Homework 2

Name – Darshan Chudiwal
PSU ID - 926435336

Abstract –

This code implements a simple Deep Q-Network (DQN) algorithm to train an agent for reinforcement learning tasks. It includes a QNetwork class that defines a neural network to approximate the action-value function, a ReplayBuffer class for storing and sampling past experiences to stabilize learning, a DQNAgent class that handles action selection, learns from experiences, and updates network parameters, and a training loop where the agent interacts with the environment, collects experiences, and learns to maximize cumulative rewards. Overall, it demonstrates how to train a DQN agent to learn optimal policies through interaction with an environment. The environment here could be any environment from the Gymnasium library, but I wrote the code with LunarLander in mind.

Code – It is something I worked on, inspired by this article – https://www.katnoria.com/nb_dqn_lunar/

QNetwork class :

```
# Define the QNetwork class inheriting from PyTorch's neural network module
class QNetwork(nn.Module):
    # Initialize the network with state and action sizes and a random seed
    def __init__(self, state_size, action_size, seed):
        super(QNetwork, self).__init__()
        # Set the random seed for reproducibility
        self.seed = torch.manual_seed(seed)
        # Define the first fully connected layer (input layer). This is a simple implementation and does not use CNN as described in the DQN paper.
        self.fc1 = nn.Linear(state_size, 128)
        # Define the second fully connected layer (hidden layer)
        self.fc2 = nn.Linear(128, 128)
        # Define the final fully connected layer (output layer)
        self.fc3 = nn.Linear(128, action_size)

    # Define the forward pass through the network
    def forward(self, x):
        # Pass input through the first layer and apply ReLU activation
        x = F.relu(self.fc1(x))
        # Pass through the second layer with ReLU activation
        x = F.relu(self.fc2(x))
        # Pass through the final layer to get action values (Q-values)
        return self.fc3(x)
```

Replay Buffer :

```
# Define the ReplayBuffer class to store and sample experiences
class ReplayBuffer:
    # Initialize the buffer with size, batch size, and seed
    def __init__(self, buffer_size, batch_size, seed):
        self.batch_size = batch_size
        # Set the random seed for reproducibility
        self.seed = random.seed(seed)
        # Initialize the memory as a deque with a maximum length
        self.memory = deque(maxlen=buffer_size)
        # Define a named tuple to store experiences and later use from
        self.experience = namedtuple("Experience", field_names=["observation", "action", "reward", "next_state", "terminated"])
```

```

    # Method to add a new experience to memory
    def add(self, observation, action, reward, next_state, terminated):
        # Handle cases where observation is a tuple (from certain environments, in this case the environment was LunarLandingv2)
        if isinstance(observation, tuple):
            observation = observation[0]
        if isinstance(next_state, tuple):
            next_state = next_state[0]
        # Create an experience tuple
        experience = self.experience(observation, action, reward, next_state, terminated)
        # Append the experience to the memory buffer
        self.memory.append(experience)

    # Method to sample a batch of experiences from memory
    def sample(self):
        # Randomly sample experiences equal to the batch size
        experiences = random.sample(self.memory, k=self.batch_size)
        # Convert observations to a tensor
        observations = torch.from_numpy(np.vstack([experience.observation for experience in experiences])).float().to(device)
        # Convert actions to a tensor and reshape
        actions = torch.from_numpy(np.vstack([experience.action for experience in experiences]).reshape(-1, 1)).long().to(device)
        # Convert rewards to a tensor and reshape
        rewards = torch.from_numpy(np.vstack([experience.reward for experience in experiences]).reshape(-1, 1)).float().to(device)
        # Convert next states to a tensor
        next_states = torch.from_numpy(np.vstack([experience.next_state for experience in experiences])).float().to(device)
        # Convert termination flags to a tensor, cast to uint8, and reshape
        terminateds = torch.from_numpy(np.vstack([experience.terminated for experience in experiences]).astype(np.uint8).reshape(-1, 1)).float().to(device)
        # Return the sampled experiences as tensors
        return (observations, actions, rewards, next_states, terminateds)

    # Return the current size of the memory buffer
    def __len__(self):
        return len(self.memory)
```

DQNAgent :

```
# Hyperparameters
BUFFER_SIZE = int(1e5)  # Size of the replay memory
BATCH_SIZE = 64         # Number of experiences to sample
GAMMA = 0.99            # Discount factor for future rewards
TAU = 1e-3              # Soft update parameter for the target network
LR = 1e-4               # Learning rate for the optimizer
UPDATE_EVERY = 4        # Frequency of network updates

# Define the DQNAgent class
class DQNAgent:
    # Initialize the agent with state and action sizes and a seed
    def __init__(self, state_size, action_size, seed):
        self.state_size = state_size
        self.action_size = action_size
        # Set the random seed for reproducibility
        self.seed = random.seed(seed)
        # Initialize the Q-network (local network)
        self.q_network = QNetwork(state_size, action_size, seed).to(device)
        # Initialize the fixed Q-network (target network)
        self.fixed_network = QNetwork(state_size, action_size, seed).to(device)
        # Define the optimizer for the Q-network
        self.optimizer = optim.Adam(self.q_network.parameters())
        # Initialize the replay memory
        self.memory = ReplayBuffer(BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize the timestep counter
        self.timestep = 0

```

```python
    # Method to process a step in the environment
    def step(self, observation, action, reward, next_state, terminated):
        # Add the experience to replay memory
        self.memory.add(observation, action, reward, next_state, terminated)
        # Increment the timestep
        self.timestep += 1
        # If it's time to update the network
        if self.timestep % UPDATE_EVERY == 0:
            # Ensure there are enough experiences in memory
            if len(self.memory) > BATCH_SIZE:
                # Sample a batch of experiences
                sampled_experiences = self.memory.sample()
                # Learn from the sampled experiences
                self.learn(sampled_experiences)

    # Method to learn from a batch of experiences
    def learn(self, experiences):
        # Unpack the experiences
        states, actions, rewards, next_states, terminateds = experiences
        # Get the Q-values from the target network for next states (detach to avoid gradients)
        action_values = self.fixed_network(next_states).detach()
        # Get the maximum Q-value for each next state
        max_action_values = action_values.max(1)[0].unsqueeze(1)
        # Compute the target Q-values using the Bellman equation
        Q_target = rewards + (GAMMA * max_action_values * (1 - terminateds))
        # Get the expected Q-values from the local network
        Q_expected = self.q_network(states).gather(1, actions)
        # Calculate the loss between expected and target Q-values
        loss = F.mse_loss(Q_expected, Q_target)
        # Zero the parameter gradients
        self.optimizer.zero_grad()
        # Perform backpropagation
        loss.backward()
        # Update the network weights
        self.optimizer.step()
        # Soft update the target network
        self.update_fixed_network(self.q_network, self.fixed_network)

    # Method to update the target network parameters
    def update_fixed_network(self, q_network, fixed_network):
        # Iterate over parameters of both networks
        for source_parameters, target_parameters in zip(q_network.parameters(), fixed_network.parameters()):
            # Perform soft update of target network parameters
            target_parameters.data.copy_(TAU * source_parameters.data + (1.0 - TAU) * target_parameters.data)

    # Method to select an action using an epsilon-greedy policy
    def act(self, observation, eps=0.0):
        # Handle cases where observation is a tuple
        if isinstance(observation, tuple):
            observation = observation[0]
        # Generate a random number for epsilon-greedy action selection
        rnd = random.random()
        # If random number is less than epsilon, select a random action (exploration)
        if rnd < eps:
            return np.random.randint(self.action_size)
        else:
            # Convert observation to a tensor and add a batch dimension
            observation = torch.from_numpy(observation).float().unsqueeze(0).to(device)
            # Set the network to evaluation mode
            self.q_network.eval()
            # Disable gradient calculation
            with torch.no_grad():
                # Get action values from the Q-network
                action_values = self.q_network(observation)
            # Set the network back to training mode
            self.q_network.train()
            # Select the action with the highest Q-value
```

```
            action = np.argmax(action_values.cpu().data.numpy())
            return action

    # Method to save the trained model weights
    def checkpoint(self, filename):
        torch.save(self.q_network.state_dict(), filename)
```

Training Loop :

```
# Instantiate the DQN agent
dqn_agent = DQNAgent(state_size, action_size, seed=0)

# Initialize a list to keep track of scores
dqn_scores = []
# Initialize a deque to store scores of the last 100 episodes
dqn_scores_window = deque(maxlen=100)
# Initialize epsilon for the epsilon-greedy policy
eps = EPS_START
# Record the start time of training
start = time()
# Initialize a list to store actions taken in each episode
dqn_actions = []

# Loop over episodes
for episode in range(1, MAX_EPISODES + 1):
    # Reset the environment and get the initial state
    state = env.reset(seed = 42)
    # Initialize the score for the current episode
    score = 0
    # Initialize a list to store actions in the current episode
    episode_actions = []

    # Loop over steps within an episode
    for t in range(MAX_STEPS):
        # Select an action using the agent's policy
        action = dqn_agent.act(state, eps)
        # Append the action to the episode's action list
        episode_actions.append(action)
        # Take the action in the environment and observe the outcome
        observation, reward, terminated, truncated, info = env.step(action)
        # Handle cases where observation is a tuple
        if isinstance(observation, tuple):
            observation = observation[0]
        # Let the agent process the step (store experience and learn)
        dqn_agent.step(state, action, reward, observation, terminated)
        # Update the state to the next state
        state = observation
        # Accumulate the reward to the score
        score += reward
        # If the episode is terminated, exit the loop
        if terminated:
            break
        # Decay epsilon
        eps = max(eps * EPS_DECAY, EPS_MIN)
        mean_score = 0
        # Every PRINT_EVERY episodes, print the progress
        if episode % PRINT_EVERY == 0:
            # Calculate the average score over the last 100 episodes
            mean_score = np.mean(dqn_scores_window)
            print('\r Progress {}/{}, average score:{:.2f}'.format(episode, MAX_EPISODES, mean_score), end="")
        # If the environment is considered solved
        if mean_score >= ENV_SOLVED:
            print('\rEnvironment solved in {} episodes, average score: {:.2f}'.format(episode, mean_score), end="")
            sys.stdout.flush()
            # Save the trained model weights
            dqn_agent.checkpoint('solved_200.pth')
```

```
            break
    # Append the score of the episode to the deque and list
    dqn_scores_window.append(score)
    dqn_scores.append(score)
    # Append the actions taken in the episode to the list
    dqn_actions.append(episode_actions)

# Record the end time of training
end = time()
# Print the total training time
print('Took {} seconds'.format(end - start))
# Store the total training time
dqn_time = end-start
```