In this assignment we are going to work with Reinforcement learning using the environment in Mountain Car and Roulette problems. Now let's look up the way we are going to work on RL first.

RL(Q-Learning) approaches to solve a problem by taking some random action and overviewing the reward allocated on reaching the goal, it learns by reinforcing the action yielding positive rewards. It maintains a Q-table that keeps track of the action taken and whether the goal is reached or not and thus allotting the reward/penalty accordingly.

Let's also overview how the reward system works and how the Q-table is updated. So, whenever the action taken by the agent accumulates to reach the goal then positive points (reward) is awarded and when the goal is not reached then negative points (penalty) is awarded. Another thing to note here is our choice of action to maintain the balance between exploration and exploitation of action. If Q-learning parameter (ϵ) allows exploration then we greedily choose a random action, else exploit the highest Q-table to decide the most favourable action to reach the goal. Now over to the Q-table updation, it is guided by the equation:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{TD error}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

Also, given below is the python implementation:

```
def update_q(Q, state, action, reward, next_state, alpha, gamma):
  Q[state][action] += alpha * (reward + gamma * np.max(Q[next_state]) - Q[state][action])
```

Now let's go through the MountainCar problem, the RL is going to use Q-Learning which defines a space of position, velocity and action taken. This helps the agent in deciding the future action and thus helps calculate the expected reward/penalty. The Q-state dimension is defined as: [position]*[velocity]*[action].

The first step of learning our agent is to discretize the position and velocity (since q-learning works on discrete space but actually position and velocity is continuous). The position and velocity space is divided into smaller discrete steps.

The agent learning the system, starts by selecting random action, evaluates the result and logs the reward/penaly to the Q-table. This step is repeatedly performed unless the agent can deterministically determine the next step of action to reach the goal.

Let’s now go through the code:

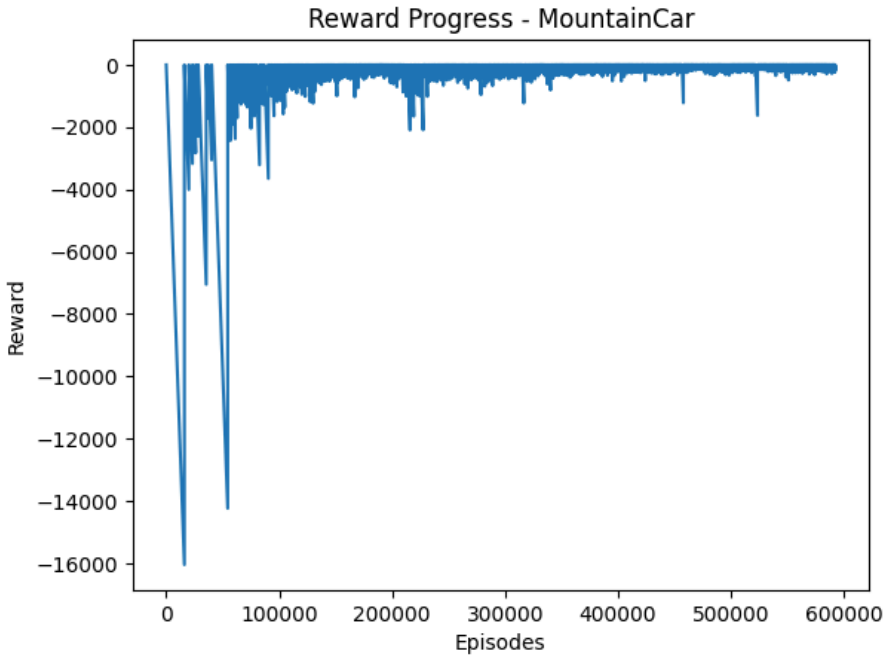
```
def create_bins(state_space, bins_per_feature=20):
  bins = [np.linspace(low, high, bins_per_feature) for low, high in
zip(state_space.low, state_space.high)]
  return bins


def discretize_state(state, bins):
  state_index = tuple(np.digitize(s, b) - 1 for s, b in zip(state, bins))
  return state_index


def choose_action(Q, state, epsilon, n_actions):
  if np.random.random() < epsilon:
    return np.random.randint(n_actions)
  else:
    return np.argmax(Q[state])


def update_q(Q, state, action, reward, next_state, alpha, gamma):
  Q[state][action] += alpha * (reward + gamma * np.max(Q[next_state]) -
Q[state][action])
def q_learning(env_name, episodes=5000, alpha=0.1, gamma=0.99,
epsilon=1.0, epsilon_decay=0.999, epsilon_min=0.05, bins_per_feature=20,
render_interval=500):
  env = gym.make(env_name)
  bins = create_bins(env.observation_space, bins_per_feature)
  n_actions = env.action_space.n


  Q = np.zeros(tuple([bins_per_feature]*len(env.observation_space.low) +
[n_actions]))
  rewards = []


  for episode in range(episodes):
    state = discretize_state(env.reset()[0], bins)
    done = False
    total_reward = 0
```

```
    while not done:
      action = choose_action(Q, state, epsilon, n_actions)
      next_state_cont, reward, done, truncated, _ = env.step(action)
      next_state = discretize_state(next_state_cont, bins)


      update_q(Q, state, action, reward, next_state, alpha, gamma)
      state = next_state
      total_reward += reward


      rewards.append(total_reward)
      epsilon = max(epsilon_min, epsilon * epsilon_decay)


      if episode % render_interval == 0:
        clear_output(wait=True)
        print(f"Episode: {episode}, Reward: {total_reward}, Epsilon:
{epsilon:.3f}")


  env.close()
  return Q, rewards
def test_mountain_car(Q, env_name='MountainCar-v0', tests=5,
bins_per_feature=20, render_mode=None):
    env = gym.make(env_name, render_mode=render_mode)
    bins = create_bins(env.observation_space, bins_per_feature)
    n_actions = env.action_space.n
    goal_reached_count = 0
    all_test_frames = []


    print(f"\nRunning {tests} tests for the agent...")


    for test in range(tests):
        state = discretize_state(env.reset()[0], bins)
        done = False
        total_reward = 0
```

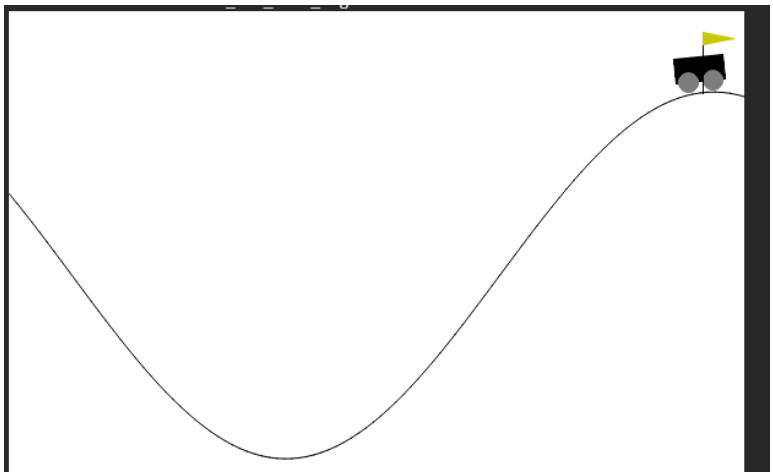
```
        goal_reached_in_test = False
        frames = []

        while not done:
            if render_mode is not None:
                frame = env.render()
                if frame is not None:
                    frames.append(frame)


            action = np.argmax(Q[state])
            next_state_cont, reward, done, truncated, _ = env.step(action)
            next_state = discretize_state(next_state_cont, bins)


            state = next_state
            total_reward += reward


            if next_state_cont[0] >= 0.5:
                goal_reached_in_test = True
                break


        if goal_reached_in_test:
            goal_reached_count += 1
            print(f"Test {test + 1}: Goal reached!")
            if render_mode is not None:
                all_test_frames.append(frames)
        else:
            print(f"Test {test + 1}: Goal not reached.")


    env.close()
    print(f"\nGoal reached in {goal_reached_count}/{tests} tests.")


    return all_test_frames, goal_reached_count
```

```
test_frames, goal_count = test_mountain_car(Q_mountain,
render_mode='rgb_array')
```

```
Running 5 tests for the agent...
Test 1: Goal reached!
Test 2: Goal reached!
Test 3: Goal reached!
Test 4: Goal reached!
Test 5: Goal reached!

Goal reached in 5/5 tests.
```



The final snapshot of the car reaching the goal is below:



Now, let's see the Roulette problem. The environment was not available in the gymnasium package hence needed to create a custom Roulette environment. The setup can be visualized as having a start balance, a target balance, bet value and discrete value to bet upon. At each round, the balances are reset and the values to bet upon are discretely divided. Then the roulette game starts where the agent puts its bet on the value and reaps its reward. This action is iteratively performed unless the agent goes broke or reaches the target mark or spin limit is reached. The action space is defined over the range of values available to bet on and the bet value available. Each action is thus a unique combination of (chosen bet number, chosen bet value). Also the reward shaping is one of the interesting parts. Whenever the iteration terminates, based on the net balance available, the reward is allotted. If the agent reaches the target balance, a positive reward is allotted. If the agent goes broke, a negative reward (penalty) is allotted. This shapes the agent's decision making in the long run and thus helps reinforcing the positive action.

Let's go through the code:

```
class RouletteEnv:
    def __init__(self, start_balance=50, target_balance=100,
max_steps=300):
        self.start_balance = start_balance
        self.target_balance = target_balance
        self.max_steps = max_steps
        self.bet_sizes = [1, 5, 10]
        self.numbers = 37
        self.action_space = self.numbers * len(self.bet_sizes)
        self.reset()

    def reset(self):
        self.balance = self.start_balance
        self.steps = 0
        return self._discretize_balance(self.balance)

    def _discretize_balance(self, balance):
        return min(balance // 2, 150)

    def step(self, action):
        number_choice = action % self.numbers
        bet_size = self.bet_sizes[action // self.numbers]

        reward = 0
        for _ in range(5):  # average spins
            spin = np.random.randint(0, 37)
```

```
            if spin == number_choice:
                reward += 35 * bet_size
            else:
                reward -= bet_size

        self.balance += reward
        self.steps += 1

        done = self.balance <= 0 or self.balance >= self.target_balance or
self.steps >= self.max_steps

        shaped_reward = reward
        if self.balance >= self.target_balance:
            shaped_reward += 100
        elif self.balance <= 0:
            shaped_reward -= 50

        next_state = self._discretize_balance(self.balance)
        return next_state, shaped_reward, done
def q_learning(
    episodes=8000,
    alpha=0.15,
    gamma=0.95,
    epsilon=1.0,
    epsilon_min=0.05,
    epsilon_decay=0.9994
):
    env = RouletteEnv()
    state_space = 151
    action_space = env.action_space
    Q = np.zeros((state_space, action_space))
    rewards_per_episode = []

    for ep in range(episodes):
        state = env.reset()
        total_reward = 0

        for _ in range(env.max_steps):
            if random.random() < epsilon:
                action = random.randint(0, action_space - 1)
```

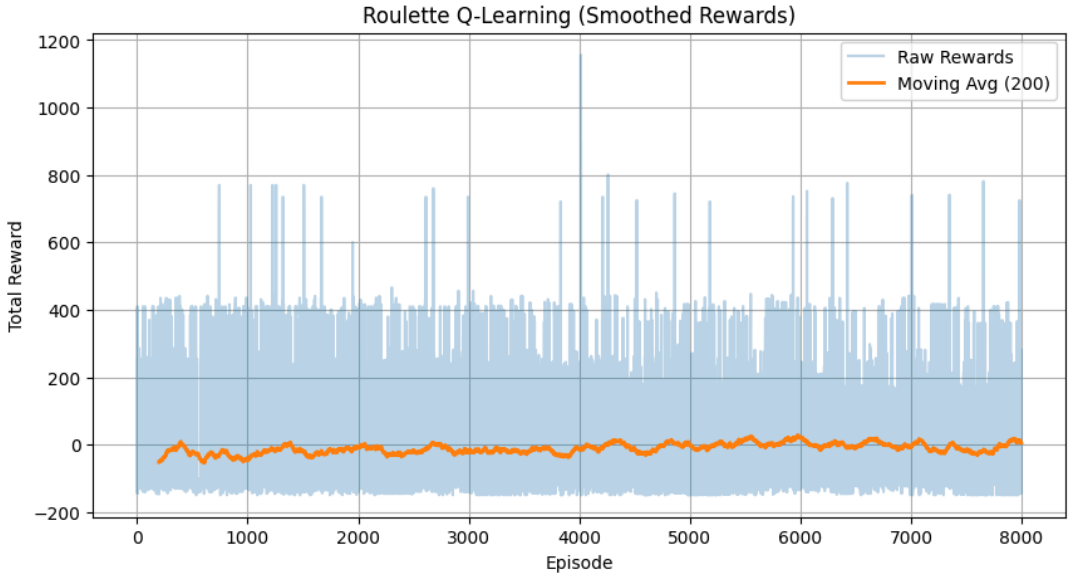
```
            else:
                action = np.argmax(Q[state])

            next_state, reward, done = env.step(action)
            total_reward += reward

            Q[state, action] += alpha * (reward + gamma *
np.max(Q[next_state]) - Q[state, action])
            state = next_state

            if done:
                break

        epsilon = max(epsilon_min, epsilon * epsilon_decay)
        rewards_per_episode.append(total_reward)

        if (ep + 1) % 1000 == 0:
            avg_reward = np.mean(rewards_per_episode[-1000:])
            print(f"Episode {ep+1}/{episodes} | Avg Reward:
{avg_reward:.2f}")

    return Q, rewards_per_episode, env
def test_agent(Q, env, runs=10):
    success_count = 0
    for r in range(runs):
        state = env.reset()
        total_reward = 0
        for _ in range(env.max_steps):
            action = np.argmax(Q[state])
            next_state, reward, done = env.step(action)
            total_reward += reward
            state = next_state
            if done:
                if env.balance >= env.target_balance:
                    success_count += 1
                break
        print(f"Test {r+1}: Final Balance = {env.balance}, Total Reward =
{total_reward}")

    print(f"\nGoal Reached in {success_count}/{runs} tests "
```

```
        f"({(success_count / runs) * 100:.1f}% success rate)")
```

After testing the agent over 10 test cases below is the result:

```
test_agent(Q, env)

Test 1: Final Balance = 103, Total Reward = 153
Test 2: Final Balance = 112, Total Reward = 162
Test 3: Final Balance = -45, Total Reward = -145
Test 4: Final Balance = 114, Total Reward = 164
Test 5: Final Balance = -24, Total Reward = -124
Test 6: Final Balance = 110, Total Reward = 160
Test 7: Final Balance = 102, Total Reward = 152
Test 8: Final Balance = 120, Total Reward = 170
Test 9: Final Balance = -48, Total Reward = -148
Test 10: Final Balance = 130, Total Reward = 180

Goal Reached in 7/10 tests (70.0% success rate)
```

The reward plot available here:



DQN over Mountain Car and Roulette problem.

Let's go over the steps taken to create the DQN solution to the above problems. The DQN solution sought to replace the Q-table with a neural network which performs the mapping between the (state, action) → reward. Q-Learning needed the state values to be in discrete forms and that aroused some problems. Too coarse values, the agent learns poorly, too fine values, the Q-table explodes in size.

The target y in DQN is calculated as:

$$y_i = \begin{cases} r_i & \text{if episode terminates at i+1} \\ r_i + \gamma \max_{a'} \hat{Q}\left(s_{i+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$$

where r_i is the reward received after taking action a_i in state s_i

s_{i+1} is the next state

γ is the discount factor

$Q(s_{i+1}, a'; \theta^-)$ is the predicted Q-value for the next state and action a', based on the target network parameters θ^-

The loss function for the DQN is:

$$\text{loss} = \left(y_i - Q\left(s_i, a_i; \theta\right)\right)^2$$

When the loss is calculated the gradient is updated in the network using the backpropagation algorithm. There is another provision called replay buffer which stores the action and reward. It also helps in training the model and breaks the sequential dependency on data such that the agent does not pick any action depending on sequential dependency and also encourages the agent to learn from the past experiences.

The approach to solve any problem remains largely similar to the Q-Learning step. The DQN teaches the agent through the following steps:

1. Reset the environment.
2. Greedily choose an action.
3. Perform the action and log the action and reward to the replay buffer.
4. Sample random batch from the replay buffer.
5. Compute the Expected Q-value
6. Compute the Current Q-value
7. Calculate the loss
8. Update the gradient
9. Update the ϵ parameter affecting the choice of action.

The Mountain Car problem was also attempted to be solved using the DQN network.

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

class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=128):
        super(QNetwork, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, action_size)
        )

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


class ReplayBuffer:
    def __init__(self, capacity=100000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
```

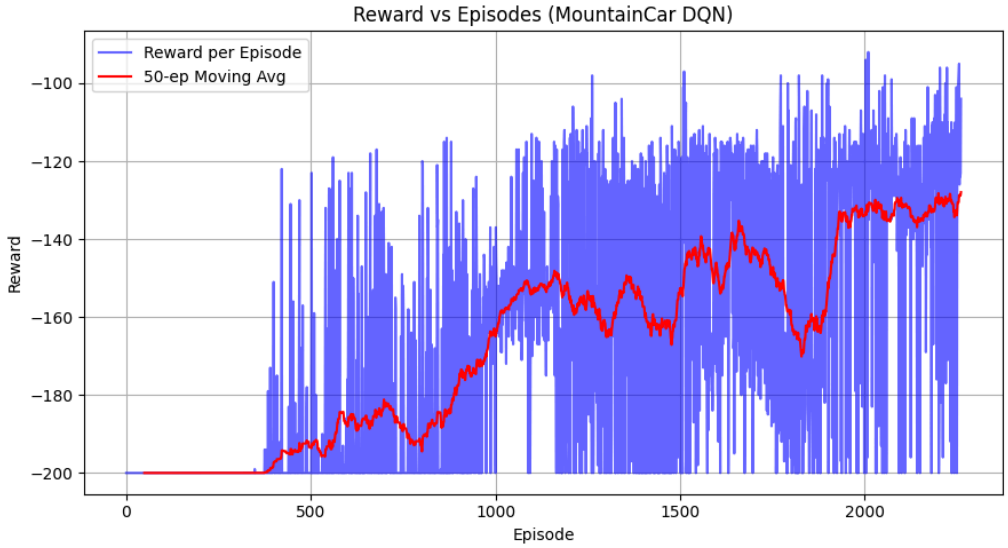
```
        return (
            torch.FloatTensor(states).to(device),
            torch.LongTensor(actions).to(device),
            torch.FloatTensor(rewards).to(device),
            torch.FloatTensor(next_states).to(device),
            torch.FloatTensor(dones).to(device),
        )

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

def plot_rewards(reward_history, window=50):
    """Plot Reward vs Episode with optional moving average."""
    plt.figure(figsize=(9, 5))
    plt.plot(reward_history, color='blue', alpha=0.6, label="Reward per
Episode")

    if len(reward_history) >= window:
        moving_avg = np.convolve(reward_history, np.ones(window)/window,
mode='valid')
        plt.plot(range(window - 1, len(reward_history)), moving_avg,
color='red', label=f"{window}-ep Moving Avg")

    plt.title("Reward vs Episodes (MountainCar DQN)")
    plt.xlabel("Episode")
    plt.ylabel("Reward")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
def train_dqn(
    env_name="MountainCar-v0",
    episodes=1000,
    batch_size=64,
    gamma=0.99,
    lr=1e-3,
    epsilon_start=1.0,
    epsilon_end=0.05,
    epsilon_decay=0.995,
    target_update_freq=10,
```

```
    device= device
):
    env = gym.make(env_name)
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    policy_net = QNetwork(state_size, action_size).to(device)
    target_net = QNetwork(state_size, action_size).to(device)
    target_net.load_state_dict(policy_net.state_dict())
    optimizer = optim.Adam(policy_net.parameters(), lr=lr)
    buffer = ReplayBuffer()

    epsilon = epsilon_start
    all_rewards = []
    goal_reached = False

    for episode in range(1, episodes + 1):
        state, _ = env.reset()
        total_reward = 0
        done = False

        while not done:

            if random.random() < epsilon:
                action = env.action_space.sample()
            else:
                with torch.no_grad():
                    state_tensor =
torch.FloatTensor(state).unsqueeze(0).to(device)
                    q_values = policy_net(state_tensor)
                    action = torch.argmax(q_values).item()

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

            position, velocity = next_state
            shaped_reward = reward + (position + 0.5)

            buffer.push(state, action, shaped_reward, next_state, done)
```

```
            state = next_state
            total_reward += reward

            if len(buffer) >= batch_size:
                states, actions, rewards, next_states, dones =
buffer.sample(batch_size)
                q_values = policy_net(states).gather(1,
actions.unsqueeze(1)).squeeze(1)
                with torch.no_grad():
                    next_q_values = target_net(next_states).max(1)[0]
                    target = rewards + gamma * next_q_values * (1 - dones)
                loss = nn.MSELoss()(q_values, target)

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

        epsilon = max(epsilon_end, epsilon * epsilon_decay)
        all_rewards.append(total_reward)

        if episode % target_update_freq == 0:
            target_net.load_state_dict(policy_net.state_dict())

        if episode % 1000 == 0:
            avg_reward = np.mean(all_rewards[-1000:])
            print(f"Episode {episode}/{episodes} | Avg Reward (last 1000):
{avg_reward:.2f} | Epsilon: {epsilon:.2f}")

        if np.mean(all_rewards[-10:]) > -110:
            print(f"Early stop at episode {episode}: near-optimal
performance.")
            goal_reached = True
            break

    env.close()
    return policy_net, goal_reached, all_rewards

def test_agent(policy_net, env_name="MountainCar-v0", runs=5,
render=False, device=device):
    env = gym.make(env_name, render_mode="human" if render else None)
```

```
    print("\n=== Running Evaluation Tests ===")
    success_count = 0
    rewards = []

    for i in range(runs):
        state, _ = env.reset()
        total_reward = 0
        done = False
        steps = 0

        while not done:
            with torch.no_grad():
                state_tensor =
torch.FloatTensor(state).unsqueeze(0).to(device)
                q_values = policy_net(state_tensor)
                action = torch.argmax(q_values).item()

            next_state, reward, terminated, truncated, _ =
env.step(action)
            done = terminated or truncated
            total_reward += reward
            steps += 1
            state = next_state

        rewards.append(total_reward)
        reached_goal = state[0] >= 0.5
        if reached_goal:
            success_count += 1
            print(f"Run {i+1}: Goal Reached in {steps} steps (Reward:
{total_reward:.2f})")
        else:
            print(f"Run {i+1}: Goal Not Reached (Final Pos:
{state[0]:.2f}, Reward: {total_reward:.2f})")

    avg_reward = np.mean(rewards)
    print(f"\n=== Summary ===")
    print(f"Average Reward: {avg_reward:.2f}")
    print(f"Goal reached in {success_count}/{runs} runs ({(success_count /
runs) * 100:.1f}% success rate)")
    env.close()
```

Let’s look over the reward plot.



Also let’s test the agent whether it is able to reach the goal or not.

```
test_agent(trained_policy, runs=5)
```

```
=== Running Evaluation Tests ===
Run 1: Goal Reached in 120 steps (Reward: -120.00)
Run 2: Goal Reached in 112 steps (Reward: -112.00)
Run 3: Goal Reached in 118 steps (Reward: -118.00)
Run 4: Goal Reached in 103 steps (Reward: -103.00)
Run 5: Goal Reached in 121 steps (Reward: -121.00)

=== Summary ===
Average Reward: -114.80
Goal reached in 5/5 runs (100.0% success rate)
```

Now let's us see the Roulette problem solved using the DQN:

```
class RouletteEnv(gym.Env):
    metadata = {"render_modes": []}

    def __init__(self, spins_per_episode=10):
        super().__init__()
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Discrete(1)
        self.state = np.array([0.0])
        self.spins_per_episode = spins_per_episode
        self.current_spin = 0

    def spin(self):
        number = np.random.randint(0, 37)
        if number == 0:
            return 2  # green
        elif 1 <= number <= 18:
            return 0  # red
        else:
            return 1  # black

    def step(self, action):
        outcome = self.spin()
        if action == outcome:
            reward = 35.0 if action == 2 else 1.0
        else:
            reward = -1.0

        # Reward normalization
        reward = reward / 10.0  # scale down large wins
        self.current_spin += 1
        done = self.current_spin >= self.spins_per_episode

        return self.state, reward, done, False, {}

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.state = np.array([0.0])
        self.current_spin = 0
        return self.state, {}
```

```
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=64):
        super(QNetwork, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, action_size)
        )

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


class ReplayBuffer:
    def __init__(self, capacity=10000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        return (
            torch.FloatTensor(states),
            torch.LongTensor(actions),
            torch.FloatTensor(rewards),
            torch.FloatTensor(next_states),
            torch.FloatTensor(dones)
        )

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

def train_dqn(env, episodes=5000, batch_size=64, gamma=0.95, lr=5e-3):
    state_size = 1
    action_size = env.action_space.n
```

```
    policy_net = QNetwork(state_size, action_size)
    target_net = QNetwork(state_size, action_size)
    target_net.load_state_dict(policy_net.state_dict())
    optimizer = optim.Adam(policy_net.parameters(), lr=lr)
    buffer = ReplayBuffer()

    epsilon = 1.0
    epsilon_decay = 0.997
    epsilon_min = 0.05
    update_target_every = 100

    all_rewards = []

    for episode in range(1, episodes + 1):
        state, _ = env.reset()
        done = False
        total_reward = 0

        while not done:

            if random.random() < epsilon:
                action = env.action_space.sample()
            else:
                with torch.no_grad():
                    q_values = policy_net(torch.FloatTensor(state))
                    action = torch.argmax(q_values).item()

            next_state, reward, done, _, _ = env.step(action)
            buffer.push(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward

            if len(buffer) >= batch_size:
                states, actions, rewards, next_states, dones =
buffer.sample(batch_size)
                q_values = policy_net(states).gather(1,
actions.unsqueeze(1)).squeeze(1)
                with torch.no_grad():
                    next_q_values = target_net(next_states).max(1)[0]
```

```
                    target = rewards + gamma * next_q_values * (1 - dones)
                loss = nn.MSELoss()(q_values, target)

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

        all_rewards.append(total_reward)
        epsilon = max(epsilon * epsilon_decay, epsilon_min)


        if episode % update_target_every == 0:
            target_net.load_state_dict(policy_net.state_dict())


        if episode % 1000== 0:
            avg_reward = np.mean(all_rewards[-1000:])
            print(f"Episode {episode}/{episodes} | Avg Reward (last 1000):
{avg_reward:.2f} | Epsilon: {epsilon:.2f}")

    return policy_net

def test_agent(policy_net, env, runs=10):
    print("\n=== Running Evaluation Tests ===")
    total_rewards = []
    success_count = 0

    for i in range(runs):
        state, _ = env.reset()
        done = False
        total_reward = 0

        while not done:
            with torch.no_grad():
                q_values = policy_net(torch.FloatTensor(state))
                action = torch.argmax(q_values).item()
            next_state, reward, done, _, _ = env.step(action)
            total_reward += reward

        total_rewards.append(total_reward)
```

```
        if total_reward >= 0:
            success_count += 1
            print(f"Run {i+1}: Total Reward = {total_reward:.2f} Goal
Reached")
        else:
            print(f"Run {i+1}: Total Reward = {total_reward:.2f} Goal Not
Reached")

    avg_reward = np.mean(total_rewards)
    print("\n=== Summary ===")
    print(f"Average Reward: {avg_reward:.2f}")
    print(f"Goal reached in {success_count}/{runs} runs
({(success_count/runs)*100:.1f}% success rate)")
```

The test was performed on the Roulette agent and below is the result of the goal reached:

```
test_agent(trained_policy, env, runs=10)


=== Running Evaluation Tests ===
Run 1: Total Reward = 0.00 Goal Reached
Run 2: Total Reward = 0.40 Goal Reached
Run 3: Total Reward = 0.20 Goal Reached
Run 4: Total Reward = -0.40 Goal Not Reached
Run 5: Total Reward = 0.40 Goal Reached
Run 6: Total Reward = 0.20 Goal Reached
Run 7: Total Reward = 0.20 Goal Reached
Run 8: Total Reward = 0.00 Goal Reached
Run 9: Total Reward = 0.20 Goal Reached
Run 10: Total Reward = -0.40 Goal Not Reached

=== Summary ===
Average Reward: 0.08
Goal reached in 8/10 runs (80.0% success rate)
```

The third and final problem is the shortest path problem achieved using the RL(Q-Learning) and DQN by constructing the graph from scratch.

```
class GraphEnv:
    def __init__(self, n_nodes, edges, start, goal, max_steps=50):
        self.n = n_nodes
        self.adj = {i: [] for i in range(n_nodes)}
        for u, v, w in edges:
            self.adj[u].append((v, w))
        self.start = start
        self.goal = goal
        self.max_steps = max_steps
        self.reset()

    def reset(self):
        self.state = self.start
        self.steps = 0
        return self.state

    def step(self, action):
        neighbors = self.adj[self.state]
        next_node, w = neighbors[action]
        self.steps += 1
        done = False
        if next_node == self.goal:
            reward = 100.0 - w
            done = True
        else:
            reward = -w
            if self.steps >= self.max_steps:
                done = True
        self.state = next_node
        return self.state, reward, done, {}

    def valid_actions(self, state=None):
        if state is None:
            state = self.state
        return list(range(len(self.adj[state])))

def train_q_learning(env, episodes=2000, alpha=0.5, gamma=0.99,
eps_start=1.0, eps_end=0.05):
```

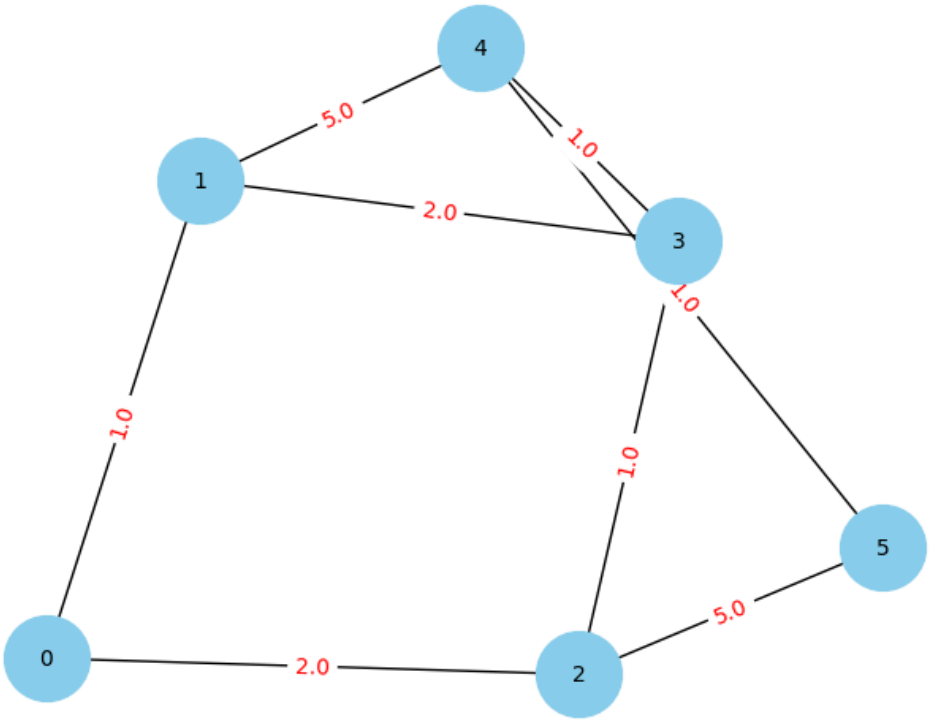
```
    Q = {s: np.zeros(len(env.adj[s])) for s in range(env.n)}
    eps_decay = (eps_start - eps_end) / episodes
    eps = eps_start
    start_time = time.time()
    for ep in range(episodes):
        s = env.reset()
        done = False
        while not done:
            valid = env.valid_actions(s)
            if random.random() < eps:
                a = random.choice(valid)
            else:
                a = int(np.argmax(Q[s]))
            next_s, r, done, _ = env.step(a)
            if not done and len(env.adj[next_s]) > 0:
                Q_next_max = np.max(Q[next_s])
            else:
                Q_next_max = 0.0
            Q[s][a] += alpha * (r + gamma * Q_next_max - Q[s][a])
            s = next_s
        eps = max(eps - eps_decay, eps_end)
    train_time = time.time() - start_time
    return Q, train_time


def evaluate_q(env, Q):
    s = env.reset()
    path = [s]
    done = False
    while not done and len(path) < env.max_steps:
        a = int(np.argmax(Q[s]))
        s, r, done, _ = env.step(a)
        path.append(s)
        if done:
            break
    return path
```

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 64), nn.ReLU(),
            nn.Linear(64, 64), nn.ReLU(),
            nn.Linear(64, action_dim)
        )

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


class ReplayBuffer:
    def __init__(self, capacity=10000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        batch = random.sample(self.buffer, min(len(self.buffer),
batch_size))
        s, a, r, s2, d = zip(*batch)
        return np.array(s), a, r, np.array(s2), d

    def __len__(self):
        return len(self.buffer)
def train_dqn(env, episodes=2000, gamma=0.99, eps_start=1.0, eps_end=0.05,
lr=1e-3, batch_size=64, target_update=50, early_stop_patience=100):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    state_dim = env.n
    action_dim = max(len(env.adj[s]) for s in range(env.n))

    policy_net = DQN(state_dim, action_dim).to(device)
    target_net = DQN(state_dim, action_dim).to(device)
    target_net.load_state_dict(policy_net.state_dict())

    optimizer = optim.Adam(policy_net.parameters(), lr=lr)
    buffer = ReplayBuffer(10000)
```

```
    eps = eps_start
    eps_decay = (eps_start - eps_end) / episodes
    start_time = time.time()

    best_reward = -float('inf')
    no_improve_count = 0

    for ep in range(episodes):
        s = env.reset()
        s_vec = np.zeros(env.n)
        s_vec[s] = 1.0
        done = False
        total_reward = 0

        while not done:
            valid = env.valid_actions(s)
            if random.random() < eps:
                a = random.choice(valid)
            else:
                with torch.no_grad():
                    q_values =
policy_net(torch.FloatTensor(s_vec).to(device))
                    mask = torch.full((action_dim,), -1e9, device=device)
                    mask[valid] = q_values[valid]
                    a = int(torch.argmax(mask).item())

            next_s, r, done, _ = env.step(a)
            total_reward += r
            ns_vec = np.zeros(env.n)
            ns_vec[next_s] = 1.0

            buffer.push(s_vec, a, r, ns_vec, done)
            s, s_vec = next_s, ns_vec

            if len(buffer) >= batch_size:
                s_b, a_b, r_b, s2_b, d_b = buffer.sample(batch_size)
                s_b = torch.FloatTensor(s_b).to(device)
                s2_b = torch.FloatTensor(s2_b).to(device)
                a_b = torch.LongTensor(a_b).to(device)
```

```
                r_b = torch.FloatTensor(r_b).to(device)
                d_b = torch.FloatTensor(d_b).to(device)

                q_values = policy_net(s_b).gather(1,
a_b.unsqueeze(1)).squeeze()
                with torch.no_grad():
                    q_next = target_net(s2_b).max(1)[0]
                    target = r_b + gamma * q_next * (1 - d_b)

                loss = nn.MSELoss()(q_values, target)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

        eps = max(eps - eps_decay, eps_end)

        if total_reward > best_reward:
            best_reward = total_reward
            no_improve_count = 0
        else:
            no_improve_count += 1

        if no_improve_count >= early_stop_patience:
            print(f"Early stopping at episode {ep} with best reward
{best_reward:.2f}")
            break

        if (ep + 1) % target_update == 0:
            target_net.load_state_dict(policy_net.state_dict())

    train_time = time.time() - start_time
    return policy_net, train_time
def evaluate_dqn(env, net):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    s = env.reset()
    path = [s]
    done = False
    action_dim = max(len(env.adj[s]) for s in range(env.n))
    while not done and len(path) < env.max_steps:
        s_vec = np.zeros(env.n)
```

```
        s_vec[s] = 1.0
        with torch.no_grad():
            q_values = net(torch.FloatTensor(s_vec).to(device))
            mask = torch.full((action_dim,), -1e9, device=device)
            mask[env.valid_actions(s)] = q_values[env.valid_actions(s)]
            a = int(torch.argmax(mask).item())
        s, _, done, _ = env.step(a)
        path.append(s)
    return path
def floyd_warshall_table(edges, n_nodes):
    INF = float('inf')
    dist = np.full((n_nodes, n_nodes), INF)
    np.fill_diagonal(dist, 0)
    for u, v, w in edges:
        dist[u][v] = w
        dist[v][u] = w
    for k in range(n_nodes):
        for i in range(n_nodes):
            for j in range(n_nodes):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    df = pd.DataFrame(dist, columns=[f'Node {i}' for i in range(n_nodes)])
    df.index = [f'Node {i}' for i in range(n_nodes)]
    return df
edges = [
    (0, 1, 1.0), (0, 2, 2.0),
    (1, 3, 2.0), (2, 3, 1.0),
    (1, 4, 5.0), (3, 4, 1.0),
    (4, 5, 1.0), (2, 5, 5.0)
]

G = nx.Graph()
G.add_weighted_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=1500,
font_size=10)
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
font_color='red')
plt.show()
```




```
print("\n--- Floyd Warshall Shortest Distance Table ---")
table = floyd_warshall_table(edges, 6)
print(table)
```

```
--- Floyd Warshall Shortest Distance Table ---
        Node 0  Node 1  Node 2  Node 3  Node 4  Node 5
Node 0     0.0     1.0     2.0     3.0     4.0     5.0
Node 1     1.0     0.0     3.0     2.0     3.0     4.0
Node 2     2.0     3.0     0.0     1.0     2.0     3.0
Node 3     3.0     2.0     1.0     0.0     1.0     2.0
Node 4     4.0     3.0     2.0     1.0     0.0     1.0
Node 5     5.0     4.0     3.0     2.0     1.0     0.0
```

```
fw_distance = table.loc['Node 0', 'Node 5']
print(f"\nInitial shortest distance (Floyd-Warshall): {fw_distance}")
```

```
Initial shortest distance (Floyd-Warshall): 5.0
```

The shortest distance problem needs to be solved using both the Q-Learning and DQN and needed to log into the table for comparison. First we calculate the all pair shortest path using the Floyd-Warshall algorithm for reporting the true distance of shortest path available.
Then, we are going to calculate the shortest path using Q-Learning and DQN and also log the shortest distance registered and also the path taken to achieve the shortest distance. Below is the comparison:

```
print("\n--- Comparison Table ---")
comparison = pd.DataFrame([
    {"Method": "Floyd-Warshall", "Shortest Distance": fw_distance, "Path": "Optimal Analytical"},
    {"Method": "Q-learning", "Shortest Distance": len(q_path), "Path": q_path},
    {"Method": "DQN (PyTorch)", "Shortest Distance": len(dqn_path), "Path": dqn_path}
])
print(comparison)
```

```
--- Comparison Table ---
           Method  Shortest Distance                Path
0  Floyd-Warshall                5.0  Optimal Analytical
1      Q-learning                5.0      [0, 1, 3, 4, 5]
2   DQN (PyTorch)                5.0      [0, 2, 3, 4, 5]
```

From here we can observe that the true distance of shortest path between node-0 and node-5 is 5. Q-Learning and DQN both performed great here as they both registered shortest distance 5 and though the path taken by them are different but by observing the graph we can conclude that both the path taken truly leads to the target using the shortest path.

Conclusion:
The Q-Learning creates a Q-table that stores the reward gained for each course of state and action. The agent performs action and thus is assigned reward/penalty based upon it. The Mountain Car problem tries to teach the agent to climb uphill to reach the goal. Upon multiple iteration and reward/penalty allotment the agent finally was able to climb the mountain. The test performed has a 100% success rate of achieving the goal. Below is the image attached:

```
Running 5 tests for the agent...
Test 1: Goal reached!
Test 2: Goal reached!
Test 3: Goal reached!
Test 4: Goal reached!
Test 5: Goal reached!

Goal reached in 5/5 tests.
```

The Roulette problem is also a very random problem in RL where the agent takes bets on a range of numbers available and targets to take its current balance to target balance. The Q-table is also created upon the (bet number, bet amount). Upon multiple iterations the agent finally learns to take bets to take the current balance amount to the target balance amount. This achievement is shown below:

```
test_agent(Q, env)

Test 1: Final Balance = 103, Total Reward = 153
Test 2: Final Balance = 112, Total Reward = 162
Test 3: Final Balance = -45, Total Reward = -145
Test 4: Final Balance = 114, Total Reward = 164
Test 5: Final Balance = -24, Total Reward = -124
Test 6: Final Balance = 110, Total Reward = 160
Test 7: Final Balance = 102, Total Reward = 152
Test 8: Final Balance = 120, Total Reward = 170
Test 9: Final Balance = -48, Total Reward = -148
Test 10: Final Balance = 130, Total Reward = 180

Goal Reached in 7/10 tests (70.0% success rate)
```

The DQN is another approach to RL where the neural network replaces the traditionally used Q-table to approximate the (state, action) → reward. This pulls down the hassle to discretize the values for action as needed in the Q-Learning. The neural network learns to model to be able to approximate the inputs. The mountain car problem is being able to learn and perform greatly. Below is the performance test:

```
test_agent(trained_policy, runs=5)
```

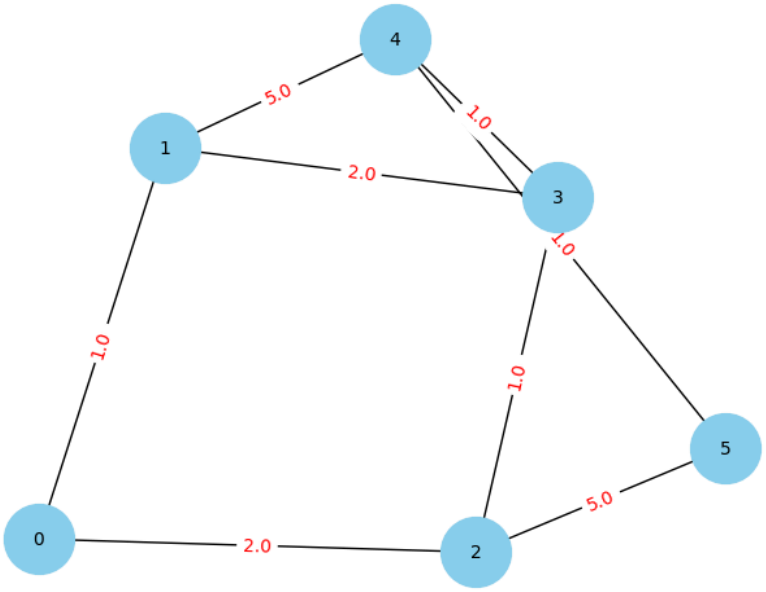
```
=== Running Evaluation Tests ===
Run 1: Goal Reached in 120 steps (Reward: -120.00)
Run 2: Goal Reached in 112 steps (Reward: -112.00)
Run 3: Goal Reached in 118 steps (Reward: -118.00)
Run 4: Goal Reached in 103 steps (Reward: -103.00)
Run 5: Goal Reached in 121 steps (Reward: -121.00)

=== Summary ===
Average Reward: -114.80
Goal reached in 5/5 runs (100.0% success rate)
```

The roulette problem is also trained using DQN and below is the performance:

```
test_agent(trained_policy, env, runs=10)
```

```
=== Running Evaluation Tests ===
Run 1: Total Reward = 0.00 Goal Reached
Run 2: Total Reward = 0.40 Goal Reached
Run 3: Total Reward = 0.20 Goal Reached
Run 4: Total Reward = -0.40 Goal Not Reached
Run 5: Total Reward = 0.40 Goal Reached
Run 6: Total Reward = 0.20 Goal Reached
Run 7: Total Reward = 0.20 Goal Reached
Run 8: Total Reward = 0.00 Goal Reached
Run 9: Total Reward = 0.20 Goal Reached
Run 10: Total Reward = -0.40 Goal Not Reached

=== Summary ===
Average Reward: 0.08
Goal reached in 8/10 runs (80.0% success rate)
```

The shortest path problem in graph is one of the classical problems and below is the graph:



The final table obtained is:

```
print("\n--- Comparison Table ---")
comparison = pd.DataFrame([
    {"Method": "Floyd-Warshall", "Shortest Distance": fw_distance, "Path": "Optimal Analytical"},
    {"Method": "Q-learning", "Shortest Distance": len(q_path), "Path": q_path},
    {"Method": "DQN (PyTorch)", "Shortest Distance": len(dqn_path), "Path": dqn_path}
])
print(comparison)
```

```
--- Comparison Table ---
           Method  Shortest Distance                Path
0  Floyd-Warshall                5.0  Optimal Analytical
1      Q-learning                5.0     [0, 1, 3, 4, 5]
2   DQN (PyTorch)                5.0     [0, 2, 3, 4, 5]
```

Github Link: [Link]