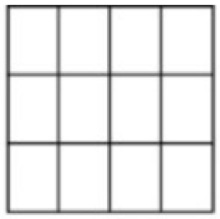
Reinforcement Learning

- **Passive Reinforcement Learning:** In case of passive RL, the agent's policy is fixed which means that it is told what to do. Agent executes a fixed policy and evaluates it.
  - **Direct Utility Estimation:** Suppose we have a 4 x 3 grid as the environment in which the agent can move either Left, Right, Up or Down (set of available actions). An example of a run and the total reward is 0.72.



4×3

$$(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (4,3)_{+1}$$

- **Active Reinforcement Learning:** In active RL, an agent needs to decide what to do as there's no fixed policy that it can act on. So, the goal of an active RL agent is to act and learn an optimal policy. Active Reinforcement Learning can be of the following categories.
  - **Q-Learning:** Q-learning is a TD learning method which does not require the agent to learn the transitional model, instead learns Q-value functions Q(s, a).

$$U(s) = \max_a Q(s, a)$$

SARSA Reinforcement Learning

- **State Action Reward State Action** (SARSA) algorithm is a slight variation of **Q-Learning** algorithm. For a learning agent in any RL algorithm, it's policy can be of two types.

  1. **On Policy:** In this case, the learning agent learns the value function according to the current action derived from the policy currently being used.

  2. **Off Policy:** In this case, the learning agent learns the value function according to the action derived from another policy.

- **Q-Learning** technique is an **Off Policy** technique and uses the greedy approach to learn the **Q-value**. **SARSA technique** is an **On Policy** and uses the action performed by the current policy to learn the **Q-value**. This difference is visible in the difference of the update statements for each technique

1. **Q-Learning:**

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

2. **SARSA:**

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

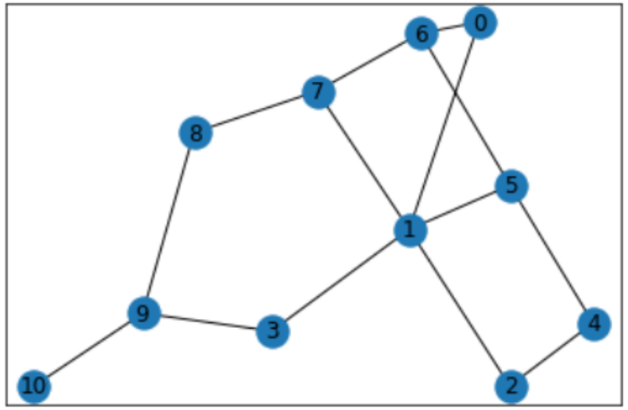
Where the update equation for SARSA depends on the current state, current action, reward obtained, next state and next action. This observation leads to the naming of the learning technique as SARSA stands for **State Action Reward State Action** which symbolizes the tuple (s, a, r, s', a').

Defining and Visualising the Graph

- We are going to demonstrate how to implement a basic Reinforcement Learning algorithm which is called as the **Q-Learning technique**.
- In this demonstration, we attempt to teach a bot to reach its destination using the **Q-Learning technique**.
- The graph may not look the same on reproduction of the code because the [networkx]() library in python produces a random graph from the given edges.

```
import numpy as np
import pylab as pl
import networkx as nx
```

```
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),
         (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),
         (8, 9), (7, 8), (1, 7), (3, 9)]

goal = 10
G = nx.Graph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_labels(G, pos)
pl.show()
```



Defining the reward the system for the bot

- **Q-learning** is a model-free reinforcement learning algorithm to learn quality of actions telling an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.
- For any finite Markov decision process (FMDP), **Q-learning** finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state.
- **Q-learning** can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that the algorithm computes with the maximum expected rewards for an action taken in a given state.

```
MATRIX_SIZE = 11
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
M *= -1

for point in edges:
    print(point)
    if point[1] == goal:
        M[point] = 100
    else:
        M[point] = 0

    if point[0] == goal:
        M[point[::-1]] = 100
    else:
        M[point[::-1]]= 0
        # reverse of point

M[goal, goal]= 100
print(M)
# add goal point round trip
```

```
(0, 1)
(1, 5)
(5, 6)
(5, 4)
(1, 2)
(1, 3)
(9, 10)
(2, 4)
(0, 6)
(6, 7)
(8, 9)
(7, 8)
(1, 7)
(3, 9)
[[ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.   0.   0.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.]
 [ -1.  -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.]
 [ -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.   0.  -1. 100.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0. 100.]]
```

Defining some utility functions to be used in the training

- The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \bigg)}^{\text{temporal difference}}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate $(0 < \alpha \leq 1)$.

Note that $Q^{new}(s_t, a_t)$ is the sum of three factors:

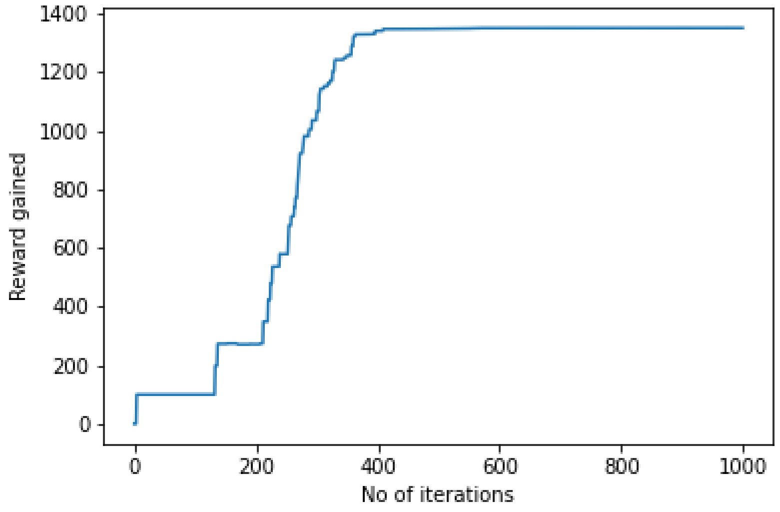
- $(1 - \alpha)Q(s_t, a_t)$: the current value weighted by the learning rate. Values of the learning rate near to 1 made faster the changes in Q.
- $\alpha\, r_t$: the reward $r_t = r(s_t, a_t)$ to obtain if action a_t is taken when in state s_t (weighted by learning rate)
- $\alpha\gamma \max_a Q(s_{t+1}, a)$: the maximum reward that can be obtained from state s_{t+1} (weighted by learning rate and discount factor)

```python
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))

gamma = 0.75
# learning parameter
initial_state = 1

# Determines the available actions for a given state
def available_actions(state):
    current_state_row = M[state, ]
    available_action = np.where(current_state_row >= 0)[1]
    return available_action

available_action = available_actions(initial_state)

# Chooses one of the available actions at random
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action

action = sample_next_action(available_action)

def update(current_state, action, gamma):

  max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
  if max_index.shape[0] > 1:
      max_index = int(np.random.choice(max_index, size = 1))
  else:
      max_index = int(max_index)
  max_value = Q[action, max_index]
  Q[current_state, action] = M[current_state, action] + gamma * max_value
  if (np.max(Q) > 0):
    return(np.sum(Q / np.max(Q)*100))
  else:
    return (0)
# Updates the Q-Matrix according to the path chosen
```

Training and evaluating the bot using the Q-Matrix

```
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)

# print("Trained Q matrix:")
# print(Q / np.max(Q)*100)
# You can uncomment the above two lines to view the trained Q matrix

# Testing
current_state = 0
steps = [current_state]

while current_state != 10:

    next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state, ]))[1]
    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)
    steps.append(next_step_index)
    current_state = next_step_index

print("Most efficient path:")
print(steps)

pl.plot(scores)
pl.xlabel('No of iterations')
pl.ylabel('Reward gained')
pl.show()
```

```
Most efficient path:
[0, 1, 3, 9, 10]
```



A Guide to the Gym Toolkit

- **OpenAI** is an artificial intelligence (AI) research organization that aims to build artificial general intelligence (AGI). **OpenAI** provides a famous toolkit called **Gym** for training a reinforcement learning agent.
- Suppose we need to train our agent to drive a car. We need an environment to train the agent. Can we train our agent in the real-world environment to drive a car? **No**, because we have learned that the reinforcement learning (RL) is a trial-and-error learning process, so while we train our agent, it will make a lot of mistakes during learning.
- For example, let's suppose our agent hits another vehicle, and it receives a negative reward.
- It will learn that hitting other vehicles is not a good action and will try not to perform this action again. But we cannot train the RL agent in the real-world environment by hitting other vehicles, right?
- That’s why we use simulators and train the RL agent in the simulated environments.
- **Gym** is a popular toolkit and It provides a variety of environments for training an RL agent ranging from classic control tasks to Atari game environments.
- We can train our RL agent to learn in these simulated environments using various RL algorithms.

Reference and Resources

- Deep Reinforcement Learning with Python - Second Edition by Sudharsan Ravichandiran Published by Packt Publishing, 2020
- https://learning.oreilly.com/library/view/deep-reinforcement-learning/9781839210686/#publisher_resources
- https://rl-book.com/supplementary_materials/
- Introduction to Reinforcement Learning – DataCamp
- https://www.geeksforgeeks.org/ml-reinforcement-learning-algorithm-python-implementation-using-q-learning/