CSCI/ROBO 7000/4830:

Deep Reinforcement Learning and Robotics

Fall 2025

State & Reward
Agent
Environment
Actions

Homework #3: Stabilizing Deep Q-Networks

Introduction

In our previous assignments, you've worked with model-based (HW1) and tabular, model-free methods (HW2). The Q-tables you built worked for small, discrete state spaces like "Cliff Walking." However, as we've seen in class, <u>these tables fail completely when a state space becomes too large or continuous</u>. In HW3, we'll explore a domain where all the things we discussed in class become self-evident.

This assignment is the bridge to modern deep RL. You will implement a Deep Q-Network (DQN), an algorithm that uses a neural network to approximate the Q-value function. To make this work, you must first implement the two key innovations that the DQN authors brought to stabilize the learning process: Experience Replay and Target Networks. Finally, you will implement a third innovation, Double DQN, to address the algorithm's tendency to overestimate Q-values.

The Environment: MiniGrid-Dynamic-Obstacles-V0

To prove our agent can handle a complex task, you will use the MiniGrid-Dynamic-Obstacles-v0 environment. More info here [LINK].

- Goal: Navigate a 2D grid from a start state to a goal state.
- Challenge: The grid is filled with obstacles (grey squares) that move randomly on every step.
- State: The agent receives a 7x7 "field of view" which is flattened into a vector. A Q-table is impossible, as the number of possible grid configurations is astronomically large. We must use VFA.
- Actions: Discrete(3): 0 (turn left), 1 (turn right), 2 (move forward).
- Rewards: A reward of ‘1 - 0.9 * (step_count / max_steps)’ is given for success, and ‘0’ for failure. A ‘-1’ penalty is subtracted if the agent collides with an obstacle, and the episode terminates.

Part 1: Implementation [50 Points]

Your task is to build a **DQNAgent** from scratch using PyTorch. You will need to implement three main components.

1.1. The Q-Network [10 Points]

Your task is to implement two agents. You should use **NumPy** for your Q-tables and mathematical operations. Create a QNetwork class that inherits from torch.nn.Module. This network is our function approximator, $\hat{q}(s, a, w)$.

- Input: The state vector (from env.observation_space).
- Output: A Q-value for each possible action (3 actions).
- Architecture: Use a simple Multi-Layer Perceptron (MLP). A good starting point is:
  - Input Layer
  - Hidden Layer 1: 64 neurons (ReLU activation)
  - Hidden Layer 2: 64 neurons (ReLU activation)
  - Output Layer: 3 neurons (one per action, linear activation)

1.2. The Replay Buffer [10 Points]

To break the correlation between sequential experiences, we must use an Experience Replay buffer. Create a ReplayBuffer class.

- It should store transitions as (state, action, reward, next_state, done) tuples.

- Implement a push() method to add new transitions. If the buffer is full, it should evict the oldest transition.
- Implement a sample(batch_size) method that returns a random mini-batch of transitions from the buffer.

1.3. The DQNAgent Training Loop [30 Points]

This is the main logic that brings everything together. Your training loop should perform the following steps for each episode (see slides for more information):

1. Initialize the environment and get the first state.
2. For each time step t:
    a. Select an action using an ϵ-greedy policy.
    b. Take that action in the environment to get reward, next_state, …
    c. Store this transition in your ReplayBuffer.
    d. Check if it's time to learn. If your buffer has enough transitions, sample a mini-batch of size batch_size.
    e. Calculate the TD Target. For your initial implementation (Part 2.1), you will use the standard DQN target calculation:
    $y_j = r_j$ (if episode terminates at step $j + 1$)
    $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_j', a', w^-)$ (otherwise)
    f. Calculate the Loss. Pass the mini-batch of states through your main network to get the predicted Q-values. The loss is the Mean-Squared Error (MSE) between the predicted Q-values and the TD targets (y_j) you just calculated.
    g. Perform Gradient Descent. Zero the gradients, call loss.backward(), and step your torch.optim optimizer.
    h. Update the Target Network. Every C steps (e.g., C=100), copy the weights from your main network to your target network.
    i. If done (episode terminated), break and start the next episode

---

Part 2: Analysis and Reasoning [50 Points]

A functioning DQN relies on stabilizing mechanisms. For this part, you will break your own code to prove you understand why they are necessary, and then improve it to fix a core flaw.

Task: You will run four experiments. For each, train for at least 1000 episodes and generate a plot of the "Sum of Rewards per Episode" (applying a rolling average of ~50 episodes to the plot is highly recommended to see the trend).

2.1 Baseline (Full DQN) [10 Points]

Run your full DQN implementation from Part 1 and submit your plot. Your agent should show a clear learning trend.

Briefly explain the high-level role of your ReplayBuffer and your target_network in achieving this stable learning.

2.2: *Experiment B - No Experience Replay [10 Points]*

Modify your code to remove the replay buffer. Instead of sampling, train online (i.e., at every step, your "batch" is just the single, most recent transition).

Submit your new plot. Compare its stability and performance to the baseline.

Using concepts from Lecture 13, explain why training on sequential, correlated samples is a problem for a neural network, which expects i.i.d. data.

2.3: *Experiment C - No Target Network [10 Points]*

Restore the replay buffer. Now, modify your update rule to remove the target network. Use your main network for both calculating the TD target and predicting the Q-values. (This is equivalent to setting the target network update frequency C to 1).

Submit your third plot. This configuration is notoriously unstable. Explain why this instability occurs by referencing the "moving target" problem.

2.4: *Experiment D - Fixing Overestimation with Double DQN [15 Points]*

Q-learning is known to suffer from overestimation bias. The standard DQN target calculation propagates this error. Implement Double DQN by modifying your TD target calculation as per slides.

Submit your fourth plot (DDQN). Compare this plot to your baseline DQN plot from 2.1. Is it more stable? Does it learn faster? Explain why this new update rule helps prevent the overestimation of Q-values.

2.5: *Conceptual Check [5 Points]*

In HW2, you used a Q-table for "Cliff Walking." Explain in detail why a Q-table is a completely impractical solution for MiniGrid-Dynamic-Obstacles-v0. How does your QNetwork from Part 1 solve this "scaling" problem?

Coding and Installation

You will need Python, `gymnasium`, `torch`, and `minigrid`.

```
pip install gymnasium torch minigrid
```

To create and wrap the environment, use the following code:

```
import gymnasium as gym

from minigrid.wrappers import FlatObsWrapper

# Create the environment
```

```
env = gym.make("MiniGrid-Dynamic-Obstacles-v0")

# Wrap the environment to get a flat observation vector

env = FlatObsWrapper(env)

print(f"Observation space: {env.observation_space.shape}")

print(f"Action space: {env.action_space.n}")
```

Implementation Guidelines & File Structure

To keep your project organized and make it easy to manage, you must structure your code in separate files. This is a common practice in software development and will make your code much easier to read, debug, and grade.

Please follow this file structure:

- q_network.py: This file must contain your QNetwork class definition (the class that inherits from torch.nn.Module).
- replay_buffer.py: This file must contain your ReplayBuffer class definition.
- dqn_agent.py: This file should contain your main DQNAgent class. This class will be the "brain" of your operation. It should:
    - Import and create instances of your QNetwork (both main and target) and your ReplayBuffer.
    - Contain the logic for selecting an action (your ϵ-greedy policy).
    - Contain the main learn() method that samples from the buffer and performs the gradient descent step.
- main_hw3.py: This is your main executable file. It should contain the code to:
    - Create the gymnasium environment (and apply the wrapper).
    - Create an instance of your DQNAgent.
    - Run the main training loop (the loop over episodes and steps).
    - Call your agent's methods to select actions and learn.
    - Store the rewards for plotting.
- plots.py (Optional but Recommended): A separate file to generate and save your analysis plots from the reward data.