Artificial intelligence

Lab - 3

May 1^{st} 2024

1 Introduction to OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It provides a diverse suite of environments ranging from simple games to complex physics simulations, allowing researchers and developers to test and benchmark their algorithms in various scenarios.

1.1 Basic Functionality

The core functionality of OpenAI Gym revolves around the concept of environments, which are defined tasks or problems that an agent interacts with. Each environment in Gym is represented by a Python class with methods to interact with it.

1.2 Methods

- **reset():** This method resets the environment to its initial state and returns an initial observation. It is typically called at the beginning of an episode or whenever the agent needs to start a new interaction with the environment. The return value is the initial observation that the agent uses to start its decision-making process.

  Example:

```python
# Python code
import gym

    # Create an environment
    env = gym.make('CartPole-v1')

    # Reset the environment
    initial_observation = env.reset()
```

  Parameters: None.

  Returns:

  - **observation:** The initial observation representing the state of the environment.

- **step():** This method takes an action as input and performs one timestep of the environment's dynamics. It returns four values: the next observation, the reward obtained from taking the action, a boolean indicating whether the episode has terminated, and additional information useful for debugging or analysis.

  Example:

```
# Take an action in the environment
action = 0  # Example action
observation, reward, done, info = env.step(action)
```

  Parameters:

  - **action:** The action to take in the environment

  Returns:

  - **observation:** The next observation after taking the action.
  - **reward:** The reward obtained from taking the action.
  - **done:** A boolean indicating whether the episode has terminated.
  - **info:** Additional information for debugging or analysis.

- **render():** The render() method renders the current state of the environment for visualization. Rendering may be disabled or implemented differently depending on the environment.

  Example:

```
env.render()
```

- **close() :**The close() method frees up resources used by the environment. It's good practice to call close() when done using the environment to clean up resources.

  Example:

```
env.close()
```

- **action_space:** Attribute representing the action space of the environment.

  Example:

```
action_space = env.action_space
```

  Attributes:

  - **action_space.n:** Number of discrete actions for discrete action spaces.
  - **action_space.shape:** Shape of the action space for continuous action spaces.

- **observation_space:** Attribute representing the observation space of the environment.

  Example:

```
observation_space = env.observation_space
```

  Attributes:

  - **observation_space.shape:** Shape of the observation space.
  - **observation_space.high:** Highest value for each dimension in the observation space for continuous spaces.
  - **observation_space.low:** Lowest value for each dimension in the observation space for continuous spaces.

These methods are fundamental for interacting with OpenAI Gym environments. By repeatedly calling **reset()** and **step()**, an agent can interact with the environment, observe its state, take actions, and receive feedback in the form of rewards. This interaction forms the basis for training reinforcement learning agents using Gym environments.

2 Exploring MarkovDecision Processes (MDP)

2.1 FrozenLake-v1

FrozenLake-v1 is a grid-world environment in OpenAI Gym where an agent navigates icy terrain to reach a goal while avoiding holes. The grid comprises safe (frozen) tiles and hazardous (hole) tiles. The agent can move in four directions: up, down, left, and right. Upon reaching the goal, the agent receives a reward of +1; falling into a hole yields a reward of 0. The environment introduces stochasticity, simulating the slippery nature of the frozen lake, where actions may not always result in the intended movement. Through reinforcement learning, agents aim to learn optimal policies to traverse the grid safely and reach the goal.

Frozen lake involves crossing a frozen lake from Start(S) to Goal(G) without falling into any Holes(H) by walking over the Frozen(F) lake. The agent may not always move in the intended direction due to the slippery nature of the frozen lake.

- **States:**The environment consists of a grid of tiles, each representing a state.
  - **Frozen tiles (F):** Safe tiles where the agent can move.
  - **Hole tiles (H):** Tiles where the agent falls into a hole and fails.
  - **Start state (S):** The initial state where the agent begins.
  - **Goal state (G):** The state the agent aims to reach to succeed.
- **Actions:** At each state, the agent can take one of four actions. These actions determine the direction in which the agent moves on the grid.
  - Move Up

  - Move Down
  - Move Left
  - Move Right

- **Rewards:** The primary objective of the agent is to maximize the cumulative reward by reaching the goal state while avoiding falling into holes. The rewards in FrozenLake-v1 are structured as follows:
  - When the agent reaches the goal state (G), it receives a reward of +1.
  - When the agent falls into a hole (H), it receives a reward of 0 (failure).
  - All other transitions yield a reward of 0.

- **Transitions:** Transitions in FrozenLake-v1 are deterministic within the grid. If the agent takes an action, it transitions to the corresponding adjacent tile unless it hits a wall, in which case it stays in the same tile. For example, if the agent is at position **(i, j)** and takes the action **"Move Right,"** it transitions to position **(i, j+1)** if possible. However, there is a stochastic element introduced in the environment. Due to the "slippery" nature of the frozen lake, the agent's intended action may not always be executed as intended. There is a small probability (0.33) that the agent will slip and move in a direction different from the intended action.

2.2 Markov Decision Process (MDP)

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making in situations where outcomes are influenced by both stochastic (random) elements and the actions taken by a decision-maker. It's widely used in reinforcement learning and operations research to solve problems involving sequential decision-making under uncertainty.

2.2.1 Markov Decision Property (MDP)

The Markov Decision Property (MDP) is a fundamental concept in reinforcement learning and stochastic processes. It states that the future state of a system, given the current state and action, depends only on the current state and action and is independent of the past states and actions.

In the context of Markov Decision Processes (MDPs), this property implies that the transition probabilities and rewards associated with state-action pairs fully capture the dynamics of the environment. Mathematically, it can be expressed as:

$$\mathbf{P}(s_{t+1}, r_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0, a_0) = \mathbf{P}(s_{t+1}, r_{t+1}|s_t, a_t)$$

Where:

- $\mathbf{s_t}$ represents the current state.
- $\mathbf{a_t}$ represents the action taken at time step t.
- $\mathbf{s_{t+1}}$ represents the next state.
- $\mathbf{r_{t+1}}$ represents the reward obtained at time step t+1.

- $\mathbf{P}(s_{t+1}, r_{t+1} s_t, a_t)$ denotes the transition probability distribution.

The Markov Decision Property simplifies the learning problem by allowing us to focus solely on the current state and action, making planning and decision-making more tractable.

2.2.2 Relevance to MDP:

- **Simplification:** MDPs are used to model decision-making in environments where outcomes are partly random. The Markov Property vastly simplifies the calculations involved because you don't need to track an endless history of states to make decisions.
- **Foundation:** The Markov Property is the underlying principle upon which MDPs are built. It allows for the modeling of these decision processes in a tractable way.

2.2.3 FrozenLake-v1 Environment as an MDP

FrozenLake-v1 can be modeled as an MDP, where states represent grid tiles, actions determine movements, transitions dictate movement outcomes (with stochasticity introduced by the slippery nature), rewards provide feedback on the agent's performance, and the policy guides the agent's decision-making process. By understanding FrozenLake-v1 as an MDP, we can apply reinforcement learning algorithms to learn optimal policies for navigating the environment and achieving the goal.

1. **States (S):**
   - Each tile on the grid represents a state in the MDP.
   - The environment consists of a grid of tiles, including frozen tiles (F), hole tiles (H), a start state (S), and a goal state (G).
   - The state encapsulates all relevant information about the agent's current position on the grid.
2. **Actions (A):**
   - At each state, the agent can take one of four actions: move up, move down, move left, or move right.
   - These actions represent the possible decisions or movements the agent can make in the environment.
3. **Transitions (P):**
   - Transitions in FrozenLake-v1 are deterministic within the grid, meaning that the outcome of an action is known with certainty.
   - If the agent takes an action, it transitions to the corresponding adjacent tile, unless it hits a wall, in which case it stays in the same tile.
   - However, there's a stochastic element introduced due to the "slippery" nature of the frozen lake, where there's a small probability (0.33) that the agent will slip and move in a direction different from the intended action.

4. **Rewards (R):**
   - The agent receives rewards based on its actions.
   - Reaching the goal state yields a reward of +1.
   - Falling into a hole results in a reward of 0 (failure).
   - All other transitions yield a reward of 0.
5. **Policy ():**
   - A policy specifies the agent's behavior, i.e., which action to take in each state.
   - The goal of the agent is to learn an optimal policy that maximizes the expected cumulative reward over time.

3 Value iteration

Value iteration is an algorithm used to compute the optimal value function for a Markov Decision Process (MDP). The optimal value function represents the expected cumulative reward that an agent can achieve from each state, following the optimal policy.

3.1 Basic of Value iteration

The algorithm iteratively updates the value function until it converges to the optimal values. Here's how the value iteration algorithm works:

- **Initialization:** Initialize the value function V(s) arbitrarily for all states s.
- **Iteration:** For each state ss, update the value function using the Bellman optimality equation:

  $\mathbf{V}(\mathbf{s}) = \max_{\mathbf{a}} \sum_{\mathbf{s}',\mathbf{r}} \mathbf{P}(\mathbf{s}', \mathbf{r}|\mathbf{s}, \mathbf{a})[\mathbf{r} + \gamma \mathbf{V}(\mathbf{s}')]$

  where:
  - a represents the actions available in state s.
  - $P(s', r|s, a)$ is the transition probability and reward function for transitioning from state s to state s with reward r after taking action a.
  - γ is the discount factor, which determines the importance of future rewards relative to immediate rewards.
- **Termination:** Repeat the iteration process until the change in the value function between consecutive iterations falls below a specified threshold, indicating convergence.
- **Policy extraction:** Once the value function has converged, extract the optimal policy by selecting the action that maximizes the value function in each state:

$\pi^*(\mathbf{s}) = \arg\max_{\mathbf{a}} \sum_{\mathbf{s}',\mathbf{r}} \mathbf{P}(\mathbf{s}', \mathbf{r} \mid \mathbf{s}, \mathbf{a})[\mathbf{r} + \gamma \mathbf{V}^*(\mathbf{s}')]$

where $\pi^*(s)$ is the optimal policy and $V^*(s)$ is the optimal value function

Value iteration converges to the optimal value function and policy for finite-state and finite-action MDPs. It is a foundational algorithm in reinforcement learning and dynamic programming, providing a principled approach to solving sequential decision-making problems under uncertainty. The graph illustrates convergence of state values in "FrozenLake-v1" using value iteration. Each line represents the estimated value of each state over iterations, demonstrating how the algorithm refines state evaluations, stabilizing towards optimal values for decision-making in a stochastic, slippery environment.

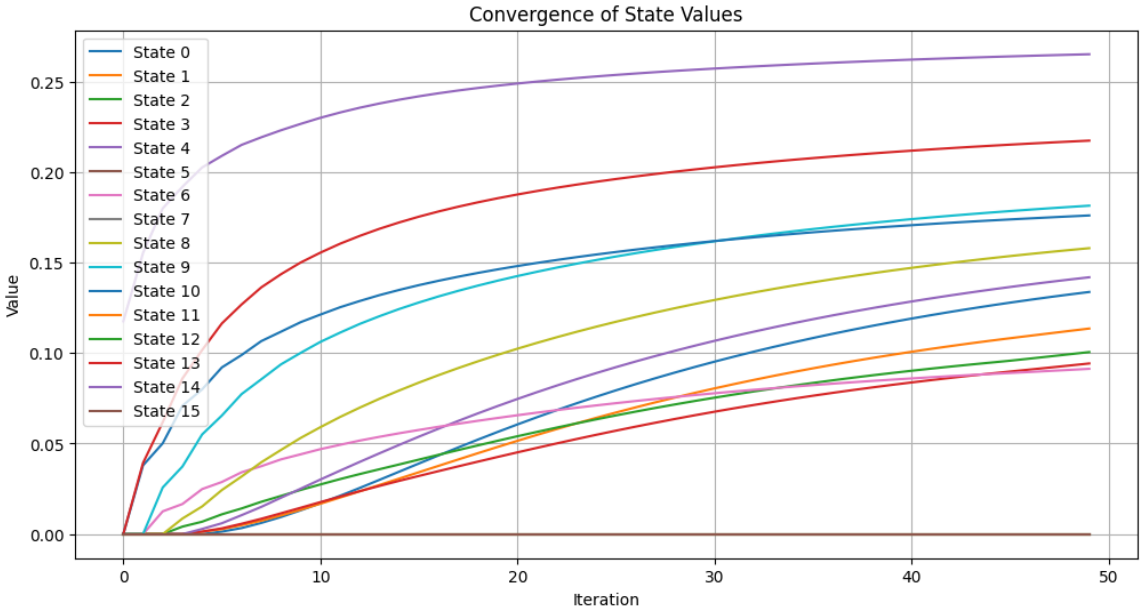

Figure 1: Convergence of State Values

The policy derived from Value Iteration generally performs well in the "FrozenLake-v1" environment. By optimizing the expected reward from each state, the policy strategically navigates the lake's tiles, aiming to reach the goal while minimizing the risk of falling into holes. The success rate of the policy, as tested in the environment, reflects its effectiveness, accounting for the inherent randomness and challenges posed by the slippery nature of the lake. The iterative refinement of state values helps in making informed decisions that improve the overall success rate over time.

3.2 Performance of the Derived Policy

The policy derived from Value Iteration typically performs very well in the "FrozenLake-v1" environment. Since it computes the maximum expected utility from each state, the derived policy is optimal with respect to the defined reward structure and transition probabilities.

- **Optimal Decision Making:**The primary strength of Value Iteration lies in its ability to compute the optimal policy by leveraging the maximum expected utility from each state. Given that the

utility values converge to their true values, the resulting policy ensures that the decisions made at every state are optimal with respect to the environment's dynamics and the reward structure. This is critical in environments like "FrozenLake-v1," where every move can potentially lead to failure (falling into a hole) or success (reaching the goal).

- **Robustness to Environmental Stochasticity:**"FrozenLake-v1" is known for its stochastic nature—actions taken by the agent do not always result in the intended outcomes due to the slippery ice. The Value Iteration-derived policy robustly accounts for this randomness by calculating utilities that incorporate the probability distributions of all possible state transitions. This allows the agent to make decisions that optimize the expected outcome, factoring in the likelihood of slipping or moving as intended.

- **Efficiency and Convergence:**Value Iteration is computationally efficient in environments with a manageable number of states and actions because it iteratively refines the utilities of states until the changes fall below a pre-set threshold, signaling convergence. This efficiency is particularly advantageous in deterministic or small stochastic environments where the state and action spaces are not overly large.

- **Evaluation and Adaptability:**The effectiveness of the policy can be empirically evaluated through simulations where the agent attempts to navigate from the start to the goal across numerous trials. In "FrozenLake-v1," an optimal policy will consistently manage to reach the goal more often than not, demonstrating a high success rate. Moreover, the adaptability of Value Iteration allows for adjustments in the model's parameters (like the discount factor) to experiment with short-term versus long-term gains, offering insights into the behavior of the policy under different strategic priorities.

- **Limitations:**Despite these strengths, the performance of the Value Iteration policy can sometimes be limited by the granularity of the state space and the accuracy of the transition probabilities provided by the environment's model. Inaccuracies in the model or an overly simplified representation of the state space can lead to suboptimal policies.

The effectiveness of the policy can be evaluated by simulating the environment with the derived policy and measuring the success rate of reaching the goal without falling into any holes. This would typically show high performance unless the environment setup or the reward structure inherently limits the achievable success.

4 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm used to learn the value of an action in a particular state. It doesn't require a model of the environment and can handle problems with stochastic transitions and rewards without needing adaptations.

4.1 Basics of Q-Learning

Q-Learning works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following a fixed policy thereafter. Here's a step-by-step explanation of how Q-Learning works:

- **Initialize the Q-values (Q-Table):** Start with a table of Q-values for each state-action pair, initialized to zero or some small random numbers.
- **Policy Execution:** At each state, select an action using a policy derived from the Q-values (commonly an -greedy policy where is the probability of choosing a random action, and 1- is the probability of choosing the action with the highest Q-value).
- **Execution and Update:**
  - Execute the chosen action, and observe the reward and the next state.
  - Update the Q-value for the state-action pair based on the reward received and the maximum Q-value of the next state (Bellman equation):

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

    Where:
    - Q(s,a) is the current Q-value of state s and action a.
    - α is the learning rate ($0 < \alpha \leq 1$).
    - r is the reward received after executing action a in state s.
    - γ is the discount factor ($0 \leq \gamma < 1$); it represents the difference in importance between future rewards and immediate rewards.
    - $\max_{a'} Q(s', a')$ is the estimate of optimal future value.
- **Repeat:** Repeat this process for each episode or until the Q-values converge.

The plot displays the success rate of a Q-learning algorithm applied on Frozen Lake-V1 over 1000 episodes of training, using a learning rate (alpha) of 0.5 and a discount factor (gamma) of 0.99.

- **Interpretation:**
  - The **high alpha (0.5)** implies that the algorithm puts significant weight on recent information, which might contribute to the observed volatility as recent states and rewards can drastically influence the policy updates.
  - The **high gamma (0.99)** suggests that future rewards are nearly as important as immediate rewards, which encourages the agent to think long-term but may also result in instability in success rate if the environment has diverse or conflicting immediate and future rewards.

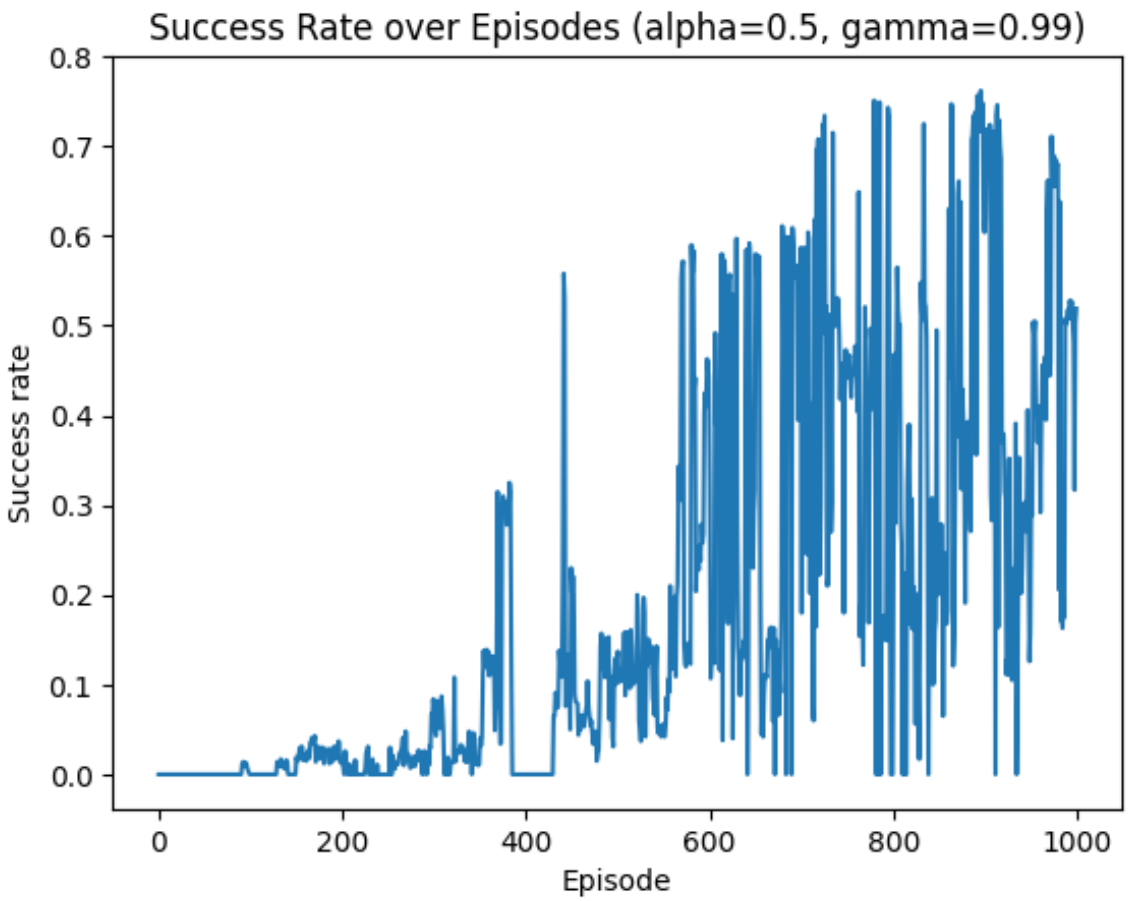

Figure 2: Q-Learning

4.2 Evaluation

```
Final success rate of Value Iteration: 76.0%
```

```
alpha = 0.5, gamma = 0.99: Final success rate of Q-Learning: 72.1%
```

Here, seems the Value Iteration got higher percentage of success rate than Q-Learning.

4.2.1 Value Iteration: Higher Success Rate

- **Optimality and Efficiency:**
  - Value Iteration achieves a higher success rate, indicating it has successfully found a policy closer to the optimal solution for the environment. This method systematically evaluates the best action from each state based on known dynamics (transition probabilities and rewards), which typically allows it to converge to the optimal policy.

  - The higher success rate of 76% suggests that Value Iteration efficiently utilized the model of the environment to compute the maximum expected utility for each state systematically, leading to more consistently successful decision-making.

- Model Dependency: The effectiveness of Value Iteration is contingent upon having accurate and complete knowledge of the environment's dynamics. This reliance on a model makes it highly effective in environments where such a model can be accurately defined and where the state and action spaces are sufficiently manageable.

4.2.2 Q-Learning: Lower Success Rate

- **Learning from Interactions:**
  - Q-learning's success rate of 56% indicates that, while it has learned to navigate the environment to a significant extent, it has not reached the level of performance of the Value Iteration policy. This discrepancy can arise from Q-learning's nature of learning solely from interactions with the environment, without prior knowledge of its dynamics.
  - This model-free approach means it discovers effective strategies through trial and error, which can be less efficient than the model-based strategies employed by Value Iteration.

- **Exploration and Exploitation Challenges:**
  - Q-learning involves a balance between exploring new actions and exploiting known rewarding actions. If not managed correctly (e.g., setting of the exploration rate, epsilon in epsilon-greedy strategy), Q-learning might not explore the environment sufficiently or might stick too quickly to suboptimal policies.
  - The convergence to an optimal policy in Q-learning is also highly dependent on parameters like the learning rate (alpha) and the discount factor (gamma), along with the number of episodes of learning allowed. Inadequate tuning or insufficient learning episodes can lead to poorer performance.

4.2.3 Comparaison

- **Robustness vs. Flexibility:** Value Iteration's robustness is evident in environments with well-understood dynamics. In contrast, Q-learning offers flexibility in unknown or complex environments but may require more iterations and careful parameter tuning to achieve optimal results.

- **Speed of Convergence:** Value Iteration typically converges faster to the optimal policy since it uses a complete model to update its estimates. Q-learning, being model-free, usually takes longer and may require more episodes to converge, reflecting in a lower success rate in environments like FrozenLake if not adequately trained.

- **Applicability:** Value Iteration is preferable in controlled settings or when the environmental model is accurate. Meanwhile, Q-learning is better suited for environments where the model is unknown or hard to estimate accurately.

4.2.4 Conclusion

The performance comparison highlights that while Value Iteration can leverage complete environmental models to quickly achieve high success rates, Q-learning's model-free approach provides valuable flexibility and adaptability, albeit often at the cost of efficiency and higher performance variability. Each method has its strengths and is best suited to different types of problems or stages within a broader machine learning strategy.