Reinforcement Learning

1. Model-Based :DP (dynamic programming)

Value Iteration using Dynamic Programming (DP) Model-Free:

- MonteCarlo: Estimates value functions based on averaging returns from episodes.
- TD: Updates value functions based on observed transitions (e.g., TD(0), SARSA, Q-learning).
- DQN: Uses deep learning to handle large state spaces, improves stability with experience replay.

Q-learning

Q-learning is an off-policy, model-free reinforcement learning algorithm. It aims to learn the action-value function $Q(s, a)$, which represents the expected return of taking action a in state s and following the optimal policy thereafter. The update rule for Q-learning is:

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

*Bellman equation has multiple representation but they all are the same (you will see more below)

where:

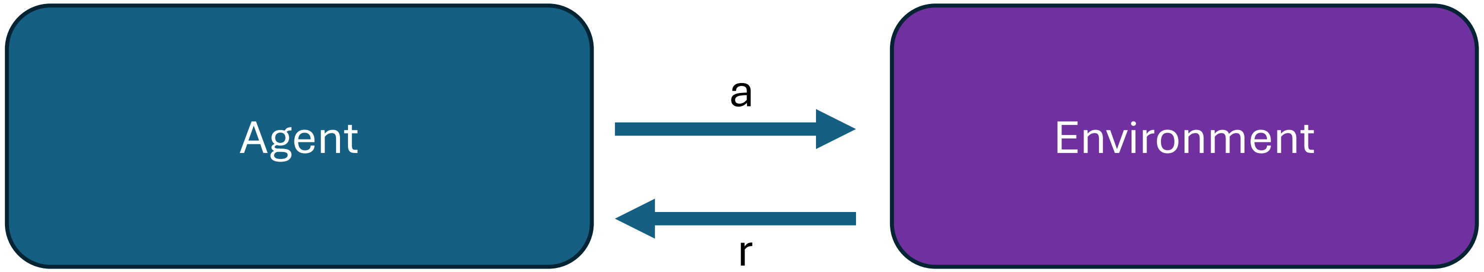
- $Q(s, a)$ is the action-value function.
- α is the learning rate.
- $R(s, a)$ is the reward received after taking action a in state s.
- γ is the discount factor.
- s' is the next state.
- a' is the next action.

	actions			
states	a_0	a_1	a_2	...
s_0	$Q(s_0, a_0)$	$Q(s_0, a_1)$	$Q(s_0, a_2)$	...
s_1	$Q(s_1, a_0)$	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...
s_2	$Q(s_2, a_0)$	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

- **Deterministic Policy π(s)π(s)**:
- A deterministic policy directly maps each state ss to a specific action aa.
- π(s)=a
- **Stochastic Policy π(a|s)** :
- A stochastic policy specifies a probability distribution over actions given a state ss. It allows the agent to select different actions with certain probabilities in each state.
- π(a|s)=P[At=a|St=s]

Policy Iteration (Python Framework provided)

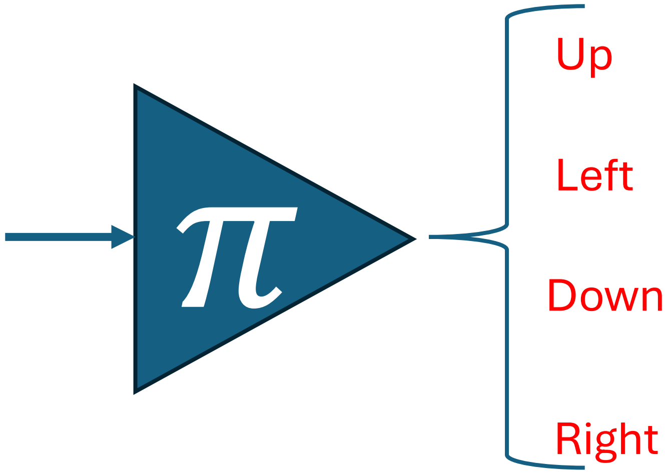
Policy tell us what is the action for a given state



Parameters

P(S)=Best Action

S1	S2	S3
S4	S5	S6
S7	S8	G



Policy

1. Bellman Expectation

The Bellman expectation equation for the state-value function under a policy

State Value

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V^\pi(S_{t+1}) | S_t = s]$$

Action Value

$$Q^\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma Q^\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

$V^\pi(s)$	The policy for a current state value
$Q^\pi(s,a)$	The policy for a current action value

Policy Optimization

Bellman Optimality Over iteration

The Bellman optimality equation for the state-value function is:

$$V^*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1})|S_t = s, A_t = a]$$

Where $V^*(s)$ is the optimal state-value function.

Bellman Optimality Equation for Action-Value Function (Q)

The Bellman optimality equation for the action-value function is:

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a')|S_t = s, A_t = a]$$

Where $Q^*(s, a)$ is the optimal action-value function.

Policy Optimization

$V^*(s)$	The policy for state value over iteration (Optimized)
$Q^*(s, a)$	The policy for action value over iteration (Optimized)

Example

```
# +---+---+---+
# | 0 | 0 | 0|
# +---+---+---+
# | -5 | 0| 0 |
# +---+---+---+
# |S  | 0 | 10 |
# +---+---+---+
```

param = Parameters((3, 3),8,{8: 10,3:-5},1.0)

If we run program

Output

```
State Values:
[[ 7.  8.  9.]
 [ 8.  9. 10.]
 [ 9. 10.  0.]]

Policy:
right down down
down down down
right right  G

Best Path from state 0 to goal:
[0, 1, 4, 7, 8]
```

Analyse the result

7	**8**	**9**
8	9	10
9	10	0

Each value show the **weight** of cell, topologically the same values have the same distance (chance) to reach the goal, at the end G will be zero to say we are on target

Best Path Actions: [0,1,4,7,8]

0	**1**	**2**
3	4	5
6	7	8=G

Policy Iteration

1. Create a data class to keep required parameters:
2. Create Environment class
3. Create Agent class

```
class Parameters:
   def __init__(self, size, goal_state, rewards,gamma):
      self.size = size
      self.goal_state = goal_state
      self.rewards = rewards
      self.gamma = gamma
```

Environment

- ```
class Environment:
    def __init__(self, size, goal_state, rewards):
        self.size = size
        self.goal_state = goal_state
        self.rewards = rewards
        self.actions = ['up', 'down', 'left', 'right']

    def get_next_state(self, state, action):
        row, col = divmod(state, self.size[1])
        if action == 'up':
            row = max(row - 1, 0)
        elif action == 'down':
            row = min(row + 1, self.size[0] - 1)
        elif action == 'left':
            col = max(col - 1, 0)
        elif action == 'right':
            col = min(col + 1, self.size[1] - 1)
        return row * self.size[1] + col

    def get_reward(self, state):
        return self.rewards.get(state, -1)
```
```

Agent(GridWord)

```
class Agent:
  def __init__(self, environment, parameters,gamma=1.0):
    self.environment = environment
    self.gamma = gamma
    self.policy = {}
    self.state_values = np.zeros(environment.size[0] * environment.size[1])
    self.initialize_policy()
    self.parameters = parameters

  #Initialization policy by random values of actions
  def initialize_policy(self):
    for state in range(self.environment.size[0] * self.environment.size[1]):
      self.policy[state] = np.random.choice(self.environment.actions)


  def policy_evaluation(self, iterations=100):
    for _ in range(iterations):
      new_state_values = np.copy(self.state_values) # create immutable to keep orignal state_values
      for state in range(self.environment.size[0] * self.environment.size[1]):
        if state == self.environment.goal_state:#if reach goal job is done just exit
          continue
        action = self.policy[state]
        next_state = self.environment.get_next_state(state, action)
        reward = self.environment.get_reward(next_state)
        new_state_values[state] = reward + self.gamma * self.state_values[next_state]
      self.state_values = new_state_values
```

……….

Example of using framework

- In Main:
-

```
import numpy as np

from RL.Agent import Agent
from RL.Environment import Environment
from RL.Parameters import Parameters

param = Parameters((3, 3),8,{8: 10,3:-5},1.0)
# Initialize the Environment
environment = Environment(param.size, param.goal_state, param.rewards)
# Initialize the Agent
agent = Agent(environment,param)

# Perform policy iteration
agent.policy_iteration()

# Print the state values and policy
print("State Values:")
print(agent.state_values.reshape(param.size))
print("\nPolicy:")
for row in range(param.size[0]):
  for col in range(param.size[1]):
    state = row * param.size[1] + col
    if state == param.goal_state:
      print(" G ", end=" ")
    else:
      print(agent.policy[state], end=" ")
  print()

 # Find and print the best path from a starting state to the goal state
start_state = 0
best_path = agent.find_best_path_for_goal(start_state)
print("\nBest Path from state 0 to goal:")
print(best_path)
```