Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Reinforcement Learning AY: 2023 - 24

Experiment 8

(SARSA Algorithm)

Name: Kresha Shah

SAP ID: 60009220080

AIM:

To implement the SARSA algorithm in the context of the Frozen Lake environment

THEORY:

Temporal Difference Learning (TD Learning)

One of the problems with the environment is that rewards usually are not immediately observable. For example, in tic-tac-toe or others, we only know the reward(s) on the final move (terminal state). All other moves will have 0 immediate rewards.

TD learning is an unsupervised technique to predict a variable's expected value in a sequence of states. TD uses a mathematical trick to replace complex reasoning about the future with a simple learning procedure that can produce the same results. Instead of calculating the total future reward, TD tries to predict the combination of immediate reward and its own reward prediction at the next moment in time.

SARSA

SARSA is a Temporal Difference (TD) method, which combines both Monte Carlo and dynamic programming methods. The update equation has the similar form of Monte Carlo's online update equation, except that SARSA uses rt + γQ (st+1, at+1) to replace the actual return Gt from the data. N(s, a) is also replaced by a parameter α.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

In Monte Carlo, we need to wait for the episode to finish before we can update the Q value. The advantage of TD methods is that they can update the estimate of Q immediately when we move one step and get a state-action pair (st, at, rt, st+1, at+1).

SARSA technique is an On Policy and uses the action performed by the current policy to learn the Q-value. The update equation for SARSA depends on the current state, current action, reward obtained, next state and next action. This observation lead to the naming of the learning technique as SARSA stands for State Action Reward State Action which symbolizes the tuple


Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)


(s, a, r, s', a').

On-Policy learning:

On-Policy learning algorithms are the algorithms that evaluate and improve the same policy which is being used to select actions. That means we will try to evaluate and improve the same policy that the agent is already using for action selection. In short, [Target Policy(the policy that the agent is trying to learn) == Behaviour Policy(the policy that is being used by an agent for action select)].

ALGORITHM:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

LAB ASSIGNMENT TO DO:

1. Implement the SARSA algorithm in the context of the Frozen Lake environment.
2. Compare the initial and final Q-tables and also record the mean episode rewards
3. Implement different values of alpha (learning rate), gamma (discount factor), number of episodes and record your observations

```
[ ] import random
    import gym
    import numpy as np
    import matplotlib.pyplot as plt
```

```
#defines the layout of the FrozenLake environment
'''"S" represents the starting point.
"F" represents a frozen surface where the agent can walk.
"H" represents a hole where the agent will fall through.
"G" represents the goal.'''
de=["SFFF", "FHFH", "FFFH", "HFFG"]


env=gym.make('FrozenLake-v1',new_step_api=True)
print("Action space:", env.action_space) #0: Left 1: Down 2: Right 3: Up
print(env.observation_space)
```

```
Action space: Discrete(4)
Discrete(16)
```

```
state_size = 16 #number of states in the environment.
action_space = env.action_space.n #the number of actions
alpha = 0.1 #learning rate of the algorithm - how much the agent updates its estimates of state-action values based on new information
gamma = 1 #discount factor - the importance of future rewards in the agent's decision-making process - future rewards equally as important as immediate rewards
state_action_vals = np.random.randn(state_size, action_space) #Q-values for each state-action pair with random values drawn from a standard normal distribution - expected cumulative rewards
policy = np.zeros(state_size, dtype=int) #agent's behavior or strategy in terms of which action to take in each state
episodes = 10000
eps = 0.2 #probability of exploration
test_episodes = 500
test_every = 1000
test_episode = []
rewards = []
```

```
def select_action(state, eps):
  sample = np.random.uniform()  #random number is used to decide whether the agent should explore or exploit between 0 and 1
  if sample < eps:
    return env.action_space.sample() #the agent chooses to explore - returns a randomly selected action from the action space
  else:
    return state_action_vals[state].argmax() #exploit - find the index of the action with the maximum Q-value in the array
```

```
act_dict={0:'Left',1:'Down', 2:"Right", 3:"Up"}
for ep in range(episodes):
  state = env.reset()
  action = select_action(state, eps) #selects the initial action for the agent using the epsilon-greedy policy
  done = False  # the episode hasn't ended yet

  while not done:
    next_state, reward, done, __= env.step(action)[0],env.step(action)[1],env.step(action)[2],env.step(action)[3]
    next_action = select_action(state, eps)
    action_value = state_action_vals[state, action]
    next_action_value = state_action_vals[next_state, next_action]
    delta = reward + gamma * next_action_value - action_value #calculates the temporal difference error - difference between the current estimate of the Q-value and the updated estimate
    state_action_vals[state, action] += alpha * delta #updates the Q-value for the current state-action pair using the temporal difference error and the learning rate
    state, action = next_state, next_action
  if ep % test_every == 0: #checks if it's time to evaluate the agent's performance on the test episodes
    total_rewards = 0
    for _ in range(test_episodes):
      done = False
      state = env.reset()

      while not done:
        action = state_action_vals[state].argmax()
        state, reward, done, _ = env.step(action)[0],env.step(action)[1],env.step(action)[2],env.step(action)[3]
        total_rewards += reward
        print(f"State: {state} \t Action:{act_dict[action]}")
    rewards.append(total_rewards / episodes)
    test_episode.append(ep)
```

```
Streaming output truncated to the last 5000 lines.
State: 0          Action:Left
State: 0          Action:Left
State: 0          Action:Left
State: 4          Action:Left
State: 12         Action:Right
State: 0          Action:Left
State: 0          Action:Left
State: 0          Action:Left
State: 4          Action:Left
State: 4          Action:Right
State: 4          Action:Left
State: 4          Action:Right
State: 4          Action:Left
State: 9          Action:Right
State: 0          Action:Left
State: 0          Action:Left
State: 0          Action:Left
State: 0          Action:Left
State: 4          Action:Left
State: 4          Action:Left
State: 0          Action:Left
State: 0          Action:Left
State: 0          Action:Left
State: 4          Action:Left
State: 12         Action:Right
State: 4          Action:Left
State: 0          Action:Right
State: 5          Action:Left
State: 4          Action:Left
State: 8          Action:Right
State: 11         Action:Down
State: 4          Action:Left
State: 0          Action:Right
State: 0          Action:Left
State: 8          Action:Left
State: 4          Action:Left
```

```
fig, ax = plt.subplots()
ax.plot(test_episode, rewards)
ax.set_title('Episodes vs average rewards')
ax.set_xlabel('Episode')
_ = ax.set_ylabel('Average reward')
```