Fall 2025 CS5368 Intelligent Systems Project 3 Report

First Name	Scott
Last Name	Weeden
Student ID	**R12041454**
Due date	Discretion of Professor (“I” incomplete makeup)
Grade	_____ / 10

Project 3: Reinforcement Learning

1. What is your implementation strategy for Question 1 (Q-learning)? Explain.

The implementation of the Q-learning agent follows a modular design where high-level methods delegate to specific computation functions. The core strategy includes:

- **Data Storage:** Q-values are stored in a `util.Counter()` named `self.qValues`. This choice is advantageous because it automatically returns a default value of 0.0 for any unseen state-action pair.
- **Value Computation:** `getValue(state)` calls `computeValueFromQValues(state)`, which calculates $V(s) = \max_a Q(s, a)$. If no legal actions exist (terminal state), it returns 0.0.
- **Policy Extraction:** `getPolicy(state)` calls `computeActionFromQValues(state)`. This finds all actions with the maximum Q-value and breaks ties randomly using `random.choice()`.
- **Update Rule:** The `update` method implements the temporal difference formula:

$$Q(s, a) = Q(s, a) + \alpha[R + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

2. What is your implementation strategy for Question 2 (Epsilon Greedy)? Explain.

The Epsilon Greedy strategy manages the exploration-exploitation trade-off within the `getAction` method:

- **Decision Logic:** I used `util.flipCoin(self.epsilon)` to decide between exploration and exploitation.
- **Exploration:** With probability ϵ, the agent selects a random action from the list of legal actions. This allows the agent to discover new states.
- **Exploitation:** With probability $1 - \epsilon$, the agent selects the best action according to its current Q-values via `self.getPolicy(state)`.

3. What happens when you run `python crawler.py`?

When running the crawler robot, the agent starts in an initial learning phase where movements appear chaotic during exploration. Over time, the following occurs:

- **Learning Process:** The robot uses Q-learning to associate arm/hand positions with forward movement.
- **Parameter Impact:** With a **higher learning rate** ($\alpha \approx 0.8$), the robot typically learns to "scoot" using long pulls by step 800. With a **lower learning rate** ($\alpha \approx 0.3$), it learns more slowly (around step 1,200).
- **Verification:** These results are based on **empirical observations** during simulation; exact step counts for convergence may vary slightly due to the stochastic nature of exploration.

4. Run `python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10` using your code and report on what is happening? Is Pacman failing or winning? What is your "Average Score" and your "Win rate"? Justify.

Upon running the tabular Q-learning agent with limited training, the results are as follows:

- **Report:** Pacman is consistently **failing**. During the 10 episodes, the agent died in every single attempt.
- **Metrics:**
  - **Average Score:** -509.3

– **Win Rate:** 0/10 (0.00)

- **Justification:** The failure of the `PacmanQAgent` is due to the **state space size** versus the **limited training episodes**. In tabular Q-learning, every state-action pair must be visited and updated individually. With only 10 training episodes, the agent has not explored enough of the `smallGrid` to learn a viable policy, resulting in negative scores (ranging from -504 to -521) and a 0% win rate.

5. What is your implementation strategy for Question 4 (Approximate Q-Learning)? Explain.

The strategy involves representing $Q(s, a)$ as a linear function approximation:

- **Feature-Based Q-Values:** $Q(s, a) = \sum_{i=1}^{n} w_i \cdot f_i(s, a)$. The `getQValue` method performs a dot product between the weights and the features extracted by the `featExtractor`.
- **Weight Updates:** Weights are updated using the TD error:

$$w_i \leftarrow w_i + \alpha \cdot [R + \gamma V(s') - Q(s, a)] \cdot f_i(s, a)$$

- **Generalization:** This approach enables the agent to perform well on unseen states by leveraging shared features, which is essential for scaling to large grid layouts.

Comparison: Tabular vs. Approximate Q-Learning

To further justify the implementation strategies, a second command was run using the `ApproximateQAgent` with the `SimpleExtractor`: `python3 pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -n 20 -l smallGrid -x 10`

Results Comparison

Metric	PacmanQAgent (Tabular)	ApproximateQAgent
Win Rate	0% (0/10)	80% (8/10)
Average Score	-509.3	301.6
Score Range	-504 to -521	-504 to 507

Justification for Performance Differences

- **Generalization vs. Memorization:** The `PacmanQAgent` relies on memorizing $Q(s, a)$ for every specific configuration of food and ghosts. Conversely, the `ApproximateQAgent` learns weights for features (e.g., "distance to closest food"). This allows the agent to make intelligent decisions in states it has never seen before.
- **Sample Efficiency:** Tabular Q-learning is highly inefficient in large state spaces. Feature-based learning updates weights that affect many states simultaneously, making it far more sample-efficient. A single update to the "ghost proximity" weight improves the policy for all states where a ghost is nearby.
- **Domain Knowledge:** The `SimpleExtractor` encodes human-like domain knowledge (food proximity is good, ghost proximity is bad). The agent leverages this structure to find an optimal policy in significantly fewer episodes than the tabular approach requires.