Islamic University of Technology

ARTIFICIAL INTELLIGENCE

CSE 4637

Lab 6 Reinforcement Learning

Author:
Atik Shahriar Ovi
ID: 210042176

September 8, 2025

Contents

1 Lab 6 : Q-Learning

In this lab,we will implement Q-learning. We will test our agent first on Gridworld, then apply them to a simulated robot controller (Crawler) and Pacman.The task focused on enabling an agent to learn an optimal policy through interaction with its environment, without having prior knowledge of transition probabilities or reward structures.

Solution Codes

Solution of the lab tasks can be found on my GitHub .

Background

Reinforcement Learning (RL) provides a framework for training agents to maximize cumulative reward. A Markov Decision Process (MDP) is the formal model for such problems, but it assumes that transition probabilities and rewards are known in advance. In contrast, Q-learning is a *model-free* approach: the agent does not need to know the full MDP, and instead learns the optimal policy through trial and error.

The central idea of Q-learning is to maintain a value function $Q(s, a)$ that estimates the expected return of taking action a in state s and then following the best possible future actions. The update rule is given by:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \Big[r + \gamma \max_{a'} Q(s', a') \Big]$$

where:

- α is the learning rate,
- γ is the discount factor,
- r is the reward received after the transition,
- s' is the next state.

Implementation Details

The implementation was carried out in the `qlearningAgents.py` file. A dictionary-like structure (`util.Counter`) was used to store Q-values, mapping state-action pairs to their learned values. Initially, unseen state-action pairs are treated as having a Q-value 0.0.

Key methods implemented include:

- `getQValue(state, action)`: Returns the current Q-value for a state-action pair (default 0.0).
- `computeValueFromQValues(state)`: Computes $\max_a Q(s, a)$ over legal actions.
- `computeActionFromQValues(state)`: Chooses the action with the highest Q-value, breaking ties randomly.
- `getAction(state)`: Implements ϵ-greedy exploration. With probability ϵ, selects a random action (exploration); otherwise, selects the best learned action (exploitation).
- `update(state, action, nextState, reward)`: Performs the Q-learning update based on the observed transition.

Exploration vs. Exploitation

A critical aspect of Q-learning is balancing exploration and exploitation. The ϵ-greedy strategy ensures that the agent does not get stuck in a potentially suboptimal policy. With probability ϵ, the agent chooses a random legal action; otherwise, it selects the best-known action. This strategy allows the agent to continually explore new actions while still exploiting its knowledge.

1.1 Question 6: Q-Learning

Question 6 required implementing the core Q-learning update rule in the `update` method. The following steps occur:

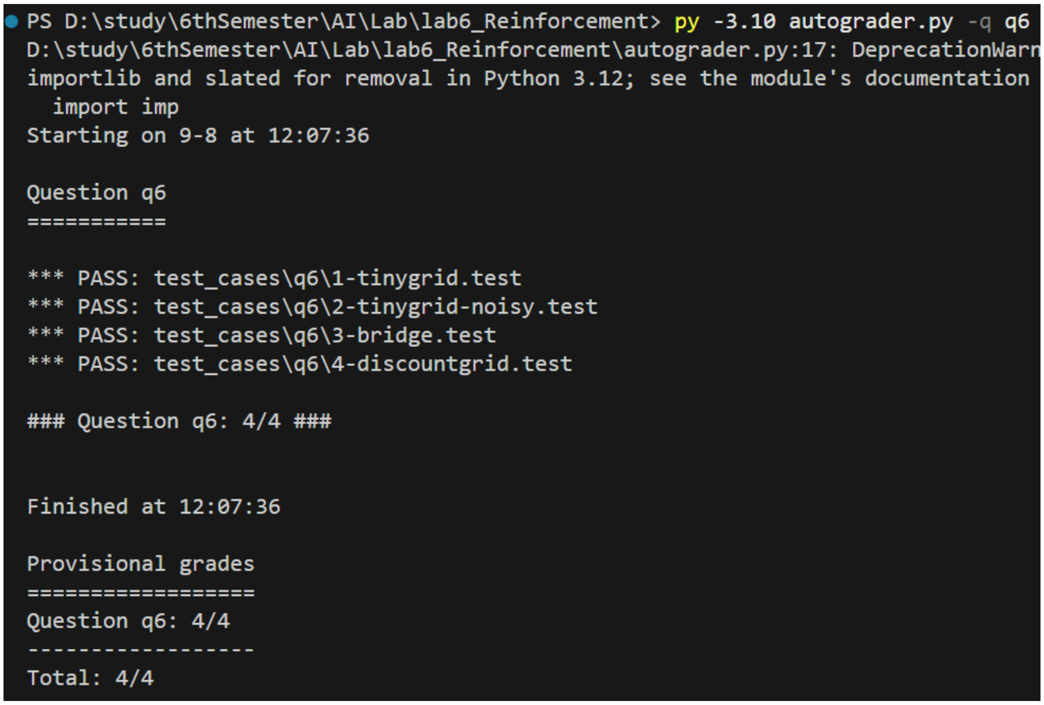
1. The agent observes a transition (s, a, s', r).
2. It computes the sample estimate:

$$\text{sample} = r + \gamma \max_{a'} Q(s', a')$$

3. It retrieves the old Q-value, $Q(s, a)$.
4. The Q-value is updated as:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot \text{sample}$$

This ensures that the Q-values are gradually adjusted toward the true expected returns, while weighting newer experiences by α.



```
PS D:\study\6thSemester\AI\Lab\lab6_Reinforcement> py -3.10 autograder.py -q q6
D:\study\6thSemester\AI\Lab\lab6_Reinforcement\autograder.py:17: DeprecationWarn
importlib and slated for removal in Python 3.12; see the module's documentation
  import imp
Starting on 9-8 at 12:07:36

Question q6
===========

*** PASS: test_cases\q6\1-tinygrid.test
*** PASS: test_cases\q6\2-tinygrid-noisy.test
*** PASS: test_cases\q6\3-bridge.test
*** PASS: test_cases\q6\4-discountgrid.test

### Question q6: 4/4 ###


Finished at 12:07:36

Provisional grades
==================
Question q6: 4/4
------------------
Total: 4/4
```

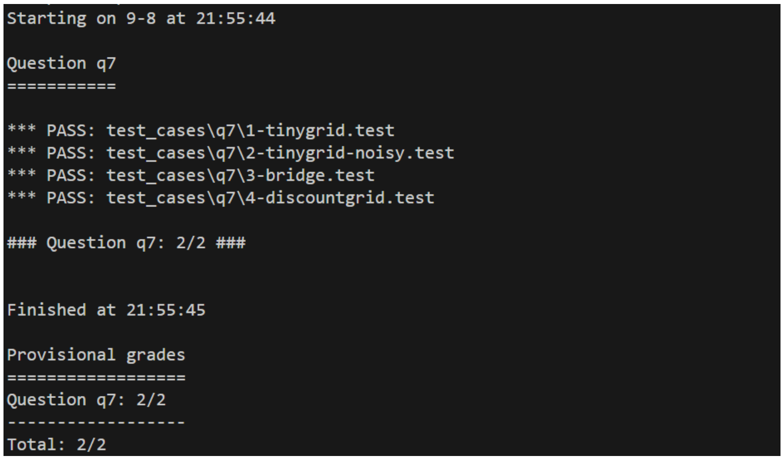
Figure 1: Grades of Q6 task solution

Finally, through repeated interaction with the Pacman environment, the Q-learning agent incrementally improves its policy. Unlike MDP-based approaches such as Value Iteration, Q-learning does not require full knowledge of transitions or rewards. Instead, it learns purely from experience. Over time, as Q-values converge, the agent is able to play optimally.

1.2 Question 7: Epsilon Greedy

The objective is to implement an ϵ-greedy action selection in `getAction` so the agent explores with probability ϵ and exploits (follows its current best

Q-values) otherwise.



```
Starting on 9-8 at 21:55:44

Question q7
===========

*** PASS: test_cases\q7\1-tinygrid.test
*** PASS: test_cases\q7\2-tinygrid-noisy.test
*** PASS: test_cases\q7\3-bridge.test
*** PASS: test_cases\q7\4-discountgrid.test

### Question q7: 2/2 ###


Finished at 21:55:45

Provisional grades
==================
Question q7: 2/2
------------------
Total: 2/2
```

Figure 2: Results for Q7

Implementation summary.

- Use `util.flipCoin(self.epsilon)` to decide whether to explore.
- If exploring, select any legal action uniformly at random: `random.choice(legalActions)`.
- If exploiting, select an action that maximizes $Q(s, a)$. If multiple actions tie for the maximum, break ties randomly.

Rationale: Exploration avoids premature convergence to a suboptimal policy; random tie-breaking prevents deterministic bias when many actions have identical Q-values (e.g., initially all are zero).

Equation: No change to the Q-update rule; only action selection changes:

$$\text{action} = \begin{cases} \text{random legal action} & \text{with probability } \epsilon, \\ \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon. \end{cases}$$

Testing & expected behaviour:

- Run: `python gridworld.py -a q -k 100 -e 0.3` (example).
- With larger ϵ (e.g., 0.9) the agent explores more and learning is slower; with small ϵ (e.g., 0.1) it exploits more and converges faster but risks local optima.

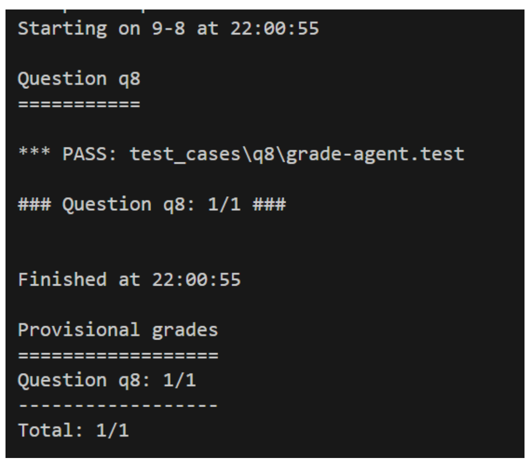
1.3 Question 8: Bridge Crossing Revisited

We need to determine whether there exists (ϵ, learning rate) pairs that make it very likely ($> 99\%$) for a Q-learner to discover the optimal policy on the noiseless `BridgeGrid` within 50 episodes.

Approach:

- Empirically testing combinations of ϵ (exploration) and α (learning rate) on the noiseless environment.
- Considering that results must be invariant to arbitrary tie-breaking and possible symmetries of the grid.

Conclusion : In practice, it is `'NOT POSSIBLE'` to guarantee $> 99\%$ success in only 50 episodes in a way that is independent of tie-breaking and initial randomness. Short episode budgets and stochastic tie-breaking make such guarantees unreliable; therefore, we return `'NOT POSSIBLE'` as a solution to the task.



```
Starting on 9-8 at 22:00:55

Question q8
===========

*** PASS: test_cases\q8\grade-agent.test

### Question q8: 1/1 ###


Finished at 22:00:55

Provisional grades
==================
Question q8: 1/1
------------------
Total: 1/1
```

Figure 3: Results for Q8

1.4 Question 9 : Q-Learning and Pacman

Train Pacman using Q-learning: a training phase with exploration and learning, followed by a testing phase where $\epsilon = 0$ and $\alpha = 0$ so the learned policy is exploited.

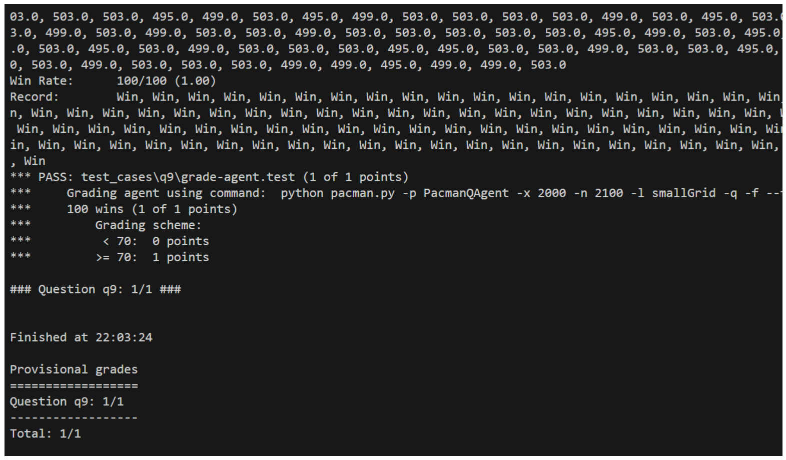


Figure 4: Results for Q9

Implementation summary.

- Use `PacmanQAgent` (inherits `QLearningAgent`) with recommended defaults: $\epsilon = 0.05, \alpha = 0.2, \gamma = 0.8$.
- Ensure `getAction` correctly accounts for unseen actions (unseen Q-values default to 0.0) so that unseen actions can be chosen if they are optimal relative to known negative Qs.
- Training is performed silently for many episodes (e.g., 2000), then testing runs with learning and exploration disabled.

Experiment commands:

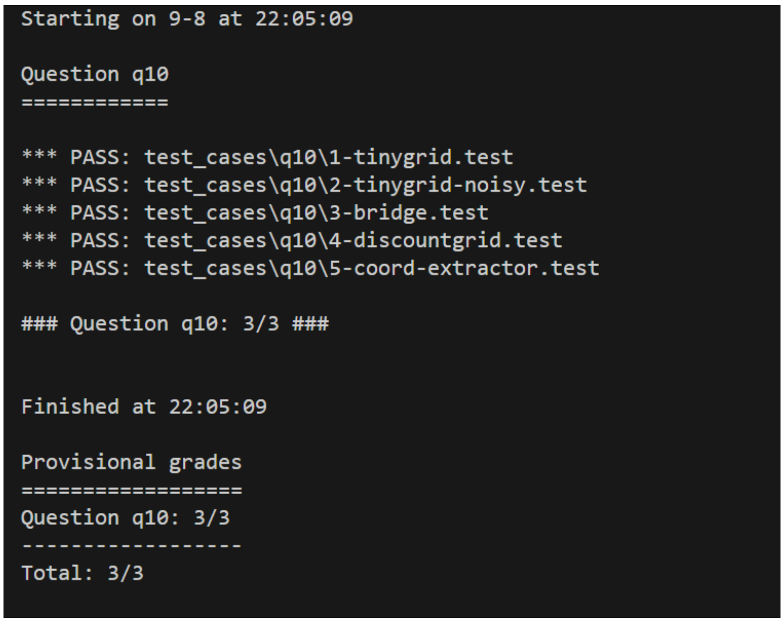
```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

Expected outcomes:

- After sufficient training (typically 1k–1.4k games), average training reward per 100-game window should become positive and improve.
- During testing, agent should win reliably (target: $\geq 80\%$ in autograder; authors report $\geq 90\%$ on small grids).
- On larger maps (e.g., `mediumGrid`), tabular Q-learning often fails because it cannot generalize: each board configuration is a separate state.

1.5 Question 10: Approximate Q-Learning

We implement an `ApproximateQAgent` that uses feature vectors and weights so that $Q(s,a) = \mathbf{w} \cdot \mathbf{f}(s,a)$and we also need update weights rather than a Q-table.



```
Starting on 9-8 at 22:05:09

Question q10
============

*** PASS: test_cases\q10\1-tinygrid.test
*** PASS: test_cases\q10\2-tinygrid-noisy.test
*** PASS: test_cases\q10\3-bridge.test
*** PASS: test_cases\q10\4-discountgrid.test
*** PASS: test_cases\q10\5-coord-extractor.test

### Question q10: 3/3 ###


Finished at 22:05:09

Provisional grades
==================
Question q10: 3/3
------------------
Total: 3/3
```

Figure 5: Results for Q10

Formulation:

$$Q(s,a) = \sum_i w_i f_i(s,a)$$

Given a transition (s, a, s', r), compute the temporal-difference error:

$$\delta = \left(r + \gamma \max_{a'} Q(s', a')\right) - Q(s,a)$$

Update weights:

$$w_i \leftarrow w_i + \alpha \cdot \delta \cdot f_i(s,a)$$

Implementation summary:

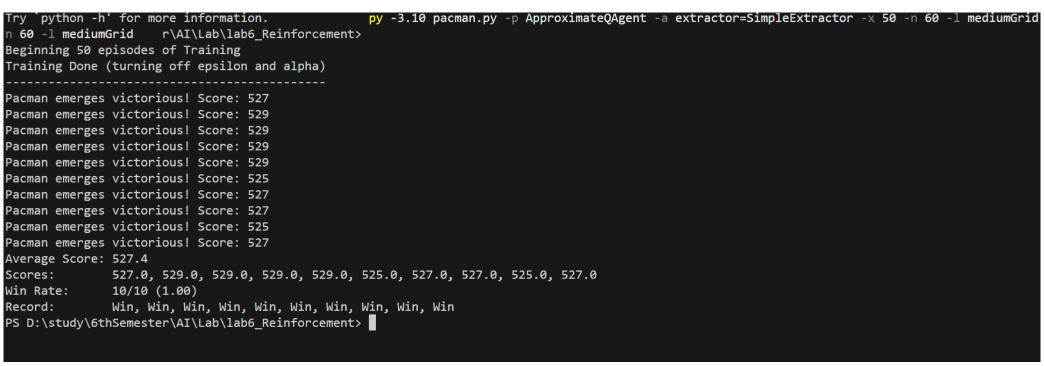
- Use provided feature extractors (feature vectors are `util.Counter` objects).

- `getQValue` should compute the dot product between weights and features.
- `update` should compute the TD error δ and update each weight accordingly.
- With `IdentityExtractor`, approximate Q-learning reduces to tabular Q-learning (one feature per state-action). With richer extractors (e.g., `SimpleExtractor`), generalization lets the agent learn effective policies on much larger maps with few training episodes.

Experiment commands:

```
# Identity extractor (sanity check)
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

# A practical feature set
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l
```



Expected outcomes:

- With the identity extractor, behavior should match tabular Q-learning.
- With informative features, the ApproximateQAgent should learn much faster and generalize to larger layouts, winning reliably after relatively few training games.

Overall Remarks

- Q-learning updates and ϵ-greedy action selection are the heart of the tabular agent.
- Approximate Q-learning replaces the Q-table with a parameterized function and is essential to scale RL to large state spaces.
- Empirical testing (multiple seeds, enough episodes) is crucial to validate claims about probability thresholds (e.g., the 99% requirement in Question 8).