2.1 Markov Decision Processes

MDP is a mathematical framework to model RL frameworks

It models states, actions, and rewards, alongside transition probabilities that represent the likelihood in moving from one state to another following an action.

The Markov property: the future state depends only on current state and action

For example, in Frozen Lake, each cell is a state. Some states are terminal, like the holes in the lake. Actions are moves (up, down, left, right).

```
env = gym.make('Frozen Lake', is_slippery=True) # is_slipped arg controls
action outcomes. True = stochastic movements. False = deterministic movements
```

in a stochastic environment, an action may lead to several outcomes
in a deterministic environment, each action leads to a specific outcome

```
print(env.action_space) # range of actions (Discrete(4)) for 4 action types
print(env.observation_space) # range of states (Discrete(16)) for 16 cells
print(env.action_space.n) # for exact number
print(env.observation_space.n)
```

env.unwrapped.P is a dictionary where keys are state-action pairs

```
print(env.unwrapped.P[state][action]) # (probability, next_state, reward,
is_terminal)
```

2.2 Policies and state-value functions

The core objective is formulating effective policies, which act as road maps guiding the agent by specifying optimal actions in each state to maximize return

We can define an example policy in code

```
policy = {
    0:1, 1:2, 2:1,
    3:1, 4:3, 5:1,
    6:2, 7:3
}

state, info = env.reset()
terminated = False
```

```
while not terminated:
    action = policy[state]
    state, reward, terminated, _, _ = env.step(action)
```

To evaluate a policy, we utilize state-value functions to assess the state's worth

We can use the Bellman equation, a recursive state-value function,
$V(s) = r_{s+1} + \gamma V(s+1)$,
that is the sum of the immediate reward and the discounted value of the next state

```
def compute_state_value(state):
    if state == terminal_state:
        return 0

    action = policy[state]
    _, next_state, reward, _ = env.unwrapped.P[state][action][0]
    return reward + gamma * compute_state_value(next_state)

terminal_state=8
gamma=1

V = {state: compute_state_value(state)
    for state in range(num_states)}
```

2.3 Action-value functions

How to improve policies through action-value functions (Q-values), which provide the expected return
$Q(s, a) = r_a + \gamma V(s+1)$
This is the sum of the reward received after performing action a in state s, and the discounted value of the next state resulting from action a

Estimates desirability of actions within states

Example:
Q(born_state, movement) = reward + gamma *state_value*
Q(4,right) = -1 + (gamma = 1) 10

```
def compute_q_value(state, action):
    if state == terminal_state:
        return None
    _, next_state, reward, _ = env.unwrapped.P[state][action][0]
    return reward + gamma * compute_state_value(next_state)

Q = {(state,action): compute_q_value(state,action)
```

```
    for state in range(num_states)
    for action in range(num_actions)}
```

Since each state has 4 Q values, we can select the highest Q value for each state to improve the policy:

```
improved_policy = {}
for state in range(num_states-1):
    max_action = max(range(num_actions), key=lambda action: Q[(state,action)])
    improved_policy[state] = max_action
```

2.4 Policy iteration and value iteration

Policy iteration is to find optimal policy via iteration

We evaluate and improve policies until policy stops changing

```
def policy_evaluation(policy):
    V = {state: compute_state_value(state, policy) for state in 
range(num_states)}
    return V

def policy_improvement(policy):
    improved_policy = {s: 0 for s in range(num_states-1)}
    Q = {(state,action): compute_q_value(state,action,policy)
    for state in range(num_states) for action in range(num_actions)}

    for state in range(num_states-1):
        max_action = max(range(num_actions), key=lambda action: 
Q[(state,action)])
        improved_policy[state] = max_action

    return improved_policy

def policy_iteration():
    policy = {# series of actions}
    while True:
        V = policy_evaluation(policy)
        improved_policy = policy_improvement(policy)

        if improved_policy == policy:
            break
        policy = improved_policy
    return policy, V
```

Value iteration simplifies and speeds up the process by combining policy evaluation and improvement
Initialize, compute Q using V, update V by selecting best actions, iterate until V stops changing

```
V = {state: 0 for state in range(num_states)}
policy = {state:0 for state in range(num_states-1)}
threshold = 0.001

while True:
    new_V = {state: 0 for state in range(num_states)}
    for state in range(num_states-1):
        max_action, max_q_value = get_max_action_and_value(state, V)
        new_V[state] = max_q_value
        policy[state] = max_action

    if all(abs(new_v[state]-V[state]) < threshold for state in V):
        break
    V = new_V

def get_max_action_and_value(state, V):
    Q_values = [compute_q_value(state, action, V) for action in
range(num_actions)]
    max_action = max(range(num_actions), key=lambda a: Q_values[a])
    max_q_value = Q_values[max_action]
    return max_action, max_q_value

# we now need to modify compute_q_value to use V dictionary

def compute_q_value(state, action, V): # V is arg now
    if state = terminal_state:
        return None
    _, next_state, reward, _ env.P[state][action][0]
    return reward + gamma * V[next_state]
```