: Environments

- Gym provides environments from **easy to advanced**:
    - Classic control problems: **CartPole**, **MountainCar**.
    - Small text environments: **Taxi**.
    - **Arcade Learning Environment (ALE)**[1]
    - Physics-based robotics: **MuJoCo**[2], **PyBullet**.
- You can:
    - Experiment with predefined environments.
    - Create your own environments.
    - Test different agent algorithms in a common interface.

[1] Volodymyr Mnih et al. 'Playing Atari with deep reinforcement learning'. In: *arXiv preprint arXiv:1312.5602* (2013).

[2] Emanuel Todorov, Tom Erez, and Yuval Tassa. 'MuJoCo: A physics engine for model-based control'. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2012, pp. 5026–5033.

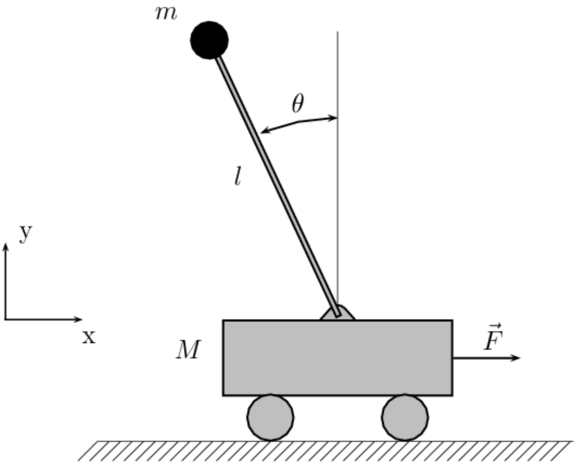
Installing Gymnasium

- Check if Python is installed and up-to-date (3.10 recommended).
- Install Gymnasium with pip:

```
pip install gymnasium
```

- Install in the same virtual environment as **PyTorch** or **TensorFlow**.
- You may need to install/update additional packages: `numpy`, `scipy`, `pyglet`, etc.
- Some environments require **OpenGL** support.

CartPole Environment



CartPole: a classic control benchmark

Testing the Installation: CartPole

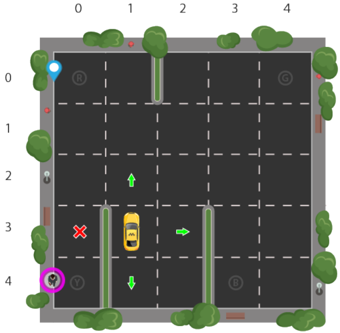
- A simple test is to run the **CartPole** environment.
- If successful, a window appears with a pole balancing on a cart.
- Random actions should move the cart and make the pole wobble.

```
import gym

env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
env.close()
```

Taxi Example: Introduction

- The **Taxi environment** is a simple Grid World from OpenAI Gym.
- Goal: Taxi must
  1. Navigate to the passenger's location.
  2. Pick up the passenger.
  3. Drive to the destination (R, G, B, Y).
  4. Drop off the passenger.
- The episode ends after successful drop-off.



Taxi Example: State Space

- The problem has a **discrete state space**:

$$25 \text{ (taxi positions)} \times 5 \text{ (passenger states)} \times 4 \text{ (destinations)} = 500$$

- Passenger states:
  - At one of 4 fixed locations (R, G, B, Y).
  - Or already inside the taxi.
- Each state fully specifies:
  - Taxi position.
  - Passenger location.
  - Destination location.

Taxi Example: Action Space

- There are **six discrete deterministic actions**:
  1. Move South
  2. Move North
  3. Move East
  4. Move West
  5. Pick up passenger
  6. Drop off passenger
- Transitions are deterministic given the current state and action.
- Illegal actions (e.g., pickup/dropoff at wrong location) are handled by rewards.

Taxi Example: Rewards

- Reward structure:
  - -1 for each time step (encourages faster completion).
  - $+20$ for successfully dropping off the passenger.
  - -10 penalty for illegal pickup or dropoff.
- This balance of positive/negative rewards:
  - Encourages efficiency.
  - Prevents random or invalid actions.

Value Iteration for Taxi

- Value iteration can be applied to this environment:
  1. Initialize $V(s)$ for all states randomly (or zeros).
  2. Iteratively update $V(s)$ using the Bellman optimality equation.
  3. Extract greedy policy $\pi(s)$ after convergence.
- OpenAI Gym provides the transition function:
  - The following code queries Gym for the next state instead of hardcoding transitions.

Value Iteration in Gym: Taxi Environment

```
import gym
import numpy as np

def iterate_value_function(v_inp, gamma, env):
    ret = np.zeros(env.nS)
    for sid in range(env.nS):
        temp_v = np.zeros(env.nA)
        for action in range(env.nA):
            for (prob, dst_state, reward, is_final) in env.P[
                 sid][action]:
                 temp_v[action] += prob*(reward + gamma*v_inp[
                     dst_state]*(not is_final))
        ret[sid] = max(temp_v)
    return ret

def build_greedy_policy(v_inp, gamma, env):
    new_policy = np.zeros(env.nS)
    for state_id in range(env.nS):
```

Value Iteration in Gym: Taxi Environment

```
        profits = np.zeros(env.nA)
        for action in range(env.nA):
            for (prob, dst_state, reward, is_final) in env.P[
                state_id][action]:
                profits[action] += prob*(reward + gamma*v[
                    dst_state])
        new_policy[state_id] = np.argmax(profits)
    return new_policy


env = gym.make('Taxi-v3')
gamma = 0.9
cum_reward = 0
n_rounds = 500
env.reset()
for t_rounds in range(n_rounds):
    # init env and value function
    observation = env.reset()
```

Value Iteration in Gym: Taxi Environment

```
v = np.zeros(env.nS)

# solve MDP
for _ in range(100):
    v_old = v.copy()
    v = iterate_value_function(v, gamma, env)
    if np.all(v == v_old):
        break
policy = build_greedy_policy(v, gamma, env).astype(np.int)

# apply policy
for t in range(1000):
    action = policy[observation]
    observation, reward, done, info = env.step(action)
    cum_reward += reward
    if done:
        break
if t_rounds % 50 == 0 and t_rounds > 0:
```

Value Iteration in Gym: Taxi Environment

```
        print(cum_reward * 1.0 / (t_rounds + 1))
env.close()
```

Taxi Example: Hands-On

- Run the provided Taxi value iteration code.
- Experiment with:
  - Discount factor γ.
  - Convergence threshold.
  - Initialization of $V(s)$.
- Try to visualize:
  - How $V(s)$ changes across iterations.
  - How the policy $\pi(s)$ emerges from the values.
- This prepares us for more complex planning and learning algorithms.

Next Lecture

- So far: Value Iteration (model-based) computes the policy using the transition model.
- Problem: In many environments, the transition probabilities are **unknown**.
- Solution: Use **model-free algorithms** that learn directly from experience.
- Key milestone: Enabled reinforcement learning to work in real-world problems.