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: 2022 - 23

Experiment 7

SARSA Algorithm

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.

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

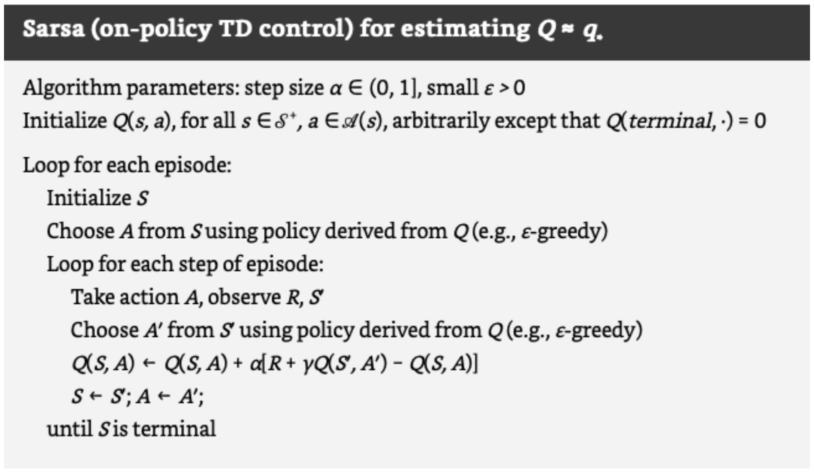
In Monte Carlo, we need to wait for the episode to finish before we can update the Q value. The advantage of TD methods is that they can update the estimate of Q immediately when we move one step and get a state-action pair (st, at, rt, st+1, at+1).

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 (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) == Behavior 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., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R, S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-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

Name : Sarvagya Singh
SAP ID : 60009200030
Batch : K1

Introduction

Reinforcement Learning: FrozenLake-v1 with SARSA

The environment

From the problem definition from gym.openai, the goal is to retrieve a frisbee from a frozen lake of size 4x4.

- **S: The starting point. This is where the agent will start traversing from.**
- **F: Frozen. The lake is frozen at this point. It's safe to step here.**
- **H: Hole. There is a hole here, no go zone. This is one of the terminal states.**
- **G: Goal. This is where the frisbee is, the zone where the agent should be going. This is the desired terminal state**

In [ ]:

```
import random

import gym
import numpy as np
import matplotlib.pyplot as plt



env = gym.make('FrozenLake-v1')
print("Action space:", env.action_space)
print("State space:", env.env.observation_space)
```

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

```
/usr/local/lib/python3.9/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `sh
ould_run_async` will not call `transform_cell` automatically in the future. Please pass t
he result to `transformed_cell` argument and any exception that happen during thetransfor
m in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
/usr/local/lib/python3.9/dist-packages/gym/core.py:317: DeprecationWarning: WARN: Initial
izing wrapper in old step API which returns one bool instead of two. It is recommended to
set `new_step_api=True` to use new step API. This will be the default behaviour in future
.
  deprecation(
/usr/local/lib/python3.9/dist-packages/gym/wrappers/step_api_compatibility.py:39: Depreca
tionWarning: WARN: Initializing environment in old step API which returns one bool instea
d of two. It is recommended to set `new_step_api=True` to use new step API. This will be
the default behaviour in future.
  deprecation(
```

The action spaces is a discrete space of size 4. That is, it has 4 distinct values (0, 1, 2 and 3). The numbers correspond to the following action: 0:left, 1:down, 2:right, 3:up. And there are 16 action spaces (0 to 15). Where 0 is the start and you move to the right, then down, then left to right etc.. to the terminal state G which is 15.

Environment dynamics

An agent on any point in the grid can take a move and slip and end up in a different grid than expected. This is the state transition probabilities $p(s^{'}, r|a, s)$ of the environment. A sample dynamics of the environment is shown below. We'll not use this dynamics while using out algorithm as it's cheating. Nevertheless, it will be

explained.

In [ ]:

```
env.env.P[0]
```

Out[ ]:

```
{0: [(0.3333333333333333, 0, 0.0, False),
  (0.3333333333333333, 0, 0.0, False),
  (0.3333333333333333, 4, 0.0, False)],
 1: [(0.3333333333333333, 0, 0.0, False),
  (0.3333333333333333, 4, 0.0, False),
  (0.3333333333333333, 1, 0.0, False)],
 2: [(0.3333333333333333, 4, 0.0, False),
  (0.3333333333333333, 1, 0.0, False),
  (0.3333333333333333, 0, 0.0, False)],
 3: [(0.3333333333333333, 1, 0.0, False),
  (0.3333333333333333, 0, 0.0, False),
  (0.3333333333333333, 0, 0.0, False)]}
```

The above is the state transition probabilities of state 0 to the neighbouring states 1 and 4(remember this is a grid). An agent moving down(1) has 0.33 chance each of ending up at states 0, 4 or 1. This is how they have emulated a slippery surface

Exploration-exploitation method

We'll be using ϵ-greey action selection of exploration and exploitation.

$$A_t = \begin{cases} argmaxQ(a), \\ \text{with probability } 1-\epsilon \\ \text{random action, with probability } \epsilon \end{cases}$$

Algorithm for SARSA:

```
Initialize state_action_vals (an array of shape states x actions (16 x 4))
Repeat for each episode:
    Select action A and state S (epsilon-greedy)
    Repeat for each step:
        Perform action A, observe reward R and next state S'
        Choose next action A' using the current policy Q (epsilon-greedy)
        delta = R + gamma * state_action_vals[S'][A'] - state_action_vals[S][A]
        state_action_vals[S][A] = state_action_vals[S][A] + alpha * delta
        S=S', A=A'
until S is terminal
```

Code

In [ ]:

```
state_size = 16
action_space = env.action_space.n
alpha = 0.1
gamma = 1
state_action_vals = np.random.randn(state_size, action_space)
policy = np.zeros(state_size, dtype=int)
episodes = 20000
eps = 0.2
test_episodes = 50
test_every = 1000
```

```
test_episode = []
rewards = []
```

In [ ]:

```
def select_action(state, eps):
    sample = np.random.uniform()
    if sample < eps:
        return env.action_space.sample()
    else:
        return state_action_vals[state].argmax()
```

In [ ]:

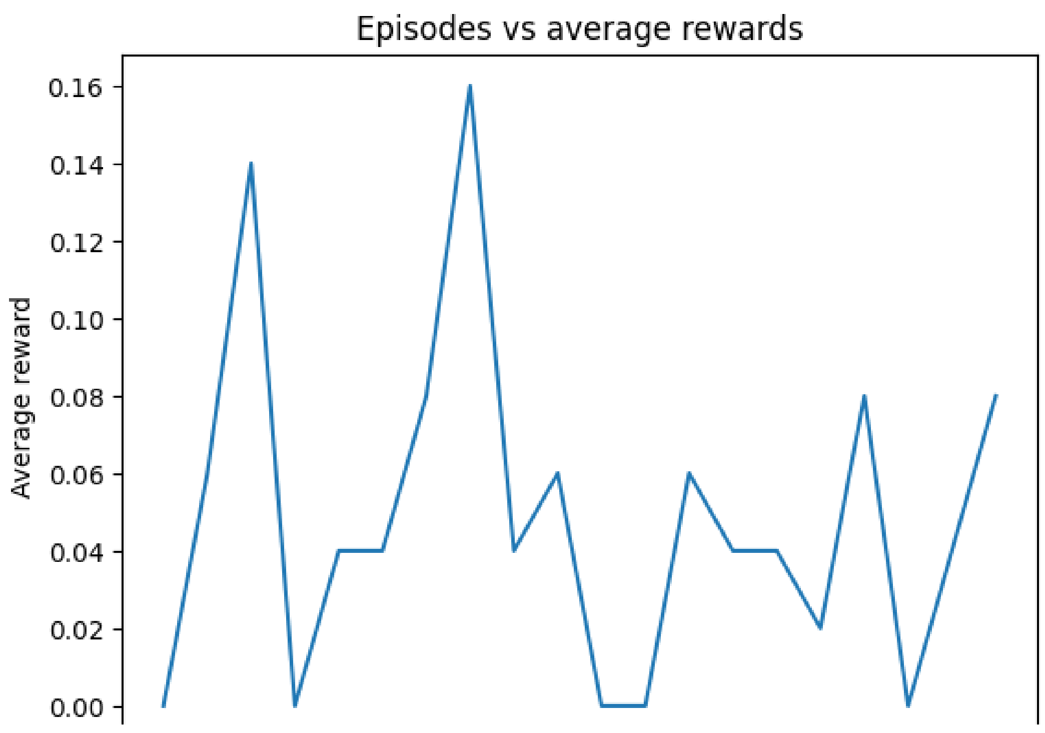
```
for ep in range(episodes):
    state = env.reset()
    action = select_action(state, eps)
    done = False
    while not done:
        next_state, reward, done, _ = env.step(action)
        next_action = select_action(state, eps)

        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
        state_action_vals[state, action] += alpha * delta
        state, action = next_state, next_action

    if ep % test_every == 0:
        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)
                total_rewards += reward
        rewards.append(total_rewards / test_episodes)
        test_episode.append(ep)
```

In [ ]:

```
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')
```



As you can see the policy has not converged and the average reward is nowhere near what is required. As the number of episodes $n \to \infty$, the policy will converge to an optimal policy according to the law of large numbers.

```
In [ ]:
```