UNIT - 4

Bootstrapping:

It is a technique used in RL to estimate the value of a state or state-action pair by using the agent's current estimate of the value funcⁿ.

→ This is done by updating the value funcⁿ using the TD error, which is the difference between the agent's current estimate value funcⁿ and the observed reward.

Why useful:

1. Efficiency
2. Scalability.
3. Better convergence
4. learn from incomplete information.
5. Improve exploration - exploitation trade-off.
6. Reduce the impact of noise & uncertainty.

How it works:

- estimating the value funcⁿ $V(s)$ which is the expected return from state s.
- Bootstrapping means that we use the current estimate of $V(s')$ (value of next state) to update $V(s)$
- This is typically done using TD learning, where updates happen in real-time as the agent moves through the environment.

TD learning:

- Includes TD(0), SARSA & Q-learning.
- general update rule in TDL for a state "s" at time "t" is

$$V(s) \leftarrow V(s) + \alpha \cdot (r + \gamma V(s') - V(s))$$

γ = discount factor

α = Alpha is learning rate

$V(s')$ = estimated next state

$V(s)$ = estimated current state

Algorithm uses in BS:

1. Q-learning
2. SARSA
3. Deep-Q-Networks (DQN)

} uses TD bootstrapping to update the action-value func?

Adv:

- Faster updates & more efficient learning.
- Provides more frequent learning signals.

Disadv:

- BS introduces "bias" because it relies on estimates rather than actual outcomes.
- This can make learning process less stable.

TD(0) Algorithm:

→ TD is a technique in RL that combines both monte carlo methods and Dynamic programming.

→ It is a model-free algorithm which does not require any information or knowledge of the environment

→ In this method Agent will updates its value estimates immediately after taking an action, based on the current state/reward & estimated value of the next state.

→ In TD(0) "0" represents single-step update. means the value of a state is updated based on the next immediate state's value, without looking further states.

→ TD(0) combines bootstrapping feature of DP and the sampling-based approach of MC.

→ It focuses only on-policy evaluation.

Objective:

Main goal of TD(0) is to estimate the value func? $v(s)$ for given policy π where

$$V(s) = E\pi[G_t | s_t = s] \qquad G_t = \text{return}$$

TD(0) update rule:

In this the value will be updated after each step by combining the current state "s" & next state "s'".

→ State value funcⁿ

$$\boxed{V(s) \leftarrow V(s) + \alpha \cdot (r + \gamma V(s') - V(s))}$$

$V(s)$ = estimate value of current state

$V(s')$ = " " " next "

γ = discount factor

α = Alpha is a learning rate

r = Immediate reward.

TD Error:

It represents the difference b/w current state and updated target. which has reward plus for next state.

→ TD error for a state sss after observing reward rrr & transitioning to state s' is

$$\boxed{\delta = r + \gamma V(s') - V(s)}$$

updated rule:

$$\boxed{V(s) \leftarrow V(s) + \alpha \cdot \delta}$$

Algorithm steps:

1. Initialize.

   $V(s)$ for all states s.

2. Repeat.

   - for each episode

     i. Take action — based on policy π & observe next state s' & reward r

     ii. compute TD error — $\delta = r + \gamma V(s') - V(s)$

     iii. update the value : $V(s) \leftarrow V(s) + \alpha \cdot \delta$

iv. Move to next state s'

5. Repeat until convergence.
(or) for a no. of episodes are completed.

characteristics of TD(0):

- on-policy
- Incremental updates
- Bootstrapping

Benefits:

- Efficiency
- suitability for continuous task
- less memory requirement

limi:

- Dependency on Discount factor "γ."
- Bias in Estimation.

Applic.

1. Robotics / Robot Navigation
2. game playing

Convergence of monte carlo & batch TD(0) Algorithm:

MC & TD are fundamental methods in RL for estimating value funcⁿ.
both algorithms are evaluate the value funcⁿ $V\pi(s)$ for a given policy π.

Monte carlo:

Imagine playing a game and making moves randomly.

→ After finishing the game, you look at the outcome and assign values to each move based on whether it led to a win or loss.

→ you update your strategy for each move based on the overall result of the entire game.

convergence of MC:

update:

$$V(s) \leftarrow V(s) + \alpha \cdot (G_t - V(s))$$

G_t = observed reward from state s.

key convergence:

1. consistency 2. weakness in Bootstrapping 3. unbiased estimate

batch TD(0):

In this Algorithm Instead of waiting until the end of the game you update your strategy after each move.

→ you estimate the value of each move by comparing it to the value of the next move.

→ This helps you learn and adjust your strategy as you go through the game.

update rule:

TD error is $V(s) \leftarrow V(s) + \alpha \cdot \delta t \; V(s)$

$$\delta t = r + \gamma V(s') - V(s)$$

Batch mode:

- TD(0) updates are applied after collecting all state transitions in a batch.
- Batch TD(0) minimizes the MSE mean square error b/w the predicted value V(s) & the BS target

$$MSE = \sum_{s} [r + \gamma V(s') - V(s)]^2$$

convergences of batch TD(0):

1. fixed-point convergence
2. linear equ^n framework
3. conditions for convergence.
4. Faster updates.

MC	batch TD(0)
1. It is better for episodic tasks where episodes are short & complete return information is readily available.	1. It is better for continuous tasks on long tasks.
2. It is computationally expensive & slow in practice	2. It often faster, but with a slight bias introduced by bootstrapping.

Model-Free ~~Algorithm~~ / control:

MFA is a type of RL Algorithm that does not require an explicit model of the environment to create an policy that maximizes the sum of future rewards.

features:

1. No model of the Environment.
2. Focus on optimal policy.
3. Exploration vs Exploitation
4. common Algorithms.
   - i. value-based methods
   - ii. policy-based "
   - iii. Actor-critic "

i: value based methods:

learn action-value func^n $Q(s,a)$ that estimates the expected return

→ it has two approaches

a. Q-learning:

- off-policy methods that learn $Q(s,a)$ using the bellman optimality equ^n.

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

updates are based on best possible action.

b. SARSA

- State - Action - Reward - State - Action
- on-policy method

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

ii. policy-based methods:

Directly learn the policy $\pi(a|s) \backslash pi(a|s) \pi(a|s)$ without estimating a value func^n

Reinforce Algorithm.

- A MC policy-gradient method:

$$\theta \leftarrow \theta + \alpha \nabla \theta \log f(1)\pi \theta (a_t|s_t) G_t$$

θ = policy parameter, G_t = reward

iii. Actor-critic method:

combine a value func^n with a policy (actor) for more stable updates

Adv:

1. Simplicity.
2. versatility.
3. Incremental learning

Chall.

1. stability
2. scalability

Appli

1. Robotics
2. Health care
3. Autonomous vehicles
4. games.

SARSA: (or) SARS'A'

State - Action - Reward - S' (Next State) - Action (for S')

-> It is an on-policy RL Algorithm that learns the action-value func^n Q(s,a) based on Agent's current policy.

update rule:

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

α = learning rate

γ = Discount factor

$Q(S',A')$ = Q value of next state & action.

→ SARSA updates the Q-value using action A' chosen according to the current policy making it on-policy.

Adv:

1. Balance exploration & exploitation.
2. Safer in environment

Limitations.

1. May converge slower than off policy methods like Q-learning

---

Expected SARSA.

It is a variant of SARSA that improves the stability of updates by using expected value of the Q-funcn at next state s' instead of relying on a single ~~chosen~~ action A'.

update rule:

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \sum_{a'} \underbrace{\pi(a'|s')}_{\substack{\text{probability of taking} \\ \text{action } a' \text{ in state} \\ s' \text{ under} \\ \text{current policy}}} Q(s',a') - Q(S,A)\right]$$

Adv:

1. more stable.
2. Improving converge in some scenarios.

limi.

1. more expensive than SARSA

SARSA	Expected SARSA
1. on-policy	1. on-policy
2. uses a single sampled action A'	2. uses expected value over all actions A'.
3. Higher variance	3. lower variance
4. less computationally expensive	4. more intensive.
5. slower in some cases	5. faster & more stable

Q-learning:

-> It is one of the most foundamental & widely used Algorithms in RL.

-> It is a model-free

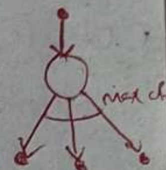
-> It is off-policy learning algorithm that can learn from other policies.

Q-learning is an off-policy LA that learns the optimal policy by directly estimating the optimal action-value func^n $Q^*(s,a)$ regardless of the current policy.

Action-value func^n:

$Q(s,a)$ represents reward (return), state s, taking action a.

$$\boxed{Q(s,a) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right]}$$



Block diagram of Q-lea

r_{t+k+1} = reward received at step $t+k+1$

γ = discount factor.

goal:

~~optima~~ To ~~learn~~ learn optimal action-value func^n $Q^*(s,a)$ such that agent can determine the optimal policy.

$$\boxed{\pi^*(s) = \arg\max_a Q^*(s,a)}$$

Q-learning Algorithm:

given state s, action a, Q-value is updated by following

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

α = learning rate

γ = discount factor

$Q(s,a)$ = current estimate Action-value func^n

$Q(s',a')$ = next state

$\max_{a'} Q(s',a')$ = maximum predicted Q-value for next state/action

r = immediate reward.

Steps:

1. Initialization:

- Initialize $Q(s,a)$

  State-action pair $= (s,a)$

  $\therefore Q(s,a) = 0$ Initially

- choose α & γ parameters

2. Interaction with environment:

- obseeue current state st
- select an action based on exploration strategy.

3. Execute the action:

peeform action "at", obseeue reward "rt" then

St + 1 S_{t+1}St + 1

4. update the Q-value:

After all these update Q-value

$$\boxed{Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha . [r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)]}$$

5. Repeat:

until it conveegence

Adv:	Limi:	Applica.
1. Mode-Free	1. Scalability	1. Game AI
2. off-policy	2. Exploration	2. Robotics
3. widely Applicable		3. Finance
		4. Healthcare.

Extensions of Q-Lea

1. Deep Q-learning – handle large datasets (or) continuous state spaces
2. Double Q-learning – Reduces overestimation.
3. Q-lambda
4.