REINFORCEMENT LEARNING

UNIT - I

Basics of probability and linear algebra, Definition of a stochastic multi-armed bandit, Definition ofregret, Achieving sublinear regret, UCB algorithm, KL-UCB, Thompson Sampling.

UNIT - II

Markov Decision Problem, policy, and value function, Reward models (infinite discounted, total, finitehorizon, and average), Episodic & continuing tasks, Bellman's optimality operator, and Value iteration& policy iteration

UNIT - III

The Reinforcement Learning problem, prediction and control problems, Model-based algorithm, MonteCarlo methods for prediction, and Online implementation of Monte Carlo policy evaluation

UNIT - IV

Bootstrapping; TD(0) algorithm; Convergence of Monte Carlo and batch TD(0) algorithms; Model-freecontrol: Q-learning, Sarsa, Expected Sarsa.

UNIT - V

n-step returns; TD(λ) algorithm; Need for generalization in practice; Linear function approximation and geometric view; Linear TD(λ). Tile coding; Control with function approximation; Policy search; Policy gradient methods; Experience replay; Fitted Q Iteration; Case studies.

REINFORCEMENT LEARNING

UNIT-1

Basics of probability and linear algebra, Definition of a stochastic multi-armed bandit, Definition of regret, Achieving sublinear regret, UCB algorithm, KL-UCB, Thompson sampling. Reinforcement - strengthening, increase, boosting

Artificial Intelligence (AI) refers to the simulation of human intelligence in machines that are programmed to think and learn like humans. Reinforcement - The process of encouraging a pattern of behaviour

It involves the development of computer systems or algorithms that can perform tasks that typically require human intelligence, such as visual perception, speech recognition, decision-making, problem-solving, and language understanding.

It enables machines to learn, understand natual language, and make decisions.

These systems can analyze data, recognize patterns, make predictions, and automate

various tasks, improving efficiency and accuracy in a variety of domains.

Artificial Intelligence has numerous applications across various fields, including healthcare, finance, transportation, manufacturing, entertainment, and more.

AI encompasses a broad range of techniques and approaches, including machine learning, neural networks, natural language processing, computer vision, expert systems, and robotics.

Machine Learning (ML) is a crucial component (subset) of AI, where algorithms are designed to learn from and make predictions or decisions based on data patterns.

There are different types of machine learning techniques, including

1) Supervised Learning: The model is trained on labeled data, where the input and the

desired output are provided. The model learns to map inputs to outputs, enabling it to make predictions on new, unseen data.

Example application

Bioinformatics Using finger prints to identify a person. To increase system security.

2) Unsupervised Learning The model learns patterns and structures in unlabeled data without any specific output labels. It aims to discover hidden patterns, group similar data objects into clusters.

Example application

customer and market segmentation clustering algorithms can help to group people that have similar traits and create customer personas for more efficient marketing and targeting campaigns.

3) Reinforcement Learning the model learns through interactions with an environment, (PTO)

persona - description of character.

receiving feedback in the form of rewards or penalties. It explores different actions and learns to maximize the cumulative reward over time.

(RL) Reinforcement learning is a type of machine learning method where an intelligent agent (computer program) interacts with the environment and learns to act within that.

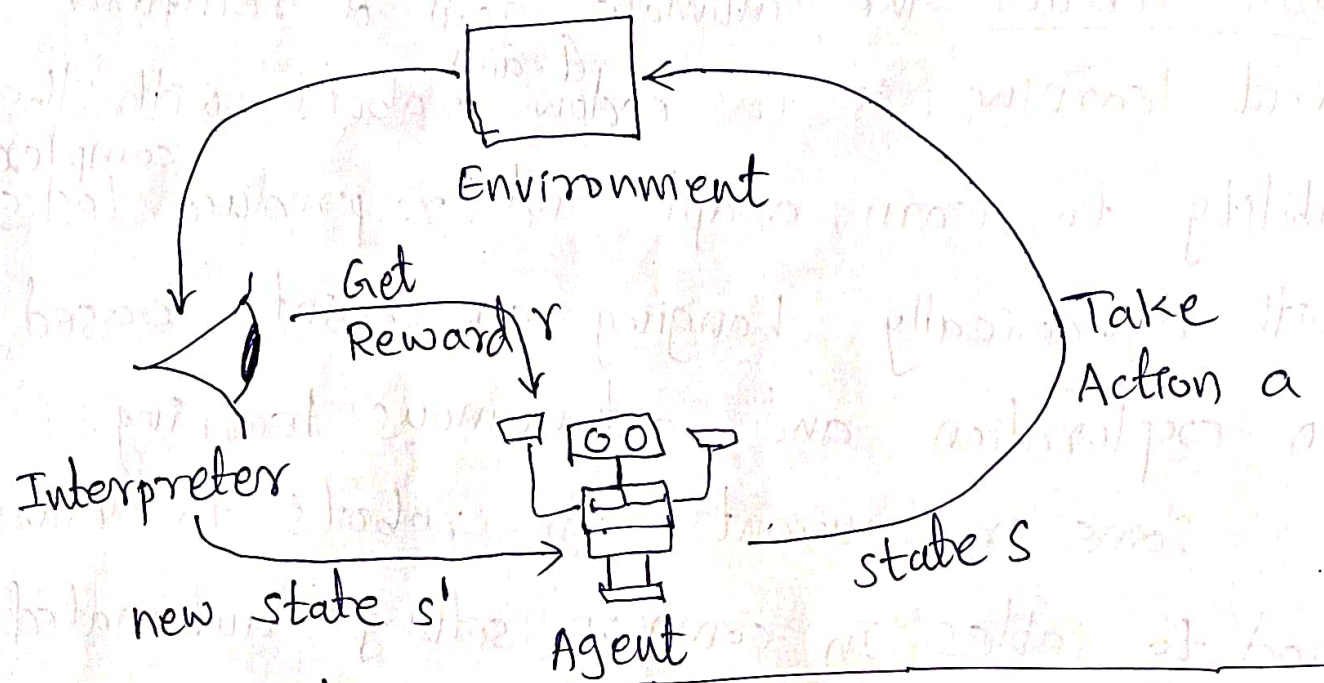
Example How a Robotic dog learns the movement of its arms.

RL is a core part of AI, and all AI agents work on the concept of RL. Here we do not need to pre-program the agent, as it learns from its own experience without any human intervention.

next page

* ML is the development of computer systems that are able to learn and adapt by using algorithms and statistical models to analyze and draw inferences from patterns in data.

adapt- become adjusted to new conditions



In RL, agents are trained on a reward and punishment mechanism. The agent is rewarded for correct moves and punished for the wrong ones. In doing so, the agent tries to minimize wrong moves and maximize the right ones.

Examples of RL

1) A toddler is sitting on the floor (it is his current state), makes an action, tries to stand up and gets a reward after this. The toddler doesn't know how to walk but he learns by trial and error

RL is based on trial (hit) and error process. (learners)

2) In robotics, the ultimate goal of reinforcement learning is to endow (train) robots with the ability to learn, adapt and reproduce complex tasks with dynamically changing constraints based on exploration and autonomous learning.

Some restaurants use robots to deliver food to tables. In common settings, automated robots have been used thus far to assemble products, inspect for defects, deliver goods, travel long and short distances, grasp and handle objects of all different shapes and sizes. As we continue to test robotic abilities, new features are being introduced (added) to expand their potential.

3) Autonomous vehicles: RL can be used to train self-driving cars, drones, or other autonomous vehicles. RL algorithms enable these vehicles to learn how to navigate roads, make decisions, and respond to different traffic scenarios by using sensors.

4) Game playing RL can be used to create intelligent game agents or opponents in video games. Agents can learn optimal strategies by playing against human players or other AI agents. providing a challenging gaming experience.

Terms used in RL

1) Agent: An entity that can perceive/explore the environment and act upon it. The agent is not instructed about the environment and what actions need to be taken. RL is based on the hit (trial) and error learning process. The agent takes the next action and changes states according to the feedback of the previous action. Ex AI robot.

2) Environment: A situation in which an agent is present or surrounded by. In RL we assume the stochastic environment which means it is random in nature. Ex a room, maze, football ground.

An agent needs to explore the environment.

3) Action: Actions are the moves taken by an agent within the environment

4) State: state is a situation returned by the environment after each action taken by the agent.

5) Reward: A feedback returned to the agent from the environment to evaluate the action of the agent.

6) Policy: Policy is a strategy applied by the agent for the next action based on the previous action and current state.

7) Value: It is expected long-term reward returned with the discount factor and opposite to the short-term reward.

8) Q-value: It is mostly similar to the value but it takes one additional parameter of a current action. Q-values or action-values are defined for states and actions. It is an estimation of how good is it to take the action at that state. Q-values are used to iteratively improve the behavior of the learning agent.. (performance)

Elements of Reinforcement Learning

Beyond the agent and environment, there are four main elements of RL. They are

1. A Policy
2. A Reward function (signal)
3. A value function
4. A model of the environment

1) Policy A policy can be defined as a way of how an agent behaves at a given time. It maps the perceived states of the environment to the next actions to be taken when in those states. A policy is the core element of the RL as it alone can define the behavior of the agent. In some cases, it may be a simple function or a lookup table. In some other cases, it may involve general computation as a search process. It could be deterministic or a stochastic policy.

For deterministic policy: $a = \pi(s)$.

a – next action

s – current state π – policy function

probabilistic or unpredictable.

For stochastic policy

$$\pi\left(\frac{a}{s}\right) = P\left(A_t = a / s_t = s\right)$$

A_t – action at time t
S_t – state at time t

2) Reward function A reward function defines the goal in a reinforcement learning problem. At each state, the environment sends an immediate signal to the learning agent, and this signal is known as a reward signal. These rewards are given according to the good and bad actions taken by the agent. The main objective of agent is to maximize the total no. of rewards received for good actions in the long-run. The reward signal can change the policy, such as if an action selected by the agent leads to low reward, then the policy may change to select other actions in the future.

3. Value function

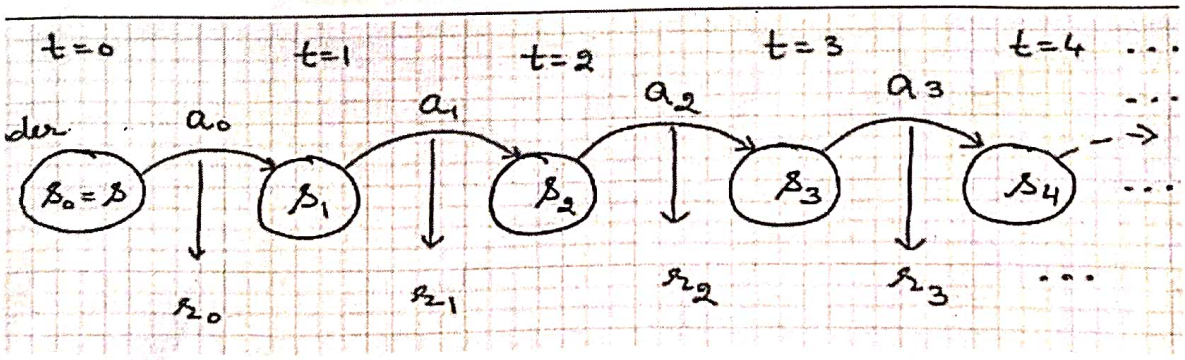
A reward function indicates what is good in an immediate sense(future), whereas a value function specifies what is good in the long-run.

The value of a state($V_t(s)$) is the total amount of reward an agent can expect to accumulate over the future starting from that state **s.**

Time steps – $t, t+1, t+2, \ldots$
States – $s_t, s_{t+1}, s_{t+2}, \cdots$
Rewards – $R_t, R_{t+1}, R_{t+2}, \ldots$

$$V_t(s) = R_t + R_{t+1} + R_{t+2} \ldots$$

For example, a state s_t might always yield a low immediate reward R_t, but still has a high value because it is regularly followed by other states $s_{t+1}, s_{t+2}, \ldots$ that yield high rewards $R_{t+1}, R_{t+2}, \ldots$



3) Value function A reward function indicates what is good in an immediate sense (future). whereas a value function specifies what is good in the long run. The value of a state is the total amount of rewards an agent can expect to accumulate over the future starting from that state. For example a state might always yield a low immediate reward, but still has a high value because it is regularly followed by others states that yield high rewards.

4) Model of the environment mimics the behavior of the environment. With the help of the model one can make inferences about how the environment will behave. Such as, if a state and an action are given, then the model can predict the next state and next reward. Models are used for planning, by which we mean any way of deciding on a course of action by

considering possible future situations before they ~~are~~ actually experiencing those situations. The approaches for solving the RL problems with the help of the model are termed as the model-based approach. comparatively, an approach without using a model is called a model-free approach.

BASICS OF PROBABILITY AND LINEAR ALGEBRA FOR RL

To understand RL concepts, it is helpful to have a basic understanding of probability theory and linear algebra.

Probability Theory

1) Probability: Probability is the measure of the likelihood of an event occurring. It ranges from 0 (impossible) to 1 (certain). In RL, probabilities are often used to represent uncertainty, such as the likeli hood of transitioning from one state to

Basics of Probability and Linear Algebra for RL

To understand RL concepts, one should have a basic understanding of probability theory and linear algebra

Probabilty Theory

1. Probability: Probability is the measure of the likelihood of an event occurring.
It ranges from 0(impossible) to 1(certain)
In RL, probabilities are often used to represent uncertainty, such as the likelihood of transitioning from one state to another state or the Probability of receiving a certain reward.

The probability formula is defined as the possibility of an event to happen is equal to the

ratio of the number of favourable outcomes and the total number of outcomes.

Probability of event(E) to happen (occur)

P(E) = Number of favourable outcomes / Total Number of outcomes.

For example, if you throw a die, then

the probability of getting 1 is 1/6.

Similarly, the probability of getting all the numbers from 2,3,4,5 and 6, one at a time is 1/6.

If you toss a coin, then we will get the outcome as head or tail.

the probability of getting head is = ½ and
the probability of getting tail is = ½

- **Conditional probability** is defined as the likelihood of an event(B) or outcome occurring, based(depending) on the occurrence of a previous event(A) or outcome.

This probability is written P(B|A), notation for the probability of B given A.
Conditional Probability Formula

P(B|A) = P(A and B) / P(A) = P(A∩B) / P(A)

- **In the case** where events **A and B are independent** (where event A has no effect on the probability of event B), the conditional probability of event B given event A is simply the probability of event B, that is

P(B|A)=P(B)

2. Random variables: A random variable is a variable whose value is determined by the outcome of a random event.

In RL, random variables can represent states, actions, rewards, and other quantities that are subject to uncertainty.

Example of a Random Variable

If the random variable Y is the number of heads we get from tossing two coins, then Y could be 0, 1, or 2.
This means that we could have no heads, one head, or both heads on a two-coin toss.
However, the two coins land in four different ways: TT, HT, TH, and HH.

3. Probability distributions: A probability distribution describes the likelihood of different outcomes for a random variable.

In RL, probability distributions are used to represent the uncertainty associated with various events, such as the probability of different actions or the distribution of events.

Probability distribution of rolling a die
Throw a dice and you will get six independent probabilities.
Explanation:
P(1)=1/6
P(2)=1/6
P(3)=1/6
P(4)=1/6
P(5)=1/6
P(6)=1/6

Probability Distribution

Probability distributions are a summary of the probabilities of all possible outcomes of an experiment or situation, known as a random variable.

For example, this table shows the probability distribution for a 4-sided spinner obtained after an experiment.

Colour	Red	Blue	Green	Yellow
Probability	0.3	0.35	0.15	0.2

These probabilities are not all the same but **sum to 1.**
0.3 + 0.35 + 0.15 + 0.2 = 1.

THIRD SPACE LEARNING

another state or the probability of receiving a certain reward.

2) Random variables: A random variable is a variable whose value is determined by the outcome of a random event. In RL random variables can represent states, actions, rewards, and other quantities that are subject to uncertainty.

3) Pobability Distributions A probability distribution describes the likelihood of different outcomes for a random variable. In RL, probability distributions are used to represent the uncertainty associated with various events, such as the probabilities of different actions or the distribution of events.

Linear Algebra

Linear algebra is the branch of mathematics concerning linear equations such as

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b.$$

Linear maps such as

$$(x_1, x_2, \cdots, x_n) \rightarrow a_1x_1 + a_2x_2 + \cdots + a_nx_n$$

and their representations in vector spaces and through matrices.

1) Vectors: Vectors are objects that represent both magnitude and direction. In RL, vectors are often used to represent states, actions, and other quantities. They can be added, subtracted, and multiplied by scalars.

2) Matrices: are used to represent transition probabilities, rewards, and value functions.

3) Matrix operations such as addition, subtraction, multiplication and inversion are used in RL, to update value functions, calculate state-action values, and perform other computations.

4) Eigenvectors and Eigenvalues can be used in algorithms like Principal Component Analysis (PCA) to extract important features from states or to perform dimensionality reduction.

In linear algebra, an eigenvector or characteristic vector of a linear transformation is a nonzero vector that changes at most by a scalar (λ) factor when that linear transformation is applied to it.

The corresponding eigenvalue denoted by λ, is the factor by which the eigenvector is scalled.

Geometrically, a transformation matrix rotates, stretches, or shears the vector it acts upon. (stretches → making longer or wider)

→ If V is a vector then
v is called an eigenvector of a linear transformation T
if T(V) is a scalar multiple of V.

This can be written as $\boxed{T(V) = \lambda V}$

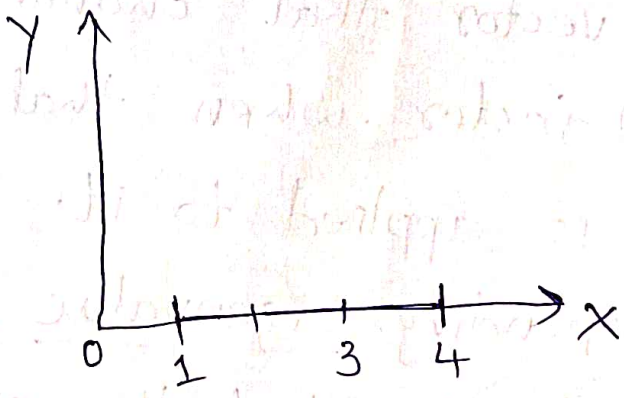
shears - larger scissors, shearing - cutting

where λ is an eigenvalue (characteristic value) associated with V.

Ex If $V = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$ and $\lambda = 20$ then

$$T(V) = \lambda V$$

$$\begin{bmatrix} 20 \\ 60 \\ 80 \end{bmatrix} \leftarrow = 20 \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$$



T(V) — Transformation of vector V.

T(V), V

$\begin{bmatrix} 20 \\ 60 \\ 80 \end{bmatrix}$ and $\begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$ are scalar multiples of each other, or parallel or colinear.

DEFINITION OF A STOCHASTIC MULTI-ARMED BANDIT PROBLEM

In reinforcement learning, a stochastic multi armed bandit problem (n-armed or K-armed bandit problem so named by analogy to a machine with n levers or arms) refers to a problem where an agent interacts with an environment [or a machine] by selecting a set of actions [or arms or hitting levers] or choices.

Each action has an associated unknown reward. These reward distributions are stochastic (probabilistic), meaning that the rewards obtained from selecting an action (arm) may vary [across different interactions] from one selection to another selection.

The agent's objective is to learn a policy that maximizes the total reward obtained over a sequence of actions.

Another analogy is that of a doctor choosing between experimental

treatments for a series of seriously ill patients. Each action is the selection of a treatment, and each reward is the survival or well-being of a patient.

Each of the k actions has an expected reward that is called the value of that action.

A_t – action selected on time step t

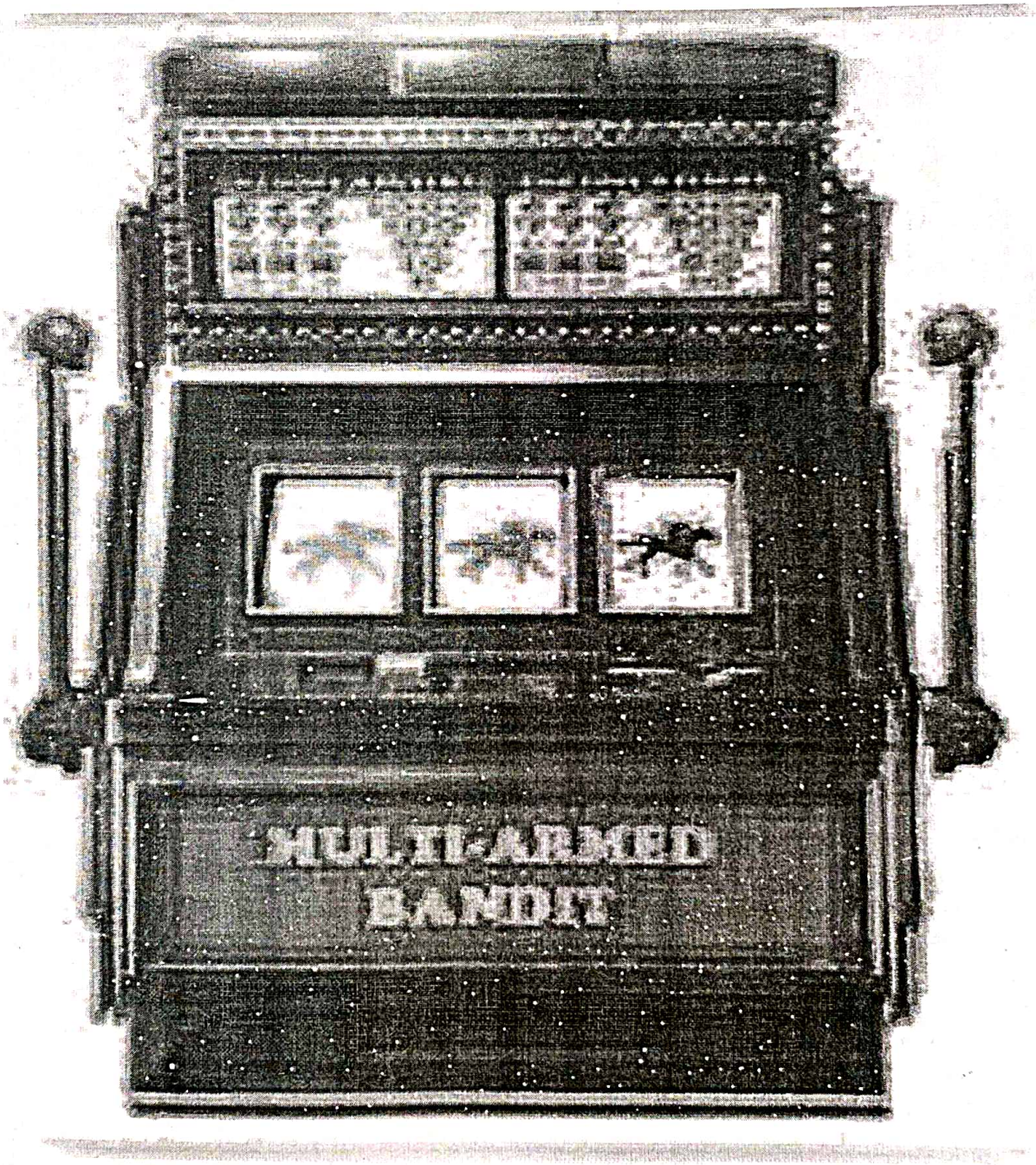
R_t – the corresponding reward.

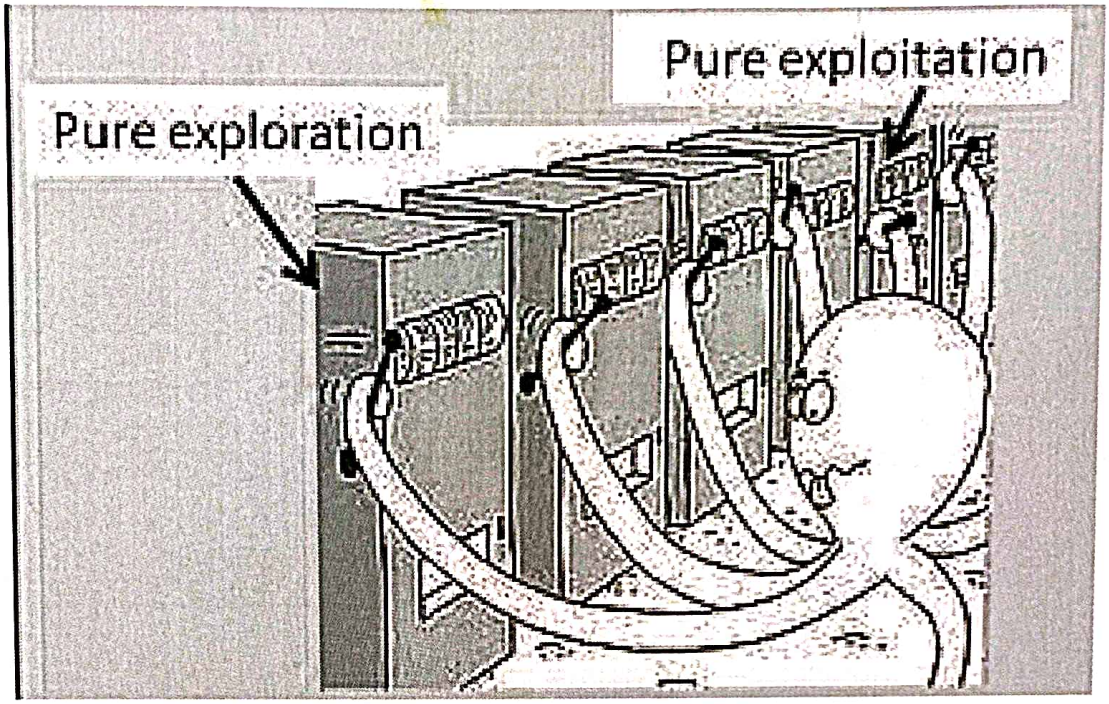
$q_*(a)$ – the value of an arbitrary action a, is the expected reward given that a is selected

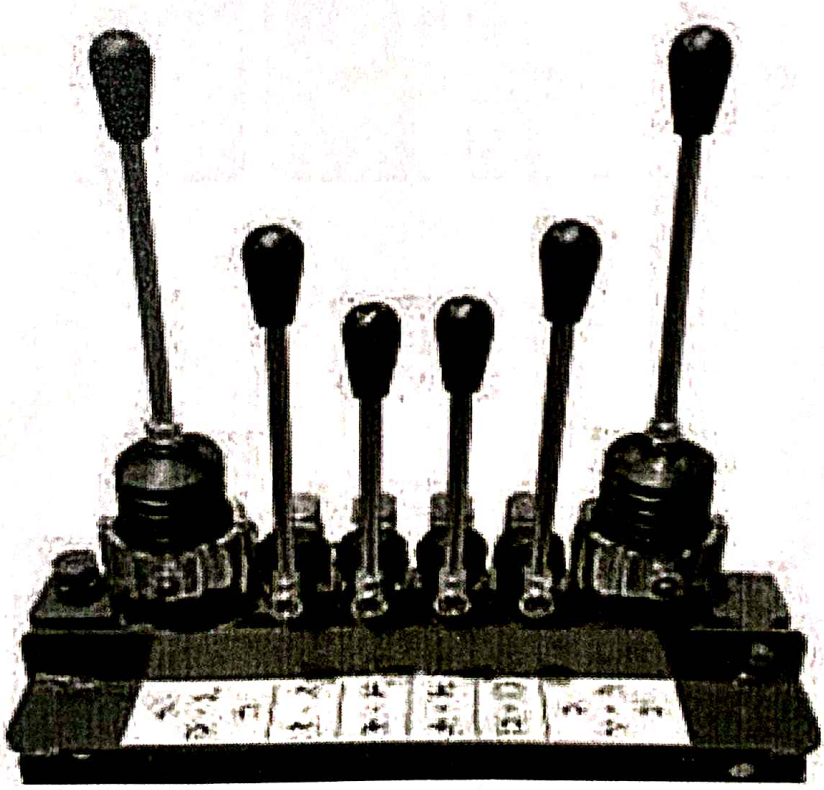
The value of action a, $q_*(a) = E[R_t / A_t = a]$

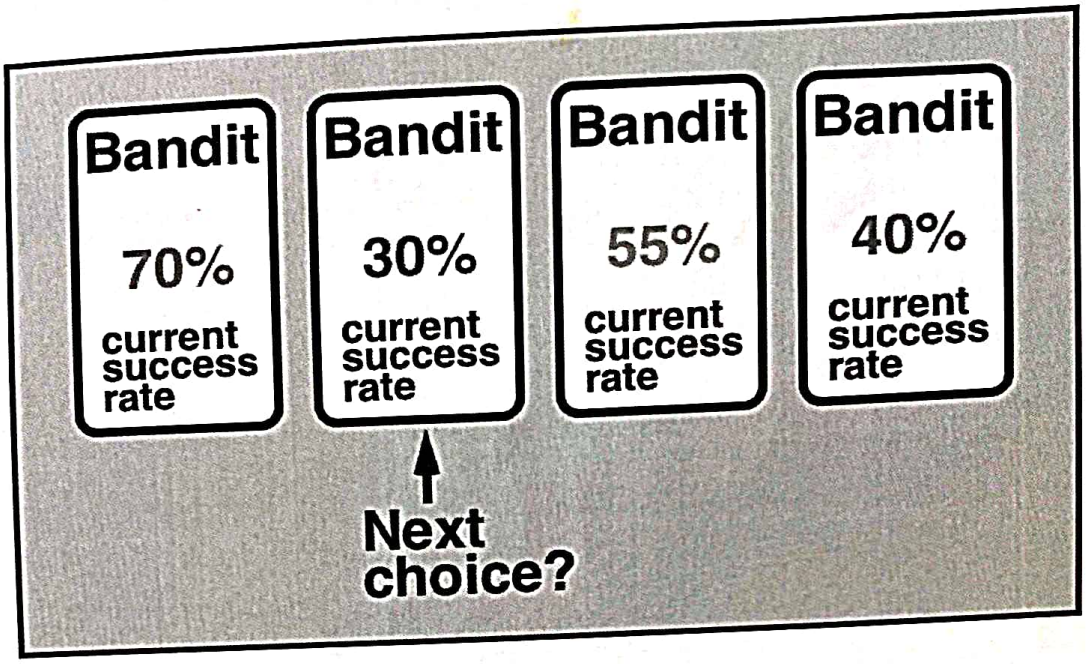
If you (the agent) knew the value of each action, then it would be trivial to solve the k-armed bandit problem: you would always select the action with highest value.

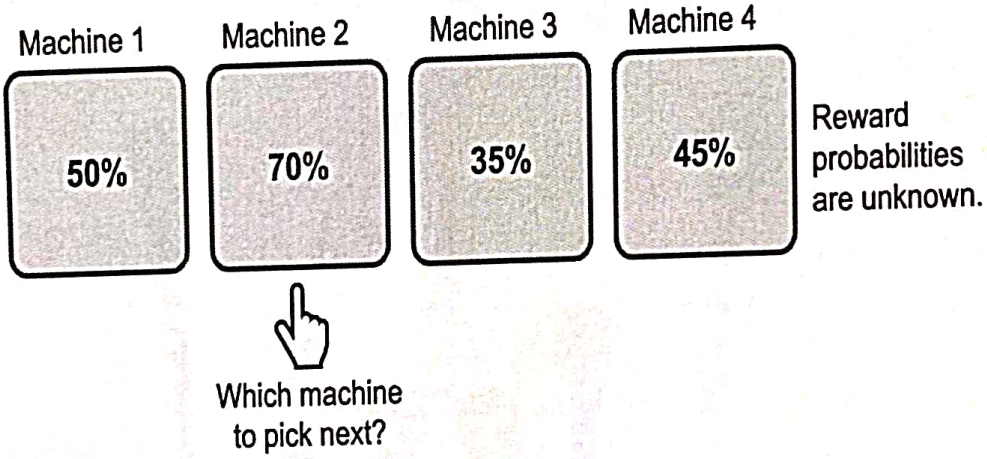
We assume that the agent do not know the action values with certainty, although you may have estimates.


MULTI-ARMED
BANDIT


Pure exploitation
Pure exploration




Bandit
70%
current success rate
Bandit
30%
current success rate
Bandit
55%
current success rate
Bandit
40%
current success rate
Next choice?


Machine 1
Machine 2
Machine 3
Machine 4
50%
70%
35%
45%
Reward probabilities are unknown.
Which machine to pick next?

We denote the estimated value of action a at time step t as $Q_t(a)$.
We would like $Q_t(a)$ to be close to $q_*(a)$.

The agent applies exploration-exploitation strategies. During the initial interactions, the agent explores all different actions to gather information about their reward distributions. Next the agent exploits the environment by selecting actions that have given higher rewards in the past.

At any time step there is at least one action whose estimated value is greatest. These actions are called greedy actions. When you select one of these actions, we say that you are exploiting your current knowledge to get maximum reward for the short-term benefit.

If instead you select one of the nongreedy actions, then we say you are exploring the environment, because this enables you to improve your estimate of the nongreedy action's value. Exploration may produce the greater total reward (a sequence of actions that depend on one another and their values) in the long-run.

As the agent continues to interact with the environment by applying RL techniques, it learns and updates its beliefs or estimates about the reward distributions of each action based on the observed rewards. The agent then uses these estimates to make informed decisions on which action (arm) to select in subsequent interactions with the environment (machine).

Action-Value methods

The true value of an action is the mean reward when that action is selected, which is calculated [estimated] by averaging the rewards actually received.

$$Q_t(a) = \frac{\text{sum of rewards of action } a, \text{ when } \underline{a} \text{ taken (selected) prior to time } t}{\text{No. of times action } \underline{a} \text{ taken or selected prior to } t.}$$

$$= \frac{\sum_{i=1}^{n} R_i}{n}$$

We call this the sampling-average method for estimating action values.

If we select one of the actions (a) with the highest estimated value. That is the greedy action selection method

$$A_t = \arg\max_a Q_t(a)$$

Exploits current knowledge to maximize immediate reward.

A simple alternative is to nongreedy actions (ε-greedy action selection) with some probability ε, randomly from among all the actions to see if they might be better.

Then the probability of selecting an optimal (max value giving) action is

$= 1 - \epsilon$

Incremental Implementation

With constant memory and constant per-time-step computation

To simplify notation we concentrate on a single action. Let

R_i – reward received after the ith selection of this action

Q_n – the estimate of its action value after it has been selected $(n-1)$ times.

which can be written as

$$Q_n = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}$$

Incremental Formulas

Given Q_n and the n^{th} reward R_n, the new average of all n rewards can be computed by

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i$$

$$= \frac{1}{n}\left[R_n + \sum_{i=1}^{n-1} R_i\right]$$

$$= \frac{1}{n}\left[R_n + (n-1)\frac{1}{(n-1)}\sum_{i=1}^{n} R_i\right]$$

$$= \frac{1}{n}\left[R_n + (n-1)Q_n\right]$$

$$= \frac{1}{n}\left[R_n + nQ_n - Q_n\right]$$

$$Q_{n+1} = Q_n + \frac{1}{n}\left[R_n - Q_n\right] \quad \text{— (1)}$$

which holds even for $n=1$.

obtaining $Q_2 = Q_1 + \frac{1}{1}\left[R_1 - Q_1\right] = Q_1 + R_1 - Q_1 = R_1$.

This implementation requires memory only for Q_n and n, and only the small computation (1) for each new reward.

This updated rule (1) is of a form that occurs frequently throughout this subject.

The general form is

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}\left[\text{Target} - \text{OldEstimate}\right]$$
$$Q_{n+1} \qquad\qquad Q_n \qquad\qquad \frac{1}{n} \qquad\qquad R_n \qquad\qquad Q_n$$

A simple bandit algorithm

Initialize, for action $a = 1$ to K

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever

$$A \leftarrow \begin{cases} \text{argmax}_a Q(a) & \text{with probability } 1-\epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$$

$R \leftarrow \text{bandit}(A)$ $\qquad$ A – action

$N(A) \leftarrow N(A) + 1$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A}\left[R - Q(A)\right]$$

The function bandit(a) is assumed to take an action and return a corresponding reward.

DEFINITION OF REGRET.

Regret refers to a metric used to evaluate the performance of an agent's decision-making process.

Regret is defined as the difference between

$$\text{Regret} = \left\{\begin{array}{l}\text{The cumulative (total)} \\ \text{reward obtained} \\ \text{by an optimal} \\ \text{(max reward giving)} \\ \text{policy}\end{array}\right\} - \left\{\begin{array}{l}\text{The cumulative} \\ \text{reward obtain} \\ \text{ed by the} \\ \text{agent's} \\ \text{policy}\end{array}\right\}$$

over a time horizon

It quantifies how much the agent could have improved its performance if it had always chosen the best possible actions at each step.

Regret is commonly used to assess the performance of RL algorithms and compare different approaches. Minimizing regret is a desirable objective for an agent, as it indicates that the agent is making decisions closer to the

optimal policy and achieving higher cumulative rewards.

ACHIEVING SUBLINEAR REGRET.

In mathematics, sublinear refers to a function or growth rate that is slower than linear growth.

Ex Linear growth rate $y = mx + c$

$y = 3x + 1$

x	y
1 → +1	4 → +3
2 → +1	7 → +3
3 → +1	10 → +3
4 → +1	13 → +3
5 → +1	16 → +3
6	19

Achieving sublinear regret in RL means that, as the agent learns and gains more experience and interacts with the environment for a longer period, the regret (difference) reduces or decreases at a rate slower than linear. It implies that the agent's policy is approaching the optimal policy, and the gap between the

agent's performance and the best possible (optimal) performance is narrowing (decreasing).

In RL, there are various techniques and algorithms for achieving sublinear regret. some of the commonly used methods.

1) Exploration-Exploitation trade-off [exchange or swap]: Balancing exploration (trying out different actions to gather information) and exploitation (using the knowledge gained so far to maximize rewards) is crucial for sub linear regret. Techniques like ϵ-greedy, UCB bounds, and Thompson sampling help strike the right balance.

2) optimism in the face of uncertainty:

Agents can incorporate optimism in their decision-making process by assigning higher values or rewards to uncertain actions. optimistic algorithms, such as optimistic initial values and optimism in the face of unknown payoffs, encourage

exploration and can lead to sublinear regret

3) Model-based approaches: Building a model of the environment allows the agent to simulate different actions and their consequences. This helps in planning and making informed decisions, reducing the regret. Techniques like Monte Carlo Tree Search (MCTS) and Model Predictive Control (MPC) fall under this category.

4) Function Approximation: Using function approximation methods, such as neural networks, allows the agent to generalize its knowledge across states and actions. This enables more efficient learning and can lead to sublinear regret.

UCB ALGORITHM

UCB - Upper Confidence Bound algorithm

It aims to strike a balance between exploration and exploitation by assigning upper confidence bounds to the estimated values of different actions. This is most widely used technique in RL and multi-armed bandit problems.

Here is how the UCB algorithm works

1) Initialization

- Initialize the estimated value $Q(a)$ and the selection count $N(a)$ for each action a.
- set the time step t to 1.

2) Action selection

- For each action a, calculate UCB by using the formula

$$UCB(a) = Q(a) + c \times \sqrt{\frac{\log(t)}{N(a)}}$$

exploit — explore

where

$Q(a)$ - is the estimated value of action a.

$N(a)$ – no. of times action a has been selected.

t – the current time step.

c – is a constant that determines the trade-off between exploration and exploitation.

- Select the action with the highest UCB value.

$$a^* = \text{argmax}(UCB(a))$$

3) Environment Interaction

- Execute action a^* and observe the reward r
- Update the estimated value $Q(a^*)$ and selection count $N(a^*)$ for the selected action.

$$Q(a^*) = \frac{[(N(a^*)-1) * Q(a^*) + r]}{N(a^*)}$$

Increment $N(a^*)$ by 1.

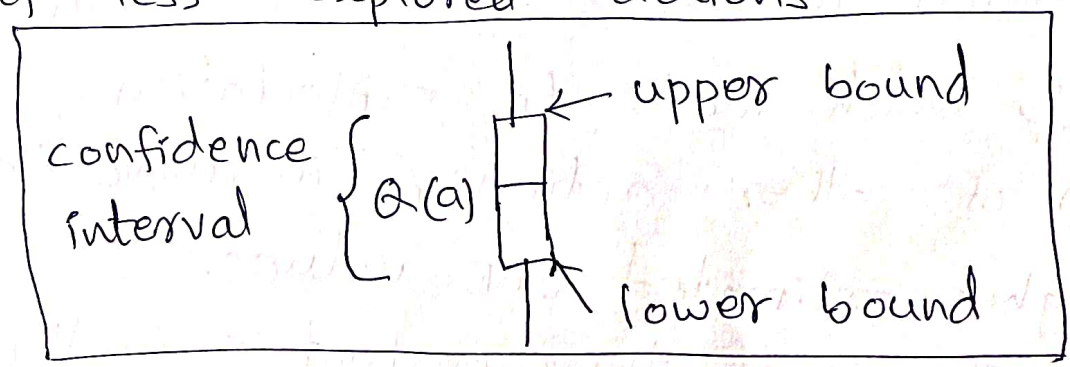
4) Repeat steps 2 and 3

Increment the time step t and continue selecting actions based on their UCB values and updating the estimated values and selection counts accordingly.

The UCB algorithm applys the concept of optimism in the face of uncertainty.

By considering UCB bound in the action

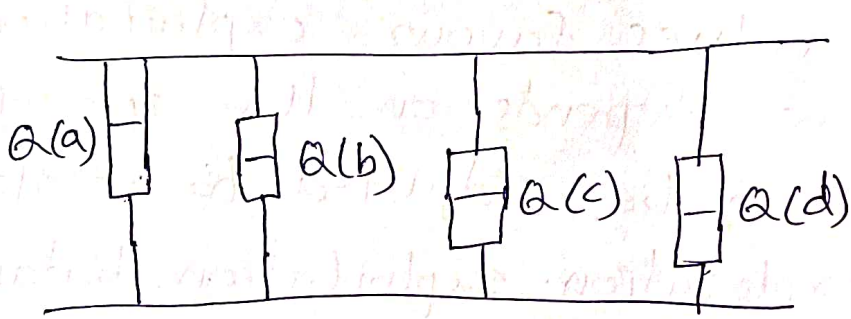
selection process, it encourages exploration of less explored actions.



From step 2) of UCB algorithm

$$UCB(a) \propto \frac{1}{N(a)}$$

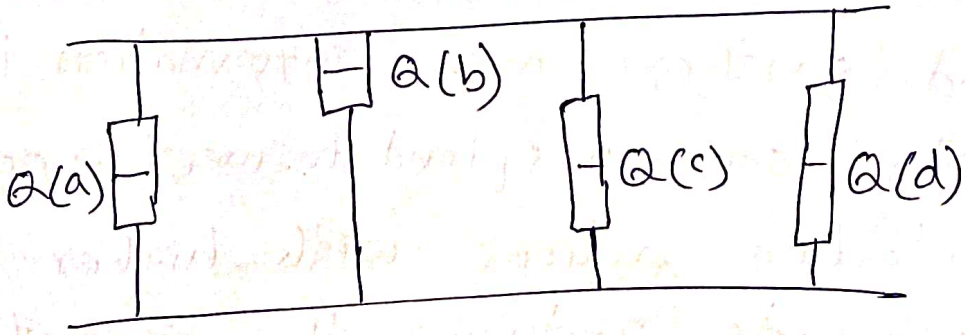
→



action a will be selected.

Initially, when an action has been selected fewer no. of times [N(a) is less], ~~and~~ then its UCB value is high, promoting its (that action's) selection.

→



action b will be selected since it's UCB is high

However, as the no. of selections (N(a)) of an action increases, its UCB value decreases, which leads to exploitation (selection) of other actions ('b' in above fig) with higher estimated UCB values.

The choice of the constant c in the UCB formula determines the level of exploration

Higher values of c lead to more exploration, while lower values favour exploitation. The value of c depends on the specific problem and can be adjusted to strike the desired exploration-exploitation balance.

The UCB algorithm achieves sublinear regret, means the regret [Total optimal reward - Total reward obtained by agent], grows slower than linearly with no. of time steps.

As the agent interacts with the environment and garthers more information, the UCB algorithm converges [tend to meet a point] towards selecting actions with higher expected rewards, reducing the regret over time.

KL-UCB Algorithm

KL- Kullback-Leibler

The KL-UCB algorithm is designed to solve multi-armed bandit problems, where an agent faces a set of actions(arms) and needs to choose(select) actions over time to maximize the cumulative reward. KL-UCB aims to balance exploration of uncertain(less frequently occurring or selected actions that give less reward) actions with exploitation of actions that seem promising(actions that are occurring or selected more no. of times that give more reward) based on the available information.

The algorithm uses the KL divergence to measure the uncertainty between the empirical(observed) distribution of rewards and the unknown

true reward distribution for each action

$$KL\left(\text{true}^{\text{actual}} \text{ reward of action } a \;\middle\|\; \text{observed reward of action } a\right) \quad (25\text{-}21)$$

It then adjusts the upper confidence bounds for actions based on this measure.

Kullback-Leibler (KL) divergence (difference) is a mathematical measure that calculates the difference between two probability distributions.

Given two probability distributions P and Q over the same (sample space, experiment), the KL divergence from P to Q is defined as follows

1. $KL(P||Q) = \sum P(x_i) \times \log\left[\frac{P(x_i)}{Q(x_i)}\right]$

for each $x_i \in$ sample space X

$X = \{x_1, x_2, \ldots, x_n\}$ outcomes

2. For continuous distributions

$$KL(P||Q) = \int P(x_i) \times \log\left[\frac{P(x_i)}{Q(x_i)}\right] dx.$$

Pseudo code of KL-UCB algorithm

1) Initialize for each arm (or action) a; a_1, a_2, a_3, a_4

$t_a = 1$ (time step for arm a)

$x_a = \text{sample\_reward}(a)$ [Initial estimate of mean reward]

$N_a = 1$ (No. of times arm a is pulled or action a is selected (executed))

2) Repeat for each time step t

For each arm a_i

- calculate the KL divergence

$$KL\left(\frac{x_a}{N_a}, \frac{x_{t_a}}{N_{t_a}}\right)$$

between the current empirical (observed) distribution and its updated version or actual. [after t_a steps]

- calculate upper confidence bound

$$U_a = \text{solveKL}\left[KL, \frac{\log(t)}{N_a}\right]$$

- choose the arm A_t with the highest upper confidence bound.

$$A_t = \text{argmax} U_a$$

- observe the reward R_t after selecting [performing or executing] action A_t
- Update estimates

$$t_a = t_a + 1$$
$$N_a = N_a + 1$$
$$X_a = \frac{(N_a - 1) X_a + R_t}{N_a}$$

- X_a (left): new updated reward value after N_a selections
- $(N_a - 1) X_a$: previous mean reward value of $(n-1)$ selections
- R_t: reward received in N_a^{th} selection.

- KL-UCB balances exploration and exploitation by adjusting the upper confidence bounds based on KL divergence.
- The choice of the function solveKL is important and depends on the specific RL problem and the properties of the reward distribution
- KL-UCB is more advanced than UCB in handling uncertain reward distribu tions, but it takes higher computational costs due to solving KL divergences.

THOMPSON SAMPLING

The same action can (may) give different rewards in different selections (executions) due to changing nature of environment

Ex Reward value distributions of

action a_1 $P(a_1) = \{15, 16, 7, 10, 11, 8, 12\}$ (29)

action a_2 $P(a_2) = \{20, 15, 18, 16, 21, 17, 25\}$. (49)

sample

a_2 will be selected

In Thompson sampling, the agent maintains a probability distribution (usually a Bayesian posterior distribution) for each arm (action), representing its beliefs about the true reward distribution of that action. At each time step, the agent samples (selects some reward sample values / draws) a reward estimate from each action's distribution and then selects the action with the highest sampled reward estimate (sum of sample values).

Thompson sampling is a heuristic for choosing actions that addresses the exploration-exploitation dilemma in the

multi-armed bandit problem. It consists of choosing the action that maximizes the expected reward w.r.t a randomly drawn belief. (randomly drawn sample).

Consider a set of contexts (states of agent) X, a set of actions A, and rewards $\mathbb{R}$. The aim of the (agent) player is to play actions under the various contexts, such as to maximize the cumulative rewards.

In each round, the player obtains a context (state) $x \in X$, plays an action $a \in A$ and receives a reward $r \in \mathbb{R}$. following a distribution that depends on the context and the issued action.

→ The elements of Thompson sampling are as follows

1) A likelihood function $P(r/\theta, a, x)$.
2) A set Θ of parameters θ for the distribution of r.

3) A prior distribution $P(\theta)$ on these parameters (reward)

4) Past observations triplets $D = \{(x, a, r)\}$.

5) A posterior distribution

$$\boxed{P(\theta/D) \propto P(D/\theta)\, P(\theta)}$$

where $P(D/\theta)$ is the likelihood function

Thompson sampling consists in playing the action $a^* \in A$ according to the probability that it maximizes the expected reward. Action a^* is chosen with probability

$$\boxed{\int [E(r/a^*, x, \theta) = \max_{a'} E(r/a', x, \theta)]\, P(\theta/D)\, d\theta.}$$

(E: expected)

The rule is implemented by sampling. In each round (selection or time step), parameters θ^* are sampled from the posterior $P(\theta/D)$ and an action a^* is chosen that maximizes $E[r/\theta^*, a^*, x]$.

That is, the expected reward given the sampled parameters (θ^*), the action a, and the current context x.

This means that the player (agent) instantiates their beliefs randomly in each round according to the posterior distribution, and then acts optimally according to them.

→ Prior probability represents what is originally believed before new evidence is introduced

P(A) = probability of event A occurring (prior probability)

P(B) = probability of event B occurring (evidence or marginal likelihood)

→ And posterior probability takes this new information into account.

P(A/B) = The probability of event A occurring, provided the evidence B.

$$P(A/B) = P(B/A)\,P(A)$$

UNIT-II

MARKOV DECISION PROBLEM (MDP)

A Markov Decision Process (MDP) is a mathematical framework used to model decision making problems with sequential actions in uncertain environments.

changing It consists of

- a set of states S
- a set of actions A
- transition probabilities $P()$
- rewards and R
- discount factor γ

At each state an agent selects an action based on a policy, the agent receives a reward and transitions to a new state.

The goal is to find an optimal policy that maximizes the expected cumulative reward over time

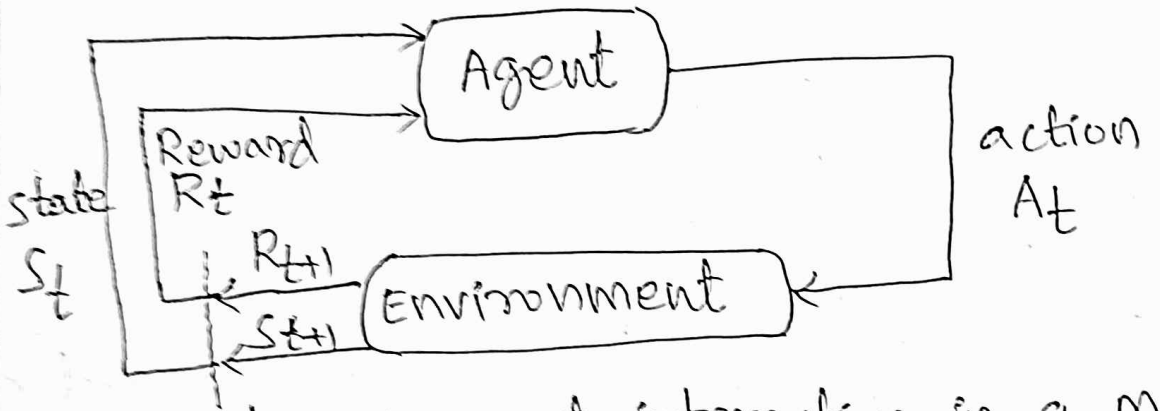
In bandit problems we estimate the value $Q_*(a)$ of each action a

But in MDPs we estimate the value $Q_*(s,a)$ of each action a in each state s.

The Agent-Environment Interface

MDPs – Frame the problem of learning from interaction to achieve a goal.

The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment.



The agent-environment interaction in a MDP

More specifically, the agent and environment interact at each of a sequence of discrete time steps $t = 0, 1, 2, 3, \ldots$

- At each time step t, the agent receives some representation of the environment's state $S_t \in S$
- and on that basis selects an action $A_t \in A(s)$.
- One time step later, the agent receives a numerical reward $R_{t+1} \in R$.
- and finds itself in a new state S_{t+1}.

The MDP and agent together give rise to a sequence or trajectory that begins like this.

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \quad \text{— (1)}$$

- For particular values of the random variables $s' \in S$ and $r \in R$, the probability of these (s', r) values occurring at time t, given particular values of the preceding state (S_{t-1}) and action (A_{t-1}) (next state, current state)

$$p(s', r \mid s, a) \doteq Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \quad \text{— (2)}$$

Function p defines the dynamics of the MDP.

$\doteq$ the dot indicates it is a definition not a fact.

The dynamics function $p : S \times R \times S \times A \rightarrow [0,1]$ is an ordinary deterministic function of four arguments.

The '|' in the middle of p specifies that it is a conditional probability

- In a finite MDP, the sets of states, actions and rewards (S, A, R) all have a finite no. of elements.

p specifies a probability distribution for each choice of s and a, that is

$$\sum_{s' \in S} \sum_{r \in R} p(s', r | s, a) = 1 \quad \text{for all } s \in S, a \in A(s) \quad -(3)$$

Markov property A state is said to have the markov property, if that state must include information about all aspects of the past agent-environment interaction that make a difference for the future.

Example MDP: Recycling Robot

A mobile robot has the job of collecting empty soda cans in an office environment. It has sensors for detecting cans, and arm and gripper that can pick them up and place them in an onboard bin. It runs on a rechargeable battery

set of states $S = \{high, low\}$

high - battery high

low - battery low.

In each state, the agent can decide whether to
1) search for a can
2) remains stationary and wait for someone to bring it a can
3) head back to its home base to recharge its battery.

Actions sets $A(high) = \{search, wait\}$

$A(low) = \{search, wait, recharge\}$

reward – No. of cans collected.

r_{search} – reward received after search

r_{wait} – reward received after wait.

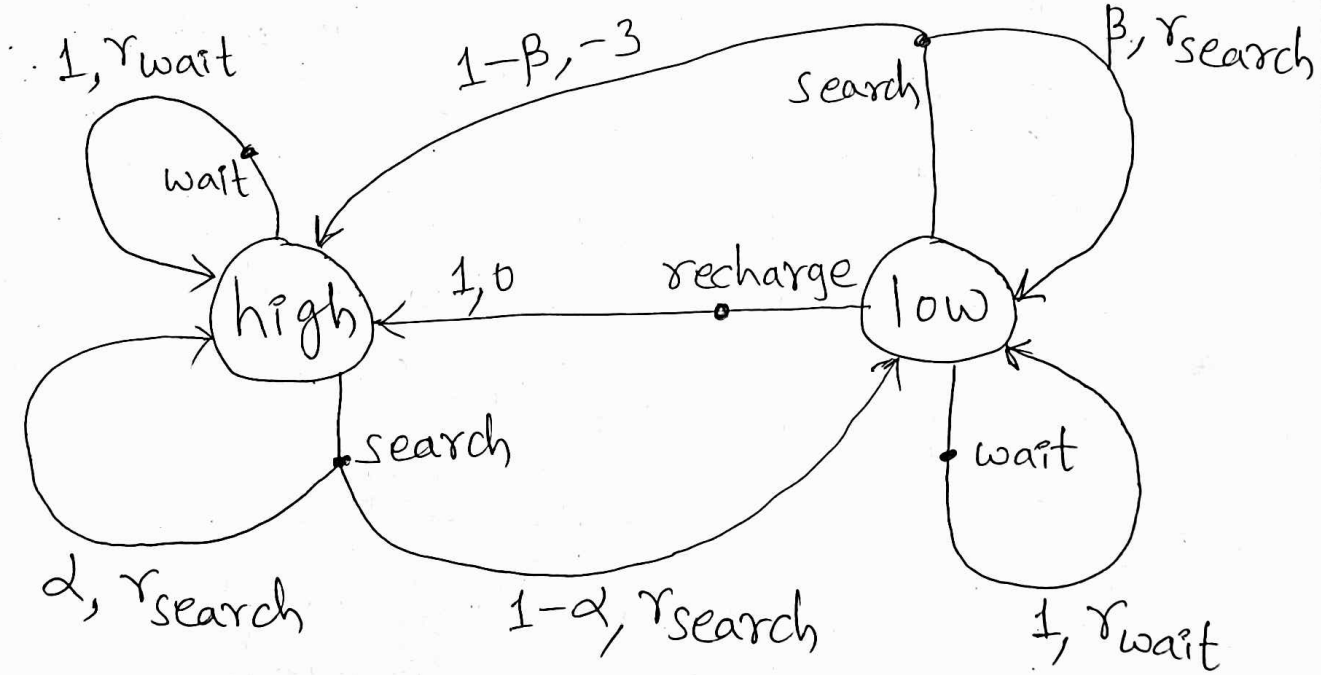
$\alpha \rightarrow$ A period of searching that begins with a high energy level, leaves the energy level high with probability α and reduces it to low with probabability $1-\alpha$.

$\beta \rightarrow$ A period searching undertaken when the energy level is low, leaves it low with probability β and depletes (consumes) the battery with probability $1-\beta$

reward = -3, whenever the robot has to be rescued and recharged.

current state s	action a	Next state s'	Transition probabilities $p(s'/s,a)$	Expected rewards $r(s,a,s')$
high	search	high	α	r_{search}
high	search	low	$1-\alpha$	r_{search}
low	search	high	$1-\beta$	-3
low	search	low	β	r_{search}
high	wait	high	1	r_{wait}
high	wait	low	0	-
low	wait	high	0	-
low	wait	low	1	r_{wait}
low	recharge	high	1	0
low	recharge	low	0	-

Transition table and transition graph



Each arrow is labeled with p, r

Some transitions have zero probability of occurring, so no expected reward (-) is specified for them.

O — state node

• — action node

The transition probabilities labeling the arrows leaving an action (•) node always sum to 1.

GOALS AND REWARDS OF AN AGENT

A reward signal is passed from the environment to the agent.

At each time step, the reward is a simple number $R_t \in \mathbb{R}$.

The agent's goal is to maximize the total amount of reward it receives in the long run.

This means maximizing not immediate reward, but cumulative reward in the long run.

RETURNS AND EPISODES

If the sequence of rewards received after time step t is denoted by

$$R_{t+1}, R_{t+2}, R_{t+3}, \cdots$$

The expected return, denoted by G_t, is defined as the sum of the rewards.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \quad - (1)$$

Return as a sum over finite no. of steps.

where T is a final time step.

This approach makes sense, when the agent-environment interaction breaks naturally into subsequences called episodes.

such as plays of a game
trips through the maze or
any sort of repeated interaction.

Each episode ends in a special state called the terminal state, followed by a reset to the starting state of next episode. This type of tasks are called episodic tasks

In episodic tasks

S denotes set of all nonterminal states

S^+ – set of nonterminal states + terminal state.

The time of termination T, is a random variable that varies from episode to episode.

On the other hand, in many cases the agent-environment interaction does not break into identifiable episodes, but goes on continually without limit.

Ex An on-going process-control task.

An application to a robot with a long life span.

These tasks are called continuing tasks.

For continuing tasks, the final time step would be $T = \infty$ and the return could be infinite.

Discounting The agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized.

It chooses A_t to maximize the expected discounted return

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad -(2)$$

$$\gamma^0 R_{t+0+1} + \gamma^1 R_{t+1+1} + \cdots$$

$$R_{t+1} + \gamma R_{t+2} + \cdots$$

Return as a sum over infinite no. of terms.

where γ is a parameter, $0 \le \gamma \le 1$, called the discount rate

As the execution of an action is delayed, it's reward value will be reduced

The discount rate determines the present value of future rewards: a reward received after k time steps in the future is worth only γ^{k-1} times [means less value] what it would be worth if it were received immediately.

Returns at successive time steps are related to each other

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots$$

$$G_t = R_{t+1} + \gamma [R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots]$$

$$G_t = R_{t+1} + \gamma G_{t+1} \quad \text{---} \quad (3).$$

REWARD MODELS

There are several types of reward models.

1. Infinite discounted reward model.
2. Total reward model
3. Finite horizon reward model.
4. Average reward model.

1) Infinite discounted reward model:

In this model, the agent aims to maximize the sum of discounted rewards over an infinite time horizon. The agent values immediate rewards more than delayed rewards since rewards in the future are discounted with a factor γ (gamma) between 0 and 1.

The agent's goal is to find a policy that maximizes the expected discounted (reduced) return (total rewards)

$$G_t = \sum_{K=0}^{\infty} \gamma^K R_{t+K+1}$$

where

- G_t is the discounted return starting at time step t

- R_{t+k+1} is the reward received at time step $t+k+1$.
- γ is the discount factor

2) Total reward model

In this model, the agent aims to maximize the total accumulated rewards over an infinite time horizon without discounting (γ). Unlike the infinite discounted reward model, the agent equally values immediate and future rewards.

The agent's goal is to find a policy that maximizes the expected total return.

$$G_t = \sum_{k=0}^{\infty} R_{t+k+1}$$

where

- G_t is the total return starting at time step t.
- R_{t+k+1} is the reward received at time step $t+k+1$.

3) Finite horizon reward model

In this model the agent aims to maximize the sum of rewards over a fixed finite time horizon T. The agent's objective is to maximize the expected return within this limited time frame

The agent's goal is to find a policy that maximizes the expected finite horizon return

$$G_t = \sum_{K=0}^{T-1} R_{t+K+1}$$

Where

- G_t is the finite horizon return starting at time step t
- R_{t+K+1} is the reward received at time step $t+K+1$
- T is the finite time horizon.

4) Average reward model

In this model, the agent aims to maximize the average reward it receives per time step over an infinite time horizon. This model is useful when the environment has a stationary distribution.

The agent's goal is to find a policy that maximizes the expected average reward.

$$G_t = \lim_{n \to \infty} \frac{1}{n} \sum_{K=0}^{n-1} R_{t+K+1}$$

where

- G_t is the average return starting at time step t
- R_{t+K+1} is the reward received at time step $t+K+1$
- n is the no. of time steps considered in the average (approaching infinity).

UNIFIED NOTATION FOR EPISODIC AND CONTINUING TASKS

In the previous section we described two kinds of reinforcement learning tasks.

1. Episodic tasks
2. continuing tasks

In episodic tasks, rather than one long sequence of time steps, we need to consider a series of episodes. Each episode consists of a finite sequence of time steps. we number the time steps of each episode starting anew from zero.

$S_{t,i}$ → state representation at time t of episode i

similarly $R_{t,i}$, $\Pi_{t,i}$, T_i etc.

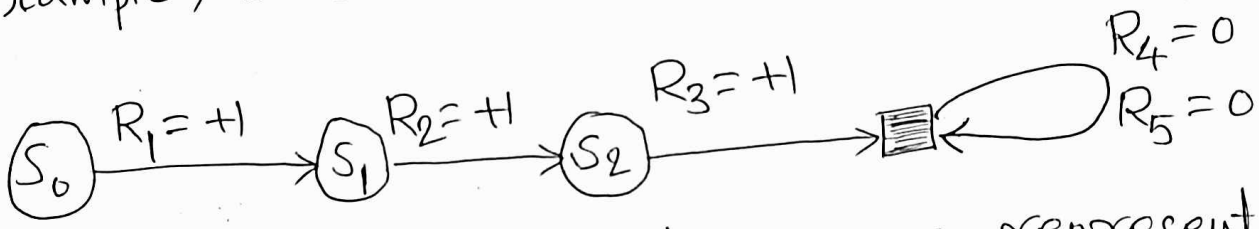
we are almost always considering a particular single episode or stating something that is true for all episodes. Accordingly, we drop the explicit reference to episode number.

That is, we write S_t to refer to $S_{t,i}$

and so on.

We need one convention to obtain a single notation that covers both episodic and continuing tasks.

Equations (1) and (2) can be unified by considering episode termination to be the entering of a special absorbing state that transitions only to itself and that generates only rewards of zero. For example, consider the state transition diagram.



Here the solid square represents the special absorbing state corresponding to the end of an episode.

Starting from S_0, we get the reward sequence $+1, +1, +1, 0, 0, 0, \ldots$ summing these, we get the same return whether we sum over the first T rewards (here T=3) or over the full infinite sequence. Thus, we can define the return, according to Equation (2)

Alternatively, we can write

$$G_t = \sum_{K=t+1}^{T} \gamma^{K-t-1} R_K$$

$$= \gamma^{t+1-t-1} R_{t+1} + \gamma^{t+2-t-1} R_{t+2} + \gamma^{t+3-t-1} R_{t+3} + \cdots$$

$$= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

including the possibility that $T = \infty$ or $\gamma = 1$. We use these conventions throughout the rest of the subject to simplify notation and to express the close parallels between episodic and continuing tasks.

POLICY FUNCTION AND VALUE FUNCTION

All RL algorithms estimate value functions - functions of states (or of state-action pairs) that estimate how good it is for the agent to perform a given action in a given state.

"How good", here is defined in terms of future rewards, in terms of expected return.

The rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting called policies.

A policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then

$\pi(a|s)$ - is the probability that $A_t = a$ if (given) $S_t = s$.

"|" represents conditional probability

RL methods specify how the agent's policy is changed as a result of its experience.

Value function: The value function of a state s under a policy π, denoted $V_\pi(s)$, is the expected return when starting in s and following π thereafter.

For MDPs, we can define v_π formally by

$$v_\pi(s) = E_\pi[G_t \mid S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \text{ for all } s \in S$$

where $E_\pi[\,]$ denotes the expected value given that the agent follows policy π, and t is any time step. we call the function v_π the state-value function for policy π.

Similarly, we define the value of taking action a in state s under a policy π, denoted by $q_\pi(s,a)$, as the expected return starting from s, taking the action a, and thereafter following policy π.

$$q_\pi(s,a) = E_\pi[G_t \mid S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

We call q_π the action-value function for policy π.

BELLMAN EQUATION

BELLMAN OPTIMALITY EQUATION

BELLMAN'S OPTIMALITY OPERATOR

For any policy π and any state s, the following consistency condition holds between the value of s and the value of its possible successor (next) states s'

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s]$$
$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']\right]$$

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\left[r + \gamma V_\pi(s')\right] \quad \text{for all } s \in S \qquad -(1)$$

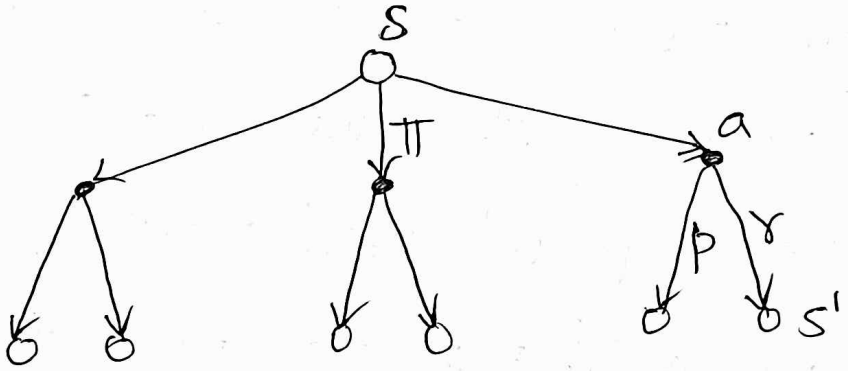
Equation (1) is called Bellman equation for V_π. It expresses a relationship between the value of a state (s) and the values of its successor (next) states s'.

In this equation we have merged the two sums, one over all the values of s'

and the other over all the values of r, into one sum over all the possible values of both. It is the expected value at state s.

It is really a sum (Σ) over all values of the three variables a, s', and r. For each triple, we compute its probability, $\pi(a|s)\,p(s', r|s, a)$, weight the quantity in brackets by that probability, then sum over all possibilities to get an expected value.

Ex



Backup diagram for V_π

Think of looking ahead from a state to its possible success (next) states, as suggested by the above diagram.

open circles ○ represent a state

solid circles ● — a state-action pair.

starting from state s, the root node at the top
The agent could take any of some set of actions
Three are shown in the diagram – based on its policy π.
From each of these, the environment could respond with one of several next states s' (two are shown in the figure), along with a reward r, depending on its dynamics given by the function p.

Bellman equation states that the value of the start state must equal the (discounted) value of the expected next state $[\gamma V_\pi(s')]$ plus the reward (γ) expected along the way.

In backup diagrams, backup operations transfer value information back to a state (or a state-action pair) from its successor states (or state-action pairs).

Optimal policies and Optimal value functions

In RL problems, the main aim is to find a policy that achieves a lot of reward over the long run.

A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states

$$\pi \geq \pi' \text{ iff } V_\pi(s) \geq V_{\pi'}(s) \text{ for all } s \in S$$

There is always at least one policy that is bettern than or equal to all other policies. That is an optimal policy. Although there may be more than one optimal policies,

π_* – all the optimal policies.

They share the same state-value function, called the optimal state-value function V_* defined as

$$V_*(s) = \max_\pi V_\pi(s). \text{ for all } s \in S$$

optimal policies also share the same optimal action-value function, q_* defined as

$$q_*(s,a) = \max_{\pi} q_\pi(s,a)$$ for all $s \in S$ and $a \in A(s)$.

For the state-action pair (s,a), this function gives the expected return for taking action a in state s and thereafter following an optimal policy.

Thus, we can write q_* in terms of v_* as follows.

$$q_*(s,a) = E\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right].$$

$v_*(s)$ optimal value function represents the maximum expected cumulative (discounted) reward achievable from a state s over all possible policies.

The Bellman optimality equation for $v_*(s)$ expresses the fact that the value of a state under an optimal policy must equal the expected return for the

best [max reward giving] action from that state s.

$$v_*(s) = \max_{a \in A(s)} q_{\pi_*}(s,a)$$

$$= \max_a E_{\pi_*}[G_t | S_t = s, A_t = a]$$

$$= \max_a E_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a]$$

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

$$= \max_a \sum_{s', r} p(s', r | s, a)[r + \gamma v_*(s')] \quad -(1)$$

written without reference to any policy π

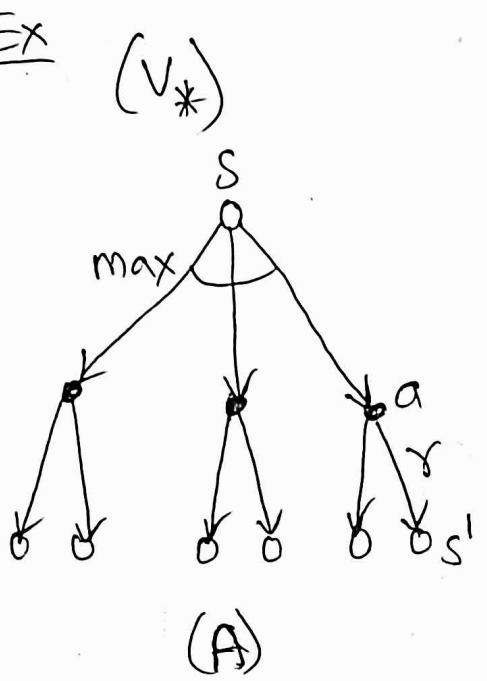
The last two equations are two forms of the Bellman optimality equation for v_*.

The Bellman optimality equation for q_* is

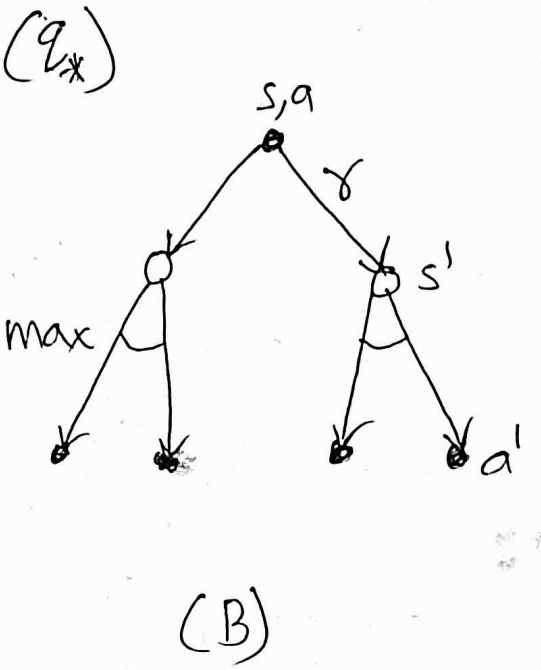
$$q_*(s,a) = E[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a]$$

$$= \sum_{s', r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')] \quad -(2)$$

Ex



(A)



(B)

The backup diagrams in the above fig shows graphically the spans of future states and actions considered in the Bellman optimality equations for V_* and q_*. These are the same as the backup diagrams for V_π and q_π presented earlier except that arcs have been added at the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy.

Diagram (A) graphically represent Eq (1)

(B) — Eq (2).

Once one has V_*, it is easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum (reward) is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy.

Having q_* makes choosing optimal actions even easier. For any state s, it can simply find any action that maximizes $q_*(s,a)$.

Bellman optimality operator (T operator)

The Bellman optimality operator, denoted as T^*, maps a value function to its corresponding optimal value function. It is used to solve for the optimal value function iteratively. The T^* operator is defined as follows

$$T^*(V)(s) = \max_a \left[R(s,a) + \gamma * \sum_{s'} P(s'|(s,a) * V(s') \right]$$

In words, the T^* operator computes the value of a state under the optimal policy by considering all possible actions and their corresponding outcomes (rewards) then taking the maximum expected cumulative reward.

DYNAMIC PROGRAMMING

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal (max reward giving) policies given a perfect model of the environment as a MDP.

Policy Evaluation (Prediction)

How to compute the state-value function V_π for a given policy π.

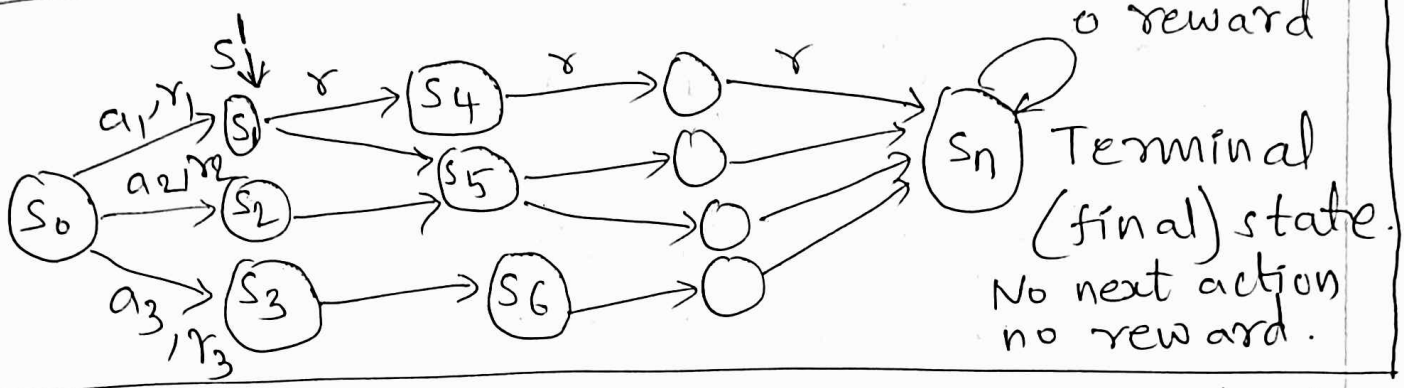
The Bellman equation for policy evaluation expresses the value of a state $[V_\pi(s)]$ in terms of the expected immediate reward and the expected value of the next state s' $[V_\pi(s')]$.

- In DP, RL problems are solved by using recursive equations.

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V_\pi(s')\right].$$

where

$\pi(a|s)$ - is the probability of taking action a in state s by following policy π.



Policy evaluation is performed iteratively $[k: 0 \to \infty]$ to estimate the value function more accurately.

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V_k(s')\right]$$

for all states $s_i \in S$

	s_0	s_1	s_2	s_3	s_4
$50 = V_k(s)$	5	10	12	15	8
$61 = V_{k+1}(s)$	6	11	10	20	14
$65 = V_{k+2}(s)$	8	11	12	19	15

$|V_{k+1}(s) - V_k(s)| = 15$

$= 15$

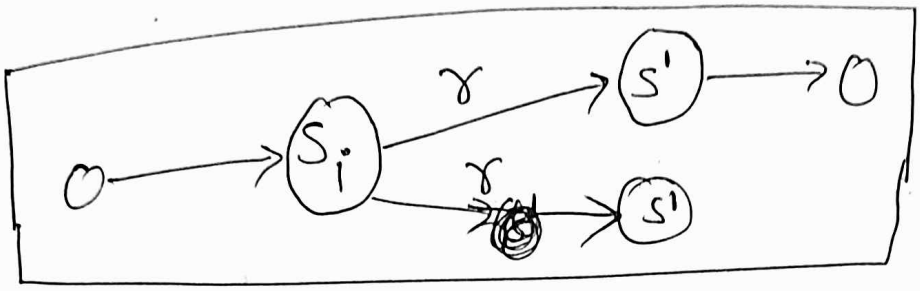
$|V_{k+2}(s) - V_{k+1}(s)|$

$= 6 < \theta$, For ex 10

$65 = V_\pi$ ↙ $65 = V_{k+2}(s)$ giving optimal (max) value

As $K \to \infty$, $V_K(s)$ converges [meeting to] $V_\pi(s)$ [max reward giving]

$$V_{K+1}(s_i) = \sum \gamma + V_K(s')$$



→ A policy (strategy) tells which path (sequence of states and actions) should be followed to get more reward.

Iterative Policy Evaluation

Given a policy π.

starting with some initial reward values of each state $V_0(s_i)$

Evaluate, update and find the more accurate values $V_K(s_i)$ of each state s_i in K iterations

$K : 0 \to \infty$

$V_0(s_i)$

$V_1(s_i)$

$V_K(s_i)$.

Iterative policy evaluation algorithm

For estimating $V(s_i) \approx V_\pi(s_i)$ $\forall$ states s_i

Input π, the policy to be evaluated

parameter θ, a small threshold $\theta > 0$ which determines the accuracy of estimation

Initialize $V_0(s_i)$ for all $s_i \in S$. For final terminal state $V_0(s_n) = 0$. It's reward $= 0$.

Loop: for $k = 0$ to ∞

$\Delta \leftarrow 0$

Loop for each state $s \in S$ set of states

$V \leftarrow V_k(s_i)$

$$V_{k+1}(s_i) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V_k(s')\right]$$

$$\Delta \leftarrow \max\left[\Delta, |V_k(s_i) - V_{k+1}(s_i)|\right]$$

until $\Delta < \theta$ → Algorithm terminates when the difference between the total value of states in two successive iterations is $[\Delta < \theta]$ less than a specified threshold.

POLICY ITERATION

Once a policy π, has been improved using V_π to yield a better policy π', we can then compute $V_{\pi'}$ and improve it again to yield an even better policy π''. We can thus obtain a sequence of monotonically improving policies and value functions.

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} V_*$$

where

$\xrightarrow{E}$ denotes a policy evaluation and

$\xrightarrow{I}$ denotes a policy improvement.

This process must converge to (meeting to) an optimal policy and optimal value function in a finite no. of iterations. This way of finding an optimal policy is called policy iteration.

Policy Iteration algorithm

using iterative policy evaluation for estimating $\pi \approx \pi_*$.

1. Initialization
$V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all states $s \in S$

2. Policy evaluation
Loop: for $k=0$ to ∞
$\Delta \leftarrow 0$
Loop for each state $s \in S$
$v \leftarrow V_k(s)$
$V_{k+1}(s) \leftarrow \sum_{s',r} p(s',r|s,\underset{\uparrow}{a}\pi(s))\left[r+\gamma V(s')\right]$.
$\Delta \leftarrow \max(\Delta, |v - V_{k+1}(s)|$
until $\Delta < \theta$

3. Policy Improvement → To check whether there is any other better policy π' than π.
policy-stable $\leftarrow$ true
For each state $s \in S$
old-action $\leftarrow \pi(s)$
$\pi'(s) \leftarrow \underset{a}{\arg\max} \sum_{s',r} p(s',r|s,a)\left[r+\gamma V(s')\right]$.
If old-action $\neq \pi'(s)$ then policy-stable $\leftarrow$ false.
If policy-stable, then stop and return $V \approx V_*$ and $\pi \approx \pi_*$] policy π is best
else goto 2 // policy π' is better than π goto step 2) and evaluate value of π'

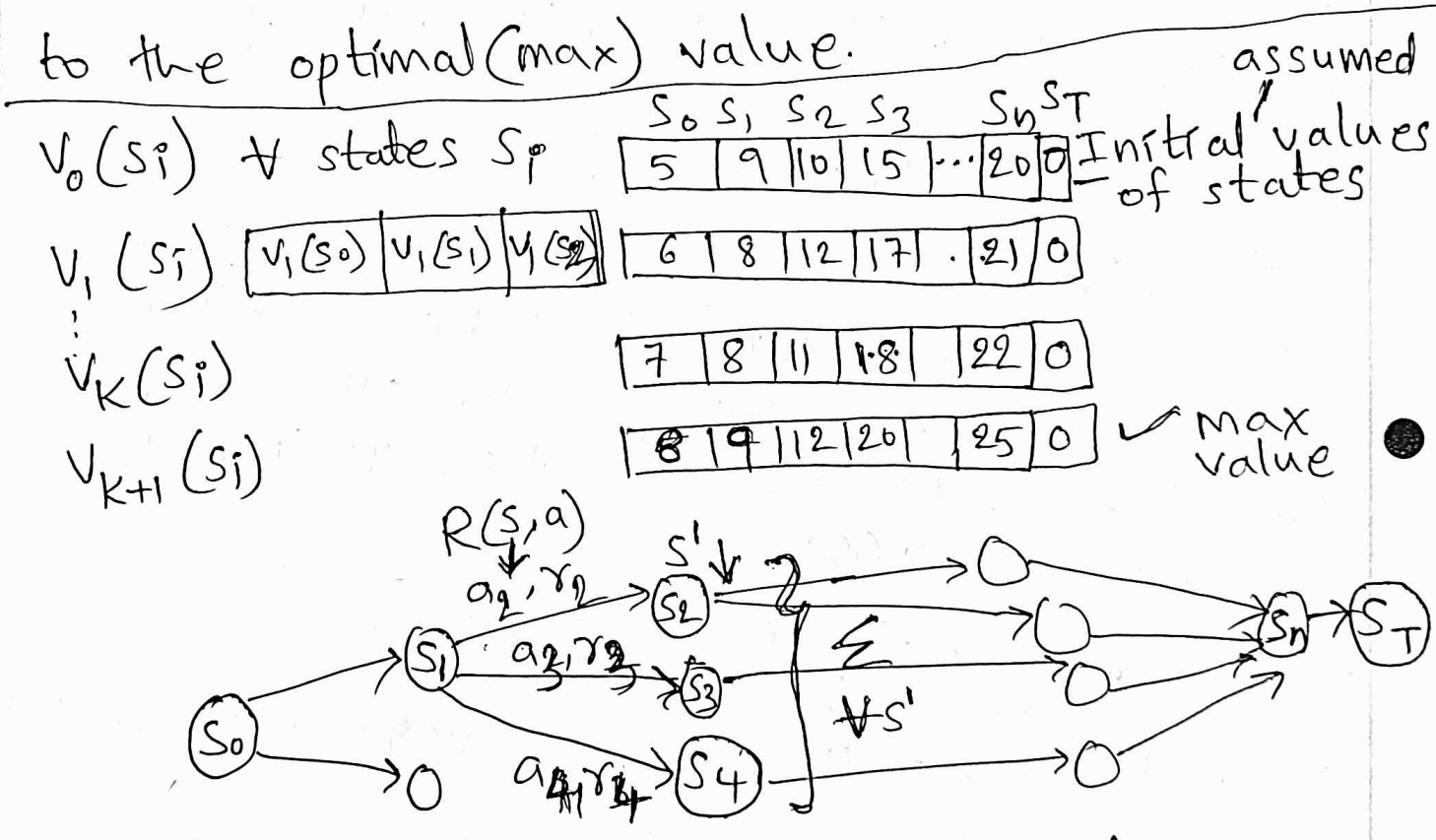
VALUE ITERATION

Value iteration is an iterative algorithm used in RL to find the optimal value function and policy for an agent in an MDP.

step1 Finding the optimal (max) value function

step2 Deriving the optimal policy based on the updated value function

These two steps are repeated iteratively until the value function converges (approaches) to the optimal (max) value.



S_T - Terminal (final) state. It returns a reward whose value is = 0. Because it is last state of RL problem from which no next action takes place.

Algorithm value Iteration | a policy π which is approximately equal to optimal policy.

For estimating $\pi \approx \pi_*$

1) Finding the optimal value function

Initialize value of each state with some arbitrary assumed value. $V_0(s_i)$

Iterative update the value estimates for each state s_i, while considering the maximum expected cumulative reward achievable from each state

$V_1(s_i)$
$V_2(s_i)$
$V_K(s_i)$
$V_{K+1}(s_i)$

Here the Bellman optimality equation is used in each value iteration to update value function.

$$V_K(s) = \max_a \left[R(s,a) + \gamma \times \sum_{\forall s'} P(s'|s,a) \times V(s') \right]$$

where,

$V_K(s)$ – value function for state s, in K^{th} iteration of the algorithm

$R(s,a)$ – immediate reward obtained when taking action a in state s.

γ (gamma) – discount factor, reduced reward value of an action in future after some delay.

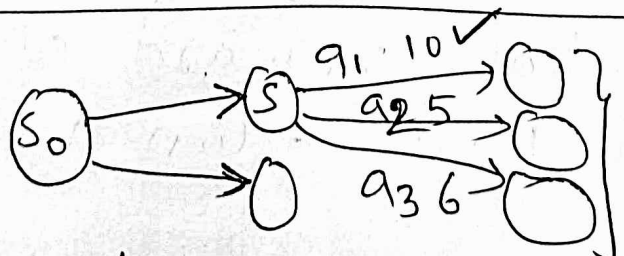
$p(s'|s,a)$ – probability of transitioning from state s to s' ~~after~~ when taking action a

$\sum_{\forall s'}$ represents the sum of rewards over all possible successor (next) states s'

2) Deriving the optimal policy.

After the value function has been updated in each iteration, the optimal policy (π) can be extracted by selecting the action (a) that maximizes ($\arg\max_a$) the ~~right~~-hand side of the Bellman optimality equation for each state (s).

$$\pi(s) \leftarrow \arg\max_a \left[R(s,a) + \gamma \times \sum_{\forall s'} p(s'|s,a) \times V(s') \right]$$



out of all actions from s to s', select an action q that maximizes reward.

UNIT-III

MONTE CARLO METHODS

→ Monte carlo methods are a class of RL algorithms used for solving Markov Decision Processes (MDPs) without requiring a model (simulation) of the environment.

→ Monte carlo methods learn by trial and error requiring only experience based on. Agent doesn't require prior knowledge of environment's dynamics.

→ These methods are particularly suited for episodic (episode-by-episode) tasks not continuous tasks.

→ An agent interacts with an environment and learns from complete episodes of experience.

→ Monte carlo methods solve the RL problem based on averaging sample returns (rewards) collected by the agent while interacting with the environment.

→ Monte carlo methods require only experience
* - sample sequences of states, actions and rewards received from actual interaction with an environment.

MONTE CARLO PREDICTION

- Monte carlo prediction is a technique in RL used to estimate the value function of a given policy by averaging the returns (rewards) observed in all sampled episodes.
- Average the returns observed after visits to a state s.

Suppose we wish to estimate $V_\pi(s)$, the optimal (max) value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a visit to s. s may be visited multiple times in the same episode. Let us call the first time s is visited in an episode the first visit to s. The first visit MC (Monte carlo) method estimates $V_\pi(s)$ as the average of the returns following first visit to s, whereas the every-visit MC method averages the

returns following all visits to s.

First-visit MC prediction algorithm

For estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all states $s \in S$

Returns(s) $\leftarrow$ an empty list, for all $s \in S$

Loop forever (for each episode):

Generate an episode following π:

$S_0, A_0, R_1, S_1, A_1, R_2, \cdots, S_{T-1}, A_{T-1}, R_T$.

$G \leftarrow 0$

Loop for each ~~episode~~ step of episode,

$t = T-1, T-2, \cdots, 1, 0$.

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \cdots S_{t-1}$

Append G to Returns(S_t)

$V(S_t) \leftarrow$ average(Returns(S_t))

Example Suppose there is an environment where we have 2 states A and B

Let's say we observed 2 sample episodes

1) $A+3 \rightarrow A+2 \rightarrow B-4 \rightarrow A+4 \rightarrow B-3 \rightarrow$ terminate

2) $B-2 \rightarrow A+3 \rightarrow B-3 \rightarrow$ terminate

$A+3 \rightarrow A$, indicates a transition from state A to A, with a reward of +3.

Let's find out the value function V(A) & V(B) using both methods.

→ In First-visit MC-prediction, we estimate the value of a state "s" by averaging the returns we observe after our first visit to "s".

From 1st episode $V(A) = +3+2+-4+4+-3 = 2$

From 2nd episode $V(A) = +3+-3 = 0$

$\therefore V(A) = \frac{2+0}{2} = 1$

From 1st episode $V(B) = -4+4+-3 = -3$

From 2nd episode $V(B) = -2+3+-3 = -2$

value of B, $V(B) = \frac{-3-2}{2} = -\frac{5}{2}$

Prediction

→ In every-visit MC, we calculate the value of a state "s" by averaging the returns after all visits to "s"

summation term adding all rewards coming after every(4)occurrence of state A

From 1st episode:

$V(A) = (3+2+-4+4+-3) + (2+-4+4+-3) + (4+-3)$

After 1st occurrence of A

After 2nd occurrence of A

$V(A) = 2 + -1 + 1$

From 2nd episode: $(3+-3) = 0$

$$\therefore V(A) = \frac{2+-1+1+0}{4} = \frac{1}{2}$$

similarly for B

From 1st episode $V(B) = (-4+4+-3) + (-3) = -3+-3$

From 2nd episode $V(B) = (-2+3+-3) + (-3) = -2+-3$

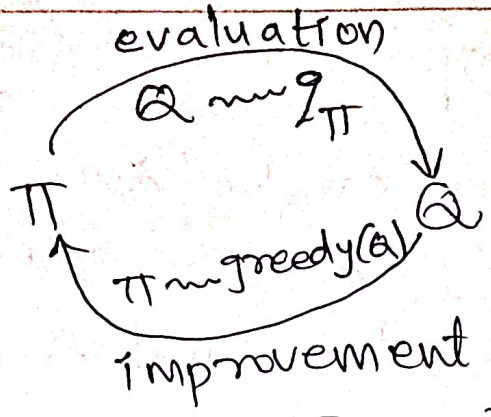
$$\therefore V(B) = \frac{-3+-3+-2+-3}{4} = \frac{-11}{4}.$$

First visit	Every visit MC Prediction.
$V(A) = 1$	$V(A) = 1/2$
$V(B) = -\frac{5}{2}$	$V(B) = -11/4$.

Exploring starts is a concept in RL that is often associated with the Monte carlo method. It is a technique used to ensure that during the learning process, the agent explores a variety of state-action (s,a) pairs by starting episodes from different initial state-action pairs, and that every pair has a nonzero probability of being selected as the start.

MONTE CARLO CONTROL

- How to use Monte carlo estimation in control, to approximate optimal policies
- We proceed according to the same idea of GPI (generalized policy iteration) of DP (Dynamic Programming).
- In Monte carlo version of policy iteration we perform alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy π_0 and ending with the optimal policy π_* and optimal action-value function q_*.



$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} q_*.$$

where

$\xrightarrow{E}$ denotes a complete policy evaluation

$\xrightarrow{I}$ denotes a complete policy improvement.

- Policy evaluation is done as many episodes are experienced, with the approximate action-value function approaching the true function asymptotically (meeting to a point). Monte carlo methods will compute each q_{π_k} exactly, for arbitrary π_k.
- Policy improvement is done by making the policy greedy w.r.to an action-value function. For any action-value function q, the corresponding (max value giving) greedy policy is the one that, for each state $s \in S$, deterministically chooses an action (a) with maximal action-value.

$$\pi(s) = \arg\max_a q(s,a)$$

- Policy improvement then can be done by constructing each π_{k+1} as the greedy policy w.r.t. to action-value function q_{π_k}.

$$q_{\pi_k} \xrightarrow{I} \pi_{k+1}.$$

Monte carlo ES (Exploring starts) algorithm

For estimating $\pi \approx \pi_*$

- Initialize:

$\pi(s) \in A(s)$ (arbitrarily), for all $s \in S$

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in S, a \in A(s)$

Returns$(s,a) \leftarrow$ empty list, for all $s \in S, a \in A(s)$

Loop forever (for each episode):

- choose $S_0 \in S, A_0 \in A(S_0)$ randomly such that all pairs have probability > 0
- Generate an episode from S_0, A_0, following π:

$\pi: S_0, A_0, R_1, S_1, A_1, R_2, \cdots S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode $t = T-1, T-2, \ldots 1, 0$.

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in
$S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$

Append G to Returns(S_t, A_t)

$Q(S_t, A_t) \leftarrow$ average (Returns(S_t, A_t))

$\pi(S_t) \leftarrow \underset{a}{\text{argmax}}\, Q(S_t, a)$

In Monte Carlo ES, all the returns for each state-action pair are accumulated and averaged, irrespective of what policy was in force when they were observed.

It is easy to see that Monte carlo ES cannot converge to any suboptimal policy.

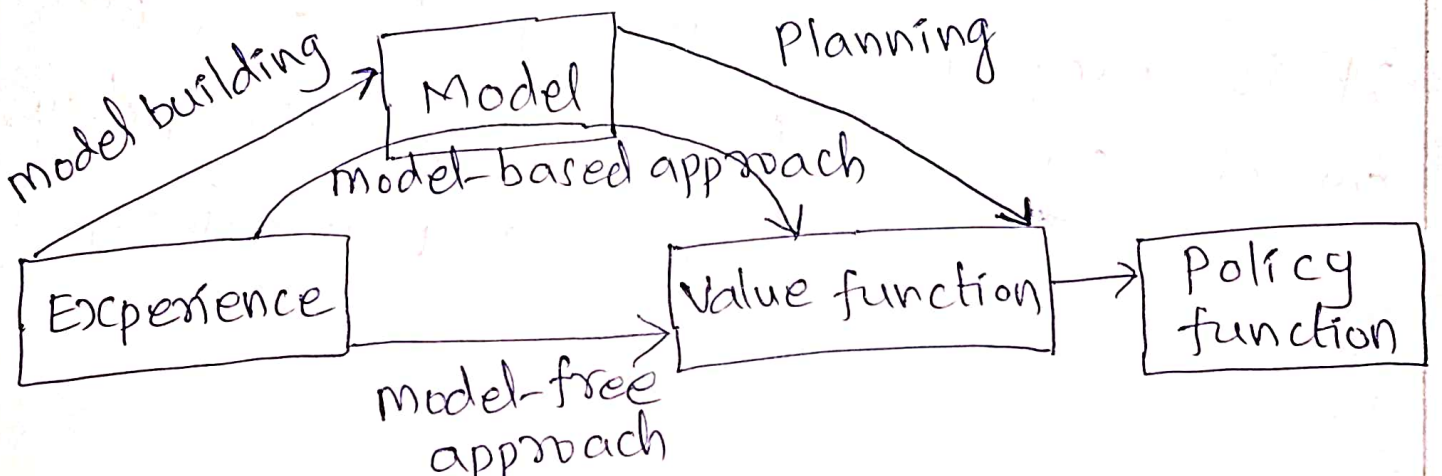
↓
less than optimal (max)

In Monte carlo control problems, the goal is to find an optimal policy π, which defines the best action to take in each state to maximize the expected return. (reward).

MODEL-BASED ALGORITHM

A model-based algorithm in RL refers to an approach that uses a learned model of the environment to make decisions and plan actions

These algorithms are distinct from model-free algorithms, which directly learn a policy or value function from interacting with the environment.



Model-based RL algorithms consist of two main components

1) Model Learning:

- The agent first learns a predictive model of the environment. This model aims to capture the dynamics of the environment,

including how states transition to new states and the expected rewards associated with those transitions.

- The predictive model can take different forms, such as a neural network, a decision tree or a parametric function that approximates the environment's behavior.

2) Planning and Decision making:

- Once the agent has learned the model, it can use this model for planning and decision making. Instead of directly interacting with the real environment to gather experience, the agent simulates interactions using its learned model.
- planning involves using algorithms like dynamic programming, search methods (e.g. Monte Carlo Tree search), or optimization techniques to find an optimal policy or value function
- The agent can simulate various actions in the model and evaluate the expected

outcomes, allowing it to select actions that maximize expected rewards.

Advantages of model-based RL algorithms

- Sample efficiency: Model-based methods often require fewer interactions with the real environment to learn effective policies. This can be crucial in situations where collecting real experience is costly or time-consuming
- Exploration: Since the agent can simulate different scenarios in the learned model, it can explore and learn faster in challenging or high-risk environments.
- Planning: Model-based algorithms can perform more sophisticated planning, which can lead to better decision-making in complex environments.

Model-based RL algorithms also have some limitations

Model Errors: The predictive model may not be perfect, and inaccuracies in the model can lead to suboptimal [less than optimal or less than max] decisions.

Computational cost: planning with a learned model can be computationally expensive, particularly if the model is complex or the state space is large.

Transferability: The model may not generalize well to unseen states or environments, limiting its ability to adapt [to work in new environments] to novel [unique and unexpected] situations.

Model-based RL is a versatile approach that finds applications in various domains including robotics, autonomous control, and game planning. Researchers explore hybrid approaches that combine the

strengths of model-based and model-free methods to achieve better performance and balance the trade-off [adjustment] between sample efficiency and model accuracy

E.g. Monte Carlo Tree Search (MCTS) is a popular model-based RL algorithm that is often used in games such as chess and Go. Go game. MCTS performs a search over the possible action sequences in the environment and uses the learned model to predict the outcomes of the actions. MCTS has been successful in learning effective policies in complex environments.

Pseudo-code for a generic model-based RL algorithm

```
# Initialization
Initialize a predictive model of the environment (e.g. neural network)
Initialize an initial policy π
Initialize discount factor γ
Initialize a planning horizon
```

i.e the no. of planning time steps.

```
# Training loop
for each episode in range(num_episodes):
    # collect data by interacting with environment
    trajectory = collect_data_with_model(π, model,
                                         planning_horizon)
    | path of states, actions, rewards

    # Update the model based on collected data
    update_model(model, trajectory)

    # Perform planning to improve the policy
    new_π = plan_with_model(model, V, planning_horizon)

    # update the policy
    π = new_π.
```

update_model function updates the model based on collected data e.g. through supervised learning

plan_with_model function uses the learned model to perform planning and generate an improved policy.

π is the current policy

new_π is the updated policy after planning

num_episodes is the no. of episodes, which determines how many iterations of data collection, model updates, and planning are performed.

MONTE CARLO METHODS FOR PREDICTION

Monte carlo methods for prediction in RL involves estimating the value function of states (or state-action pairs) based on random sampling of episodes. There are several methods, each with its own variations and specific purposes.

1) First-visit Monte carlo prediction:

This method estimates the value function by averaging the returns observed only during the first visit to each state (or state-action pair) in each episode. It is suitable for episodic tasks.

2) Every-visit Monte carlo prediction:
In this approach, all visits to a state (or state-action pair) during an episode contribute to the estimate of the value function. It can be used for both episodic and continuing tasks.

3) Monte carlo Exploring starts:
In this method, episodes start from all possible state-action pairs with a nonzero probability. This ensures that all states and actions are explored and contributes to more accurate value estimates.

4) off-policy monte carlo prediction:
This method estimates the value function of one policy while following a different policy during episode sampling. Techniques like importance sampling are used to adjust for the differences between the behavior policy (used for exploration) and the target policy (for which the value function is estimated).

5) Incremental Monte carlo prediction:
Instead of waiting until the end of an episode to update value estimates, incremental Monte carlo methods update the estimates incrementally after each time step within an episode. This can save memory and be computationally more efficient

6) Weighted importance sampling Monte carlo (MC)
This method improves the efficiency of off-policy Monte carlo methods by using importance sampling weights that adjust the returns to account for differences between the behavior policy and the taget policy

7) Temporal-Difference Monte carlo:
Temporal-Difference (TD) Monte carlo methods combine the ideas of TD learning and Monte carlo methods. They update value estimates at each time step within an episode while still using MC sampling to

estimate the return

8) Recurrent Monte carlo methods:
These methods extend MC prediction to recurrent (occurring repeatedly) or partially observable environments, where the agent doesn't have full visibility of the environment state. Recurrent Neural Networks (RNNs) or other memory-based models are often used in this context.

9) Bootstrapped Monte carlo methods:
Bootstrapping combines Monte carlo sampling with other methods (e.g. TD learning or model-based methods) to balance the trade-off between accuracy and computational efficiency in value estimation.

10) Exploration strategies:
While not a Monte carlo method per se, the choice of exploration strategies, such as ϵ-greedy, UCB (Upper Confidence Bound) or Bayesian methods, can significantly impact the effectiveness of MC prediction

The choice of which Monte carlo method to use depends on the specific characteristics of the RL problem you are addressing, such as

The type of task (episodic or continuous)

The availability of a behavior policy. and

The trade-off between computational complexity and estimation accuracy.

UNIT- IV

TEMPORAL-DIFFERENCE LEARNING

BOOTSTRAPPING

Bootstrapping in temporal-difference (TD) learning and RL refers to "updating the value estimate of a state or action based on the estimated value of a subsequent state or action".

In TD learning, the agent updates its estimate of the value of a state or action based on the current reward (R_{t+1}) and the estimated value of the next state or action. In other words, it "bootstraps" its estimate using information it has already learned.

$$V(S_t) = V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

V - is the estimate of the value of a state or action

Agent receiving an immediate reward

Temporal = relating to time

R_{t+1}, while transitioning from current state S_t to next state S_{t+1}.

Bootstrap: start, load, reboot, reset, restart.

Bootstrapping - refers to a self-starting process that is supposed to continue or grow without external input.

In general, bootstrapping in RL means that you "update a value based on some estimates and not on some exact values".

In RL, Temporal-difference(TD)learning is a method of updating the value function of a state or a state-action pair by estimating the difference between the predicted reward and the actual reward received at each time step.

TD learning idea is central and novel to RL. TD learning is a combination Monte carlo ideas and Dynamic Programming (DP) ideas. Like monte carlo methods, TD methods can learn directly from

raw experience without using a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome or exact values (TD methods use bootstrapping)

TD PREDICTION

Both TD and Monte Carlo (MC) methods use experience to solve the prediction problem. Given some experience following a policy π, both methods update their estimate V of V_π for the nonterminal states S_t occurring in that experience.

MC methods wait until the return (G_t) following the visit is known, then use that return as a target for $V(S_t)$. A simple every-visit MC method suitable for nonstationary (changing) environment is

$$V(S_t) \leftarrow V(S_t) + \alpha [G_t - V(S_t)]$$

where G_t is the actual return following

time t

α is a constant step-size parameter

Let us call this method constant-α MC

The above equation is similar to the eq

$$Q_{n+1} \leftarrow Q_n + \frac{1}{n}[R_t - Q_n]$$

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}[\text{Target} - \text{OldEstimate}]$$

Whereas MC methods must wait until the end of the episode to determine the increment to $V(S_t)$ (only then is G_t known)

TD methods need to wait only until the next time step. At time t+1 they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$. The simplest TD method makes the update

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

immediately on transition to S_{t+1} and receiving R_{t+1}.

- Target for MC update is G_t
- Target for TD update is $R_{t+1} + \gamma V(S_{t+1})$.

This TD method is called TD(0), or one-step TD, because it is a special case of the TD(λ) and n-step TD methods which will be covered in next unit.

TD(0) ALGORITHM

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated

Parameter: step-size $\alpha \in (0,1]$

Initialize state-value function $V(s)$ for all states $s \in S^+$, arbitrarily except that for terminal or final state $V(\text{terminal}) = 0$.

Loop for each episode:

Initialize S

Loop for each step of episode:

$A \leftarrow$ action given by π for S

Take action (execute action) A, observe R, S'

R - reward, S' — next state, S - current state

$$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$$

$S \leftarrow S'$ (next state becomes current state)

until S is terminal (final state is reached)

In TD(0) algorithm, the value function of each state is updated based on the difference between the estimated value of the current state(S) and the estimated value of the next state (S'), weighted by the learning rate α.

TD(0) algorithm is computationally efficient and requires less memory than MC methods because it updates the value function iteratively after each time step.

Because TD(0) bases its update in part on an existing estimate, we say that it is a bootstrapping method, like DP.

$$V_\pi(s) = E_\pi[G_t | S_t = s] \quad \text{---(1)}$$
$$= E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s]$$
$$= E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s] \quad \text{---(2)}$$

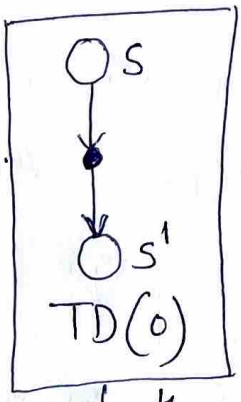
MC methods use an estimate of (1) as a target.

whereas

DP methods use an estimate of (2) as a target.

- TD methods combine the sampling of MC with the bootstrapping of DP.
- TD takes advantages of both MC and DP methods.

shown to the right is the backup diagram for tabular TD(0). The value estimate for the state node at the top of the backup diagram is updated on the basis of the one sample transition from it (s) to the immediately following state (s'). we refer to TD and MC updates as <u>sample updates</u> because they involve looking ahead to a sample successor state (or state-action pair), using the value of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state-action pair) accordingly.



<u>Sample updates</u> differ from <u>expected updates</u> of DP methods in that they are

based on a single sample successor rather than on a complete distribution of all possible successors (states).

The quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$.

This quantity, called the TD error, arises in various forms throughout RL.

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

EXAMPLE: DRIVING HOME

Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. Say on this Friday you are leaving at exactly 6 o' clock, and you estimate that it will take 30 minutes to get home.

As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes.

state	Elapsed Time (min)	Predicted Time to go	predicted Total Time
leaving office, friday at 6 PM	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass.

You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home.

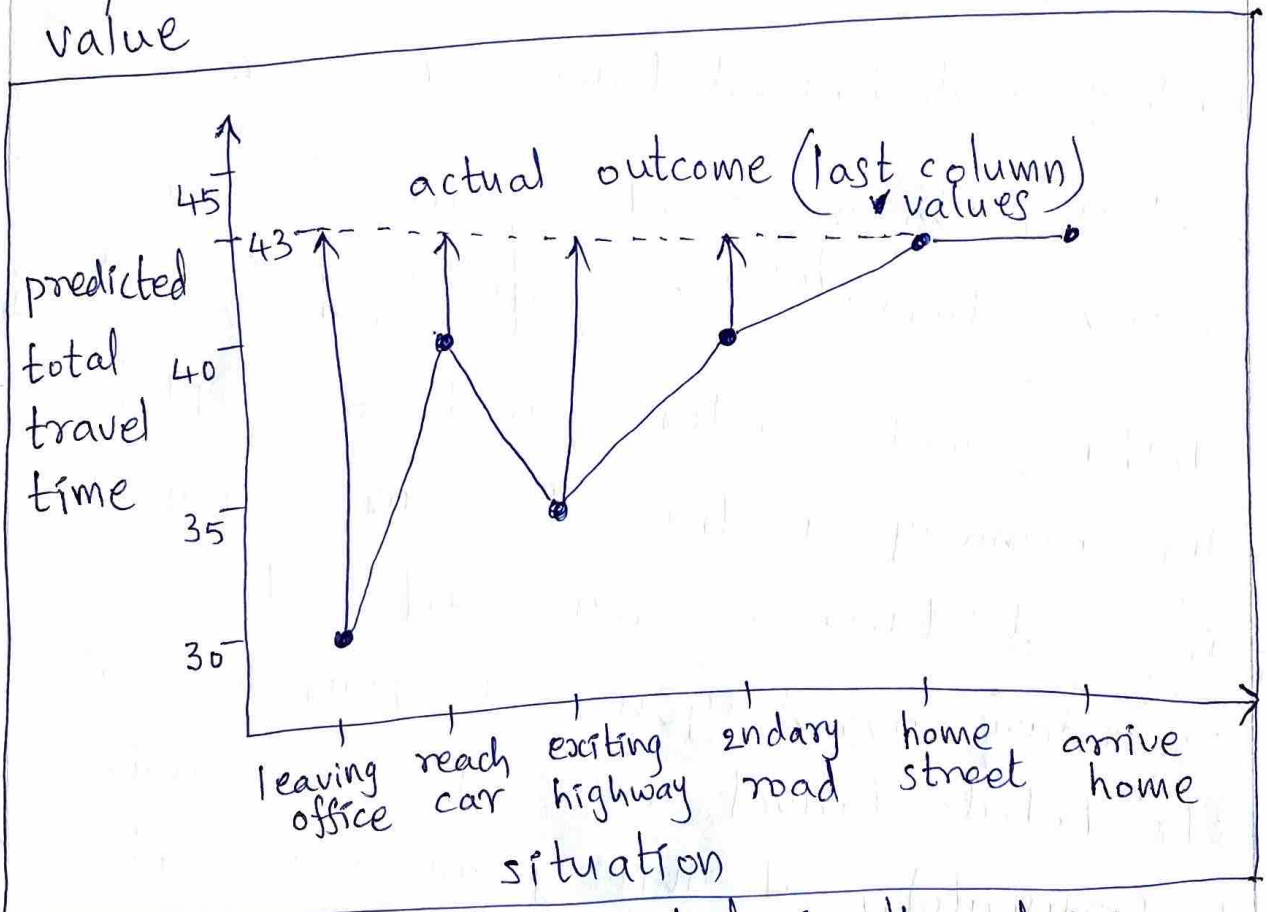
The sequence of states, times, and predictions is shown in the above table.

In this example

rewards are elapsed times after each state.

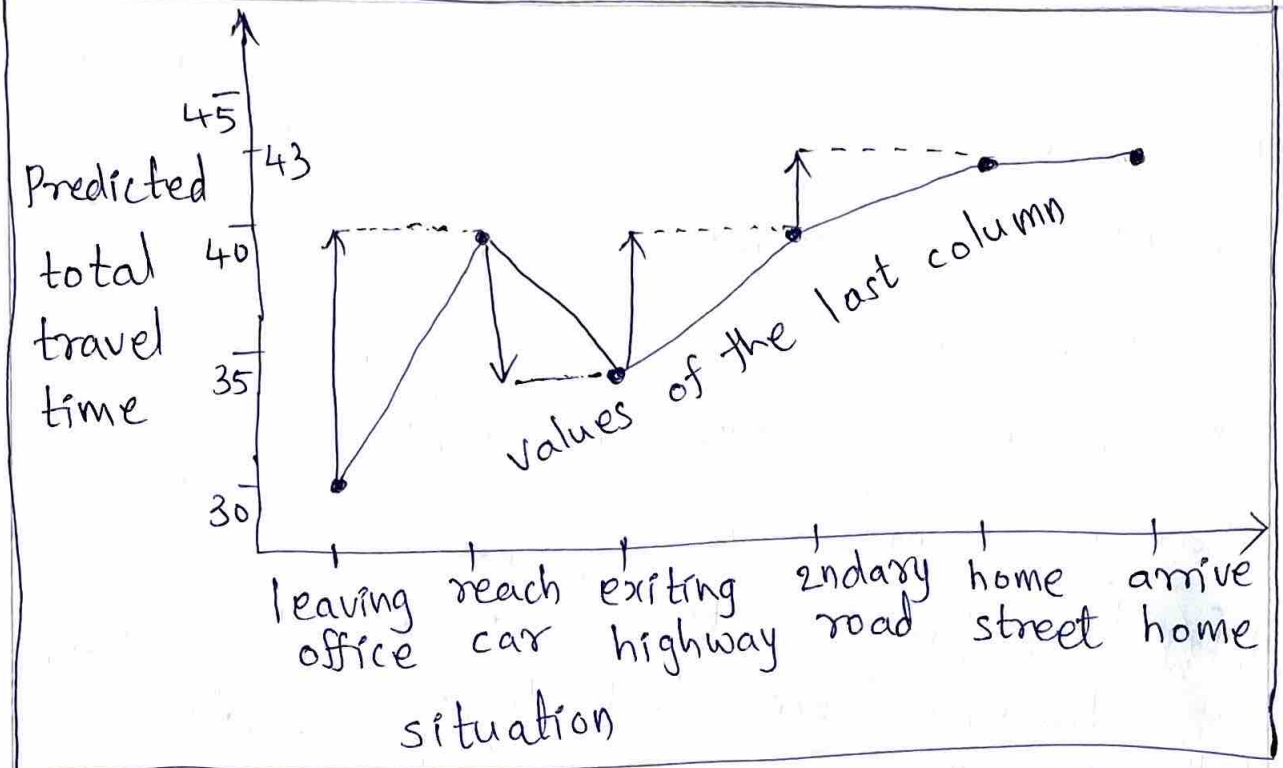
return for each state - actual time to go from that state.

value



changes recommended in the driving home example by Monte carlo methods.

value function updated after completion of entire episode.



changes recommended in the driving home example by TD methods.

-value function updated after each state transition (change).

CONVERGENCE OF MONTE CARLO AND BATCH TD(0) ALGORITHMS

Batch TD(0) is RL algorithm that aims to estimate the value function of a given policy by using batches of transitions collected from the environment. It belongs to the family of TD learning algorithms that combine MC and DP methods in estimating the value function.

The convergence of MC and batch TD algorithms refers to their ability to identify the optimal policy in a given environment.

MC methods require the entire episode to be completed before updating the value function. This results in a lower of estimation, but a higher bias (partiality) which may lead to slower convergence.

difference
variance
variation

With an infinite number of episodes, the value function estimate becomes more and more accurate, and the algorithm converg~~e~~ to the optimal value function. MC methods are guaranteed to converge to the optimal policy as long as the policy space is explored fully.

Batch TD(0) methods, on the other hand update the value function after each state transition (change). This results in a higher variance of estimation

but lower bias, which may lead to faster convergence. However, batch TD(0) algorithms may converge to a suboptimal (less than optimal (max)) policy if the exploration space is not fully covered.

Both algorithms can converge to the optimal policy (max reward giving), but their convergence rates may vary depending on the specific environment and exploration strategy used.

Variance degree of fluctuation or inconsistency in the estimation of value functions.

Bias degree to which the estimate deviates from the true value.

Q: In Q-learning, Q refers to the quality or value of a specific action taken in a particular state of the environment, and is represented as a function of $Q(s,a)$.

MODEL-FREE CONTROL

Model-free control is a type of RL algorithm that does not require an explicit model of the environment to (learn) create a policy that maximizes the sum of future rewards. Instead, it uses trial-and-error interaction with the environment to estimate the value function and select actions that maximize the expected sum of future rewards.

Two popular model-free control algorithms in TD learning and RL are Q-learning and SARSA.

In Q-learning, the algorithm learns the optimal action-value function by updating the Q-values for each state-action pair based on the difference between the current estimate of the Q-value and the maximum Q-value for the next state.

In SARSA (state-Action-Reward-state-Action), the algorithm learns the optimal policy by estimating the Q-values for each state-action pair and selecting actions based on the current estimate of the Q-value and the probability of taking each action in the current state.

Both Q-learning and SARSA are considered off-policy algorithms, meaning that the policy used for learning (the behavior policy) can be different from the policy used for action selection (the target policy). This property allows for exploration and does not require a separate exploration policy.

Model-free control algorithms are particularly useful in environments where the dynamics of the environment are unknown or change over time. They can also learn optimal policies in

high-dimensional or continuous-state spaces which are often difficult for model-based methods.

Model-free control in TD learning and RL is a powerful approach for learning optimal policies without using an explicit model of the environment.

SARSA, Q-LEARNING, and EXPECTED SARSA

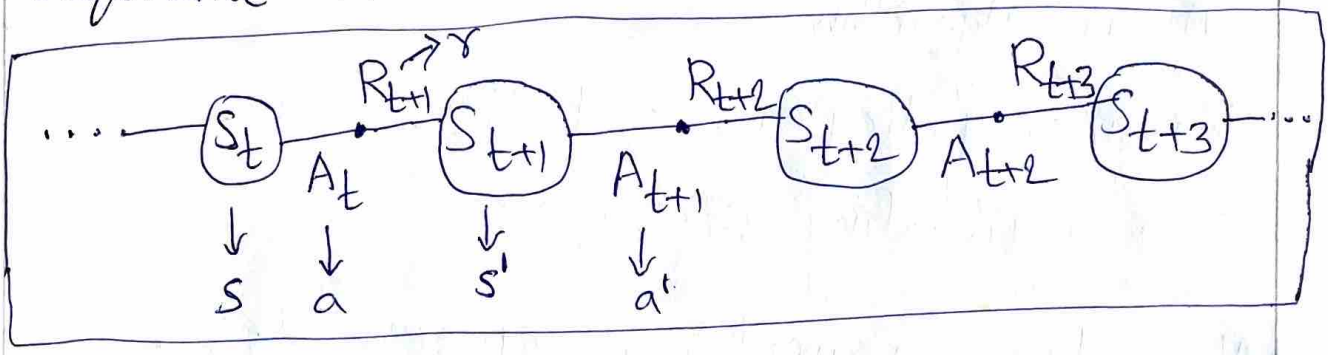
SARSA, Q-LEARNING, and EXPECTED SARSA are TD learning algorithms that estimate the optimal action-value function in RL.

SARSA: On-policy TD control method

In SARSA (State-Action-Reward-State-Action) the agent learns the optimal policy by estimating the action-value function $Q(s,a)$ for the current policy. The algorithm learns incrementally from each experience quintuple (s, a, r, s', a').

SARSA is an on-policy TD learning algorithm where the agent selects an action (a) based on the current policy and updates its Q-value based on the observed reward and the next state-action pair.

An episode consists of an alternating sequence of states and state-action pairs



The update rule for SARSA is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

or

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

where,

- $Q(s,a)$ is the estimated action-value of state s and action a
- α is the learning rate.

- r is the reward obtained after taking action a in state s
- γ (gamma) is the discount factor which determines the importance (value) of future action rewards
- s' is the next state reached after taking action a in state s
- a' is the next action taken in state s'

SARSA algorithm

Sarsa (on-policy TD control)

for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$
small $\epsilon > 0$

Initialize $Q(s,a)$ for all states $s \in S^+$, $a \in A(s)$ arbitrarily except that $Q(\text{terminal}, \cdot) = 0$.

Loop for each episode:

Initialize S

Choose A from S using policy derived from Q (e.g., ϵ-greedy)

ϵ-greedy policy selects a greedy (highest reward giving) action with probability ϵ selects other random actions with probability $1-\epsilon$

Loop for each step of episode:
Take action A, observe R, S'
choose A' from S' using policy derived from Q (e.g. ϵ-greedy or ϵ-soft)

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

$S \leftarrow S'$, $A \leftarrow A'$;
until S is terminal (final state)

Backup diagram for Sarsa

Sarsa uses current policy to explore the environment and update their Q-value Q(S,a) accordingly.
Q-learning: Off-policy TD control method

Q-learning is an off-policy learning algorithm that can learn from other policies.

Q-learning is an off-policy TD learning algorithm that learns the optimal policy by directly estimating the optimal action-value function $Q^*(s,a)$ regardless

of the current policy (policy being followed).

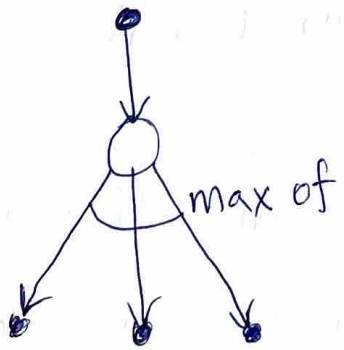
In Q-learning the agent selects an action(a) that maximizes the Q-value $Q(s,a)$ for the current state(s) and updates its Q-value based on the observed reward and the maximum Q-value for the next state.

The update rule for Q-learning is.

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

$\max_{a'} Q(s',a')$ is the maximum action-value of the next state s'

Backup diagram for Q-learning



Q-learning algorithm

Q-learning (off-policy TD control)

for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$,
small $\epsilon > 0$

Initialize $Q(s,a)$ for all states s arbitrarily
For terminal (final) state $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:
Initialize S
Loop for each step of episode:
choose A from S using policy derived from Q (e.g., ϵ-greedy)

Take action A, observe R, S'

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

$S \leftarrow S'$
until S is terminal.

ϵ-soft policy

selects the optimal (max reward giving) action with probability $1-\epsilon$
selects a random action with probability ϵ
to ensure exploration.

Expected SARSA

Expected SARSA is an on-policy TD learning algorithm that is similar to SARSA, but

→ instead of using the next action a' to update the action value as in SARSA

→ Expected SARSA uses the expected value of all possible actions a' for the next state. It takes average of all possible actions based on the current policy.

The update rule for expected SARSA is

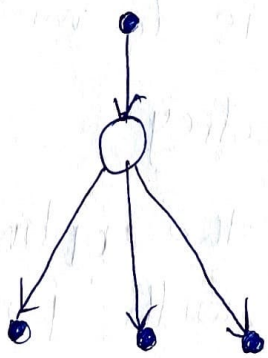
$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma E_{\pi}\left[Q(s',a') \mid s'\right] - Q(s,a)\right]$$

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

where,

$\sum_{a'} \pi(a'|s')$ is the expected value of all possible actions a' for the next states' (from)

Backup diagram for expected SARSA



UNIT-5

n-STEP RETURNS

What is the space of methods lying between Monte carlo and TD methods? consider estimating the value function V_π from sample episodes generated using policy function π.

Monte carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode.

The update of one-step TD methods is based on just the one next reward. and the estimated value of next state.

~~A two~~

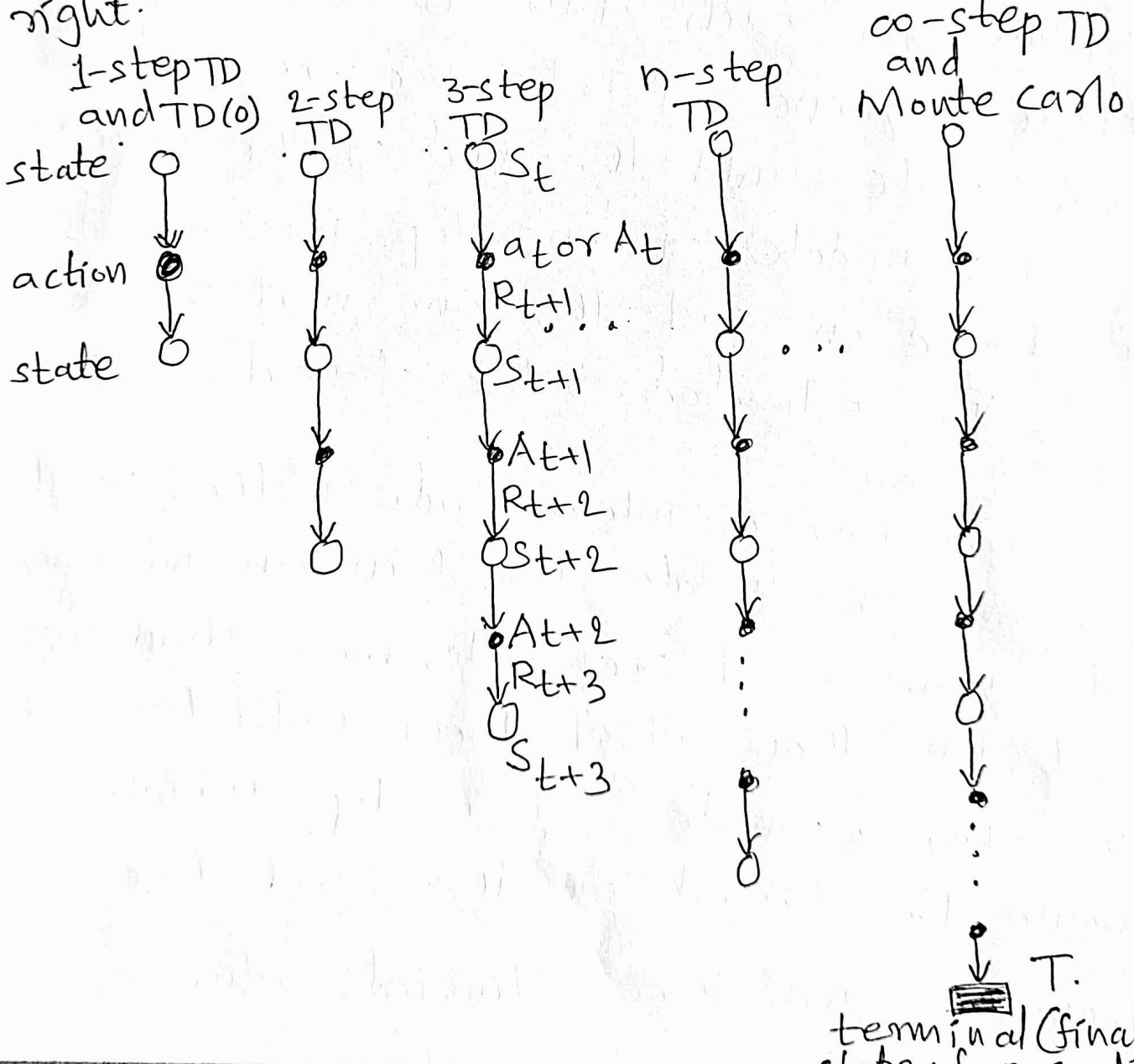
One kind of intermediate method would perform an update based on an intermediate number of rewards: more than one, but less than all of them until termination.

For example, a two-step update would be based on the first two rewards and the estimated value of

the state two steps later.

Similarly, we could have 3-step updates, 4-step updates, and so on.

The below figure shows the backup diagrams of the spectrum of n-step updates for V_π, with the one-step TD update on the left and the up-until-termination Monte Carlo update on the right.



Target of Monte carlo (MC) update

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

where T is the last time step of the episode.

1-step return

In one-step updates the target is the first reward (R_{t+1}) plus the discounted estimated value ($\gamma V_t(S_{t+1})$) of the next state S_{t+1}.

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1})$$

$t:t+1$ time steps t to t+1.

2-step return

$$G_{t:t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

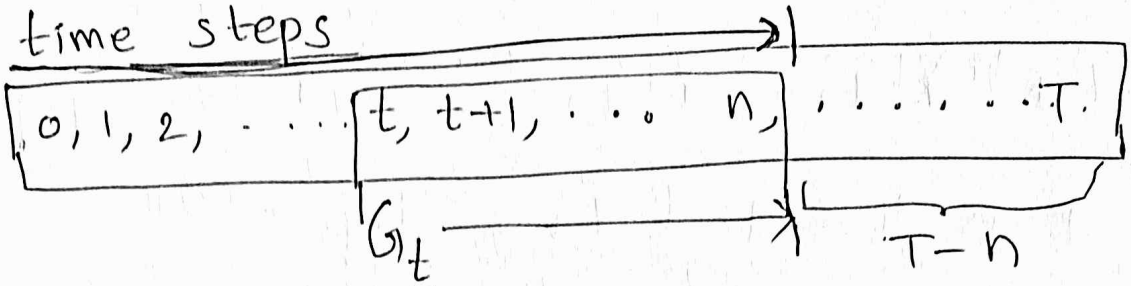
$t:t+2$ means from time step t to t+2

n-step return

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

from step t, ~~t+1~~ → to → t+n

for all n, t such that $n \geq 1$ and $0 \leq t < T-n$.



If $t+n \geq T$ (if the n-step return extends to or beyond termination step T), then all the missing terms are taken as zero, and the n-step return = ordinary full return. ($G_{t:t+n} = G_t$)

The natural state-value learning algorithm for using n-step returns is thus

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha \left[G_{t:t+n} - V_{t+n-1}(S_t) \right]$$

$0 \leq t \leq T.$

This algorithm is called n-step TD algorithm

n-step TD algorithm

n-step TD for estimating $V \approx V_\pi$

Input: a policy π

Algorithm parameters: step size $\alpha \in (0,1]$,
a positive integer n

Initialize $V(s)$ arbitrarily, for all $s \in S$

All store and access operations (for S_t and R_t) can take their index mod $n+1$.

Loop for each episode:
Initialize and store $S_0 \neq$ terminal
$T \leftarrow \infty$
Loop for time step $t = 0, 1, 2, \cdots$
{
- if $t < T$, then:
- Take (execute) an action according to policy function $\pi(\cdot|S_t)$
- observe and store the next reward (of that action) as R_{t+1} and the next state as S_{t+1}
- If S_{t+1} is terminal (final) state, then $T \leftarrow t+1$.

- $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated).

- If $\tau \geq 0$:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$$

- If $\tau + n < T$, then:

$$G \leftarrow G + \gamma^n V(S_{\tau+n}) \quad (G_{\tau:\tau+n}).$$

$$V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$$

}
until $\tau = T - 1$.

TD(λ) ALGORITHM

The λ-return

The λ-return is a way to combine the returns at different time steps. It is useful in situations where you want to credit not just the immediate reward but also rewards that may occur further in the future

λ-return is calculated by using the following formula. λ-return at time step t is

$$G_t^{\lambda} = (1-\lambda) G_t + \lambda V(S_{t+1})$$

where

- λ (lambda) is a parameter that controls the trade-off (adjustment) between considering immediate reward G_t and expected future rewards $V(S_{t+1})$.
- G_t is the actual return (cumulative reward) starting from time step t.
- $V(S_{t+1})$ is the expected future rewards

from the next state s_{t+1}.

→ When $\lambda = 0$

$$G_t^{\lambda} = (1-0)G_t + 0\,V(s_{t+1}) = G_t + 0 = G_t$$

only the immediate reward will be considered

→ when $\lambda = 1$

$$G_t^{\lambda} = (1-1)G_t + 1\,V(s_{t+1}) = 0 + V(s_{t+1}) = V(s_{t+1})$$

only the expected future rewards from state s_{t+1} will be considered.

→ In practice, intermediate values ($0 < \lambda < 1$) are often used to balance the contribution of immediate and future rewards.

Weight vector w_t of state s_t

vector → one dimensional array

$w_t = [w_1, w_2, w_3, w_4]$ → all states have 4 features

any time step t.

→ Each state has a set of features

→ these features are represented in the form of a vector, which is also called as weight vector.

weight vector w is vector of weights of feature values of states.

d = dimensionality of w

$$w = [w_1, w_2, w_3, \cdots, w_d]$$

→ changing one weight changes the estimated value of many states

→ consequently, when a single state value is updated, the change generalizes from that state to affect the values of many other states.

→ similar to a multi-layer ANN, with w the vector of connection weights in all the layers.

Eligibility trace vector

It is a mechanism used to assign credit or blame to different past (state-action) pairs for the outcomes of an agent's interactions with its environment.

Based on that past states, actions, rewards the agent determines how good it is (or eligibility of take action

a to take in state s) in future.

With function approximation, the eligibility trace is a vector $z_t \in \mathbb{R}^d$ with the same number of components as the weight vector w_t. (vector with d values) Whereas the weight vector is a long-term memory, accumulating over the lifetime of the system, the eligibility trace is a short-term memory, typically lasting less than the length of the episode.

Eligibility trace assist in the learning process, their only consequence is that they (z_t) affect the weight vector (w_t) and then the weight vector determines the estimated value.

In TD(λ), the eligibility trace vector is initialized to zero at the beginning of the episode, is incremented on each time step by the value gradient $\nabla v(s_t, w_t)$ [represents the.

change in the value function for a state S_t with respect to the weight vector w_t. It signifies the update to the value estimate for that state-action pair. This update is used to adjust the weights in the function approximation of the value function] and then Z_{t-1} fades away by $\gamma\lambda$.

$$Z_{-1} = 0$$
$$Z_t = \gamma\lambda Z_{t-1} + \nabla\hat{v}(S_t, w_t), \quad 0 \le t \le T.$$

The TD error for state-value prediction is

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)$$

where
$\hat{v}(s, w)$ is the approximate value of state s given weight vector w.

In TD(λ) the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace.

$$w_{t+1} = w_t + \alpha\delta_t Z_t$$

TD(λ) algorithm

For estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : S^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$.

Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$

Initialize value-function weights w arbitrarily (e.g. $w = 0$).

Loop for each episode:

Initialize S $z \leftarrow 0$ [a d-dimensional vector]

Loop for each time step of episode:

choose Action $A \sim \pi(\cdot|S)$

Take action A, observe R, S'

$z \leftarrow \gamma\lambda z + \nabla \hat{v}(S, w)$

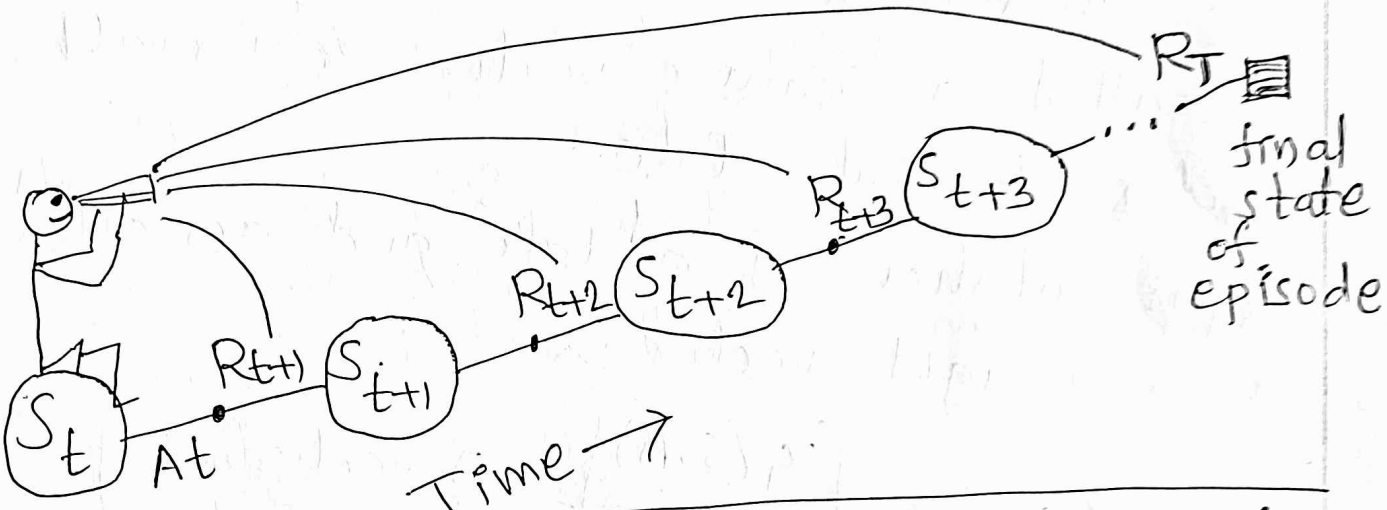
$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$

$w \leftarrow w + \alpha \delta z$

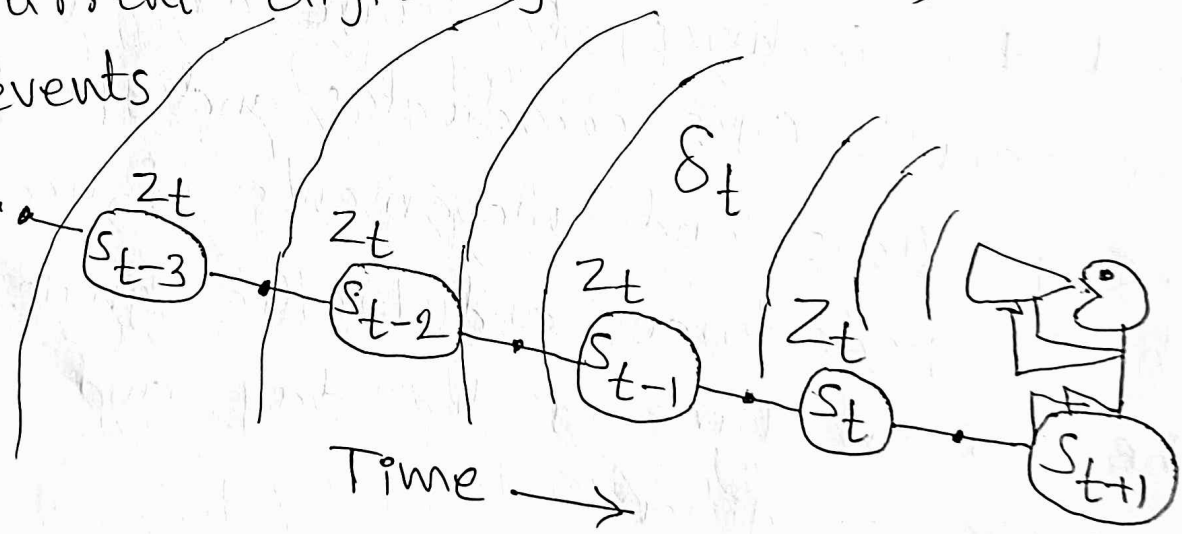
$S \leftarrow S'$

until S' is terminal [final state of episode].

In forward view of learning algorithm, we decide how to update each state by looking forward to future rewards and states



In the backward or mechanistic view of TD(1), each update depends on the current TD error(δ_t) combined with the current eligibility traces (z_t) of past events



Feature vector (or) weight vector

Each state has a set of features. These features are represented as a vector [one dimensional vertical array] which is called a feature vector or weight vector of that state.

Representation of a state S and an action A by a weight vector X

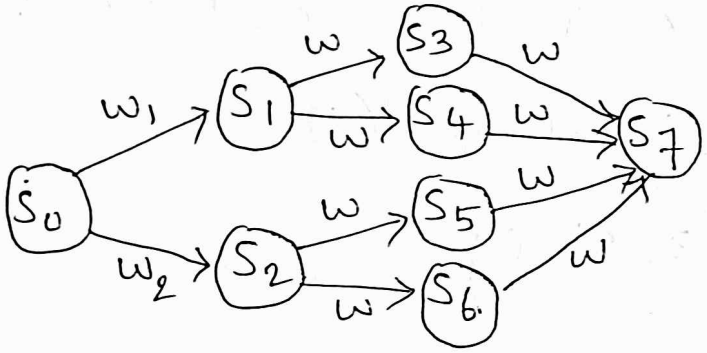
$$w = X(S,A) = \begin{bmatrix} x_1(S,A) \\ x_2(S,A) \\ \vdots \\ x_n(S,A) \end{bmatrix}$$

A vector of n features

For an example of autonomous driving agent the features of a state are like

1) The agent's GPS coordinates and speed.
2) The positions and movements of nearby vehicles, pedestrians and traffic signals.
3) The current time of the day and weather condition
4) The road type and traffic density.
5) The recent history of its actions

such as whether it has recently accelerated, changed lanes, or come to a stop. Weight vectors of states can be represented as an ANN (Artificial Neural Network).



changing one weight changes the estimated value of one state.
consequently, when a single state value is updated (changed), the change generalizes from that state to affect the values of many other states.

LINEAR FUNCTION APPROXIMATION AND GEOMETRIC VIEW.

Linear function approximation is a technique used in RL to estimate the value function (or action-value function) using a linear combination of features or

input variables associated with states (or state-action pairs).

Linear function approximation assumes that the value function $V(s)$ or $Q(s,a)$ can be approximated as a linear combination of feature vectors and weight.

For state-value function $V(s)$

$$V(s) = \sum_i \phi_i(s) \times w_i$$

For action-value function $Q(s,a)$

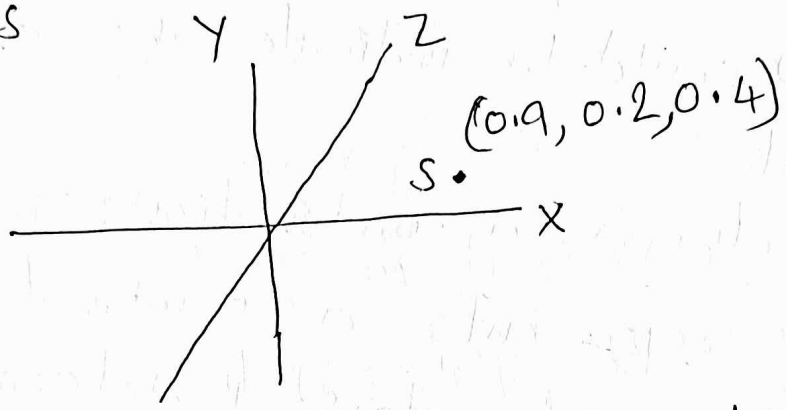
$$Q(s,a) = \sum_i \phi_i(s,a) \times w_i$$

where

- $V(s)$ is the estimated value of state s (or $Q(s,a)$ for state-action pairs.
- $\phi_i(s)$ (or $\phi_i(s,a)$ represents the i^{th} feature vector associated with state s (or state-action pair (s,a)).
- w_i is the weight parameter associated with the i^{th} feature.

Geometric view of Linear function approximation

Geometric view of a state s with three features



In the geometric view, each state is represented as a point in a high-dimensional input space, and the features are vectors in this space that define specific direction or axes.

The geometric view helps us understand how linear function approximation works by considering the relationship between feature vectors, weight vectors, and value estimation

Feature vectors as a Basis: Think of the feature vectors $\phi_i(s)$ as forming a basis for the space of states. These features capture relevant information

about the state that the value function should take into account. For example $\phi_1(s)$ could be a binary indicator for whether the agent is in a specific room, while $\phi_2(s)$ could ~~be~~ indicate the distance to a goal.

Weight vector as a combination: The weight vector "w" represents a combination of these (above) basis vectors. It determines how much importance is given to each feature when estimating the value of a state. The weight vector defines a plane in the feature space.

Value estimation as Projection: The value estimation $V(s)$ is essentially the projection of the state s onto this (above) plane defined by the weight vector. The closer the state is to this plane, the more accurate the value estimate. In other words

$$V(s) = w \cdot \phi(s)$$

where $\cdot$ represents the dot product.

Learning Weights: During training, the RL algorithm adjusts the weights (w_i) to minimize the error between the estimated value and the true value. This adjustment effectively rotates the weight vector in the feature space, finding the best plane that approximates the value function.

NEED FOR GENERALIZATION IN PRACTICE

Generalization is a crucial concept in RL and function approximation in practice. It refers to the ability of a learned value function to make reasonable predictions and estimate the value of states (or state-action pairs) that it has not encountered during training. Generalization allows RL algorithms to transfer knowledge from observed states to unseen or unvisited states and thus handle large or continuous state spaces effectively.

The following are a few reasons why generalization is essential in practice.

1) Scalability: In many real world problems, the state space can be vast or continuous. Training a value function for every possible state is often infeasible or impractical. Generalization allows an agent to generalize its knowledge from a limited set of observations to effectively navigate this large state space.

2) Efficiency: Generalization can significantly reduce the number of samples required for learning. Instead of learning from every state individually, the agent can learn from a representative subset and generalize its knowledge to similar states.

3) Adaptability: Generalization enables the agent to adapt to changes in the environment. If the environment changes slightly or if new states are introduced, a well generalized value function can still make reasonable predictions in these new situations.

4) Transferability: Generalization facilitates knowledge transfer between related tasks. A value function learned in one task can often be adapted for use in a similar but distinct task.

To achieve generalization in RL, function approximation techniques are often employed, including linear function approximation, neural networks, decision trees, and more. These techniques allow the value function to generalize by approximating the value of unseen states based on the features extracted from the observed states.

LINEAR TD(λ)

Linear TD(λ) is a RL algorithm that combines linear function approximation and eligibility traces for estimating the value function efficiently. It is suitable for RL tasks with large state spaces.

Key components

1) Linear function approximation

$$\hat{V}(s,w) = x(s)^T w$$

- Represents the value function as a linear combination of features
- w is the weight vector
- x(s) is the feature vector of state s

2) Temporal difference (TD) learning

$$\delta_t = R_t + \gamma \hat{V}(s', w) - \hat{V}(s, w)$$

3) Eligibility traces (λ)

$$z_{t+1} = \lambda \gamma z_t + x_t$$

4) weight update rule

$$w_{t+1} = w_t + \alpha \delta_t z_t .$$

Applications

1) Efficient approximation: Useful in environments with large state spaces where a linear approximation is effective

2) Allows generalization across similar states reducing the need for specific values for each state.

3) memory efficiency: Eligibility traces efficiently manage credit assignment over time, reducing computational costs.

Limitations

1) Linear assumption: The assumption that the true value function is linear may not hold in all environments

2) Choosing λ: The choice of the eligibility trace parameter (λ) can impact performance and requires careful tuning.

3) Overestimation (or) Underestimation: Linear function approximation may lead to overestimation or underestimation of values especially in complex environments.

TILE CODING

Tile coding is a method used to discretize continuous state spaces in RL. It involves partitioning the state space into multiple tiles or grids and then assigning each tile a unique identifier or index. This provides a

way to map continuous states to a discrete representation

How it works

1) Tiling

- The state space is divided into multiple overlapping tiles or grids
- Each tile is a subset of the state space and has a unique identifier

2) Encoding

- A state is encoded by determining which tiles it intersects with
- Each tile gets assigned a binary value indicating whether the state is inside or outside that tile.

3) Representation The combination of binary values across all tiles forms a compact, discrete representation of the continuous state.

Benefits of Tile coding in function approximation

1) Generalization

- Tile coding allows for generalization across similar states within a tile.

- Encoded states in the same tile share similar features, aiding generalization in function approximation.

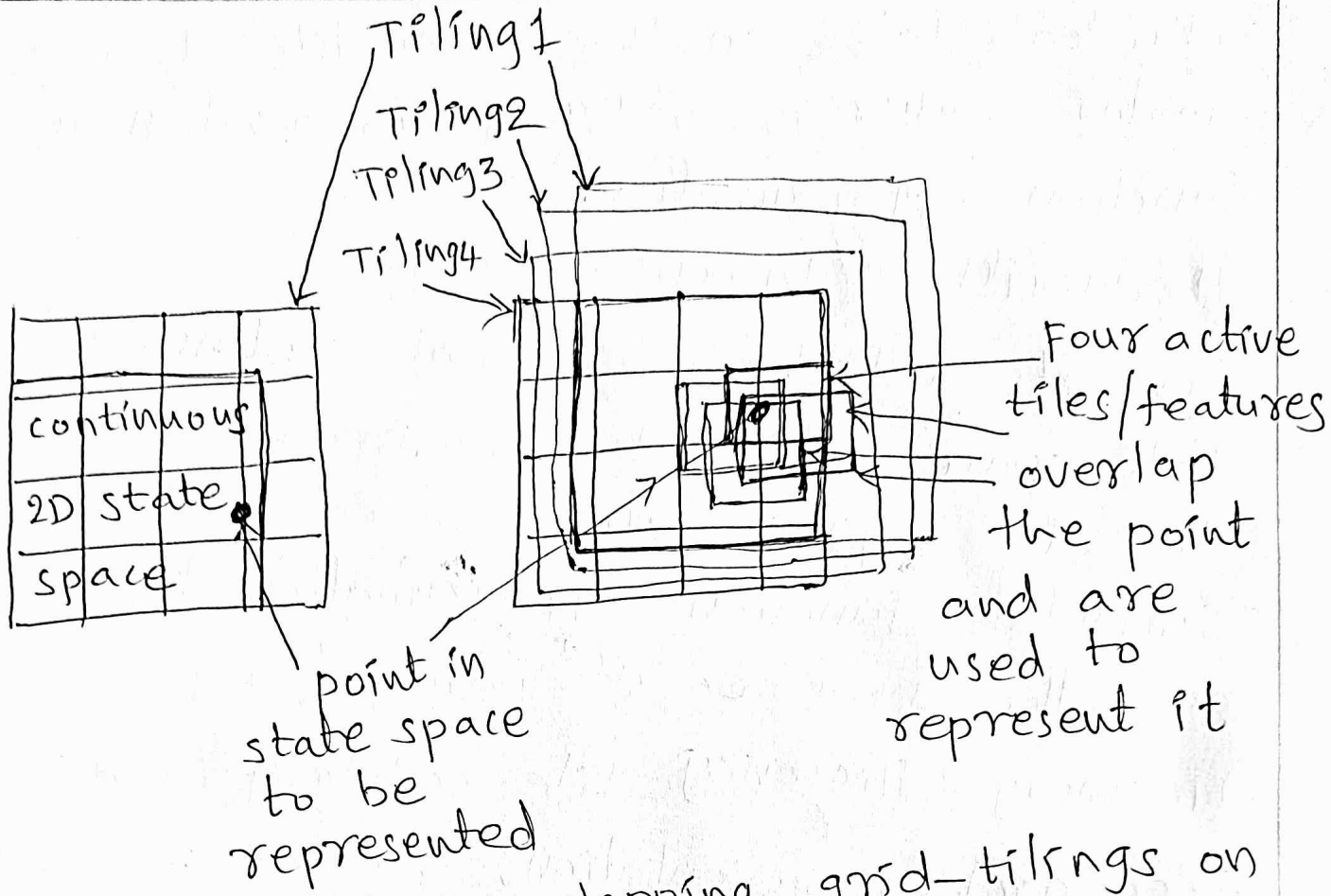
2) Reduction of Dimensionality

- converts a high-dimensional continuous state space into a lower-dimensional discrete representation
- Simplifies function approximation by reducing the number of parameters.

3) Memory Efficiency Tile coding provides a compact representation, reducing memory requirements compared to storing values for all possible states.

4) Improved Learning Helps in learning the value function in a more sample-efficient manner by discretizing the state space and focusing on critical regions.

5) Handling continuous Features Suitable for problems with continuous features where traditional table-based methods become impractical.



Figure; multiple, overlapping grid-tilings on a limited 2-D space. These tilings are offset from one another by a uniform amount in each dimension

CONTROL WITH FUNCTION APPROXIMATION

Control with function approximation in RL is a technique used to handle large or continuous state or action spaces by approximating the value or action-value function instead of explicitly representing all states and actions.

In traditional tabular RL, the agent learns the value or action-value function for each combination of state and action. However, in scenarios with large or continuous state or action spaces, this approach becomes impractical due to the large number of possible combinations.

In control with function approximation, a function approximator is used to estimate the value or action-value function. This function approximator maps a state or state-action pair to its corresponding value.

There are various types of function approximators that can be used, such as

Linear regression
Artificial Neural Networks
Decision Trees and
Kernel methods.

These approximators learn a mapping between the state or state-action space and the estimated values through training.

During training, the function approximator is updated to minimize the error between the predicted values and the true values. This is typically accomplished using optimization algorithms like gradient descent or its variants.

POLICY SEARCH

Policy is a strategy the agent uses to decide what actions to take in different states.

Policy search is a class of methods in RL that directly optimize the policy itself, rather than estimating the value or action-value functions. The key idea behind policy search is to find the optimal (maximum reward giving) policy by searching directly in the space of policies.

The key ideas behind policy search methods are as follows

1) Parameterized policy: A policy is represented by a set of parameters that can be adjusted to improve the policy's performance. Common parameterizations include probability distributions over actions or explicit parameterizations like a neural network.

2) objective Function: Policy search methods define an objective function that quantifies the quality or performance of a policy. This objective function measures how good the policy is in terms of expected cumulative rewards

3) optimization: The objective function is maximized or minimized to find the best set of parameters that correspond to an optimal policy. This optimization can be performed using various techniques like gradient descent, evolutionary algorithms, Bayesian optimizations, or natural gradient methods.

Policy search methods differ from value-based methods in several ways.

1) Policy Vs value: value-based methods aim to estimate the value or action-value functions and then derive an optimal policy based on these estimates. In contrast, policy search methods directly optimize the policy itself

2) Model-Free: Policy search methods are model-free, meaning that they do not require knowledge of the environment's dynamics. They solely rely on interacting directly with the environment to learn and improve the policy.

3) Search in policy space: Instead of searching in the space of value functions, policy search methods search in the space of policies, adjusting the policy parameters to optimize the objective function

4) Handling large or continuous spaces: policy search methods are well suited to handling large or continuous state or action spaces as they directly optimize

the policy, bypassing the need for explicit value function estimation.

POLICY GRADIENT METHODS

Definition Policy gradient methods are a class of RL algorithms that directly optimize the policy parameters to find the optimal policy. Instead of estimating value or action-value functions, these methods search in the policy space by adjusting the policy parameters to maximize a performance measure, typically the expected cumulative rewards.

Different policy gradient methods (VPG)

1) Vanilla Policy Gradient (REINFORCE): one of the simplest policy gradient algorithms that uses Monte Carlo estimation of the expected return to compute the gradients.

2) Proximal Policy Optimization (PPO): A popular policy gradient algorithm that uses a surrogate objective function and uses a loss function that encourages small parameter updates to improve

stability and mitigate large policy changes

3) Trust Region Policy Optimization (TRPO): Another policy gradient algorithm that enforces a constraint on the size of policy updates to avoid large policy divergences during optimization

4) Actor-Critic Methods: These combine the benefits of both policy-based and value-based methods by incorporating both an actor (policy) and a critic (value function). The actor provides policy update directions, while the critic estimates the value function and provides a baseline for the policy improvement

Use of policy gradient methods in RL

Policy gradient methods are used in RL to optimize parameterized policies directly.

They offer several advantages such as handling continuous action spaces, incorporating exploration naturally, optimizing stochastic policies, and providing.

greater sample efficiency compared to value-based methods

These methods have been successful in various domains (applications), including Robotics, Game playing, Simulation, NLP and have contributed significantly to the advancements in RL algorithms and applications.

Deep Reinforcement Learning is a branch of ML that combines deep learning with RL techniques. It involves training a network of ANNs to make sequential decisions in an environment to maximize a reward signal. The agent learns complex tasks by iteratively exploring and interacting with the environment, using a combination of raw sensory input and feedback received after each action taken. Through the integration of deep neural networks, it can handle high-dimensional and unstructured data, enabling the learning of intricate behaviors and strategies. This approach has been

successful in solving challenging tasks like playing complex games, robotic control and autonomous systems.

EXPERIENCE REPLAY

Experience replay is a fundamental technique in deep reinforcement learning (DRL) that enhances the stability and efficiency of learning algorithms.

It involves storing and reusing past experiences (i.e, sequences of states, actions, rewards) in a replay buffer memory.

During learning, instead of basing updates solely on the current state-action pair, the algorithm samples a batch of experiences from the replay buffer, breaking the correlation between consecutive experiences. By doing so, experience replay helps stablize the learning process, reduce the variance (difference) of updates, and prevent the network from overfitting to recent experiences.

Experience replay is particularly valuable in DRL due to two main reasons.

First, it enables efficient reuse of experiences, as NNs can quickly compute multiple updates from a single batch of experiences. This improves data efficiency by extracting more information each experience.

second, experience replay breaks the temporal correlation between experiences, which are critical in RL scenarios. With this, the network can learn from a more diverse set of experiences and explore different decision-making strategies, leading to better overall performance and faster convergence.

Therefore, experience replay plays a crucial role in DRL algorithms, enhancing their stability, efficiency, and ability to learn complex tasks.

FITTED Q ITERATION (FQI)

Fitted Q Iteration is an approach to approximate (update) Q-values using function approximation in RL. It works by iteratively updating the Q-values of state-action pairs.

using a set of observed transitions (s, a, r, s') collected during interaction with the environment. The key idea is to approximate the optimal Q-function, by fitting a function approximator, such as a Neural network, to the observed transitions.

The core equation for FQI can expressed as follows.

$$Q_{K+1}(s,a) = \underset{\theta}{\operatorname{argmin}} \sum_i \left[Q_K(s_i, a_i; \theta) - \left(r_i + \gamma \max_{a'} Q_K(s_{i+1}, a'; \theta) \right) \right]^2$$

where

$Q_{K+1}(s,a)$ represents the updated Q-function after the $K+1^{th}$ iteration

$Q_K(s_i, a_i; \theta)$ is the current estimate of the Q-function with parameters θ at state s_i and action a_i

r_i is the observed reward after taking action a_i in state s_i

s_{i+1} is the next state after taking action a_i

γ is the discount factor

θ represents the parameters of the function

approximation used to estimate the Q-values. Typically θ corresponds to the weights of a neural network if a neural network is employed as the function approximator for the Q-function

The optimization process involves updating these parameters to improve the accuracy of the Q-value estimates during each iteration of the FQI algorithm

FQI works as follows

Data collection: Initially, the agent interacts with the environment to collect a dataset of state-action (s,a) pairs along with their corresponding observed rewards (r) and next states (s') (s,a,r,s') or (s_i, a, r_i, s_{i+1}).

Initialize Q-function: An initial estimate of the Q-function is made, typically using a function approximator such as a neural network.

FQI Iterations: The Q-function is iteratively refined (improved) using a series of updates. In each iteration, the dataset is used to

create a training set for the Q-function. It works by iteratively updating the Q-values of state-action pairs using a set of observed transitions (s, a, r, s') collected during interaction with the environment. The Q-values are updated [see the above equation] by minimizing the difference between the predicted Q-values and the target values, which are calculated based on the observed rewards and the estimated Q-values from the previous iteration.

Convergence: The iterations continue until the Q-function converges (meets) to a satisfactory approximation of the true Q-values.

Advantages of FQI

Function Approximation: FQI allows for the use of function approximation techniques, such as NNs, to represent the Q-function. This is crucial in handling large and continuous state spaces where a tabular representation is infeasible

Sample efficiency: FQI can be more sample-efficient compared to traditional Q-learning methods, as it leverages [use something

to maximum advantage] the collected dataset to iteratively update the Q-function, making the most out of the available data.

Generalization: The use of function approximation enables the Q-function to generalize across similar states, improving the agent's ability to transfer knowledge to unseen situations.

Suitability for Deep learning: FQI is well-suited for DRL, where NNs are employed as function approximators. This allows for the exploration of DL's capacity to learn complex and hierarchical representations from raw sensory input.

FQI is an effective approach to approximate Q-values using function approximation, making it suitable for high-dimensional state spaces and Deep Reinforcement Learning (DRL) scenarios.

** APPLICATIONS AND CASE STUDIES OF RL

* Real-world RL scenarios

1) Tic-Tac-Toe game
2) TD-Gammon

3) Samuel's checkers Player

4) The Acrobot game

A robotic arm is composed of two points and two links and the joint between the two links is actued. (move or activate) ~~or~~

5) Elevator Dispatching

6) Dynamic channel Allocation for networks

7) Job-Shop scheduling.

Multiple jobs are processed on several machines.

8) Mastering the game of Go (Go game)

- Alpha Go
- AlphaGo Zero