Contents

1 Overview

This report describes an alternative implementation of a reinforcement learning (RL) assignment. The goal is to solve the same set of RL tasks using a slightly different style of coding and some advanced design choices.

The implementation includes:

- Task 1: Checking and exploring the Gymnasium environment setup
- Task 2: Solving FrozenLake using tabular Q-learning
- Task 3: Solving MountainCar using a Deep Q-Network (DQN)

This report is written in simple language and focuses on explaining what was done and why.

2 Task 1: Environment Setup

2.1 Goal

The goal of Task 1 was to verify that Gymnasium is working correctly and that the environment can be created and interacted with properly.

2.2 What I Did

I created a script named `task1_environment_setup.py`. The script does the following:

1. Imports Gymnasium
2. Creates the `FrozenLake-v1` environment
3. Resets the environment and prints the initial state
4. Runs a few random actions to confirm the environment responds normally
5. Prints environment information like state space and action space
6. Tests reproducibility using seeds (same seed should give similar behaviour)
7. Tracks the short trajectory (sequence of states visited)

2.3 Observations

Important observations from this setup test:

- FrozenLake has a small discrete state space ($4 \times 4 = 16$ states)
- It also has a discrete action space (4 moves: Left, Down, Right, Up)

- The environment can be stochastic when `slippery=True`, meaning actions are not always executed perfectly
- Using seeds helps test if results can be reproduced

3 Task 2: Q-Learning on FrozenLake

3.1 Environment Description

FrozenLake is a grid-world environment:

- **S** = Start position
- **G** = Goal position
- **H** = Hole (episode ends, no reward)
- **F** = Safe frozen tile

The reward is simple:

- Reward = 1 only when the agent reaches the goal
- Reward = 0 otherwise

Because of slippery ice, the same action can sometimes lead to different next states.

3.2 Algorithm Used: Tabular Q-Learning

Q-learning is a model-free RL method that learns a table of values $Q(s, a)$, where:

- s is the state
- a is the action
- $Q(s, a)$ tells how good it is to take action a in state s

The update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

3.3 Hyperparameters Used

Parameter	Value	Simple Meaning
Learning rate (α)	0.15	How fast Q-values update
Discount factor (γ)	0.99	How much future reward matters
Initial epsilon (ϵ)	1.0	Start with full exploration
Minimum epsilon	0.01	Keep a bit of exploration
Epsilon decay	0.997	Slowly reduce exploration
Episodes	15000	Train long enough to learn
Max steps/episode	100	Stop if too many steps

3.4 How Training Works (Simple Steps)

Training follows this loop:

1. Start episode, reset environment
2. Choose action using epsilon-greedy:
    - With probability ϵ: pick random action
    - Otherwise: pick best action from the Q-table
3. Take action, observe next state and reward
4. Update Q-table using the update rule
5. Repeat until episode ends
6. Reduce epsilon a little (epsilon decay)

3.5 Outputs Saved

This task generates:

- `task2_results.png` (learning curves and heatmaps)
- `task2_qtable.npy` (final learned Q-table)

3.6 Common Learning Behaviour

A simple pattern usually happens:

- Early training: mostly random, success is low
- Middle: agent starts finding useful safe paths
- Late: agent improves and becomes stable

4 Task 3: Deep Q-Learning on MountainCar

4.1 Environment Description

MountainCar is an RL environment where:

- The car must reach a target position (flag)
- The engine is weak, so it cannot climb directly
- The correct trick is to build momentum by moving back and forth

State is continuous:

- Position (continuous)
- Velocity (continuous)

Actions are discrete:

- Push left
- No push
- Push right

The reward is usually:

- -1 every time step until the goal is reached

4.2 Why We Need a Neural Network

Because the state space is continuous, we cannot store a Q-table for every possible state. So we approximate $Q(s, a)$ using a neural network. This is called a Deep Q-Network (DQN).

4.3 DQN Architecture Used

The neural network used was:

- Input layer: 2 values (position, velocity)
- Hidden layer 1: 256 neurons (ReLU)
- Hidden layer 2: 128 neurons (ReLU)
- Hidden layer 3: 64 neurons (ReLU)
- Output layer: 3 values (Q-values for each action)

A small dropout (0.1) was also used to reduce overfitting.

4.4 Experience Replay

The agent stores transitions like:

$$(state, action, reward, next\_state, done)$$

in a replay buffer. Then it trains by sampling random batches. This helps because RL data is highly correlated if we train only in strict time order.

4.5 Prioritized Replay (Simplified)

Instead of sampling all experiences equally, the implementation tries to sample more "important" experiences more often (like surprising transitions). This makes learning faster in many cases.

4.6 Target Network

DQN uses a separate target network to compute stable targets. The target network is updated every few episodes by copying weights from the main network.

4.7 Reward Shaping

To make learning faster, reward shaping was added:

- Small bonus for moving right
- Extra bonus for higher velocity in the correct direction
- Large bonus for reaching the goal

This gives intermediate feedback instead of only punishment each step.

4.8 Hyperparameters Used

Parameter	Value	Simple Meaning
Learning rate	0.001	Step size for optimizer
Discount factor (γ)	0.99	Future reward importance
Initial epsilon	1.0	Full exploration first
Minimum epsilon	0.01	Always keep some exploration
Epsilon decay	0.995	Slowly reduce randomness
Batch size	128	Train on 128 samples at a time
Replay buffer size	20000	Store many experiences
Target update	Every 10 episodes	Keep training stable
Episodes	600	Training length
Dropout	0.1	Regularization for network

4.9 Outputs Saved

This task generates:

- `task3_results.png` (training plots)
- `task3_model.pth` (saved network weights)

5 Common Issues and Fixes

5.1 Q-Learning Not Improving

Possible fixes:

- Increase number of episodes
- Keep exploration longer (slower epsilon decay)
- Check if learning rate is too small or too large

5.2 DQN Loss Becomes Very Large

Possible fixes:

- Reduce learning rate (example: 0.0001)
- Clip gradients to prevent explosions
- Reduce reward shaping magnitude if it is too large

5.3 Training Too Slow

Possible fixes:

- Reduce episodes temporarily during debugging
- Reduce replay buffer size
- Use smaller batch sizes

6 Experimentation Ideas

Some simple experiments that can be tried later:

6.1 FrozenLake

- Learning rate: 0.05, 0.10, 0.15, 0.20
- Epsilon decay: 0.990, 0.995, 0.997, 0.999
- Gamma: 0.90, 0.95, 0.99, 0.995

6.2 MountainCar

- Network sizes: 128–64, 256–128–64, 512–256–128
- Dropout: 0, 0.1, 0.2, 0.3
- Batch size: 32, 64, 128, 256
- Buffer size: 5000, 10000, 20000, 50000