UCS21H24 PRINCIPLES OF REINFORCEMENT LEARNING

Unit 3

Integration of Tabular Methods

n-step Bootstrapping : TD Prediction - SARSA - OFF policy learning; Planning and Learning with Tabular Methods

* Temporal Difference (TD) Learning

→ a method in RL aimed at predicting future outcomes or optimizing decisions in uncertain environments.

→ combines aspects of

(i) Monte Carlo methods - estimate outcomes based on complete episodes

(ii) Dynamic Programming - uses known models of the environment.

→ The primary goal of TD learning is to estimate the value of a state/action in a given environment. It involves updating value estimates based on the difference (error) between predicted rewards and actually observed rewards over time.

→ This difference is called the temporal difference error.

* TD Learning Process

→ When an agent is in a particular state, it makes a prediction about the value of that state.

→ The agent takes an action, receives a reward, and moves to the next state.

→ It then makes a prediction about the value of the next state.

→ The TD error is computed as the difference between the new prediction and the new prediction (adjusted by the received reward) and the old prediction.

→ This error is used to update the value of the initial state, bringing it closer to the true value.

$$\text{TD Error} = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Update Rule :-

$$V(s_t) \leftarrow V(s_t) + \alpha \times \text{TD Error}$$

$$V(s_t) \leftarrow V(s_t) + \alpha [\, r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \,]$$

* Goal of TD Learning

→ Estimate the value of states to improve decision-making.

→ Imagine playing a game where each state corresponds to a situation with potential rewards.

→ You can take actions, which lead to new states and give one some points (rewards).

→ TD Learning helps understand how good each state is so that one can make better decisions and maximize one's total points (rewards) in the game.

* Why TD Learning

(i) Fast Updates - Don't need to wait until an episode ends to update values. Updates occur after each action.

(ii) Model Free - TD Learning doesn't require knowledge of the transition rules or environment model; it learns through experience alone.

* Examples of TD Learning?

① Weather Prediction - You think today it will be sunny (current prediction). Suddenly, you see clouds forming (new info, a new state) You update your prediction to it might rain soon, based on the new state.

This update is like the TD error. You adjust your prediction because you learned something new.

② Delivery Robot - You have a delivery robot that needs to learn the best path to delivery packages from the lobby to various offices in a building.

The robot receives a reward for delivering the package successfully to the correct office. The robot's goal is to learn which paths lead to faster deliveries (higher rewards) and avoid paths that are slow or have obstacles (lower rewards).

* Steps in TD Learning

Step 1 : Initial Estimate

→ At the beginning, you have some guess about how good each state is. This is called the value of the state. Can start w/ random guesses initially

Step 2 : Experience a transition

→ Let the current state be s.

→ Take an action and move to a new state s'

→ Receive an ~~action~~ reward r for this action

Step 3 : Make a prediction

→ Before taking the action, you had a prediction about the value of the state s.

→ After moving to state s', you have a new prediction based on s' and the reward recieved.

Step 4 : Calculate TD Error.

→ The TD error is the difference between what you predicted for state s and what you learned after moving to state s'.

* TD Learning vs. Monte Carlo Methods

TD Learning	Monte Carlo Methods
1. One-step TD allows for updates at each step in an episode. - calculate reward diff. when moving from the current state to the next	1. MC updates every state after analyzing the whole episode. (roughly equivalent to an update for each state occurring after n steps)
2. $V(s_t) \leftarrow V(s_t) + \alpha\left[\underbrace{R_{t+1} + \gamma V(s_{t+1})}_{\text{Bootstrapping target}} - V(s_t)\right]$ TD Error	$V(s_t) \leftarrow V(s_t) + \alpha\left[\underbrace{G_t}_{\text{return}} - V(s_t)\right]$ MC Error

(-X-) n-step Bootstrapping generalizes TD-algorithms

one-step TD Learning ——— Monte Carlo

$n=1$ $\qquad$ $n=\infty$

* n-step Bootstrapping

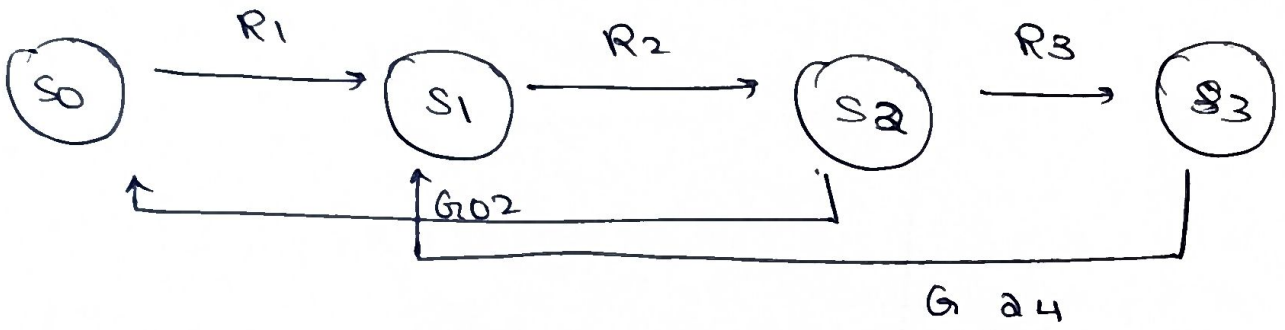
→ The idea of 1-step TD learning can be extended to multiple steps.

→ To do this, introduce n-step return, which calculates the accumulated discount reward, between the current state t and the future state at step $t+n$. In addition, it also adds state value at step $t+n$.

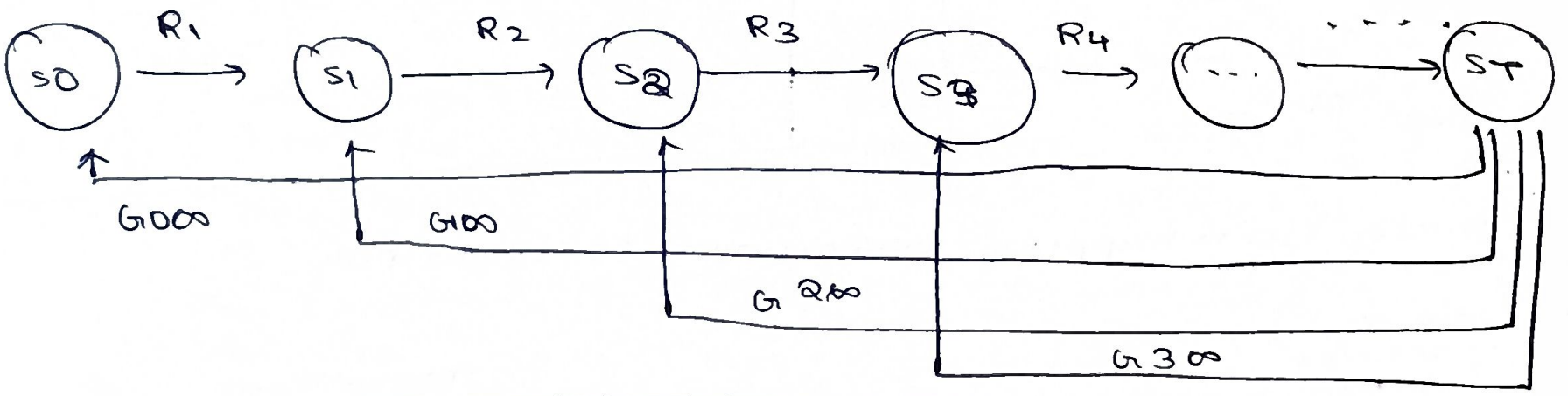
→ The update rule for n-step bootstrapping is

$$V(S_t) = V(S_t) + \alpha\left(G_{t:t+n} - V(S_t)\right)$$

1-step TD



2 step TD



∞ - step TD = Monte Carlo

example : in 3-step TD

→ beginning of episode is generated till state S_3

→ S_0 is updated using 3-step return, which sums up rewards R_1, R_2, R_3 and the value of state 3

Eq for S_1 → sum R_2, R_3 & R_4 and S_4

* Algorithm for n-step Bootstrapping

Input : policy π

Parameters : step size $\alpha \in (0,1]$, a +ve int n

Initialize $v(s)$ arbitrarily $\forall \; s \in S$

Loop for each episode

Initialize & store s_0

$T \leftarrow \infty$

Loop for $t = 0, 1, 2 \ldots$

If $t < T$ then

Take an action according to $\pi(\cdot | s_t)$

Observe and stored R_{t+1} & s_{t+1}

If s_{t+1} is interminal $T \leftarrow t+1$

$\gamma \leftarrow t - n + 1$

If $\gamma \geq 0$:

$G \leftarrow \sum \gamma^{i-\gamma-1} R_i$

If $\gamma + n < T$, $G \leftarrow G + \gamma^n v(s_{\gamma+n})$

$v(s_\gamma) \leftarrow v(s_\gamma) + \alpha [G - v(s_\gamma)]$

until $\gamma = T-1$

* On-Policy and off-Policy learning Methods

→ In RL, on-policy and off-policy methods refer to how an agent learns from the data it collects during interaction with an environment.

A. On-Policy Methods

Definition : On-policy methods learn the value of the same policy that the agent follows to make decisions. This means that the agent both explores and learns using the same policy.

Characteristics

(i) Policy Improvement : The agent continuously improves the policy it is currently using to interact with the environment

(ii) Exploration and Exploitation Balance : Both managed by a single policy, that tries to balance trying new actions (exploration) with choosing the best-known action (exploitation).

eg. ε-greedy policy

Example Algorithm — SARSA

Advantages — (i) more stable updates as they are based on the current behavior policy

Disadvantages — (i) slower convergence compared to off-policy methods

(ii) may not fully explore the environment if the exploration strategy is not effective.

B. Off-Policy Methods

Definition - allows the agent to learn the value of an optimal policy independently of the actions it takes.

→ The agent can use one policy for exploration (called the behavior policy), while learning or improving another policy (often, an optimal policy, known as the target policy)

Characteristics -

Separate Learning and Exploration Policies - The agent gathers experience with one policy and learns with another, more optimal policy.

Example Algorithm - Q-Learning - agent uses the experience of its exploratory behaviour to update the value of the optimal policy.

Advantages (i) more flexible as it allows for off-policy exploration

(ii) can learn from stored data (replay buffer), which is useful in environments with large state-action spaces

Disadvantages : (i) may be less stable due to the difference between the behavior and target policies.

(ii) may require more careful tuning of exploration strategies.

* Q- Learning

→ Q-Learning is an RL technique that uses Q-values (or action-values) to guide an agent in selecting actions that yield the highest rewards over time.

→ The primary goal is to build a Q-table that guides the agent towards optimal actions at each state.

Terms

(i) Q-values : These values represent the quality of an action Q(s,a) taken at a state s, indicating how rewarding the action is expected to be.

→ Q(s,a) is iteratively computed using the TD-update rule

(ii) Reward : After each transition, the agent observes a reward for every action from the environment, and then transits to another state.

* Process of Q-Learning

→ Initially, the agent explores the environment and updates the Q-Table. When the Q-Table is ready, the agent will start to exploit the environment and take better actions.

→ It is an off-policy algorithm, meaning that it learns the optimal policy independently of the agent's current behavior policy, allowing it to refine its strategy.

→ Q-Learning estimates the value of Q at each time step, using the TD-update rule.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \underbrace{\left(R + \gamma \left(\max \left(Q(s',A') - Q(S,A) \right) \right) \right)}_{\text{TD - error}}$$

(R: reward; γ: discount factor)

* Q-Learning Algorithm

1. Set the γ parameter
2. Set environment rewards in matrix
3. Initialize matrix Q as zero matrix

(rows denote the current state of the agent, columns denote the possible actions leading to the next state)

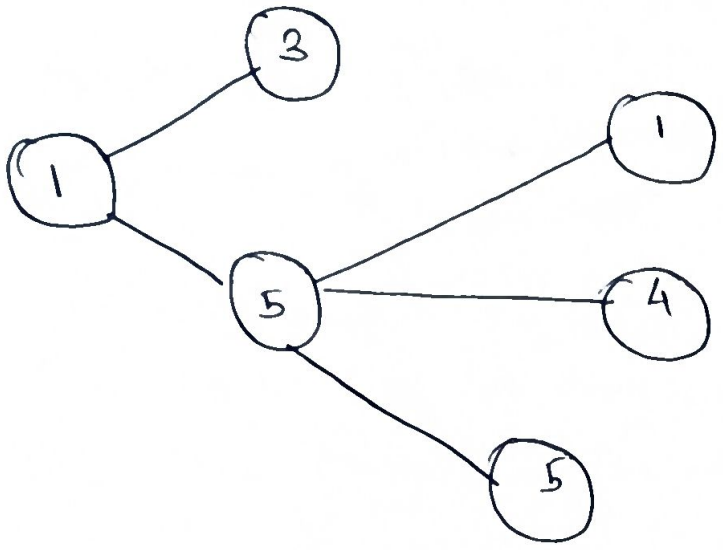
4. Select random initial source state
   - Set initial state s = current state
   - Select one action 'a' among all possible actions using the exploratory policy
   - Take this action 'a', go to next state s'
   - Observe reward r
   - Get max Q-value to go to the next state based on all possible actions
5. Compute

$$Q(s,a) = Q(s,a) + \alpha \left(R + \gamma \max \left(Q(s',a') - Q(s,a) \right) \right)$$

6. Repeat the above steps until current state = goal state.

Example

Consider the following graph:



Let $\gamma = 0.8$

The Q matrix is initialized to 0.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

The R-matrix is

$$R = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

From state 1, agent can go to 3 or 5

Go to 5 (say)

Compute Q value

$\Rightarrow Q(s,a) = R(s,a) + \gamma * \max[Q(\text{next state}, \text{all actions})]$

$\Rightarrow Q(1,5) = 100 + 0.8 * \max[Q(5,1), Q(5,4), Q(5,5)]$

$\Rightarrow Q(1,5) = 100 + 0.8 * \max(0,0,0)$

$Q(1,5) = 100 \rightarrow$ Fill this value in Q table.

* SARSA

→ an RL algorithm that learns by interacting with its environment.

→ an on-policy TD-learning algorithm. It updates the action-value function $Q(s,a)$, when starting from state s, taking action a, and following the current policy thereafter.

→ The agent learns from its own experiences & refines it policy based on the observed rewards & state transition

→ The update rule for SARSA is

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

Algorithm

1. Initialize Q-table - fill with random values or zeroes, representing the estimated value of taking each action in each state

2. Choose action - The agent chooses an action based on its current state using a policy, such as ϵ-greedy, which balances exploration & exploitation

3. Take action - The agent takes the chosen action and observes the new state and the reward it receives from the environment

4. Update Q-Table - The Q-table is updated using the SARSA update rule.

5. Repeat - The agent continues to iterate through the process, observing outcomes & updating its Q-table, till it converges

* Advantages and Disadvantages of SARSA

Advantages	Disadvantages
→ more stable, less prone to oscillations compared to off-policy methods	→ can be sensitive to learning rate and exploration parameters → can be slow to converge in environments w/ high variance or complex reward structures

* Planning and Learning with Tabular Methods

* Planning and Learning

Planning - any computational process that uses a model to create or improve a policy

Learning - the acquisition of knowledge or skills through experience, study or being taught.

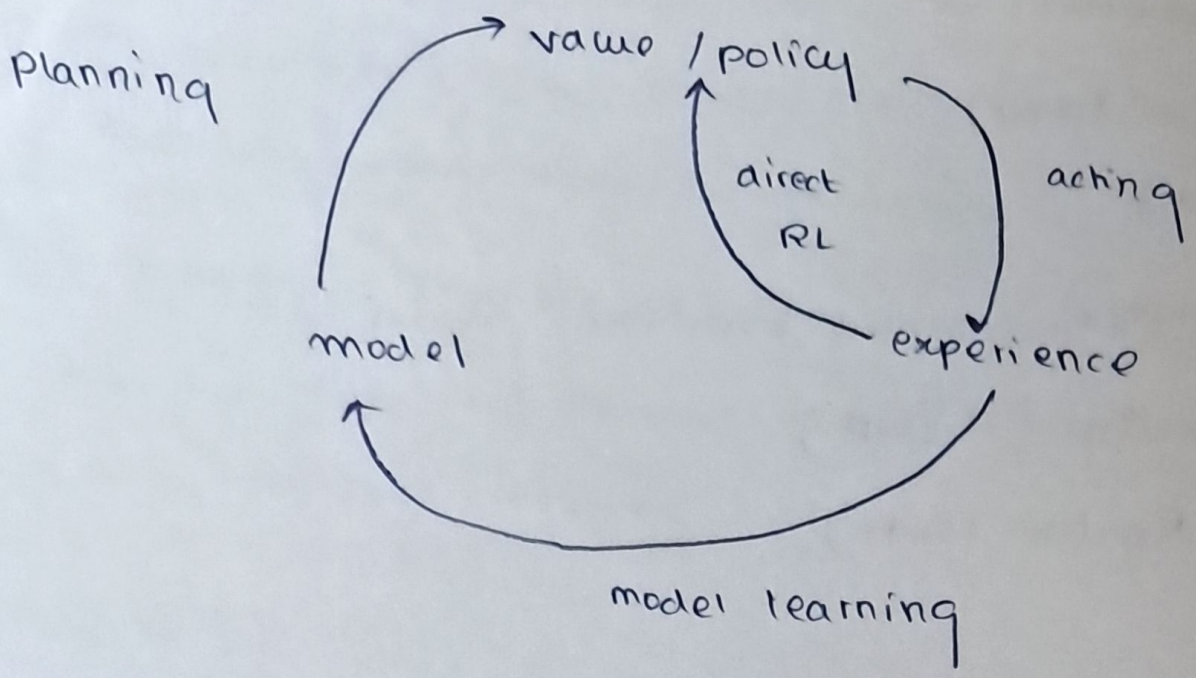
* Planning and Learning in RL

→ Planning involves the agent using a model of the environment to simulate interactions and update its policy w/o directly interacting with the environment

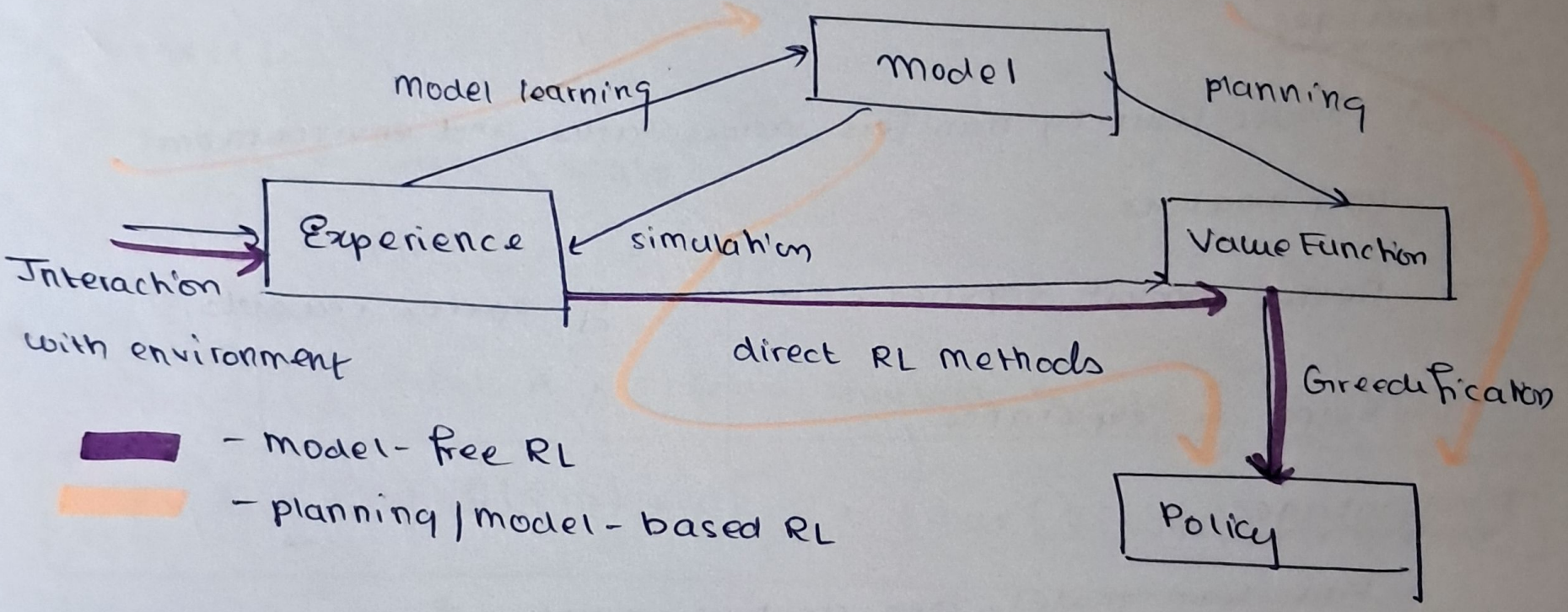
eg. Value Iteration, Policy Iteration

→ Learning involves the agent interacting with the environment to improve its knowledge about the reward structure / transition dynamics . eg. Q-Learning

Basic Structure



Model-free vs. Model-based RL



* Features of Planning / Model-based RL

Combine real and simulated experience

1. If the model is unknown, learn the model
2. Learn value functions using both real experience & the model
3. Learning value functions online using model-based lookahead search

i.e Let the real experience be:

$$S' \sim T(s'|s,a)$$
$$R = r(s,a)$$

and the simulated experience, sampled from the model is

$$s' \sim T_\eta(s'|s,A)$$
$$R = R_\eta(R|s,A)$$

* Advantages and Disadvantages of model-based RL

Advantages

1. Model learning transfers across tasks and environment configurations
2. Better exploits experience in case of sparse rewards
3. Helps in exploration

Disadvantage

1. First learn model, then construct a value function: Two sources of approximation error

* Dyna-Q

→ In dyna-Q, learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning.

Key Concepts

① Direct RL : Involves the agent updating its Q-values directly from real interactions with the environment.

② Model Learning? - The agent learns a model of the environment by storing the transitions it observed. This model can be deterministic or stochastic.

③ Planning? - Involves updating Q-values based on simulated experience generated from the learned model. This allows the agent to improve its policy without requiring additional interactions with the environment.

Algorithm

Initialize $Q(s,a)$ and $Model(s,a)$ for $\forall\ s \in S$ and $a \in A(s)$

Do forever:

(a) $S \leftarrow$ current state

(b) $A \leftarrow \varepsilon$-greedy(S,Q)

(c) Execute action A, observe resultant reward R and state S'

(d) $Q(S,a) \leftarrow Q(S,a) + \alpha[R + \gamma \max(Q(S',a)) - Q(S,A)]$

↳ Direct RL

(e) $Model(S,A) \leftarrow RS'$

↳ Model learning

(f) Repeat n times:

↳ Planning?

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow Model(S,A)$

$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max Q(S',a) - Q(S,A)]$

* Prioritized Sweeping

→ Simulating with random transitions might be very heavy in some applications.

→ Instead of performing uniform or random updates (as in Dyna-Q), Prioritized Sweeping selectively updates state-action pairs that are more likely to lead to significant changes in the value function

→ A queue is maintained of every state-action pair whose estimated value would change non-trivially if updated, prioritized by the size of the change.

→ Prioritized sweeping has been found to dramatically increase the speed at which optimal solutions are found in maze tasks, often by a factor of 5 to 10.

→ The priority is calculated based on the magnitude of the temporal difference (TD) - error

$$\text{priority}(S', A') = \left| R + \gamma \max_{a} Q(S', a) - Q(S, A) \right|$$

→ If the TD error is large, it indicates that the value estimate for (S', A') is significantly outdated, so it should be prioritized for updating

Steps in Prioritized Sweeping

1. Compute Priority - After observing a transition, and updating the value function for a state, action pair (S, A), the algorithm computes the priority for predecessor states that can transition into s via some action A'

2. Push into priority queue - The calculated priority is then pushed into the priority queue. The larger the TD-error, the higher the priority, meaning this state-action pair will be update sooner

3. Prioritized Updates - The planning loop then performs updates on state-action pairs with the highest priority first

Algorithm

Initialize $Q(s,a)$, $Model(s,a)$ for $\forall s,a$ and $PQueue$ to empty

Do forever:

(a) $S \leftarrow$ current state

(b) $A \leftarrow policy(S, Q)$

(c) Execute action A, observe reward R and state S'

(d) $Model(S, A) \leftarrow R, S'$

(e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$

(f) If $P > \theta$, then insert S, A into $PQueue$ with priority P

(g) Repeat n times, while $PQueue$ is not empty:

$S, A \leftarrow$ First (PQueue)

$R, S' \leftarrow$ Model (S, A)

$Q(S', A)' \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$

Repeat $\forall \bar{S}, \bar{A}$ predicted to lead to S':

$\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$

$P \leftarrow \left| \bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A}) \right|$

If $P > 0$ then insert $\bar{S}, \bar{A}$ into PQueue with priority P

Disadvantage of Prioritized Sweeping

→ It uses expected updates, which in stochastic environments may waste lots of computation on low-probability transitions.