ECE 60146 and BME 64600: Homework 11

Triparna Mahata

May 3, 2026

3 Programming Tasks

3.1 Task 1: Tabular Double Q-Learning on CartPole

3.1.1 Reproducing the baseline

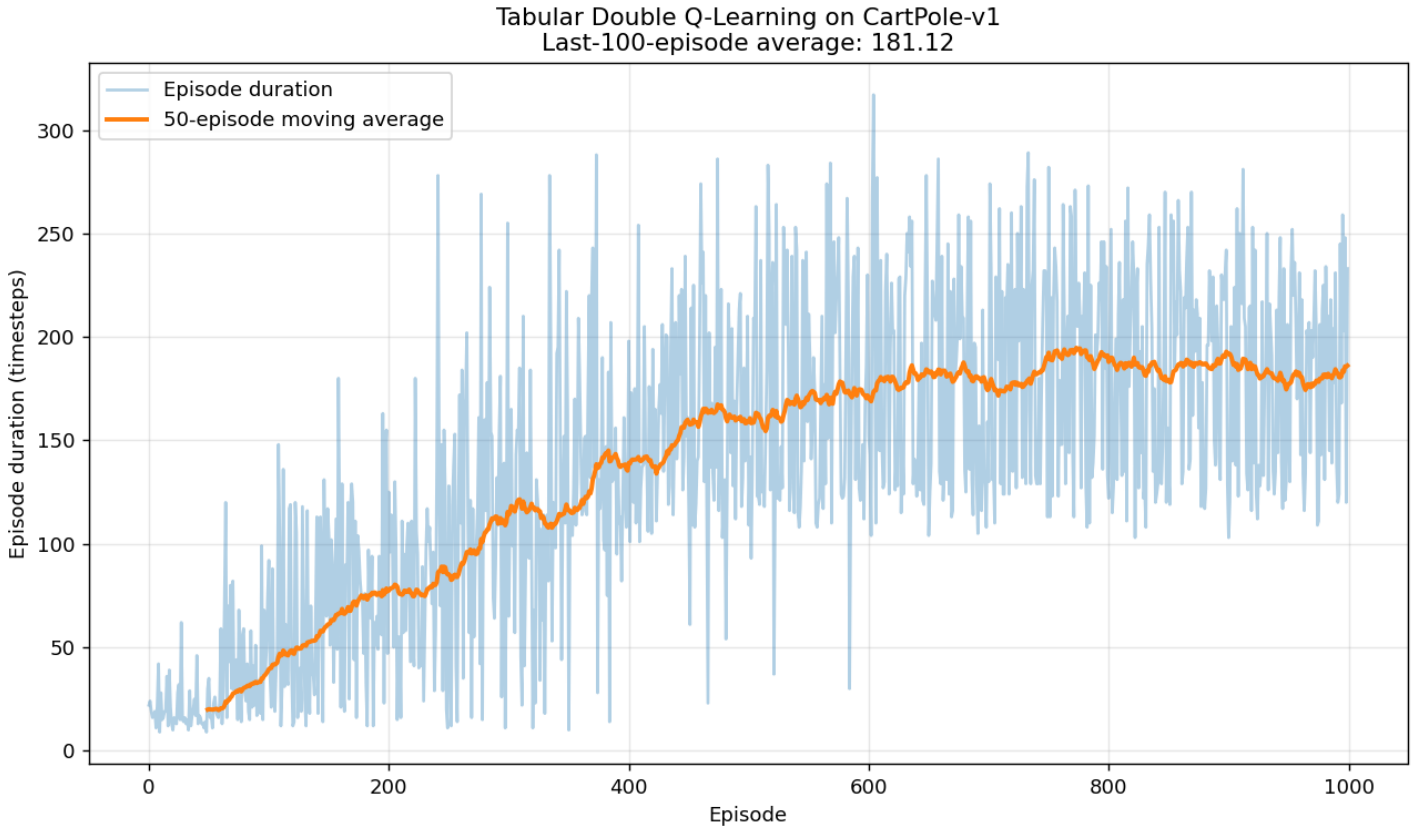
Code:

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import random

# ------------------------------------------------------------------------
# Reproducibility
# ------------------------------------------------------------------------
seed = 0
random.seed(seed)
np.random.seed(seed)

# ------------------------------------------------------------------------
# Initialize parameters and environment
# ------------------------------------------------------------------------
env = gym.make('CartPole-v1')                          # (A) v0 -> v1
# In gymnasium the action space carries its OWN RNG, separate from
# numpy.random and Python's random module.  Must be seeded explicitly,
# otherwise env.action_space.sample() (used by epsilon-greedy) is
# nondeterministic across runs.
env.action_space.seed(seed)

GAMMA              = 0.85
MAX_ITER           = 1000
EXPLORATION_MAX    = 1.0
EXPLORATION_MIN    = 0.01
EXPLORATION_DECAY  = 1 + np.log(EXPLORATION_MIN) / MAX_ITER
REP_MEM_SIZE       = 100000
MINIBATCH_SIZE     = 64

N_states  = 162
N_actions = 2

Q1 = np.zeros((N_states, N_actions))
Q2 = np.zeros((N_states, N_actions))

alpha    = 0.001
eps_rate = EXPLORATION_MAX


# ------------------------------------------------------------------------
# Map the four-dimensional continuous state to a discrete state in [0, 161]
# Credit: http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf
# ------------------------------------------------------------------------
def map_discrete_state(cs):                            # (B)
    ds_vector = np.zeros(4)
```

```
47
48      # x  (cart position)
49      if abs(cs[0]) <= 0.8:
50          ds_vector[0] = 1
51      elif cs[0] < -0.8:
52          ds_vector[0] = 0
53      elif cs[0] > 0.8:
54          ds_vector[0] = 2
55
56      # x' (cart velocity)
57      if abs(cs[1]) <= 0.5:
58          ds_vector[1] = 1
59      elif cs[1] < -0.5:
60          ds_vector[1] = 0
61      elif cs[1] > 0.5:
62          ds_vector[1] = 2
63
64      # theta (pole angle, in degrees)
65      angle = 180 / 3.1428 * cs[2]
66      if   -12 < angle <= -6: ds_vector[2] = 0
67      elif  -6 < angle <= -1: ds_vector[2] = 1
68      elif  -1 < angle <=  0: ds_vector[2] = 2
69      elif   0 < angle <=  1: ds_vector[2] = 3
70      elif   1 < angle <=  6: ds_vector[2] = 4
71      elif   6 < angle <= 12: ds_vector[2] = 5
72
73      # theta' (pole angular velocity)
74      if abs(cs[3]) <= 50:
75          ds_vector[3] = 1
76      elif cs[3] < -50:
77          ds_vector[3] = 0
78      elif cs[3] > 50:
79          ds_vector[3] = 2
80
81      ds = int(ds_vector[0] * 54 + ds_vector[1] * 18
82               + ds_vector[2] * 3 + ds_vector[3])
83      return ds
84
85
86  # ------------------------------------------------------------------------
87  # Epsilon-greedy action selection
88  # ------------------------------------------------------------------------
89  def optimal_action(Q, state, eps_rate, env):         # (C)
90      if np.random.random() < eps_rate:
91          return env.action_space.sample()
92      return int(np.argmax(Q[state, :]))
93
94
95  # ------------------------------------------------------------------------
96  # Double Q-learning update
97  # ------------------------------------------------------------------------
98  def updateQValues(state, action, reward, next_state, alpha, eps_rate, env):  # (D)
99      if np.random.random() < 0.5:
100         next_action = optimal_action(Q1, next_state, eps_rate, env)
101         Q1[state][action] += alpha * (
102             reward + GAMMA * Q2[next_state][next_action] - Q1[state][action]
103         )
104     else:
105         next_action = optimal_action(Q2, next_state, eps_rate, env)
106         Q2[state][action] += alpha * (
107             reward + GAMMA * Q1[next_state][next_action] - Q2[state][action]
108         )
109     return next_action
110
111
112 # ------------------------------------------------------------------------
```

```
# Replay memory
# ------------------------------------------------------------------------
class ReplayMemory:                                          # (E)
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory   = []

    def push(self, s, a, r, ns):
        self.memory.append([s, a, r, ns])
        if len(self.memory) > self.capacity:
            del self.memory[0]

    def sample(self, MINIBATCH_SIZE):
        return random.sample(self.memory, MINIBATCH_SIZE)

    def __len__(self):
        return len(self.memory)


# ------------------------------------------------------------------------
# Training loop
# ------------------------------------------------------------------------
if __name__ == "__main__":                                   # (F)

    duration_history = []          # raw episode durations (timesteps)
    epsilon_history  = []
    repmemory        = ReplayMemory(REP_MEM_SIZE)

    for i_episode in range(MAX_ITER):

        # --- gymnasium reset returns (obs, info) ---
        state, info = env.reset(seed=seed if i_episode == 0 else None)
        state = map_discrete_state(state)
        action = optimal_action(0.5 * (Q1 + Q2), state, 0, env)

        # epsilon decay
        eps_rate *= EXPLORATION_DECAY
        eps_rate  = max(EXPLORATION_MIN, eps_rate)

        t = 0
        while True:
            # --- gymnasium step returns 5 values ---
            next_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated

            if done:
                reward = -10

            next_state = map_discrete_state(next_state)
            repmemory.push(state, action, reward, next_state)

            # one on-policy Double-Q update
            next_action = updateQValues(state, action, reward, next_state,
                                        alpha, eps_rate, env)

            # replay updates
            if len(repmemory) > MINIBATCH_SIZE:
                experiences = repmemory.sample(MINIBATCH_SIZE)
                for ts, ta, tr, tns in experiences:
                    updateQValues(ts, ta, tr, tns, alpha, eps_rate, env)

            state  = next_state
            action = next_action
            t += 1

            if done:
```

```
                break

        duration_history.append(t)
        epsilon_history.append(eps_rate)

        if (i_episode + 1) % 50 == 0:
            recent = np.mean(duration_history[-50:])
            print(f"Episode {i_episode+1:4d} | duration {t:4d} "
                  f"| 50-ep avg {recent:6.2f} | epsilon {eps_rate:.4f}")

    env.close()

    # ------------------------------------------------------------------
    # Save Q tables
    # ------------------------------------------------------------------
    np.save('Q1.npy', Q1)
    np.save('Q2.npy', Q2)

    # ------------------------------------------------------------------
    # Average episode duration over the LAST 100 training episodes
    # ------------------------------------------------------------------
    last100_avg = float(np.mean(duration_history[-100:]))
    print("\n" + "=" * 60)
    print(f"Average episode duration over the last 100 episodes: "
          f"{last100_avg:.2f}")
    print("=" * 60)

    # ------------------------------------------------------------------
    # Plot: episode duration vs. episode number with moving-average overlay
    # ------------------------------------------------------------------
    durations = np.asarray(duration_history)
    window    = 50
    moving_avg = np.convolve(durations,
                             np.ones(window) / window,
                             mode='valid')
    ma_x = np.arange(window - 1, len(durations))

    plt.figure(figsize=(10, 6))
    plt.plot(durations, alpha=0.35, label='Episode duration')
    plt.plot(ma_x, moving_avg, linewidth=2.2,
             label=f'{window}-episode moving average')
    plt.xlabel('Episode')
    plt.ylabel('Episode duration (timesteps)')
    plt.title('Tabular Double Q-Learning on CartPole-v1\n'
              f'Last-100-episode average: {last100_avg:.2f}')
    plt.legend(loc='upper left')
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig('tabular_dq_training_curve.png', dpi=130)
    print("\nSaved training curve to tabular_dq_training_curve.png")
```



3.1.2 One ablation

Code:

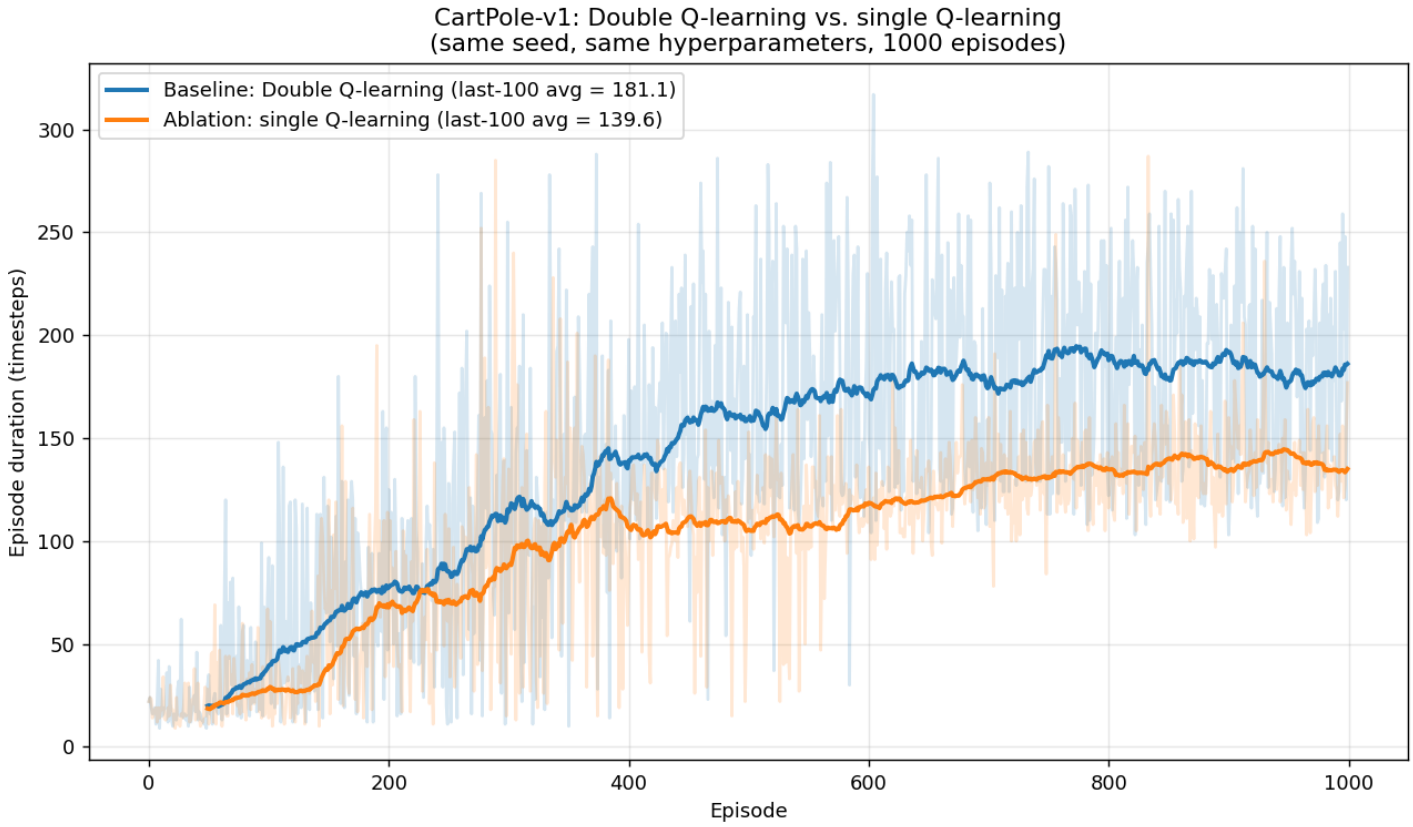
```python
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import random


# ------------------------------------------------------------------------
# Hyperparameters (identical across baseline and ablation)
# ------------------------------------------------------------------------
SEED              = 0
GAMMA             = 0.85
MAX_ITER          = 1000
EXPLORATION_MAX   = 1.0
EXPLORATION_MIN   = 0.01
EXPLORATION_DECAY = 1 + np.log(EXPLORATION_MIN) / MAX_ITER
REP_MEM_SIZE      = 100000
MINIBATCH_SIZE    = 64
ALPHA             = 0.001
N_STATES          = 162
N_ACTIONS         = 2


# ------------------------------------------------------------------------
# Discrete state mapping (from the lecture)
# ------------------------------------------------------------------------
def map_discrete_state(cs):
    ds_vector = np.zeros(4)

    if abs(cs[0]) <= 0.8:     ds_vector[0] = 1
    elif cs[0] < -0.8:        ds_vector[0] = 0
    elif cs[0] > 0.8:         ds_vector[0] = 2

```

```
    if abs(cs[1]) <= 0.5:    ds_vector[1] = 1
    elif cs[1] < -0.5:       ds_vector[1] = 0
    elif cs[1] > 0.5:        ds_vector[1] = 2

    angle = 180 / 3.1428 * cs[2]
    if   -12 < angle <= -6: ds_vector[2] = 0
    elif  -6 < angle <= -1: ds_vector[2] = 1
    elif  -1 < angle <=  0: ds_vector[2] = 2
    elif   0 < angle <=  1: ds_vector[2] = 3
    elif   1 < angle <=  6: ds_vector[2] = 4
    elif   6 < angle <= 12: ds_vector[2] = 5

    if abs(cs[3]) <= 50:    ds_vector[3] = 1
    elif cs[3] < -50:       ds_vector[3] = 0
    elif cs[3] > 50:        ds_vector[3] = 2

    return int(ds_vector[0] * 54 + ds_vector[1] * 18
               + ds_vector[2] * 3 + ds_vector[3])


def epsilon_greedy(Q, state, eps_rate, env):
    if np.random.random() < eps_rate:
        return env.action_space.sample()
    return int(np.argmax(Q[state, :]))


class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory   = []
    def push(self, s, a, r, ns):
        self.memory.append([s, a, r, ns])
        if len(self.memory) > self.capacity:
            del self.memory[0]
    def sample(self, k):
        return random.sample(self.memory, k)
    def __len__(self):
        return len(self.memory)


# ------------------------------------------------------------------------
# Generic training loop, parameterised by mode ("double" or "single")
# ------------------------------------------------------------------------
def train(mode):
    """
    mode = "double" -> baseline Double Q-learning (two tables Q1, Q2)
    mode = "single" -> ablation, standard Q-learning (one table Q)
    """
    assert mode in ("double", "single")

    # Re-seed every RNG so the two runs share an identical starting point.
    random.seed(SEED)
    np.random.seed(SEED)

    env = gym.make('CartPole-v1')
    env.action_space.seed(SEED)        # gymnasium action-space RNG (separate)

    if mode == "double":
        Q1 = np.zeros((N_STATES, N_ACTIONS))
        Q2 = np.zeros((N_STATES, N_ACTIONS))
    else:
        Q  = np.zeros((N_STATES, N_ACTIONS))

    eps_rate         = EXPLORATION_MAX
    duration_history = []
    repmemory        = ReplayMemory(REP_MEM_SIZE)
```

```
99
100     # --- update rule ---
101     def update(s, a, r, ns, eps_rate):
102         if mode == "double":
103             if np.random.random() < 0.5:
104                 na = epsilon_greedy(Q1, ns, eps_rate, env)
105                 Q1[s][a] += ALPHA * (r + GAMMA * Q2[ns][na] - Q1[s][a])
106             else:
107                 na = epsilon_greedy(Q2, ns, eps_rate, env)
108                 Q2[s][a] += ALPHA * (r + GAMMA * Q1[ns][na] - Q2[s][a])
109             return na
110         else:
111             # Standard single Q-learning: target uses max over a' of the SAME table.
112             na = epsilon_greedy(Q, ns, eps_rate, env)
113             Q[s][a] += ALPHA * (r + GAMMA * np.max(Q[ns, :]) - Q[s][a])
114             return na
115
116     # --- training ---
117     for i_episode in range(MAX_ITER):
118         state, _ = env.reset(seed=SEED if i_episode == 0 else None)
119         state = map_discrete_state(state)
120
121         if mode == "double":
122             action = epsilon_greedy(0.5 * (Q1 + Q2), state, 0, env)
123         else:
124             action = epsilon_greedy(Q, state, 0, env)
125
126         eps_rate = max(EXPLORATION_MIN, eps_rate * EXPLORATION_DECAY)
127
128         t = 0
129         while True:
130             next_state, reward, terminated, truncated, _ = env.step(action)
131             done = terminated or truncated
132             if done:
133                 reward = -10
134             next_state = map_discrete_state(next_state)
135             repmemory.push(state, action, reward, next_state)
136
137             next_action = update(state, action, reward, next_state, eps_rate)
138
139             if len(repmemory) > MINIBATCH_SIZE:
140                 for ts, ta, tr, tns in repmemory.sample(MINIBATCH_SIZE):
141                     update(ts, ta, tr, tns, eps_rate)
142
143             state, action = next_state, next_action
144             t += 1
145             if done:
146                 break
147
148         duration_history.append(t)
149
150         if (i_episode + 1) % 100 == 0:
151             recent = np.mean(duration_history[-50:])
152             print(f"  [{mode:6s}] episode {i_episode+1:4d} | "
153                   f"50-ep avg {recent:6.2f} | epsilon {eps_rate:.4f}")
154
155     env.close()
156     return np.asarray(duration_history)
157
158
159 # ------------------------------------------------------------------------
160 # Run both experiments
161 # ------------------------------------------------------------------------
162 if __name__ == "__main__":
163
164     print("Training BASELINE (Double Q-learning) ...")
```

```
    baseline = train("double")
    print(f"  -> last-100-episode average: {baseline[-100:].mean():.2f}\n")

    print("Training ABLATION (single Q-learning) ...")
    ablation = train("single")
    print(f"  -> last-100-episode average: {ablation[-100:].mean():.2f}\n")

    # ------------------------------------------------------------------
    # Comparison plot (50-episode moving averages on the same axes)
    # ------------------------------------------------------------------
    window = 50
    def moving_avg(x):
        return np.convolve(x, np.ones(window) / window, mode='valid')

    x_axis = np.arange(window - 1, MAX_ITER)

    plt.figure(figsize=(10, 6))
    plt.plot(baseline, alpha=0.18, color='C0')
    plt.plot(ablation, alpha=0.18, color='C1')
    plt.plot(x_axis, moving_avg(baseline), color='C0', linewidth=2.2,
             label=f'Baseline: Double Q-learning '
                   f'(last-100 avg = {baseline[-100:].mean():.1f})')
    plt.plot(x_axis, moving_avg(ablation), color='C1', linewidth=2.2,
             label=f'Ablation: single Q-learning '
                   f'(last-100 avg = {ablation[-100:].mean():.1f})')
    plt.xlabel('Episode')
    plt.ylabel('Episode duration (timesteps)')
    plt.title('CartPole-v1: Double Q-learning vs. single Q-learning\n'
              '(same seed, same hyperparameters, 1000 episodes)')
    plt.legend(loc='upper left')
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig('tabular_dq_ablation.png', dpi=130)

    print("=" * 64)
    print(f"Baseline (Double Q) last-100 avg: {baseline[-100:].mean():.2f}")
    print(f"Ablation (single Q) last-100 avg: {ablation[-100:].mean():.2f}")
    print("Saved comparison plot to tabular_dq_ablation.png")
    print("=" * 64)
```

I replaced the Double Q-learning update with the standard single-table Q-learning update, $Q[s][a] \leftarrow Q[s][a] + \alpha\big(r + \gamma \max_{a'} Q[s'][a'] - Q[s][a]\big)$, while keeping every other hyperparameter identical. The single-Q agent learns more slowly and plateaus around a 50-episode moving average of around 135 timesteps, while the Double-Q baseline continues climbing to around 185, which is about a 23% drop in final performance. The most likely reason is maximization bias: with a single table, the bootstrap target $\max_{a'} Q[s'][a']$ uses the same noisy estimates both to select the next action and to evaluate it, which produces a systematic positive bias when the Q-values still have substantial estimation noise. Double Q-learning decorrelates the two operations,i.e, one table picks the action, the other evaluates it, which removes most of that bias and yields more reliable bootstrap targets. With only 1000 episodes and $\alpha = 0.001$, the Q-tables are still quite noisy throughout training, which is exactly the regime where the double-table trick helps the most, so the gap we observe is consistent with the theory.



3.2 Task 2: DQN on LunarLander

3.2.1 Background

Background provided in Homework document.

3.2.2 Implementation requirements

Code:

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
dqn_lander.py

Deep Q-Network on LunarLander-v3.

Usage:
    python dqn_lander.py                    # train
    python dqn_lander.py --mode record      # roll out trained agent to GIF
    python dqn_lander.py --mode both        # train and then record

References:
    Lecture slides 83-95 (Kak, RL).
    Mnih et al., "Human-level control through deep RL", Nature 2015.
"""

import argparse
import random
from collections import deque

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
import torch.optim as optim
import matplotlib.pyplot as plt


class QNet(nn.Module):

    def __init__(self, state_dim: int = 8, action_dim: int = 4):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 128), nn.ReLU(),
            nn.Linear(128, 128),       nn.ReLU(),
            nn.Linear(128, action_dim),
        )

    def forward(self, s: torch.Tensor) -> torch.Tensor:
        return self.net(s)


def select_action(state, model):
    """
    Args:
        state : np.ndarray of shape (8,)
        model : a QNet instance

    Returns:
        int in {0, 1, 2, 3}
    """
    model.eval()
    with torch.no_grad():
        s = torch.as_tensor(state, dtype=torch.float32).unsqueeze(0)
        q = model(s)
        return int(q.argmax(dim=1).item())


# ============================================================================
# Replay buffer
# ============================================================================
class ReplayBuffer:
    """FIFO buffer storing (s, a, r, s', terminated) tuples."""

    def __init__(self, capacity: int = 100_000):
        self.buf = deque(maxlen=capacity)

    def push(self, s, a, r, s_next, terminated):
        self.buf.append((s, a, r, s_next, terminated))

    def sample(self, batch_size: int):
        batch = random.sample(self.buf, batch_size)
        s, a, r, s_next, term = zip(*batch)
        return (
            torch.as_tensor(np.array(s),       dtype=torch.float32),
            torch.as_tensor(a,                 dtype=torch.long).unsqueeze(1),
            torch.as_tensor(r,                 dtype=torch.float32).unsqueeze(1),
            torch.as_tensor(np.array(s_next),  dtype=torch.float32),
            torch.as_tensor(term,              dtype=torch.float32).unsqueeze(1),
        )

    def __len__(self):
        return len(self.buf)


# ============================================================================
# Training
# ============================================================================
def train(args):
    import gymnasium as gym
```

```
92
93      # ---- reproducibility ----
94      random.seed(args.seed)
95      np.random.seed(args.seed)
96      torch.manual_seed(args.seed)
97
98      env = gym.make(args.env)
99      env.action_space.seed(args.seed)          # gymnasium action-space RNG
100
101     state_dim  = env.observation_space.shape[0]
102     action_dim = env.action_space.n
103     assert state_dim == 8 and action_dim == 4, \
104         "This script targets LunarLander (8-d state, 4 discrete actions)."
105
106     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
107     print(f"device          : {device}")
108     print(f"env             : {args.env}")
109     print(f"episodes        : {args.episodes}")
110     print(f"gamma           : {args.gamma}")
111     print(f"lr              : {args.lr}")
112     print(f"buffer size     : {args.buffer_size}")
113     print(f"batch size      : {args.batch_size}")
114     print(f"target update   : Polyak tau = {args.tau} every gradient step")
115     print(f"epsilon decay   : 1.00 -> 0.05 linearly over first "
116           f"{args.eps_decay_episodes} episodes")
117     print()
118
119     # ---- networks ----
120     online = QNet(state_dim, action_dim).to(device)
121     target = QNet(state_dim, action_dim).to(device)
122     target.load_state_dict(online.state_dict())
123     for p in target.parameters():
124         p.requires_grad = False
125
126     optimizer = optim.Adam(online.parameters(), lr=args.lr)
127     buffer    = ReplayBuffer(args.buffer_size)
128
129     EPS_START, EPS_END = 1.0, 0.05
130     episode_returns = []
131     best_avg100     = -float('inf')
132     total_steps     = 0
133     solved          = False
134     solved_episode  = None
135
136     for episode in range(args.episodes):
137         frac    = min(1.0, episode / max(1, args.eps_decay_episodes))
138         epsilon = EPS_START + frac * (EPS_END - EPS_START)
139
140         state, _ = env.reset(seed=args.seed if episode == 0 else None)
141         done       = False
142         ep_return  = 0.0
143
144         while not done:
145             # ---- epsilon-greedy action ----
146             if random.random() < epsilon:
147                 action = env.action_space.sample()
148             else:
149                 with torch.no_grad():
150                     s = torch.as_tensor(state, dtype=torch.float32,
151                                         device=device).unsqueeze(0)
152                     action = int(online(s).argmax(dim=1).item())
153
154             next_state, reward, terminated, truncated, _ = env.step(action)
155             done = terminated or truncated
156
157             buffer.push(state, action, reward, next_state, float(terminated))
```

```
            state       = next_state
            ep_return  += reward
            total_steps += 1

            # ---- gradient step ----
            if (len(buffer) >= args.warmup
                    and total_steps % args.train_every == 0):

                s_b, a_b, r_b, s2_b, term_b = buffer.sample(args.batch_size)
                s_b   = s_b.to(device)
                a_b   = a_b.to(device)
                r_b   = r_b.to(device)
                s2_b  = s2_b.to(device)
                term_b = term_b.to(device)

                # current Q(s, a)
                q_sa = online(s_b).gather(1, a_b)

                # TD target  y = r + gamma * (1 - terminated) * max_a' Q_tgt(s', a')
                with torch.no_grad():
                    q_next = target(s2_b).max(dim=1, keepdim=True)[0]
                    y      = r_b + args.gamma * (1.0 - term_b) * q_next

                loss = F.smooth_l1_loss(q_sa, y)

                optimizer.zero_grad()
                loss.backward()
                nn.utils.clip_grad_norm_(online.parameters(), max_norm=10.0)
                optimizer.step()

                with torch.no_grad():
                    for p, p_t in zip(online.parameters(),
                                      target.parameters()):
                        p_t.data.mul_(1.0 - args.tau)
                        p_t.data.add_(args.tau * p.data)

        # ---- end of episode ----
        episode_returns.append(ep_return)
        avg100 = float(np.mean(episode_returns[-100:]))

        if (episode + 1) % 10 == 0:
            print(f"ep {episode+1:4d} | return {ep_return:8.2f} "
                  f"| avg100 {avg100:8.2f} | eps {epsilon:.3f} "
                  f"| buf {len(buffer):6d} | steps {total_steps}")

        # checkpoint best running average
        if len(episode_returns) >= 100 and avg100 > best_avg100:
            best_avg100 = avg100
            torch.save(online.state_dict(), args.checkpoint)

        if (not solved and len(episode_returns) >= 100 and avg100 >= 200.0):
            solved         = True
            solved_episode = episode + 1
            print(f"\n*** Solved at episode {solved_episode} "
                  f"(avg100 = {avg100:.2f}) ***\n")
            if args.stop_when_solved:
                break

    env.close()

    last100 = float(np.mean(episode_returns[-100:]))
    if last100 >= best_avg100:
        torch.save(online.state_dict(), args.checkpoint)
    print(f"\nSaved checkpoint to {args.checkpoint}")
```

```
    # ------------------------------------------------------------------
    # Plot training curve with 100-episode moving average
    # ------------------------------------------------------------------
    returns = np.asarray(episode_returns)
    window  = 100
    if len(returns) >= window:
        ma   = np.convolve(returns, np.ones(window) / window, mode='valid')
        ma_x = np.arange(window - 1, len(returns))
    else:
        ma, ma_x = np.array([]), np.array([])

    status = "SOLVED" if last100 >= 200.0 else "not solved"

    plt.figure(figsize=(10, 6))
    plt.plot(returns, alpha=0.3, label='Episode return')
    if len(ma):
        plt.plot(ma_x, ma, linewidth=2.2, label='100-episode moving average')
    plt.axhline(200, linestyle='--', color='gray', alpha=0.6,
                label='Solved threshold (+200)')
    plt.xlabel('Episode')
    plt.ylabel('Episode return')
    plt.title(f'DQN on {args.env}\n'
              f'Last-100-episode avg: {last100:.2f}  ({status})')
    plt.legend(loc='lower right')
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig(args.plot, dpi=130)
    print(f"Saved training curve to {args.plot}")

    print("=" * 64)
    print(f"Final 100-episode average return: {last100:.2f}  ({status})")
    if solved:
        print(f"First solved at episode {solved_episode}")
    print("=" * 64)


# ============================================================================
# Video recording of a trained episode
# ============================================================================
def record_video(args):
    import gymnasium as gym
    import imageio

    env = gym.make(args.env, render_mode="rgb_array")
    model = QNet().to(torch.device("cpu"))
    model.load_state_dict(torch.load(args.checkpoint, map_location="cpu"))
    model.eval()

    frames = []
    state, _ = env.reset(seed=args.seed)
    done, total = False, 0.0
    while not done:
        action = select_action(state, model)
        state, r, terminated, truncated, _ = env.step(action)
        frames.append(env.render())
        total += r
        done = terminated or truncated
    env.close()

    imageio.mimsave(args.gif, frames, fps=30)
    print(f"Saved trained-agent rollout to {args.gif}  (return = {total:.2f})")


def get_parser():
    p = argparse.ArgumentParser(description="DQN on LunarLander")
    p.add_argument('--env', type=str, default='LunarLander-v3',
```

```
                    help='Use LunarLander-v2 if v3 is not available')
    p.add_argument('--mode', type=str, default='train',
                   choices=['train', 'record', 'both'])

    # training hyperparameters
    p.add_argument('--episodes', type=int, default=1000)
    p.add_argument('--gamma', type=float, default=0.99)
    p.add_argument('--lr', type=float, default=5e-4)
    p.add_argument('--batch_size', type=int, default=64)
    p.add_argument('--buffer_size', type=int, default=100_000)
    p.add_argument('--warmup', type=int, default=1_000,
                   help='Min transitions in buffer before training begins')
    p.add_argument('--train_every', type=int, default=4,
                   help='Take a gradient step every N env steps')
    p.add_argument('--tau', type=float, default=0.005,
                   help='Polyak averaging coefficient for target network')
    p.add_argument('--eps_decay_episodes', type=int, default=300,
                   help='Linearly anneal eps from 1.0 to 0.05 over this many '
                        'episodes (10-30%% of training)')
    p.add_argument('--stop_when_solved', action='store_true',
                   help='Stop training as soon as the env is solved')

    # bookkeeping
    p.add_argument('--seed', type=int, default=0)
    p.add_argument('--checkpoint', type=str, default='dqn_lander.pt')
    p.add_argument('--plot', type=str,
                   default='dqn_lander_training_curve.png')
    p.add_argument('--gif', type=str, default='dqn_lander.gif')
    return p


if __name__ == "__main__":
    args = get_parser().parse_args()
    if args.mode in ('train', 'both'):
        train(args)
    if args.mode in ('record', 'both'):
        record_video(args)
```

I used option (b): Polyak averaging with $\tau = 0.005$, applied after every gradient step. The code that implements it in `train()`:

```
                with torch.no_grad():
                    for p, p_t in zip(online.parameters(),
                                      target.parameters()):
                        p_t.data.mul_(1.0 - args.tau)
                        p_t.data.add_(args.tau * p.data)
```

Implementation:

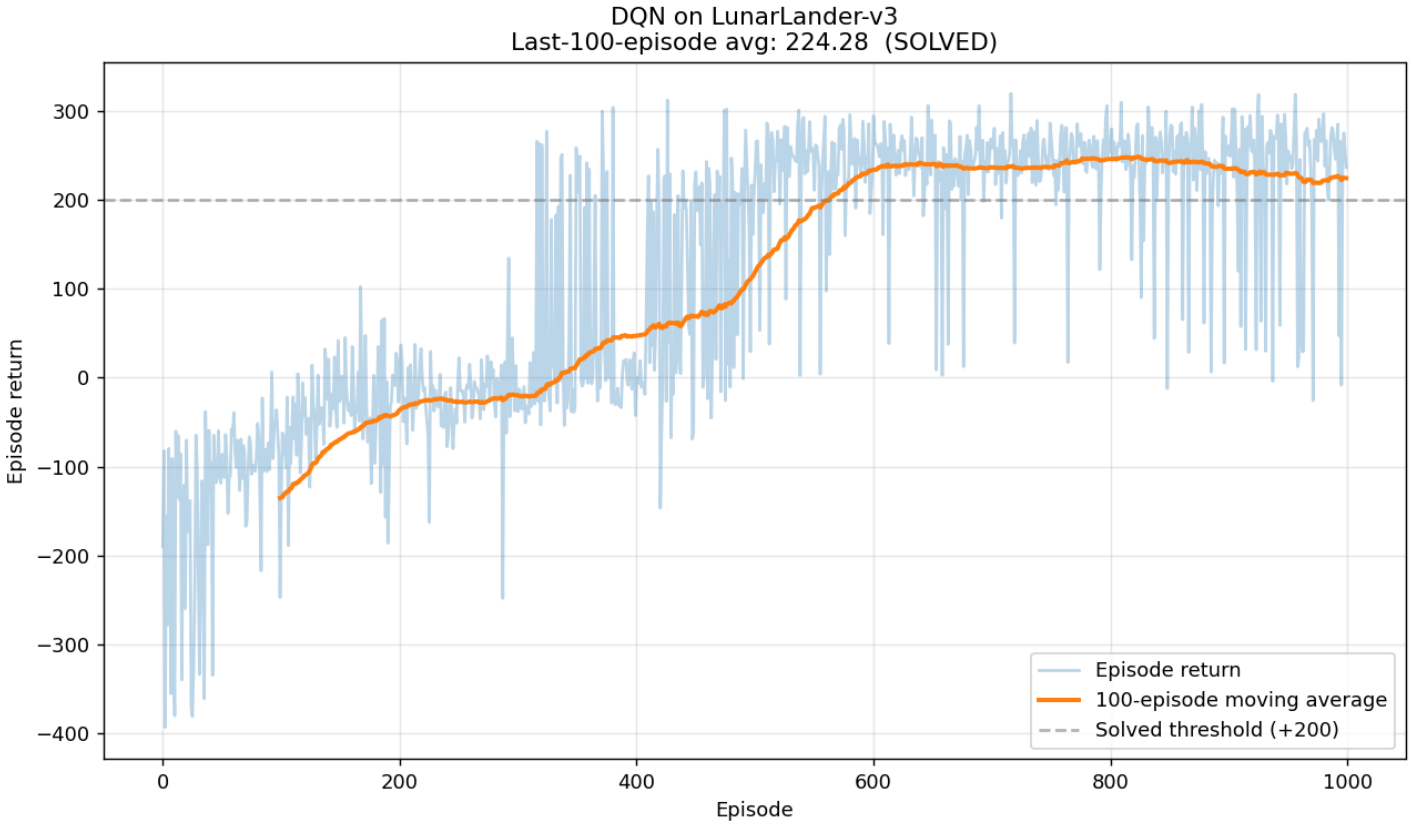
```
!python dqn_lander.py
```

Output:

```
device          : cuda
env             : LunarLander-v3
episodes        : 1000
gamma           : 0.99
lr              : 0.0005
buffer size     : 100000
batch size      : 64
target update   : Polyak tau = 0.005 every gradient step
epsilon decay   : 1.00 -> 0.05 linearly over first 300 episodes

ep   10 | return  -272.00 | avg100  -209.22 | eps 0.972 | buf    971 | steps 971
ep   20 | return  -259.54 | avg100  -191.44 | eps 0.940 | buf   1889 | steps 1889
ep   30 | return  -101.58 | avg100  -194.78 | eps 0.908 | buf   2892 | steps 2892
...
...
```

```
ep   540 | return    234.76 | avg100    175.79 | eps 0.050 | buf 100000 | steps 304355
ep   550 | return    255.69 | avg100    190.89 | eps 0.050 | buf 100000 | steps 308232
ep   560 | return    293.89 | avg100    198.03 | eps 0.050 | buf 100000 | steps 312223

*** Solved at episode 564 (avg100 = 200.91) ***

ep   570 | return    228.26 | avg100    206.80 | eps 0.050 | buf 100000 | steps 318193
ep   580 | return    223.63 | avg100    218.28 | eps 0.050 | buf 100000 | steps 322994
...
...
ep   990 | return    256.91 | avg100    225.43 | eps 0.050 | buf 100000 | steps 493556
ep  1000 | return    236.30 | avg100    224.28 | eps 0.050 | buf 100000 | steps 496811

Saved checkpoint to dqn_lander.pt
Saved training curve to dqn_lander_training_curve.png
============================================================
Final 100-episode average return: 224.28  (SOLVED)
First solved at episode 564
============================================================
```

Training curve showing episode return vs. episode number. Overlay a moving average over 100 episodes.



3.2.3 Brief analysis

Question: Why a target network? What would go wrong if you used the same Q_ϕ to produce both the predicted value and the TD target? Frame your answer in terms of what happens to the regression target during a single gradient step.

Answer: In a single gradient step, the regression target $y = r + \gamma \max_{a'} Q_\phi(s', a')$ is itself a function of the parameters ϕ being updated, so when the optimizer nudges $Q_\phi(s, a)$ toward y it simultaneously shifts y in (typically) the same direction,i.e. the target chases the prediction instead of standing still. This positive feedback loop amplifies any over-estimation in $Q_\phi(s', a')$, especially because the max operator already biases the bootstrap target upward by selecting the noisiest action's value. Holding a separate, slowly-updated copy Q_{ϕ^-} for the target makes the regression target a near-constant within each step, which is what gives gradient descent a stable thing to fit.

Question: ϵ-greedy decay. You annealed ϵ from 1.0 to 0.05 over training. Why not start at $\epsilon = 0.05$? Why not stay at $\epsilon = 1.0$?
Answer: Starting at $\epsilon = 0.05$ means the agent commits almost immediately to the argmax of an untrained, essentially random Q-network, so it would lock onto whatever bad action the initialization happens to favor and never visit enough of the state space (the landing pad, leg-contact transitions, the +100 success reward) to learn anything. The replay buffer would fill with garbage from one tiny region. Staying at $\epsilon = 1.0$ fixes the exploration problem but wastes the learning: every action is a coin flip regardless of how good Q has become, so the agent never actually exploits its policy and never sees the long, well-controlled landing trajectories needed to drive the 100-episode average toward +200. Annealing gives us both – broad exploration while the Q-values are still noise, then a gradual handoff to the learned greedy policy as those values become trustworthy, with the 0.05 floor preserving just enough exploration to keep escaping local optima late in training.

3.3 Task 3: Reward modeling on real human preferences

3.3.1 Background: what is reward modeling, and who is the 'critic'?

3.3.2 The dataset

3.3.3 One small adaptation of Slide 20

3.3.4 Starter code

3.3.5 Load the dataset and look at it

```python
from lm_utils import load_preference_dataset

triples = load_preference_dataset(n_examples=500)
print(f"loaded {len(triples)} triples")

for i in range(3):
    t = triples[i]
    print(f"\n=== Example {i+1} ===")
    print(f"POST:     {t['post']!r}")
    print(f"CHOSEN:   {t['chosen']!r}")
    print(f"REJECTED: {t['rejected']!r}")
```

```
loaded 500 triples

=== Example 1 ===
POST:     "My boyfriend and I are long distance. We have a trip planned this summer which
    involves me going over to him in the USA. This will be the second time I have actually been
    with him in person. I am flying from the UK with my mum to the east coast. The original plan
    was for me to fly over to my boyfriend in the west coast (my parents are holidaying on the
    east coast) but because my mum was freaking out so much about me going to meet my boyfriend i
    said we can all road trip there together. I even invited her on the trip with us. I have given
     her all of our dates so that she can travel around with us.\n\nThe plan was for me to stay on
     the 4th July and fly back on the 5th. Mum knew this. I told her I had booked a flight back
    already from the west coast to east coast (where she would pick me up and we would fly back to
     the UK together). She has gone mad at me because she can't believe I would book a flight when
     she told me she didn't want me flying on my own. At the time I had booked it she told me she
    wasn't gonna road trip with us. She knew the trip was happening.......how else was I to get
    home if I don't fly? \n\nI am fine flying on my own it doesn't bother me at all. I feel like I
     have done everything I can to make her feel comfortable with this trip and she is just trying
     to sabotage it. Thoughts??"
CHOSEN:   ' I have made sure my mother is comfortable with my boyfriend travelling on a trip and
    now my mother is mad because I booked it.'
REJECTED: ' Mum is mad at me for not flying on my own trip to meet my boyfriend.'

=== Example 2 ===
POST:     "My boyfriend and I are long distance. We have a trip planned this summer which
    involves me going over to him in the USA. This will be the second time I have actually been
    with him in person. I am flying from the UK with my mum to the east coast. The original plan
```

```
    was for me to fly over to my boyfriend in the west coast (my parents are holidaying on the
    east coast) but because my mum was freaking out so much about me going to meet my boyfriend i
    said we can all road trip there together. I even invited her on the trip with us. I have given
     her all of our dates so that she can travel around with us.\n\nThe plan was for me to stay on
     the 4th July and fly back on the 5th. Mum knew this. I told her I had booked a flight back
    already from the west coast to east coast (where she would pick me up and we would fly back to
     the UK together). She has gone mad at me because she can't believe I would book a flight when
     she told me she didn't want me flying on my own. At the time I had booked it she told me she
    wasn't gonna road trip with us. She knew the trip was happening.......how else was I to get
    home if I don't fly? \n\nI am fine flying on my own it doesn't bother me at all. I feel like I
     have done everything I can to make her feel comfortable with this trip and she is just trying
     to sabotage it. Thoughts??"
CHOSEN:   " mum isn't speaking to me because I booked a flight and she doesn't want me flying on
    my own."
REJECTED: ' I have made sure my mother is comfortable with my boyfriend travelling on a trip and
    now my mother is mad because I booked it.'

=== Example 3 ===
POST:     "My boyfriend and I are long distance. We have a trip planned this summer which
    involves me going over to him in the USA. This will be the second time I have actually been
    with him in person. I am flying from the UK with my mum to the east coast. The original plan
    was for me to fly over to my boyfriend in the west coast (my parents are holidaying on the
    east coast) but because my mum was freaking out so much about me going to meet my boyfriend i
    said we can all road trip there together. I even invited her on the trip with us. I have given
     her all of our dates so that she can travel around with us.\n\nThe plan was for me to stay on
     the 4th July and fly back on the 5th. Mum knew this. I told her I had booked a flight back
    already from the west coast to east coast (where she would pick me up and we would fly back to
     the UK together). She has gone mad at me because she can't believe I would book a flight when
     she told me she didn't want me flying on my own. At the time I had booked it she told me she
    wasn't gonna road trip with us. She knew the trip was happening.......how else was I to get
    home if I don't fly? \n\nI am fine flying on my own it doesn't bother me at all. I feel like I
     have done everything I can to make her feel comfortable with this trip and she is just trying
     to sabotage it. Thoughts??"
CHOSEN:   " mum isn't speaking to me because I booked a flight and she doesn't want me flying on
    my own."
REJECTED: ' Mum thought I was going to road trip with my boyfriend. I cancelled the flight. Mum
    is annoyed because she thought she would be traveling with me. I am fine with that.'
```

The chosen summaries are visibly better, particularly the one favored in Examples 2 and 3. They improve primarily in factual accuracy and hallucination reduction. The lower-ranked summaries either completely misinterpret the text (such as Example 1's chosen summary claiming the boyfriend is the one traveling) or invent entirely false details (such as Example 3's rejected summary claiming the author canceled the flight). The ultimately preferred summary ("mum isn't speaking to me because I booked a flight and she doesn't want me flying on my own") is the only one that more or less summarizes the actual core conflict without fabricating information.

3.3.6 Embed everything

```python
from lm_utils import load_gpt2, embed_triples

tokenizer, base, device = load_gpt2()
embedded = embed_triples(triples, tokenizer, base, device)
print(f"embedded {len(embedded)} triples")
print(f"post embedding shape:  {embedded[0]['post_emb'].shape}")

# 80/20 train/validation split
n_train = int(0.8 * len(embedded))
train_set = embedded[:n_train]
val_set   = embedded[n_train:]
print(f"train set: {len(train_set)} triples")
print(f"val set:   {len(val_set)} triples")
```

```
  embedded 50/500
  embedded 100/500
  embedded 150/500
```

```
  embedded 200/500
  embedded 250/500
  embedded 300/500
  embedded 350/500
  embedded 400/500
  embedded 450/500
  embedded 500/500
embedded 500 triples
post embedding shape:  torch.Size([768])
train set: 400 triples
val set:   100 triples
```

3.3.7 Train the RewardNetwork

Code:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
train_reward_model.py

HW11 Task 3.3: train Prof. Kak's RewardNetwork (Slide 35 of the RL lecture)
on the OpenAI summarize_from_feedback preference dataset.
"""

from __future__ import annotations

import argparse
import random

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt

from lm_utils import (
    load_preference_dataset,
    load_gpt2,
    embed_triples,
)

class RewardNetwork(torch.nn.Module):
    def __init__(self, embedding_size):
        torch.nn.Module.__init__(self)
        self.linear1 = nn.Linear(3 * embedding_size, 2 * embedding_size)
        self.linear2 = nn.Linear(2 * embedding_size, embedding_size)
        self.linear3 = nn.Linear(embedding_size, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, xSEPy):
        x = self.linear1(xSEPy).clamp(min=0)
        x = self.linear2(x).clamp(min=0)
        x = self.linear3(x)
        r = self.sigmoid(x)
        return r


def make_input(post_emb, sep_emb, response_emb):
    """
    Concatenate [post, SEP, response] into a single (3 * 768,) = (2304,)
    tensor in float32, matching the input shape RewardNetwork expects.
    """
    return torch.cat([post_emb, sep_emb, response_emb], dim=0).float()


```

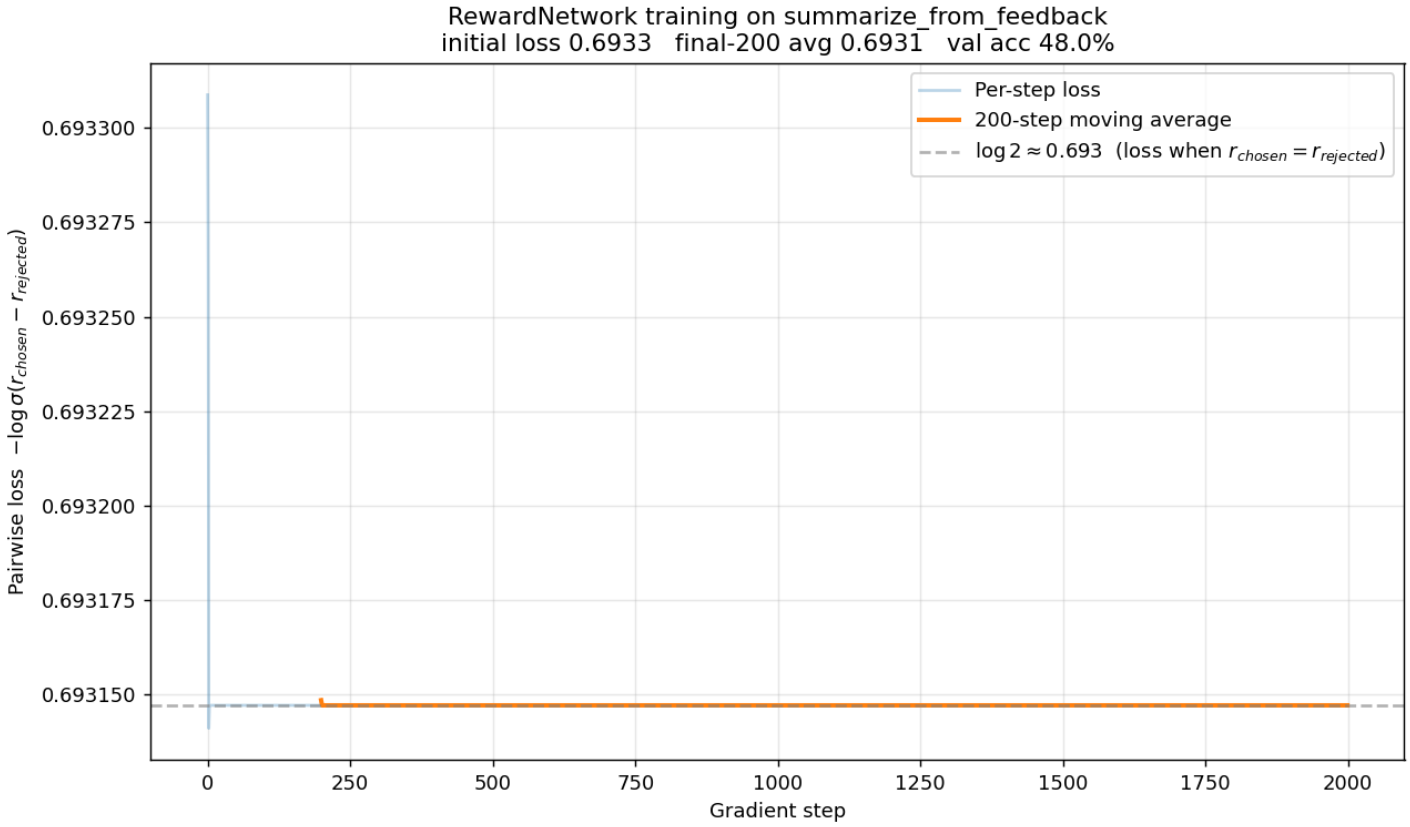
```python
# ============================================================================
# Training
# ============================================================================
def train_reward_model(train_set,
                       embedding_size: int = 768,
                       n_steps: int = 2000,
                       lr: float = 1e-3,
                       seed: int = 0):
    random.seed(seed)
    torch.manual_seed(seed)
    np.random.seed(seed)

    model     = RewardNetwork(embedding_size)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    losses = []
    for step in range(n_steps):
        triple = random.choice(train_set)

        # Build the two 2304-dim inputs.
        x_chosen   = make_input(triple['post_emb'],
                                triple['sep_emb'],
                                triple['chosen_emb'])
        x_rejected = make_input(triple['post_emb'],
                                triple['sep_emb'],
                                triple['rejected_emb'])

        # Forward: scalar reward for each response.
        r_chosen   = model(x_chosen).squeeze()
        r_rejected = model(x_rejected).squeeze()

        # Bradley-Terry pairwise loss (Eq. 6 specialized to two responses).
        loss = -F.logsigmoid(r_chosen - r_rejected)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

        if (step + 1) % 200 == 0:
            recent = float(np.mean(losses[-200:]))
            print(f"  step {step+1:4d} | loss {loss.item():.4f} "
                  f"| 200-step avg {recent:.4f}")

    return model, losses


# ============================================================================
# Validation
# ============================================================================
@torch.no_grad()
def evaluate(model, val_set):
    """Return fraction of validation triples with r_chosen > r_rejected."""
    model.eval()
    correct = 0
    for t in val_set:
        x_c = make_input(t['post_emb'], t['sep_emb'], t['chosen_emb'])
        x_r = make_input(t['post_emb'], t['sep_emb'], t['rejected_emb'])
        if model(x_c).item() > model(x_r).item():
            correct += 1
    model.train()
    return correct / len(val_set)


# ============================================================================
```

```python
# Main
# ================================================================================
def main(args):
    # ----  Load preference triples ---------------------------------------
    print(f"Loading {args.n_examples} preference triples ...")
    triples = load_preference_dataset(n_examples=args.n_examples)
    print(f"  loaded {len(triples)} triples\n")

    # ----  Embed everything with frozen distilgpt2 ------------------------
    print("Embedding posts and summaries with distilgpt2 ...")
    tokenizer, base, device = load_gpt2()
    embedded = embed_triples(triples, tokenizer, base, device)
    print(f"  embedded {len(embedded)} triples")
    print(f"  post embedding shape: {embedded[0]['post_emb'].shape}\n")

    # ----  80/20 split (no shuffle: keep duplicate-prompt rows together) ---
    n_train   = int(0.8 * len(embedded))
    train_set = embedded[:n_train]
    val_set   = embedded[n_train:]
    print(f"  train: {len(train_set)} | val: {len(val_set)}\n")

    # ---- Train ----------------------------------------------------------
    print(f"Training RewardNetwork for {args.n_steps} steps "
          f"(Adam, lr={args.lr}, batch size 1) ...")
    model, losses = train_reward_model(
        train_set,
        embedding_size=embedded[0]['post_emb'].shape[0],   # 768
        n_steps=args.n_steps,
        lr=args.lr,
        seed=args.seed,
    )

    # ---- Validation accuracy --------------------------------------------
    val_acc = evaluate(model, val_set)

    # ---- Print ----------------------------------------------
    initial_loss = losses[0]
    final_loss   = float(np.mean(losses[-200:]))

    print("\n" + "=" * 64)
    print(f"Initial loss (step 1):              {initial_loss:.4f}")
    print(f"Final loss (last-200-step average): {final_loss:.4f}")
    print(f"Validation accuracy (held-out 20%): {val_acc*100:.2f}%  "
          f"(random = 50%)")
    print("=" * 64)

    # ---- Save checkpoint ------------------------------------------------
    torch.save(model.state_dict(), args.checkpoint)
    print(f"\nSaved reward model to {args.checkpoint}")

    # ---- Loss curve -----------------------------------------------------
    losses_arr = np.asarray(losses)
    window     = 200
    if len(losses_arr) >= window:
        ma   = np.convolve(losses_arr, np.ones(window) / window, mode='valid')
        ma_x = np.arange(window - 1, len(losses_arr))
    else:
        ma, ma_x = np.array([]), np.array([])

    plt.figure(figsize=(10, 6))
    plt.plot(losses_arr, alpha=0.3, label='Per-step loss')
    if len(ma):
        plt.plot(ma_x, ma, linewidth=2.2,
                 label=f'{window}-step moving average')
    plt.axhline(np.log(2), linestyle='--', color='gray', alpha=0.6,
                label=r'$\log 2 \approx 0.693$  '
```

```
                      r'(loss when $r_{chosen}=r_{rejected}$)')
    plt.xlabel('Gradient step')
    plt.ylabel(r'Pairwise loss  $-\log \sigma(r_{chosen}-r_{rejected})$')
    plt.title('RewardNetwork training on summarize_from_feedback\n'
              f'initial loss {initial_loss:.4f}   '
              f'final-200 avg {final_loss:.4f}   '
              f'val acc {val_acc*100:.1f}%')
    plt.legend(loc='upper right')
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig(args.plot, dpi=130)
    print(f"Saved loss curve to {args.plot}")


def get_parser():
    p = argparse.ArgumentParser(
        description="Train RewardNetwork on summarize_from_feedback")
    p.add_argument('--n_examples', type=int, default=500)
    p.add_argument('--n_steps',    type=int, default=2000)
    p.add_argument('--lr',         type=float, default=1e-3)
    p.add_argument('--seed',       type=int, default=0)
    p.add_argument('--checkpoint', type=str, default='reward_model.pt')
    p.add_argument('--plot',       type=str,
                   default='reward_model_loss.png')
    return p


if __name__ == "__main__":
    main(get_parser().parse_args())
```

Output:

```
  embedded 50/500
  embedded 100/500
  embedded 150/500
  embedded 200/500
  embedded 250/500
  embedded 300/500
  embedded 350/500
  embedded 400/500
  embedded 450/500
  embedded 500/500
  embedded 500 triples
  post embedding shape: torch.Size([768])

  train: 400 | val: 100

Training RewardNetwork for 2000 steps (Adam, lr=0.001, batch size 1) ...
  step  200 | loss 0.6931 | 200-step avg 0.6931
  step  400 | loss 0.6931 | 200-step avg 0.6931
  step  600 | loss 0.6931 | 200-step avg 0.6931
  step  800 | loss 0.6931 | 200-step avg 0.6931
  step 1000 | loss 0.6931 | 200-step avg 0.6931
  step 1200 | loss 0.6931 | 200-step avg 0.6931
  step 1400 | loss 0.6931 | 200-step avg 0.6931
  step 1600 | loss 0.6931 | 200-step avg 0.6931
  step 1800 | loss 0.6931 | 200-step avg 0.6931
  step 2000 | loss 0.6931 | 200-step avg 0.6931

============================================================
Initial loss (step 1):              0.6933
Final loss (last-200-step average): 0.6931
Validation accuracy (held-out 20%): 48.00%  (random = 50%)
============================================================

Saved reward model to reward_model.pt
Saved loss curve to reward_model_loss.png
```



3.3.8 Evaluate and critique

```
Validation accuracy (held-out 20%): 48.00%  (random = 50%)
```

Question: Report your validation accuracy and what happened to the training loss. How did the values you saw compare to what you would expect from a reward model that is learning to capture human preferences?
Answer: Validation accuracy was 48.0% which is indistinguishable from random guessing and the training loss only moved from 0.6933 to a final 200-step average of 0.6931, a change of approx 2×10^{-4} over 2000 gradient steps. A reward model that was actually learning human preferences would push the loss meaningfully below $\ln 2 \approx 0.693$ and reach higher than 50% validation accuracy on this dataset.

Question: Look carefully at the `RewardNetwork` class from Slide 35. Work out what values the forward pass can return: start with the three `clamp(min=0)` operations, then the final sigmoid, and describe the range of outputs. Given the pairwise loss $-\log \sigma(r_{\text{chosen}} - r_{\text{rejected}})$, how does this output range affect the loss values you can reach during training?
Answer: The two `clamp(min=0)` ops act as ReLUs and keep the hidden activations in $[0, \infty)$; `linear3` produces an unbounded scalar, and the final sigmoid squashes it to $(0, 1)$. So $r_{\text{chosen}} - r_{\text{rejected}} \in (-1, 1)$, which means the pairwise loss is bounded below by $-\log \sigma(1) \approx 0.313$ and starts near $-\log \sigma(0) = \log 2 \approx 0.693$ at initialization. The narrow window combined with sigmoid gradient saturation as the score difference approaches ± 1 leaves almost no room for the loss to descend, which is exactly why mine got pinned at approx 0.693.

Question: Suggest one modification to the RewardNetwork that you think would expand the range of its outputs and allow the loss to decrease further. You are not required to implement it or verify that it works; a short description is enough.
Answer: Removing the final nn.Sigmoid() and letting linear3 output a raw, unbounded scalar will lead to $r_{\text{chosen}} - r_{\text{rejected}}$ ranging over all of R, so the loss can approach 0 and gradients no longer would vanish near the bound.

4 Bonus Task: PPO on LunarLander

4.1 What to implement

4.2 Deliverables

```
"""
ppo_lander.py

HW11 Bonus Task: Proximal Policy Optimization on LunarLander-v3.

Adapted from CleanRL's reference implementation `cleanrl/ppo.py`
(Huang et al., JMLR 2022; https://github.com/vwxyzjn/cleanrl).
The structure below mirrors CleanRL closely.
  - separate Actor and Critic networks rather than a shared trunk
  - LunarLander-specific hyperparameters,
  - logging is per-episode-return rather than wall-clock SPS,
  - saves episode returns to a .npy so you can overlay the DQN training
    curve from Task 2 on the same axes.

Citation:
    Huang, Dossa, Raffin, Kanervisto, Wang.
    "CleanRL: High-quality Single-file Implementations of Deep
     Reinforcement Learning Algorithms." JMLR, 2022.
    https://github.com/vwxyzjn/cleanrl

Usage:
    python ppo_lander.py                      # train
    python ppo_lander.py --mode compare       # overlay DQN vs PPO curves
    python ppo_lander.py --mode record        # roll out trained agent to GIF
"""

from __future__ import annotations

import argparse
import os
import random
from dataclasses import dataclass

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
import matplotlib.pyplot as plt



# Networks
def layer_init(layer: nn.Linear, std: float = np.sqrt(2),
               bias_const: float = 0.0) -> nn.Linear:
    nn.init.orthogonal_(layer.weight, std)
    nn.init.constant_(layer.bias, bias_const)
    return layer


class Actor(nn.Module):
    def __init__(self, state_dim: int = 8, action_dim: int = 4):
        super().__init__()
        self.net = nn.Sequential(
            layer_init(nn.Linear(state_dim, 128)), nn.Tanh(),
            layer_init(nn.Linear(128, 128)),       nn.Tanh(),
            layer_init(nn.Linear(128, action_dim), std=0.01),
        )

    def forward(self, s: torch.Tensor) -> torch.Tensor:
```

```
        return self.net(s)                          # raw logits

    def get_action_and_logprob(self, s: torch.Tensor,
                               action: torch.Tensor = None):
        """Sample (or evaluate) an action and return (action, logp, entropy)."""
        logits = self.net(s)
        dist   = Categorical(logits=logits)
        if action is None:
            action = dist.sample()
        return action, dist.log_prob(action), dist.entropy()


class Critic(nn.Module):
    def __init__(self, state_dim: int = 8):
        super().__init__()
        self.net = nn.Sequential(
            layer_init(nn.Linear(state_dim, 128)), nn.Tanh(),
            layer_init(nn.Linear(128, 128)),       nn.Tanh(),
            layer_init(nn.Linear(128, 1), std=1.0),
        )

    def forward(self, s: torch.Tensor) -> torch.Tensor:
        return self.net(s).squeeze(-1)



# Rollout buffer
@dataclass
class Rollout:
    """Storage for one rollout of length T transitions."""
    states:    torch.Tensor                         # (T, state_dim)
    actions:   torch.Tensor                         # (T,)
    logprobs:  torch.Tensor                         # (T,)
    rewards:   torch.Tensor                         # (T,)
    dones:     torch.Tensor                         # (T,) -- terminated only
    values:    torch.Tensor                         # (T,)


def compute_gae(rollout: Rollout, last_value: torch.Tensor,
                gamma: float, lam: float):
    """
    Generalized Advantage Estimation (Schulman 2016, Eq. 16).

        delta_t = r_t + gamma * (1 - done_t) * V(s_{t+1}) - V(s_t)
        A_t     = delta_t + gamma * lam * (1 - done_t) * A_{t+1}
        R_t     = A_t + V(s_t)        # value-fn regression target

    Returns (advantages, returns), each shape (T,).
    """
    T = rollout.rewards.shape[0]
    advantages = torch.zeros_like(rollout.rewards)
    last_gae   = 0.0
    for t in reversed(range(T)):
        if t == T - 1:
            next_value   = last_value
            next_nonterm = 1.0 - rollout.dones[t]
        else:
            next_value   = rollout.values[t + 1]
            next_nonterm = 1.0 - rollout.dones[t]
        delta = (rollout.rewards[t]
                 + gamma * next_value * next_nonterm
                 - rollout.values[t])
        last_gae      = delta + gamma * lam * next_nonterm * last_gae
        advantages[t] = last_gae
    returns = advantages + rollout.values
    return advantages, returns
```

```python


# Training
def train(args):
    import gymnasium as gym

    # ---- reproducibility ----
    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.backends.cudnn.deterministic = True

    env = gym.make(args.env)
    env.action_space.seed(args.seed)

    state_dim  = env.observation_space.shape[0]
    action_dim = env.action_space.n
    assert state_dim == 8 and action_dim == 4, \
        "This script targets LunarLander (8-d state, 4 discrete actions)."

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"device          : {device}")
    print(f"env             : {args.env}")
    print(f"total timesteps : {args.total_timesteps}")
    print(f"rollout length  : {args.num_steps}")
    print(f"gamma           : {args.gamma}")
    print(f"gae lambda      : {args.gae_lambda}")
    print(f"clip coef       : {args.clip_coef}")
    print(f"update epochs   : {args.update_epochs}")
    print(f"minibatch size  : {args.minibatch_size}")
    print(f"lr              : {args.lr} (annealed: {args.anneal_lr})\n")

    # ---- networks & optimizer ----
    actor  = Actor(state_dim, action_dim).to(device)
    critic = Critic(state_dim).to(device)
    optimizer = optim.Adam(
        list(actor.parameters()) + list(critic.parameters()),
        lr=args.lr, eps=1e-5,                       # CleanRL uses Adam eps=1e-5
    )

    # ---- rollout storage (reused each iteration) ----
    states_buf   = torch.zeros((args.num_steps, state_dim)).to(device)
    actions_buf  = torch.zeros((args.num_steps,), dtype=torch.long).to(device)
    logprobs_buf = torch.zeros((args.num_steps,)).to(device)
    rewards_buf  = torch.zeros((args.num_steps,)).to(device)
    dones_buf    = torch.zeros((args.num_steps,)).to(device)
    values_buf   = torch.zeros((args.num_steps,)).to(device)

    # ---- env state ----
    obs, _ = env.reset(seed=args.seed)
    next_state = torch.as_tensor(obs, dtype=torch.float32, device=device)
    next_done  = torch.zeros(1, device=device)

    # ---- bookkeeping ----
    episode_returns = []
    current_return  = 0.0
    current_length  = 0
    global_step     = 0
    best_avg100     = -float('inf')
    solved          = False
    solved_episode  = None

    n_iterations = args.total_timesteps // args.num_steps

    for iteration in range(n_iterations):
```

```
        # ---- optional learning-rate annealing ----
        if args.anneal_lr:
            frac    = 1.0 - iteration / n_iterations
            new_lr  = frac * args.lr
            for g in optimizer.param_groups:
                g['lr'] = new_lr

        # ROLLOUT  --  collect num_steps transitions
        for step in range(args.num_steps):
            global_step += 1
            states_buf[step] = next_state
            dones_buf[step]  = next_done

            with torch.no_grad():
                action, logp, _ = actor.get_action_and_logprob(
                    next_state.unsqueeze(0))
                value = critic(next_state.unsqueeze(0))

            actions_buf[step]  = action.squeeze()
            logprobs_buf[step] = logp.squeeze()
            values_buf[step]   = value.squeeze()

            # Step env
            obs, reward, terminated, truncated, _ = env.step(
                int(action.item()))
            rewards_buf[step] = float(reward)
            done = terminated or truncated

            current_return += float(reward)
            current_length += 1

            if done:
                episode_returns.append(current_return)
                avg100 = float(np.mean(episode_returns[-100:]))

                if len(episode_returns) % 10 == 0:
                    print(f"  ep {len(episode_returns):4d} | "
                          f"step {global_step:7d} | "
                          f"return {current_return:8.2f} | "
                          f"avg100 {avg100:8.2f} | "
                          f"len {current_length}")

                if (len(episode_returns) >= 100
                        and avg100 > best_avg100):
                    best_avg100 = avg100
                    torch.save({'actor':  actor.state_dict(),
                                'critic': critic.state_dict()},
                               args.checkpoint)

                if (not solved and len(episode_returns) >= 100
                        and avg100 >= 200.0):
                    solved         = True
                    solved_episode = len(episode_returns)
                    print(f"\n*** Solved at episode {solved_episode} "
                          f"(global step {global_step}, "
                          f"avg100 = {avg100:.2f}) ***\n")
                    if args.stop_when_solved:
                        break

                obs, _ = env.reset()
                current_return = 0.0
                current_length = 0

            next_state = torch.as_tensor(obs, dtype=torch.float32,
                                         device=device)
            next_done  = torch.tensor([float(terminated)], device=device)
```

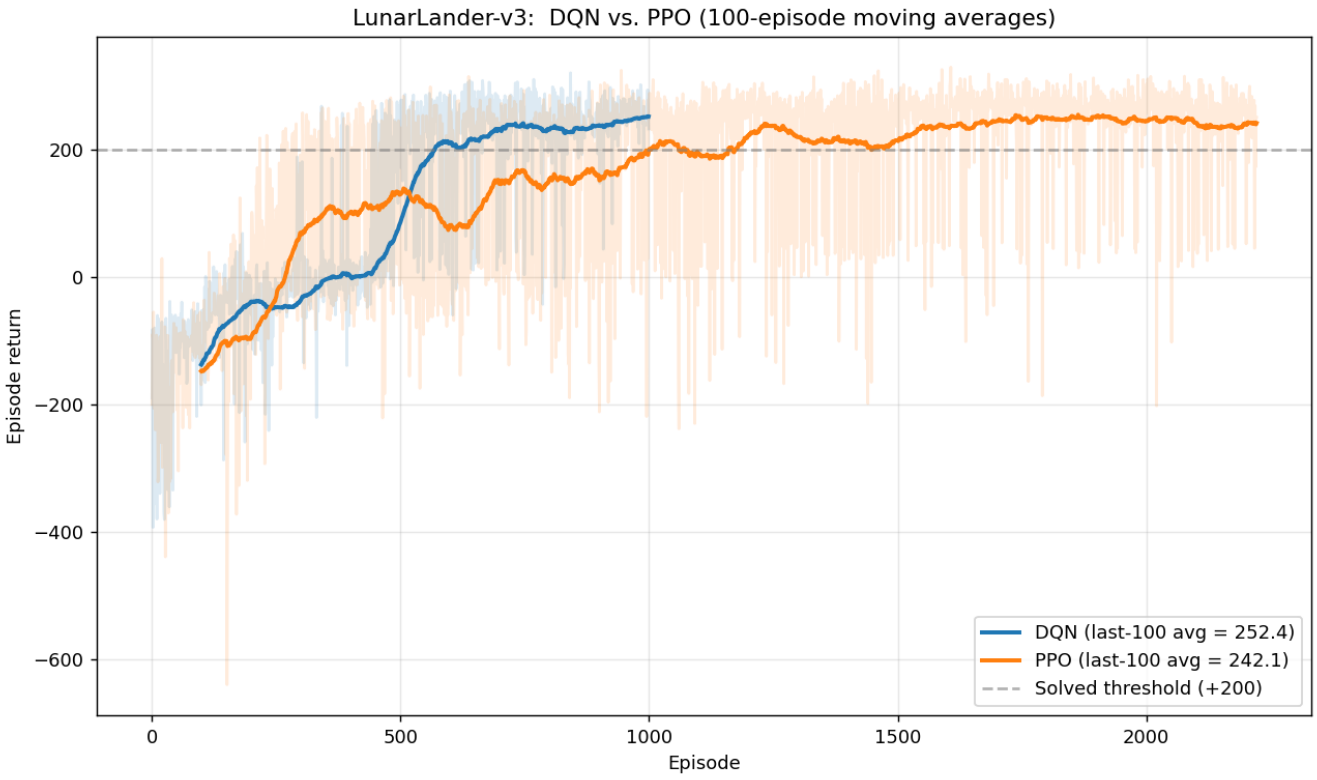
```
        if solved and args.stop_when_solved:
            break

        # GAE  --  compute advantages and value targets
        with torch.no_grad():
            last_value = critic(next_state.unsqueeze(0)).squeeze()
        rollout = Rollout(states_buf, actions_buf, logprobs_buf,
                          rewards_buf, dones_buf, values_buf)
        advantages, returns = compute_gae(rollout, last_value,
                                          args.gamma, args.gae_lambda)

        # UPDATE  --  K epochs over the rollout in random minibatches
        b_states     = states_buf
        b_actions    = actions_buf
        b_logprobs   = logprobs_buf
        b_advantages = advantages
        b_returns    = returns
        b_values     = values_buf

        idxs = np.arange(args.num_steps)
        for epoch in range(args.update_epochs):
            np.random.shuffle(idxs)
            for start in range(0, args.num_steps, args.minibatch_size):
                end = start + args.minibatch_size
                mb  = idxs[start:end]

                _, new_logp, entropy = actor.get_action_and_logprob(
                    b_states[mb], b_actions[mb])
                new_value = critic(b_states[mb])

                # ---- policy loss (clipped surrogate, Schulman 2017 Eq. 7) ----
                ratio   = (new_logp - b_logprobs[mb]).exp()
                mb_adv  = b_advantages[mb]
                mb_adv  = (mb_adv - mb_adv.mean()) / (mb_adv.std() + 1e-8)

                pg_loss1 = -mb_adv * ratio
                pg_loss2 = -mb_adv * torch.clamp(ratio,
                                                 1 - args.clip_coef,
                                                 1 + args.clip_coef)
                pg_loss = torch.max(pg_loss1, pg_loss2).mean()

                # ---- value loss (clipped, CleanRL convention) ----
                v_unclipped = (new_value - b_returns[mb]).pow(2)
                v_clipped   = b_values[mb] + torch.clamp(
                    new_value - b_values[mb],
                    -args.clip_coef, args.clip_coef)
                v_clipped_loss = (v_clipped - b_returns[mb]).pow(2)
                v_loss = 0.5 * torch.max(v_unclipped, v_clipped_loss).mean()

                # ---- entropy bonus (encourages exploration) ----
                ent_loss = entropy.mean()

                loss = (pg_loss
                        + args.vf_coef * v_loss
                        - args.ent_coef * ent_loss)

                optimizer.zero_grad()
                loss.backward()
                nn.utils.clip_grad_norm_(
                    list(actor.parameters()) + list(critic.parameters()),
                    args.max_grad_norm)
                optimizer.step()

    env.close()
```

```
    # ------------------------------------------------------------------------
    # Save final stats
    # ------------------------------------------------------------------------
    last100 = float(np.mean(episode_returns[-100:])) \
        if len(episode_returns) >= 100 else float(np.mean(episode_returns))
    if last100 >= best_avg100:
        torch.save({'actor':  actor.state_dict(),
                    'critic': critic.state_dict()}, args.checkpoint)
    np.save(args.returns_file, np.asarray(episode_returns))
    print(f"\nSaved checkpoint to {args.checkpoint}")
    print(f"Saved per-episode returns to {args.returns_file}")

    # ------------------------------------------------------------------------
    # Plot training curve
    # ------------------------------------------------------------------------
    returns = np.asarray(episode_returns)
    window  = 100
    if len(returns) >= window:
        ma   = np.convolve(returns, np.ones(window) / window, mode='valid')
        ma_x = np.arange(window - 1, len(returns))
    else:
        ma, ma_x = np.array([]), np.array([])

    status = "SOLVED" if last100 >= 200.0 else "not solved"

    plt.figure(figsize=(10, 6))
    plt.plot(returns, alpha=0.3, label='Episode return')
    if len(ma):
        plt.plot(ma_x, ma, linewidth=2.2, label='100-episode moving average')
    plt.axhline(200, linestyle='--', color='gray', alpha=0.6,
                label='Solved threshold (+200)')
    plt.xlabel('Episode')
    plt.ylabel('Episode return')
    plt.title(f'PPO on {args.env}\n'
              f'Last-100-episode avg: {last100:.2f}  ({status})')
    plt.legend(loc='lower right')
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig(args.plot, dpi=130)
    print(f"Saved training curve to {args.plot}")

    print("=" * 64)
    print(f"Final 100-episode average return: {last100:.2f}  ({status})")
    if solved:
        print(f"First solved at episode {solved_episode}")
    print("=" * 64)


# ============================================================================
# Comparison plot:  DQN (Task 2) vs. PPO (Task 4)
# ============================================================================
def compare(args):
    """Overlay DQN's per-episode returns and PPO's per-episode returns."""
    if not os.path.exists(args.dqn_returns):
        raise FileNotFoundError(
            f"Cannot find DQN returns at {args.dqn_returns}.")
    if not os.path.exists(args.returns_file):
        raise FileNotFoundError(
            f"Cannot find PPO returns at {args.returns_file}.")

    dqn = np.load(args.dqn_returns)
    ppo = np.load(args.returns_file)

    window = 100
    def ma(x):
        if len(x) < window:
```

```
            return np.array([]), np.array([])
        m = np.convolve(x, np.ones(window) / window, mode='valid')
        return np.arange(window - 1, len(x)), m

    dqn_x, dqn_ma = ma(dqn)
    ppo_x, ppo_ma = ma(ppo)

    plt.figure(figsize=(10, 6))
    plt.plot(dqn, alpha=0.15, color='C0')
    plt.plot(ppo, alpha=0.15, color='C1')
    if len(dqn_ma):
        plt.plot(dqn_x, dqn_ma, color='C0', linewidth=2.2,
                 label=f'DQN (last-100 avg = {dqn[-100:].mean():.1f})')
    if len(ppo_ma):
        plt.plot(ppo_x, ppo_ma, color='C1', linewidth=2.2,
                 label=f'PPO (last-100 avg = {ppo[-100:].mean():.1f})')
    plt.axhline(200, linestyle='--', color='gray', alpha=0.6,
                label='Solved threshold (+200)')
    plt.xlabel('Episode')
    plt.ylabel('Episode return')
    plt.title(f'LunarLander-v3:  DQN vs. PPO (100-episode moving averages)')
    plt.legend(loc='lower right')
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.savefig(args.compare_plot, dpi=130)
    print(f"Saved comparison plot to {args.compare_plot}")


# ============================================================================
# Video recording of a trained agent
# ============================================================================
def record_video(args):
    import gymnasium as gym
    import imageio

    env = gym.make(args.env, render_mode="rgb_array")
    actor = Actor()
    ckpt = torch.load(args.checkpoint, map_location="cpu")
    actor.load_state_dict(ckpt['actor'])
    actor.eval()

    frames = []
    obs, _ = env.reset(seed=args.seed)
    done, total = False, 0.0
    while not done:
        with torch.no_grad():
            s      = torch.as_tensor(obs, dtype=torch.float32).unsqueeze(0)
            logits = actor(s)
            action = int(logits.argmax(dim=1).item())   # greedy at eval time
        obs, r, terminated, truncated, _ = env.step(action)
        frames.append(env.render())
        total += r
        done = terminated or truncated
    env.close()
    imageio.mimsave(args.gif, frames, fps=30)
    print(f"Saved trained-agent rollout to {args.gif}  (return = {total:.2f})")


def get_parser():
    p = argparse.ArgumentParser(description="PPO on LunarLander")
    p.add_argument('--env', type=str, default='LunarLander-v3',
                   help='Use LunarLander-v2 if v3 is not available')
    p.add_argument('--mode', type=str, default='train',
                   choices=['train', 'compare', 'record'])

    # PPO hyperparameters (CleanRL defaults tuned for classic control)
```

```
    p.add_argument('--total_timesteps', type=int, default=600_000)
    p.add_argument('--num_steps',       type=int, default=1024,
                   help='Steps per rollout, before each PPO update.')
    p.add_argument('--gamma',           type=float, default=0.99)
    p.add_argument('--gae_lambda',      type=float, default=0.95)
    p.add_argument('--clip_coef',       type=float, default=0.2)
    p.add_argument('--ent_coef',        type=float, default=0.01)
    p.add_argument('--vf_coef',         type=float, default=0.5)
    p.add_argument('--max_grad_norm',   type=float, default=0.5)
    p.add_argument('--update_epochs',   type=int,   default=10)
    p.add_argument('--minibatch_size',  type=int,   default=64)
    p.add_argument('--lr',              type=float, default=3e-4)
    p.add_argument('--anneal_lr',       action='store_true', default=True)
    p.add_argument('--stop_when_solved', action='store_true')

    # Bookkeeping
    p.add_argument('--seed',            type=int, default=0)
    p.add_argument('--checkpoint',      type=str, default='ppo_lander.pt')
    p.add_argument('--plot',            type=str,
                   default='ppo_lander_training_curve.png')
    p.add_argument('--returns_file',    type=str, default='ppo_returns.npy')
    p.add_argument('--dqn_returns',     type=str, default='dqn_returns.npy')
    p.add_argument('--compare_plot',    type=str,
                   default='dqn_vs_ppo.png')
    p.add_argument('--gif',             type=str, default='ppo_lander.gif')
    return p


if __name__ == "__main__":
    args = get_parser().parse_args()
    if args.mode == 'train':
        train(args)
    elif args.mode == 'compare':
        compare(args)
    elif args.mode == 'record':
        record_video(args)
```

Training curve comparison:



Average return over the last 100 episodes for PPO:

```
============================================================
Final 100-episode average return: 242.07  (SOLVED)
First solved at episode 1001
============================================================
```

DQN solved the environment significantly faster (episode 569 vs. PPO's 1001) and achieved a slightly higher final average return (252.4 vs. 242.1). However, because DQN heavily relies on environment-specific hyperparameters such as the hardcoded 300-episode epsilon decay schedule and target network updates, PPO is generally considered easier to tune and is the standard default choice for new problems.