Reinforcement Learning (RL) – A Deep Dive

1. What is Reinforcement Learning (RL)?

Reinforcement Learning (RL) is a **learning paradigm** where an **agent** interacts with an **environment** and learns **optimal actions** by maximizing cumulative rewards.

Example Scenario

- Imagine **training a robot to walk**.
- The **agent** (robot) takes **actions** (move left, right, forward).
- The **environment** (ground) responds with **rewards** (+1 for moving forward, -1 for falling).
- The agent **learns a policy** to maximize rewards **over time**.

Key RL Concepts

Concept	Definition
Agent	The decision-maker (e.g., robot, AI playing chess)
Environment	The world the agent interacts with (e.g., chessboard, game, robot's surroundings)
State (s)	A representation of the environment at a given time
Action (a)	A choice the agent makes in a given state
Reward (r)	A signal that tells the agent if an action was good or bad
Policy (π)	The strategy the agent uses to choose actions
Value Function ($V(s)$)	The expected long-term reward from state s
Q-Value ($Q(s,a)$)	The expected reward for taking action a in state s

2. Markov Decision Processes (MDP) – The RL Framework

What is an MDP?

An **MDP (Markov Decision Process)** provides the **mathematical foundation** for RL. It consists of:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$$

where:

- $\mathcal{S}$ → Set of **states**
- $\mathcal{A}$ → Set of **actions**
- $P(s'|s, a)$ → **Transition probability** (probability of moving to state s' from s after action a)
- $R(s, a)$ → **Reward function** (immediate reward for taking action a in state s)
- γ → **Discount factor** (how much future rewards matter)

Key MDP Properties

- **Markov Property**: The future state **only depends on the current state and action**, not past history.
- **Goal**: Find an **optimal policy** $\pi^*(s)$ that **maximizes cumulative rewards** over time.

3. Policy & Value Functions

What is a Policy ($\pi(s)$)?

A **policy** is the agent's strategy for choosing actions.

- **Deterministic Policy**: Always picks the best action.

$$\pi(s) = a$$

- **Stochastic Policy**: Picks actions based on probabilities.

$$\pi(a|s) = P(a|s)$$

Value Functions

- **State-Value Function ($V(s)$)**: Expected total reward **starting from state** s.

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, \pi\right]$$

- **Action-Value Function ($Q(s,a)$)**: Expected total reward **starting from** s **and taking action** a.

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right]$$

Bellman Equation (Recursion)

The **Bellman equation** expresses $V(s)$ in terms of **future rewards**:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) \left[R(s,a) + \gamma V^\pi(s')\right]$$

This lets us compute **optimal policies** efficiently.

4. Q-Learning: Learning Without a Model

Q-Learning is a **model-free RL algorithm** that learns **Q-values** without knowing the environment dynamics.

Q-Learning Update Rule

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

where:

- α = learning rate.
- r = reward.
- γ = discount factor.

Deep Q-Learning (DQN)

Deep Q-Networks (DQN) use **neural networks** to approximate $Q(s,a)$ when the state space is large.

- **Replace Q-table with a deep neural network**.
- **Uses Experience Replay** → Stores past experiences and learns from random samples.
- **Target Network Stabilization** → Uses two networks to improve stability.

5. Policy Gradient Methods

Instead of learning $Q(s, a)$, **policy gradient methods** directly optimize the policy.

Policy Gradient Loss

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \log \pi_\theta(a_t|s_t)\right]$$

This is optimized using **stochastic gradient ascent**.

Actor-Critic Methods

- **Actor** updates the policy $\pi_\theta(s)$.
- **Critic** estimates $V(s)$ to guide learning.

6. Fully Commented Manual Implementation of Q-Learning

python

Copy Edit

```python
import numpy as np
import random

class QLearningAgent:
    def __init__(self, n_states, n_actions, alpha=0.1, gamma=0.9, epsilon=0.1):
        self.q_table = np.zeros((n_states, n_actions))  # Initialize Q-table
        self.alpha = alpha  # Learning rate
        self.gamma = gamma  # Discount factor
        self.epsilon = epsilon  # Exploration rate

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:  # Exploration
            return random.randint(0, self.q_table.shape[1] - 1)
        return np.argmax(self.q_table[state])  # Exploitation
```

```python
def update(self, state, action, reward, next_state):
    best_next_action = np.argmax(self.q_table[next_state])  # Best future action
    target = reward + self.gamma * self.q_table[next_state, best_next_action]
    self.q_table[state, action] += self.alpha * (target - self.q_table[state, action]
```

```python
# Example: 5 states, 2 actions per state
agent = QLearningAgent(n_states=5, n_actions=2)

# Simulate an experience
state, action, reward, next_state = 0, 1, 10, 3
agent.update(state, action, reward, next_state)

# Print updated Q-table
print(agent.q_table)
```

```python
"""
CS 229 Machine Learning
Question: Reinforcement Learning - The Inverted Pendulum
"""
from __future__ import division, print_function
from env import CartPole, Physics
import matplotlib.pyplot as plt
import numpy as np
from scipy.signal import lfilter

"""
Parts of the code (cart and pole dynamics, and the state
discretization) are inspired from code available at the RL repository
http://www-anw.cs.umass.edu/rlr/domains.html

Briefly, the cart-pole system is described in `cart_pole.py`. The main
simulation loop in this file calls the `simulate()` function for
simulating the pole dynamics, `get_state()` for discretizing the
otherwise continuous state space in discrete states, and `show_cart()`
for display.

Some useful parameters are listed below:

`NUM_STATES`: Number of states in the discretized state space
You must assume that states are numbered 0 through `NUM_STATES` - 1. The
state numbered `NUM_STATES` - 1 (the last one) is a special state that
marks the state when the pole has been judged to have fallen (or when
the cart is out of bounds). However, you should NOT treat this state
any differently in your code. Any distinctions you need to make between
states should come automatically from your learning algorithm.

After each simulation cycle, you are supposed to update the transition
counts and rewards observed. However, you should not change either
your value function or the transition probability matrix at each
cycle.

Whenever the pole falls, a section of your code below will be
executed. At this point, you must use the transition counts and reward
observations that you have gathered to generate a new model for the MDP
(i.e. transition probabilities and state rewards). After that, you
must use value iteration to get the optimal value function for this MDP
model.
```

```
`TOLERANCE`: Controls the convergence criteria for each value iteration
run. In value iteration, you can assume convergence when the maximum
absolute change in the value function at any state in an iteration
becomes lower than `TOLERANCE.

You need to write code that chooses the best action according
to your current value function, and the current model of the MDP. The
action must be either 0 or 1 (corresponding to possible directions of
pushing the cart)

Finally, we assume that the simulation has converged when
`NO_LEARNING_THRESHOLD` consecutive value function computations all
converged within one value function iteration. Intuitively, it seems
like there will be little learning after this, so we end the simulation
here, and say the overall algorithm has converged.


Learning curves can be generated by calling a code snippet at the end
(it assumes that the learning was just executed, and the array
`time_steps_to_failure` that records the time for which the pole was
balanced before each failure is in memory). `num_failures` is a variable
that stores the number of failures (pole drops / cart out of bounds)
till now.

Other parameters in the code are described below:

`GAMMA`: Discount factor to be used

The following parameters control the simulation display; you dont
really need to know about them:

`pause_time`: Controls the pause between successive frames of the
display. Higher values make your simulation slower.
`min_trial_length_to_start_display`: Allows you to start the display only
after the pole has been successfully balanced for at least this many
trials. Setting this to zero starts the display immediately. Choosing a
reasonably high value (around 100) can allow you to rush through the
initial learning quickly, and start the display only after the
performance is reasonable.
"""
```

```python
def initialize_mdp_data(num_states):
    """
    Return a variable that contains all the parameters/state you need for your MDP.
    Feel free to use whatever data type is most convenient for you (custom classes, tuples, dicts, etc)

    Assume that no transitions or rewards have been observed.
    Initialize the value function array to small random values (0 to 0.10, say).
    Initialize the transition probabilities uniformly (ie, probability of
        transitioning for state x to state y using action a is exactly
        1/num_states).
    Initialize all state rewards to zero.

    Args:
        num_states: The number of states

    Returns: The initial MDP parameters
    """
    transition_counts = np.zeros((num_states, num_states, 2))
    transition_probs = np.ones((num_states, num_states, 2)) / num_states
    #Index zero is count of rewards being -1 , index 1 is count of total num state is reached
    reward_counts = np.zeros((num_states, 2))
    reward = np.zeros(num_states)
    value = np.random.rand(num_states) * 0.1

    return {
        'transition_counts': transition_counts,
        'transition_probs': transition_probs,
        'reward_counts': reward_counts,
        'reward': reward,
        'value': value,
        'num_states': num_states,
    }

def sample_random_action():
    return 0 if np.random.uniform() < 0.5 else 1
```

```python
def choose_action(state, mdp_data):
    """
    Choose the next action (0 or 1) that is optimal according to your current
    mdp_data. When there is no optimal action, return a random action using
    sample_random_action.

    Args:
        state: The current state in the MDP
        mdp_data: The parameters for your MDP. See initialize_mdp_data.

    Returns:
        0 or 1 that is optimal according to your current MDP
    """

    # *** START CODE HERE ***
    pi0 = np.sum(mdp_data['transition_probs'][state, :, 0] * mdp_data['value'][:])
    pi1 = np.sum(mdp_data['transition_probs'][state, :, 1] * mdp_data['value'][:])
    if pi0 == pi1:
        return sample_random_action()
    else:
        if pi0 > pi1:
            return 0
        else:
            return 1

    # *** END CODE HERE ***

def update_mdp_transition_counts_reward_counts(mdp_data, state, action, new_state, reward):
    """
    Update the transition count and reward count information in your mdp_data.
    Do not change the other MDP parameters (those get changed later).

    Record the number of times `state, action, new_state` occurs.
    Record the rewards for every `new_state`
    (since rewards are -1 or 0, you just need to record number of times reward -1 is seen in 'reward_counts' index new_state,0)
    Record the number of time `new_state` was reached (in 'reward_counts' index new_state,1)

    Args:
        mdp_data: The parameters of your MDP. See initialize_mdp_data.
        state: The state that was observed at the start.
        action: The action you performed.
```

```python
        action: The action you performed.
        new_state: The state after your action.
        reward: The reward after your action (i.e. reward corresponding to new_state).

    Returns:
        Nothing
    """

    # *** START CODE HERE ***
    mdp_data['transition_counts'][state, new_state, action] += 1
    mdp_data['reward_counts'][new_state, 1] += 1
    if reward == -1:
        mdp_data['reward_counts'][new_state, 0] += 1
    # *** END CODE HERE ***

    # This function does not return anything
    return

```

```python
def update_mdp_transition_probs_reward(mdp_data):
    """
    Update the estimated transition probabilities and reward values in your MDP.

    Make sure you account for the case when a state-action pair has never
    been tried before, or the state has never been visited before. In that
    case, you must not change that component (and thus keep it at the
    initialized uniform distribution).

    Args:
        mdp_data: The data for your MDP. See initialize_mdp_data.

    Returns:
        Nothing

    """

    # *** START CODE HERE ***
    for i in range(mdp_data['transition_counts'].shape[0]):
        if mdp_data['reward_counts'][i, 1] == 0:
            continue
        else:
            for j in range(2):
                denom = 0
                denom = np.sum(mdp_data['transition_counts'][i, :, j])
                if denom == 0:
                    continue
                else:
                    for k in range(mdp_data['transition_counts'].shape[0]):
                        mdp_data['transition_probs'][i, k, j] = mdp_data['transition_counts'][i, k, j] / denom
            mdp_data['reward'][i] = mdp_data['reward_counts'][i, 0] / mdp_data['reward_counts'][i, 1]

    # *** END CODE HERE ***

    # This function does not return anything
    return
```

```python
    if plot:
        # plot the learning curve (time balanced vs. trial)
        log_tstf = np.log(np.array(time_steps_to_failure))
        plt.plot(np.arange(len(time_steps_to_failure)), log_tstf, 'k')
        window = 30
        w = np.array([1/window for _ in range(window)])
        weights = lfilter(w, 1, log_tstf)
        x = np.arange(window//2, len(log_tstf) - window//2)
        plt.plot(x, weights[window:len(log_tstf)], 'r--')
        plt.xlabel('Num failures')
        plt.ylabel('Log of num steps to failure')
        plt.savefig('./control.pdf')

    return np.array(time_steps_to_failure)

if __name__ == '__main__':
    main()
```

```python
            # Recompute MDP model whenever pole falls
            # Compute the value function V for the new model
            if new_state == NUM_STATES - 1:

                update_mdp_transition_probs_reward(mdp_data)

                converged_in_one_iteration = update_mdp_value(mdp_data, TOLERANCE, GAMMA)

                if converged_in_one_iteration:
                    consecutive_no_learning_trials = consecutive_no_learning_trials + 1
                else:
                    consecutive_no_learning_trials = 0

            # Do NOT change this code: Controls the simulation, and handles the case
            # when the pole fell and the state must be reinitialized.
            if new_state == NUM_STATES - 1:
                num_failures += 1
                if num_failures >= max_failures:
                    break
                print('[INFO] Failure number {}'.format(num_failures))
                time_steps_to_failure.append(time - time_at_start_of_current_trial)
                # time_steps_to_failure[num_failures] = time - time_at_start_of_current_trial
                time_at_start_of_current_trial = time

                if time_steps_to_failure[num_failures - 1] > min_trial_length_to_start_display:
                    display_started = 1

                # Reinitialize state
                # x = 0.0
                x = -1.1 + np.random.uniform() * 2.2
                x_dot, theta, theta_dot = 0.0, 0.0, 0.0
                state_tuple = (x, x_dot, theta, theta_dot)
                state = cart_pole.get_state(state_tuple)
            else:
                state = new_state

```

```python
    mdp_data = initialize_mdp_data(NUM_STATES)

    # This is the criterion to end the simulation.
    # You should change it to terminate when the previous
    # 'NO_LEARNING_THRESHOLD' consecutive value function computations all
    # converged within one value function iteration. Intuitively, it seems
    # like there will be little learning after this, so end the simulation
    # here, and say the overall algorithm has converged.

    consecutive_no_learning_trials = 0
    while consecutive_no_learning_trials < NO_LEARNING_THRESHOLD:

        action = choose_action(state, mdp_data)

        # Get the next state by simulating the dynamics
        state_tuple = cart_pole.simulate(action, state_tuple)
        # x, x_dot, theta, theta_dot = state_tuple

        # Increment simulation time
        time = time + 1

        # Get the state number corresponding to new state vector
        new_state = cart_pole.get_state(state_tuple)
        # if display_started == 1:
        #     cart_pole.show_cart(state_tuple, pause_time)

        # reward function to use - do not change this!
        if new_state == NUM_STATES - 1:
            R = -1
        else:
            R = 0

        update_mdp_transition_counts_reward_counts(mdp_data, state, action, new_state, R)

```

```python
def main(plot=True):
    # Seed the randomness of the simulation so this outputs the same thing each time
    np.random.seed(0)

    # Simulation parameters
    pause_time = 0.0001
    min_trial_length_to_start_display = 100
    display_started = min_trial_length_to_start_display == 0

    NUM_STATES = 163
    GAMMA = 0.995
    TOLERANCE = 0.01
    NO_LEARNING_THRESHOLD = 20

    # Time cycle of the simulation
    time = 0

    # These variables perform bookkeeping (how many cycles was the pole
    # balanced for before it fell). Useful for plotting learning curves.
    time_steps_to_failure = []
    num_failures = 0
    time_at_start_of_current_trial = 0

    # You should reach convergence well before this
    max_failures = 500

    # Initialize a cart pole
    cart_pole = CartPole(Physics())

    # Starting `state_tuple` is (0, 0, 0, 0)
    # x, x_dot, theta, theta_dot represents the actual continuous state vector
    x, x_dot, theta, theta_dot = 0.0, 0.0, 0.0, 0.0
    state_tuple = (x, x_dot, theta, theta_dot)

    # `state` is the number given to this state, you only need to consider
    # this representation of the state
    state = cart_pole.get_state(state_tuple)
    # if min_trial_length_to_start_display == 0 or display_started == 1:
    #     cart_pole.show_cart(state_tuple, pause_time)
```

```
    # *** START CODE HERE ***
    flag = False
    flag_convergence = False
    cont_convergence = 0
    deltas = np.zeros(mdp_data['transition_counts'].shape[0])

    while flag_convergence == False:
        cont_convergence += 1
        for i in range(mdp_data['transition_counts'].shape[0]):
            value = mdp_data['value'][i]
            sum0 = np.sum(mdp_data['transition_probs'][i, :, 0] * mdp_data['value'][:])
            sum1 = np.sum(mdp_data['transition_probs'][i, :, 1] * mdp_data['value'][:])
            V0 = mdp_data['reward'][i] + gamma * sum0
            V1 = mdp_data['reward'][i] + gamma * sum1
            mdp_data['value'][i] = max(V0, V1)
            delta = abs(mdp_data['value'][i] - value)
            deltas[i] = delta

        delta_max = np.max(deltas)
        if delta_max < tolerance:
            if cont_convergence == 1:
                flag = True
            flag_convergence = True

    return flag
    # *** END CODE HERE ***
```

```python
def update_mdp_value(mdp_data, tolerance, gamma):
    """
    Update the estimated values in your MDP.


    Perform value iteration using the new estimated model for the MDP.
    The convergence criterion should be based on `TOLERANCE` as described
    at the top of the file.

    Return true if it converges within one iteration.

    Args:
        mdp_data: The data for your MDP. See initialize_mdp_data.
        tolerance: The tolerance to use for the convergence criterion.
        gamma: Your discount factor.

    Returns:
        True if the value iteration converged in one iteration

    """
```

Final Takeaways

✅ RL is about maximizing cumulative rewards by learning an optimal policy.
✅ Q-Learning is a fundamental technique for model-free RL.
✅ Deep Q-Networks (DQN) use neural networks to approximate Q-values.
✅ Policy Gradient methods learn policies directly.
✅ Actor-Critic methods combine value-based and policy-based learning.