ece60146hw11

Xinyu Chen

May 2, 2026

3 Programming Tasks

3.1 Task 1: Tabular Double Q-Learning on CartPole (15 points)

3.1.1 Reproducing the baseline (8 points)

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
## Cartpole_DQL.v2.py
## Balancing a cartpole using Double Q-learning (without a neural network) and using
## a discretized state space. The algorithm is trained using epsilon-greedy approach
## with Replay Memory
##
## @author: Rohan Sarkar (sarkarr@purdue.edu)
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import random

seed = 0
random.seed(seed)
np.random.seed(seed)

## Initialize parameters and environment variables:
ENV_NAME = "CartPole-v1"
BASELINE_GAMMA = 0.85
ABLATION_GAMMA = 0.99
# MAX_ITER = 500
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
cum_reward = 0
train_reward_history = []


## Map the four dimensional continuous state-space to discretized state-space:
## Credit: http://www.derongliu.org/adp/adp-cdrom/Barto1983.pdf.
def map_discrete_state(cs):  ## (B)
    ds_vector = np.zeros(4)
    ds = -1

    # Discretize x (position)
    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

    # Discretize x' (velocity)
    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

    # Discretize theta (angle)
    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

    # Discretize theta' (angular velocity)
```

```
    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

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


## Return the most optimal action and the corresponding Q
    value:
def optimal_action(Q, state, eps_rate, env):  ## (C)
    p = np.random.random()

    # Choose random action if in 'exploration' mode
    if p < eps_rate:
        action = env.action_space.sample()  # choose
            randomly between 0 and 1
    else:
        # Choose optimal action based on learned weights if
            in 'exploitation' mode
        action = np.argmax(Q[state, :])

    return action


## Update Qvalues based on the logic of the Double Q-
    learning: ## (D)
def updateQValues(state, action, reward, next_state, alpha,
    eps_rate, env, gamma, Q1, Q2):
    p = np.random.random()

    if p < 0.5:
        # Update Qvalues for Table Q1
        next_action = optimal_action(Q1, next_state,
            eps_rate, env)
        Q1[state][action] = Q1[state][action] + alpha * (
            reward + gamma * Q2[next_state][next_action] -
                Q1[state][action]
        )
    else:
        # Update Qvalues for Table Q2
        next_action = optimal_action(Q2, next_state,
            eps_rate, env)
        Q2[state][action] = Q2[state][action] + alpha * (
            reward + gamma * Q1[next_state][next_action] -
                Q2[state][action]
```

```
        )

    return next_action


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)


def run_training(gamma, label):
    ## Learn the weights for Double Q learning:
    random.seed(seed)
    np.random.seed(seed)

    env = gym.make(ENV_NAME)  ## (A)
    env.action_space.seed(seed)
    Q1 = np.zeros((N_states, N_actions))
    Q2 = np.zeros((N_states, N_actions))
    eps_rate = EXPLORATION_MAX
    cum_reward = 0
    max_t = 0
    duration_history = []
    history = []
    epsilon_history = []
    train_reward_history = []
    repmemory = ReplayMemory(REP_MEM_SIZE)

    for i_episode in range(MAX_ITER):
        ## state is a 4-tuple: [cart_pos, cart_vel,
           pole_angle, pole_ang_vel]
        if i_episode == 0:
            state, info = env.reset(seed=seed)
        else:
            state, info = env.reset()  ### state set to
                uniformly dist random bet -0.05 and 0.05
        done = False

```

```
        # Initial state and action selection when environment restarts
        state = map_discrete_state(state)
        action = optimal_action(0.5 * (Q1 + Q2), state, 0, env)
        t = 0

        # Decay the exploration parameter in epsilon-greedy approach.
        eps_rate *= EXPLORATION_DECAY
        eps_rate = max(EXPLORATION_MIN, eps_rate)

        while True:
            # env.render()
            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)

            # Update Q table using Double Q learning and get the next optimal action.
            next_action = updateQValues(
                state, action, reward, next_state, alpha, eps_rate, env, gamma, Q1, Q2
            )

            # Update Q values by randomly sampling experiences in Replay Memory
            if len(repmemory) > MINIBATCH_SIZE:
                experiences = repmemory.sample(MINIBATCH_SIZE)
                for experience in experiences:
                    ts, ta, tr, tns = experience
                    updateQValues(ts, ta, tr, tns, alpha, eps_rate, env, gamma, Q1, Q2)

            state = next_state
            action = next_action
            t += 1

            if done:
                break

        history.append(t)
```

```
        if i_episode > 50:
            latest_duration = history[-50:]
        else:
            latest_duration = history

        # print(latest_duration)
        duration_run_avg = np.mean(latest_duration)
        # print(duration_run_avg)
        duration_history.append([t, duration_run_avg])
        epsilon_history.append(eps_rate)
        cum_reward += t

        if t > max_t:
            max_t = t

        train_reward_history.append([cum_reward / (i_episode
            + 1), max_t])
        print(
            "\n%s Episode: %d Episode duration: %d timesteps
                | epsilon %f"
            % (label, i_episode + 1, t + 1, eps_rate)
        )

    last_100_avg = np.mean(history[-100:])
    print("%s average episode duration over last 100
        training episodes: %.2f" % (label, last_100_avg))
    env.close()
    return Q1, Q2, history, duration_history,
        epsilon_history, last_100_avg


if __name__ == "__main__":  ## (F)
    baseline_Q1, baseline_Q2, baseline_history,
        baseline_duration_history, baseline_epsilon_history,
        baseline_avg = run_training(
        BASELINE_GAMMA, "Baseline gamma=0.85"
    )
    ablation_Q1, ablation_Q2, ablation_history,
        ablation_duration_history, ablation_epsilon_history,
        ablation_avg = run_training(
        ABLATION_GAMMA, "Ablation gamma=0.99"
    )

    np.save("Q1.npy", ablation_Q1)
    np.save("Q2.npy", ablation_Q2)

    fig = plt.figure(1)
    plt.clf()
```

```
    baseline_episodes = np.arange(1, len(baseline_history) +
        1)
    ablation_episodes = np.arange(1, len(ablation_history) +
        1)
    baseline_moving_average = np.asarray(
        baseline_duration_history)[:, 1]
    ablation_moving_average = np.asarray(
        ablation_duration_history)[:, 1]

    plt.subplot(1, 2, 1)
    plt.title("Training History")
    plt.plot(baseline_episodes, baseline_moving_average,
        label="gamma=0.85 moving average")
    plt.plot(ablation_episodes, ablation_moving_average,
        label="gamma=0.99 moving average")
    plt.xlabel("Episodes")
    plt.ylabel("Episode Duration")
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.title("Epsilon for Episode")
    plt.plot(baseline_episodes, np.asarray(
        baseline_epsilon_history), label="gamma=0.85")
    plt.plot(ablation_episodes, np.asarray(
        ablation_epsilon_history), label="gamma=0.99")
    plt.xlabel("Episodes")
    plt.ylabel("Epsilon Value")
    plt.legend()

    plt.tight_layout()
    plt.savefig("Cartpole_DoubleQ_Learning.png")
    plt.savefig("Cartpole_DoubleQ_Gamma_Comparison.png")
    plt.close(fig)  ## (G)
```

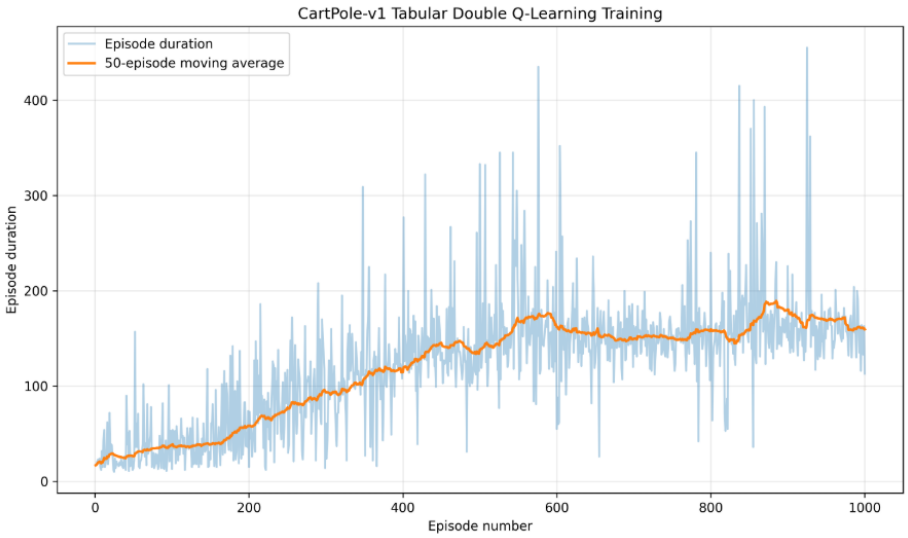

Figure 1: Loss curve

Average episode duration over last 100 training episodes: 163.13

3.1.2 One ablation (7 points)

I pick change discount factor to 0.99

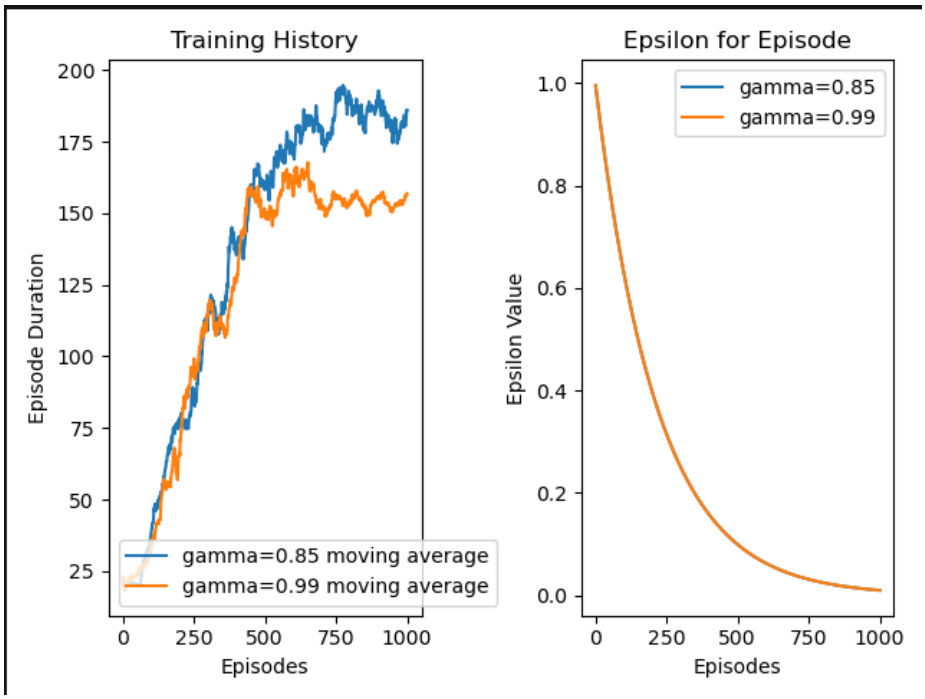

Figure 2: Comparison

For the ablation, I changed the discount factor from 0.85 to 0.99 while keeping the same random seed, number of episodes, and other hyperparameters. The training curves show that both agents improve over time, but the baseline with discount factor = 0.85 reaches a higher episode duration. With discount factor = 0.99, the moving average increases in the first half of training. This suggests that using a very large discount factor made learning less effective in this tabular CartPole setting. One possible reason is that discount factor = 0.99 puts more weight on long-term future rewards, which can make the Q-value estimates less stable when the state space is discretized. In contrast, discount factor = 0.85 focuses more on shorter term rewards, which may be easier to learn. So the ablation make the training worse.

3.2 Task 2: DQN on LunarLander (35 points)

3.2.1 Background

3.2.2 Implementation requirements (30 points)

```
import random
from collections import deque

import gymnasium as gym
```

```
import imageio
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim



EPISODES = 1000
MAX_STEPS = 1000

GAMMA = 0.99
LR = 1e-3

BATCH_SIZE = 64
REPLAY_SIZE = 50000
MIN_REPLAY_SIZE = 1000

EPS_START = 1.0
EPS_END = 0.05
EPS_DECAY_EPISODES = int(0.2 * EPISODES)  # first 20% of
    training

TARGET_UPDATE_FREQ = 1000  # hard update every 1000
    environment steps

CHECKPOINT_PATH = "dqn_lander.pt"
PLOT_PATH = "dqn_lander_training_curve.png"
GIF_PATH = "dqn_lander.gif"

SEED = 0


random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)



device = torch.device("cuda" if torch.cuda.is_available()
    else "cpu")


class QNet(nn.Module):
    def __init__(self, state_dim=8, action_dim=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):
        return self.net(s)



class ReplayBuffer:
    def __init__(self, capacity=50000):
        self.buf = deque(maxlen=capacity)

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

    def sample(self, batch_size):
        batch = random.sample(self.buf, batch_size)

        states, actions, rewards, next_states, dones = zip(*
            batch)

        states = torch.tensor(np.array(states), dtype=torch.
            float32, device=device)
        actions = torch.tensor(actions, dtype=torch.long,
            device=device).unsqueeze(1)
        rewards = torch.tensor(rewards, dtype=torch.float32,
             device=device)
        next_states = torch.tensor(np.array(next_states),
            dtype=torch.float32, device=device)
        dones = torch.tensor(dones, dtype=torch.float32,
            device=device)

        return states, actions, rewards, next_states, dones

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



def get_epsilon(episode):
    """
    Linearly anneal epsilon from 1.0 to 0.05 over the first
        20% of training.
    """
    if episode >= EPS_DECAY_EPISODES:
        return EPS_END

```

```
    frac = episode / EPS_DECAY_EPISODES
    return EPS_START + frac * (EPS_END - EPS_START)



def select_action(state, model):
    state_t = torch.tensor(state, dtype=torch.float32).
        unsqueeze(0)

    with torch.no_grad():
        q_values = model(state_t)
        action = torch.argmax(q_values, dim=1).item()

    return action


def select_action_epsilon(q_net, state, epsilon, action_dim)
    :
    if random.random() < epsilon:
        return random.randrange(action_dim)

    state_t = torch.tensor(state, dtype=torch.float32,
        device=device).unsqueeze(0)

    with torch.no_grad():
        q_values = q_net(state_t)
        action = torch.argmax(q_values, dim=1).item()

    return action



def train_step(q_net, target_net, optimizer, replay_buffer):
    if len(replay_buffer) < MIN_REPLAY_SIZE:
        return None

    states, actions, rewards, next_states, dones =
        replay_buffer.sample(BATCH_SIZE)

    # Q_phi(s, a)
    current_q = q_net(states).gather(1, actions).squeeze(1)

    # y = r + gamma * (1 - done) * max_a' Q_target(s', a')
    with torch.no_grad():
        max_next_q = target_net(next_states).max(dim=1)[0]
        target_q = rewards + GAMMA * (1.0 - dones) *
            max_next_q

    loss = nn.functional.mse_loss(current_q, target_q)

```

```
    optimizer.zero_grad()
    loss.backward()

    # optional gradient clipping for stability
    torch.nn.utils.clip_grad_norm_(q_net.parameters(),
        max_norm=10.0)

    optimizer.step()

    return loss.item()



def moving_average(data, window=100):
    if len(data) < window:
        return np.array([])

    return np.convolve(data, np.ones(window) / window, mode=
        "valid")



def plot_training_curve(episode_returns):
    plt.figure(figsize=(10, 6))

    plt.plot(episode_returns, label="Episode return", alpha
        =0.5)

    ma = moving_average(episode_returns, window=100)
    if len(ma) > 0:
        plt.plot(
            range(99, 99 + len(ma)),
            ma,
            label="100-episode moving average",
            linewidth=2,
        )

    plt.xlabel("Episode number")
    plt.ylabel("Episode return")
    plt.title("DQN on LunarLander")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(PLOT_PATH, dpi=300)
    plt.show()



def record_gif(q_net, env_name, gif_path=GIF_PATH):
    """
```

```
    Record one episode using the trained policy.
    """
    env = gym.make(env_name, render_mode="rgb_array")

    state, info = env.reset(seed=SEED)
    frames = []
    total_return = 0.0
    done = False
    steps = 0

    while not done and steps < MAX_STEPS:
        action = select_action(state, q_net)

        state, reward, terminated, truncated, info = env.
            step(action)
        frames.append(env.render())
        done = terminated or truncated

        total_return += reward
        steps += 1

    env.close()

    imageio.mimsave(gif_path, frames, fps=30)
    print(f"Saved GIF to {gif_path}")
    print(f"Recorded episode return: {total_return:.2f}")


def main():
    # Try LunarLander-v3 first. If unavailable, use
        LunarLander-v2.
    try:
        env = gym.make("LunarLander-v3")
        env_name = "LunarLander-v3"
    except Exception:
        env = gym.make("LunarLander-v2")
        env_name = "LunarLander-v2"

    print("Using environment:", env_name)

    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.n

    print("State dim:", state_dim)
    print("Action dim:", action_dim)

    q_net = QNet(state_dim, action_dim).to(device)
    target_net = QNet(state_dim, action_dim).to(device)

    # Initial target network copy
```

```
    target_net.load_state_dict(q_net.state_dict())
    target_net.eval()

    optimizer = optim.Adam(q_net.parameters(), lr=LR)
    replay_buffer = ReplayBuffer(REPLAY_SIZE)

    episode_returns = []
    losses = []

    global_step = 0
    solved = False

    for episode in range(EPISODES):
        state, info = env.reset(seed=SEED + episode)

        episode_return = 0.0
        epsilon = get_epsilon(episode)

        for step in range(MAX_STEPS):
            action = select_action_epsilon(q_net, state,
                epsilon, action_dim)

            next_state, reward, terminated, truncated, info
                = env.step(action)
            done = terminated or truncated

            replay_buffer.push(state, action, reward,
                next_state, done)

            loss = train_step(q_net, target_net, optimizer,
                replay_buffer)
            if loss is not None:
                losses.append(loss)

            state = next_state
            episode_return += reward
            global_step += 1

            # Hard update target network
            if global_step % TARGET_UPDATE_FREQ == 0:
                target_net.load_state_dict(q_net.state_dict
                    ())

            if done:
                break

        episode_returns.append(episode_return)

        avg_last_100 = np.mean(episode_returns[-100:])

```

```
        print(
            f"Episode {episode + 1:4d} | "
            f"Return: {episode_return:8.2f} | "
            f"Avg100: {avg_last_100:8.2f} | "
            f"Epsilon: {epsilon:.3f} | "
            f"Replay: {len(replay_buffer)}"
        )

        # Check solved condition
        if len(episode_returns) >= 100 and avg_last_100 >=
            200:
            print(f"Solved at episode {episode + 1}!")
            solved = True
            break

    env.close()

    last_100_avg = np.mean(episode_returns[-100:])

    print("\nTraining finished.")
    print(f"Average return over last 100 episodes: {
        last_100_avg:.2f}")
    print("Solved environment:", solved)

    # Save checkpoint
    torch.save(q_net.state_dict(), CHECKPOINT_PATH)

    print(f"Saved checkpoint to {CHECKPOINT_PATH}")

    # Save plot
    plot_training_curve(episode_returns)
    print(f"Saved training curve to {PLOT_PATH}")

    # Save GIF
    record_gif(q_net, env_name, GIF_PATH)


if __name__ == "__main__":
    main()
```

Average return over last 100 episodes: 200.49 It did solved environment.

3.2.3 Brief analysis (5 points)

- Why a target network? A target network make tarining more stable. If we use the same Q network to predict both current and target, the target will change every time the network update. It make mode unstable. Using a separate network, the target will be fixed, so it has more stable value to learn.

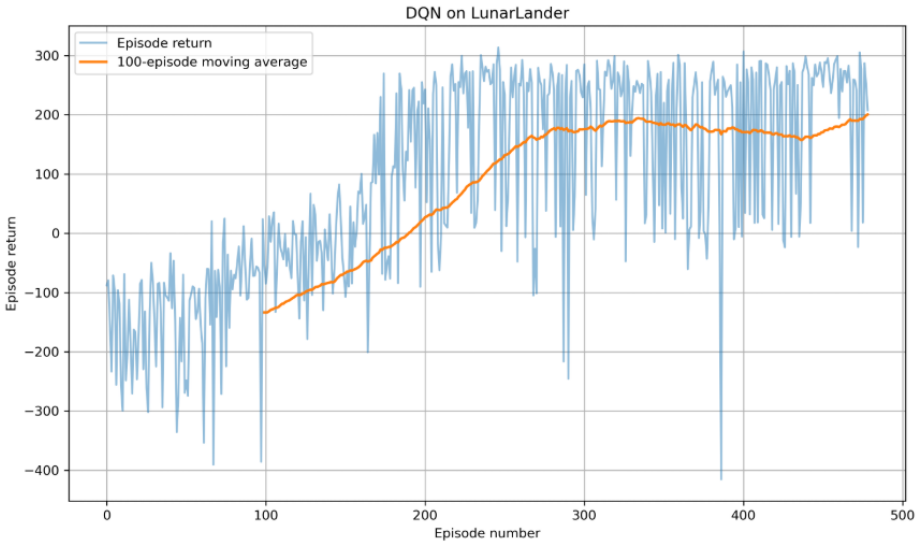

Figure 3: episode return vs. episode number

- epsilon-greedy decay : Because the agent should explore a lot at the beginning and use what it learned later.If epsilon starts at 0.05, the agent explores too little and may learn from poor early actions. If epsilon stays at 1.0, the agent keeps acting randomly and never really uses the learned policy. So changing from 1 to 0.05 can give balance.

3.3 Task 3: Reward modeling on real human preferences (50 points)

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 (8 points)

```
import sys
from pathlib import Path


STARTER_DIR = Path(__file__).resolve().parent / "
    HW11_starter_code"
sys.path.insert(0, str(STARTER_DIR))

```

```python
from lm_utils import load_gpt2, load_preference_dataset,
    embed_triples


def main():
    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'][:200]!r}...")
        print(f"CHOSEN: {t['chosen']!r}")
        print(f"REJECTED: {t['rejected']!r}")
```

```
=== 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 flyi'...
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 flyi'...
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 flyi'...
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 example come from the reddit post. The chosen summaries are only slightly better to me: it capture the main issue more directly, especially that the mother is upset because of the flight. However, the difference is not always very obvious, because some rejected summaries also contain similar information and the summaries are all quite short.

3.3.6 Embed everything (12 points)

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

    split_idx = int(0.8 * len(embedded))
    train_embedded = embedded[:split_idx]
    val_embedded = embedded[split_idx:]
    print(f"train triples: {len(train_embedded)}")
    print(f"validation triples: {len(val_embedded)}")
```

embedded 500 triples post embedding shape: torch.Size([768]) train triples: 400 validation triples: 100

3.3.7 Train the RewardNetwork (20 points)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random
import sys
from pathlib import Path

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


STARTER_DIR = Path(__file__).resolve().parent / "
    HW11_starter_code"
sys.path.insert(0, str(STARTER_DIR))

from lm_utils import HIDDEN_SIZE, embed_triples, load_gpt2,
    load_preference_dataset


SEED = 2
N_EXAMPLES = 500
TRAIN_STEPS = 2000
LR = 1e-3
LOSS_PLOT_PATH = Path(__file__).resolve().parent / "
    reward_model_loss_curve.png"


```

```
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)


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(example, response_key, device):
    return torch.cat(
        [
            example["post_emb"],
            example["sep_emb"],
            example[response_key],
        ]
    ).to(device)


def pairwise_loss(reward_model, example, device):
    chosen_input = make_input(example, "chosen_emb", device)
    rejected_input = make_input(example, "rejected_emb",
        device)
    r_chosen = reward_model(chosen_input)
    r_rejected = reward_model(rejected_input)
    return -torch.nn.functional.logsigmoid(r_chosen -
        r_rejected).mean()


def moving_average(values, window=200):
    if len(values) < window:
        return np.array([])
    return np.convolve(values, np.ones(window) / window,
        mode="valid")
```

```
72
73
74  def plot_losses(losses):
75      steps = np.arange(1, len(losses) + 1)
76      avg = moving_average(losses, window=200)
77
78      plt.figure(figsize=(10, 6))
79      plt.plot(steps, losses, label="Per-step loss", alpha
            =0.35)
80      if len(avg) > 0:
81          plt.plot(np.arange(200, 200 + len(avg)), avg, label=
                "Moving average (200)", linewidth=2)
82      plt.xlabel("Gradient step")
83      plt.ylabel("Loss")
84      plt.title("Reward Model Training Loss")
85      plt.legend()
86      plt.grid(True)
87      plt.tight_layout()
88      plt.savefig(LOSS_PLOT_PATH, dpi=300)
89      plt.close()
90
91
92  def validation_accuracy(reward_model, val_data, device):
93      correct = 0
94      reward_model.eval()
95      with torch.no_grad():
96          for example in val_data:
97              chosen_input = make_input(example, "chosen_emb",
                    device)
98              rejected_input = make_input(example, "
                    rejected_emb", device)
99              r_chosen = reward_model(chosen_input)
100             r_rejected = reward_model(rejected_input)
101             if r_chosen.item() > r_rejected.item():
102                 correct += 1
103     return correct / len(val_data)
104
105
106 def main():
107     triples = load_preference_dataset(n_examples=N_EXAMPLES)
108     print(f"loaded {len(triples)} triples")
109
110     tokenizer, base, embed_device = load_gpt2()
111     embedded = embed_triples(triples, tokenizer, base,
            embed_device)
112     print(f"embedded {len(embedded)} triples")
113     print(f"post embedding shape: {embedded[0]['post_emb'].
            shape}")
114
115     split_idx = int(0.8 * len(embedded))
```

```
    train_data = embedded[:split_idx]
    val_data = embedded[split_idx:]
    print(f"train triples: {len(train_data)}")
    print(f"validation triples: {len(val_data)}")

    del base
    if torch.cuda.is_available():
        torch.cuda.empty_cache()

    device = torch.device("cuda" if torch.cuda.is_available
        () else "cpu")
    reward_model = RewardNetwork(HIDDEN_SIZE).to(device)
    optimizer = torch.optim.Adam(reward_model.parameters(),
        lr=LR)

    losses = []
    order = list(range(len(train_data)))
    random.shuffle(order)

    reward_model.train()
    for step in range(TRAIN_STEPS):
        if step > 0 and step % len(order) == 0:
            random.shuffle(order)

        example = train_data[order[step % len(order)]]
        loss = pairwise_loss(reward_model, example, device)

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

        losses.append(loss.item())
        if (step + 1) % 200 == 0:
            print(f"step {step + 1:4d} | loss {loss.item()
                :.4f}")

    initial_loss = losses[0]
    final_loss = float(np.mean(losses[-200:]))
    val_acc = validation_accuracy(reward_model, val_data,
        device)
    plot_losses(losses)

    print(f"\ninitial loss: {initial_loss:.4f}")
    print(f"final loss (last-200-step average): {final_loss
        :.4f}")
    print(f"validation accuracy: {val_acc:.4f}")
    print(f"saved loss curve to: {LOSS_PLOT_PATH}")
```

```
if __name__ == "__main__":
    main()
```

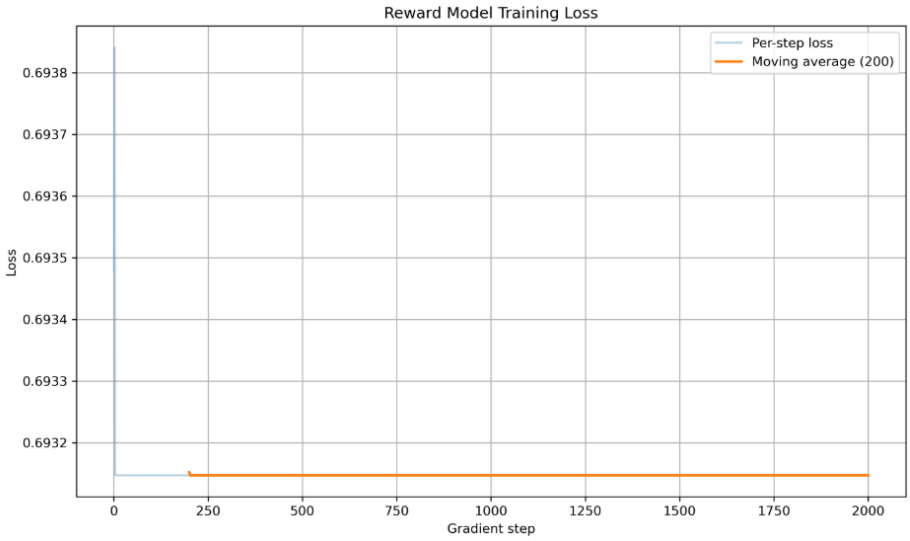

Figure 4: Loss curve

initial loss: 0.6935 final loss (last-200-step average): 0.6931

3.3.8 Evaluate and critique (10 points)

- The validation accuracy is 0.4600, The loss start with 0.6935 and the final last 200 step average is 0.6931. This is much worse than I expected, since the random guess will give 50% accuracy. So the model is leanring nothing, gave the same reward.
- In the rewardnetwork, the first two layer will clipped activations to 0. And the final layer is passed to sigmoid. So the reward should be between 0 adn 1. In this case, $-1 < r_{chose} - r_{rejected} < 1$. So the sigmoid prevents the model from producing large reward, which make model harder to decrease the loss.
- one possible modification is to remove the final sigmoid. And try iwth linear layer. So the $r_{chose} - r_{rejected}$ will become larger than 1, and the loss will decrease further.

3.4 Bonus Task: PPO on LunarLander (20 points)

I am using the code from CleanRL

```python
import random
import re
import time
from pathlib import Path

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


SEED = 0
ENV_ID = "LunarLander-v3"
TOTAL_TIMESTEPS = 500000
NUM_ENVS = 4
NUM_STEPS = 128
LEARNING_RATE = 2.5e-4
GAMMA = 0.99
GAE_LAMBDA = 0.95
UPDATE_EPOCHS = 4
NUM_MINIBATCHES = 4
CLIP_COEF = 0.2
ENT_COEF = 0.01
VF_COEF = 0.5
MAX_GRAD_NORM = 0.5

PPO_PLOT_PATH = "ppo_lander_training_curve.png"
COMPARISON_PLOT_PATH = "ppo_vs_dqn_comparison.png"
REPORT_PATH = "ppo_lander_report.txt"
DQN_LOG_PATH = "task2_v2.log"


def make_env(env_id, seed, idx):
    def thunk():
        env = gym.make(env_id)
        env = gym.wrappers.RecordEpisodeStatistics(env)
        env.action_space.seed(seed + idx)
        env.observation_space.seed(seed + idx)
        return env

    return thunk


def layer_init(layer, std=np.sqrt(2), bias_const=0.0):
    torch.nn.init.orthogonal_(layer.weight, std)
    torch.nn.init.constant_(layer.bias, bias_const)
```

```
    return layer


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

    def forward(self, x):
        return torch.softmax(self.net(x), dim=-1)


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

    def forward(self, x):
        return self.net(x)


class Agent(nn.Module):
    def __init__(self, state_dim=8, action_dim=4):
        super().__init__()
        self.actor = Actor(state_dim, action_dim)
        self.critic = Critic(state_dim)

    def get_value(self, x):
        return self.critic(x)

    def get_action_and_value(self, x, action=None):
        probs = self.actor(x)
        dist = Categorical(probs=probs)
        if action is None:
            action = dist.sample()
        return action, dist.log_prob(action), dist.entropy(), self.critic(x)
```

```
98
99
100 def moving_average(values, window=100):
101     values = np.asarray(values, dtype=np.float32)
102     if len(values) < window:
103         return np.array([])
104     return np.convolve(values, np.ones(window) / window,
        mode="valid")
105
106
107 def parse_dqn_returns(log_path):
108     path = Path(log_path)
109     if not path.exists():
110         return []
111     text = path.read_text()
112     return [
113         float(match.group(1))
114         for match in re.finditer(r"Return:\s*([-+]?\d+(?:\.\
            d+)?)", text)
115     ]
116
117
118 def save_ppo_plot(ppo_returns):
119     plt.figure(figsize=(10, 6))
120     plt.plot(ppo_returns, label="PPO episode return", alpha
        =0.45)
121     ma = moving_average(ppo_returns, window=100)
122     if len(ma) > 0:
123         plt.plot(range(99, 99 + len(ma)), ma, label="PPO
            100-episode moving average", linewidth=2)
124     plt.xlabel("Episode")
125     plt.ylabel("Return")
126     plt.title("PPO on LunarLander")
127     plt.grid(True, alpha=0.3)
128     plt.legend()
129     plt.tight_layout()
130     plt.savefig(PPO_PLOT_PATH, dpi=300)
131     plt.close()
132
133
134 def save_comparison_plot(ppo_returns, dqn_returns):
135     plt.figure(figsize=(10, 6))
136
137     ppo_ma = moving_average(ppo_returns, window=100)
138     if len(ppo_ma) > 0:
139         plt.plot(range(99, 99 + len(ppo_ma)), ppo_ma, label=
            "PPO Avg100", linewidth=2)
140
141     dqn_ma = moving_average(dqn_returns, window=100)
142     if len(dqn_ma) > 0:
```

```
        plt.plot(range(99, 99 + len(dqn_ma)), dqn_ma, label="DQN Avg100", linewidth=2)

    plt.xlabel("Episode")
    plt.ylabel("100-episode moving average return")
    plt.title("LunarLander Training Curve Comparison")
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(COMPARISON_PLOT_PATH, dpi=300)
    plt.close()




def main():
    total_timesteps = TOTAL_TIMESTEPS
    seed = SEED

    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.backends.cudnn.deterministic = True

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    try:
        envs = gym.vector.SyncVectorEnv(
            [make_env(ENV_ID, seed, i) for i in range(NUM_ENVS)]
        )
        env_id = ENV_ID
    except Exception:
        env_id = "LunarLander-v2"
        envs = gym.vector.SyncVectorEnv(
            [make_env(env_id, seed, i) for i in range(NUM_ENVS)]
        )

    state_dim = int(np.array(envs.single_observation_space.shape).prod())
    action_dim = envs.single_action_space.n
    agent = Agent(state_dim, action_dim).to(device)
    optimizer = optim.Adam(agent.parameters(), lr=LEARNING_RATE, eps=1e-5)

    batch_size = NUM_ENVS * NUM_STEPS
    minibatch_size = batch_size // NUM_MINIBATCHES
    num_iterations = total_timesteps // batch_size

```

```
    obs = torch.zeros((NUM_STEPS, NUM_ENVS) + envs.
        single_observation_space.shape, device=device)
    actions = torch.zeros((NUM_STEPS, NUM_ENVS), device=
        device)
    logprobs = torch.zeros((NUM_STEPS, NUM_ENVS), device=
        device)
    rewards = torch.zeros((NUM_STEPS, NUM_ENVS), device=
        device)
    dones = torch.zeros((NUM_STEPS, NUM_ENVS), device=device
        )
    values = torch.zeros((NUM_STEPS, NUM_ENVS), device=
        device)

    ppo_returns = []
    global_step = 0
    start_time = time.time()
    next_obs, info = envs.reset(seed=seed)
    next_obs = torch.tensor(next_obs, dtype=torch.float32,
        device=device)
    next_done = torch.zeros(NUM_ENVS, device=device)

    print(f"Using environment: {env_id}")
    print(f"Using device: {device}")

    for iteration in range(1, num_iterations + 1):
        frac = 1.0 - (iteration - 1.0) / num_iterations
        optimizer.param_groups[0]["lr"] = frac *
            LEARNING_RATE

        for step in range(NUM_STEPS):
            global_step += NUM_ENVS
            obs[step] = next_obs
            dones[step] = next_done

            with torch.no_grad():
                action, logprob, entropy, value = agent.
                    get_action_and_value(next_obs)
                values[step] = value.flatten()

            actions[step] = action
            logprobs[step] = logprob

            next_obs_np, reward, terminated, truncated,
                infos = envs.step(action.cpu().numpy())
            done = np.logical_or(terminated, truncated)
            rewards[step] = torch.tensor(reward, dtype=torch
                .float32, device=device)
            next_obs = torch.tensor(next_obs_np, dtype=torch
                .float32, device=device)
            next_done = torch.tensor(done, dtype=torch.
```

```
                float32, device=device)

            if "final_info" in infos:
                for info in infos["final_info"]:
                    if info and "episode" in info:
                        episode_return = float(info["episode"]["r"])
                        ppo_returns.append(episode_return)
                        avg100 = np.mean(ppo_returns[-100:])
                        print(
                            f"Episode {len(ppo_returns):4d} | Return: {episode_return:8.2f} | "
                            f"Avg100: {avg100:8.2f} | Global step: {global_step}"
                        )

        with torch.no_grad():
            next_value = agent.get_value(next_obs).reshape(1, -1)
            advantages = torch.zeros_like(rewards, device=device)
            lastgaelam = 0
            for t in reversed(range(NUM_STEPS)):
                if t == NUM_STEPS - 1:
                    nextnonterminal = 1.0 - next_done
                    nextvalues = next_value
                else:
                    nextnonterminal = 1.0 - dones[t + 1]
                    nextvalues = values[t + 1]
                delta = rewards[t] + GAMMA * nextvalues * nextnonterminal - values[t]
                advantages[t] = lastgaelam = delta + GAMMA * GAE_LAMBDA * nextnonterminal * lastgaelam
            returns = advantages + values

        b_obs = obs.reshape((-1,) + envs.single_observation_space.shape)
        b_logprobs = logprobs.reshape(-1)
        b_actions = actions.reshape(-1).long()
        b_advantages = advantages.reshape(-1)
        b_returns = returns.reshape(-1)
        b_values = values.reshape(-1)

        b_inds = np.arange(batch_size)
        for epoch in range(UPDATE_EPOCHS):
            np.random.shuffle(b_inds)
            for start in range(0, batch_size, minibatch_size):
```

```
                mb_inds = b_inds[start : start +
                    minibatch_size]
                _, newlogprob, entropy, newvalue = agent.
                    get_action_and_value(
                     b_obs[mb_inds], b_actions[mb_inds]
                )
                logratio = newlogprob - b_logprobs[mb_inds]
                ratio = logratio.exp()

                mb_advantages = b_advantages[mb_inds]
                mb_advantages = (mb_advantages -
                    mb_advantages.mean()) / (mb_advantages.
                    std() + 1e-8)

                pg_loss1 = -mb_advantages * ratio
                pg_loss2 = -mb_advantages * torch.clamp(
                    ratio, 1 - CLIP_COEF, 1 + CLIP_COEF)
                pg_loss = torch.max(pg_loss1, pg_loss2).mean
                    ()

                newvalue = newvalue.view(-1)
                v_loss = 0.5 * ((newvalue - b_returns[
                    mb_inds]) ** 2).mean()
                entropy_loss = entropy.mean()
                loss = pg_loss - ENT_COEF * entropy_loss +
                    VF_COEF * v_loss

                optimizer.zero_grad()
                loss.backward()
                nn.utils.clip_grad_norm_(agent.parameters(),
                     MAX_GRAD_NORM)
                optimizer.step()

        if iteration % 10 == 0:
            sps = int(global_step / (time.time() -
                start_time))
            print(f"Iteration {iteration:4d}/{num_iterations
                } | SPS: {sps}")

    envs.close()

    save_ppo_plot(ppo_returns)
    dqn_returns = parse_dqn_returns(DQN_LOG_PATH)
    save_comparison_plot(ppo_returns, dqn_returns)
    print(f"Saved PPO curve to {PPO_PLOT_PATH}")
    print(f"Saved comparison plot to {COMPARISON_PLOT_PATH}"
        )


if __name__ == "__main__":
```

```
301     main()
```

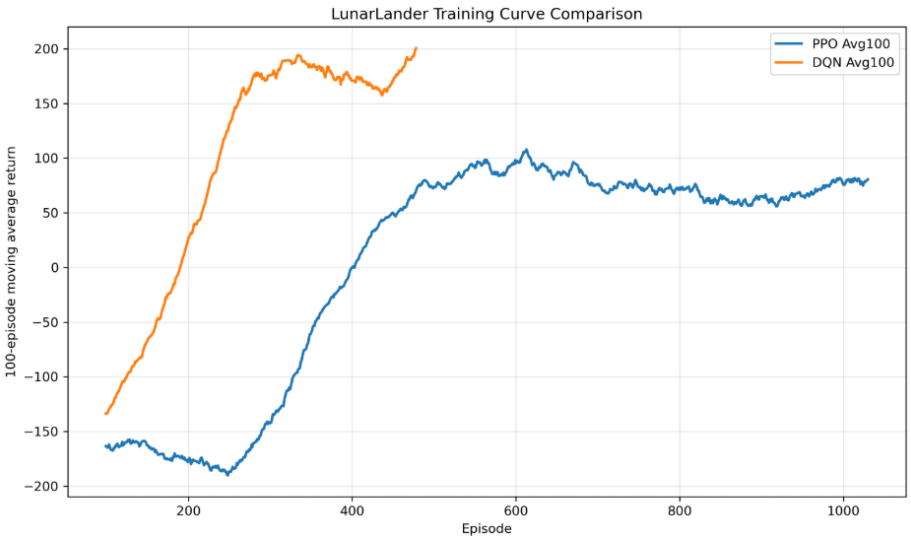

Figure 5: comparison

Average return over the last 100 episodes for PPO: 80.56.
DQN solved the environment much faster than PPO, it using around 500 eposide to reach 200 return and ppo use 1000 eposide.DQN appears easier to tune for this task, since it achieved strong performance with fewer episodes.I think for a new problem, I would like to test on DQN first, since it is fast.