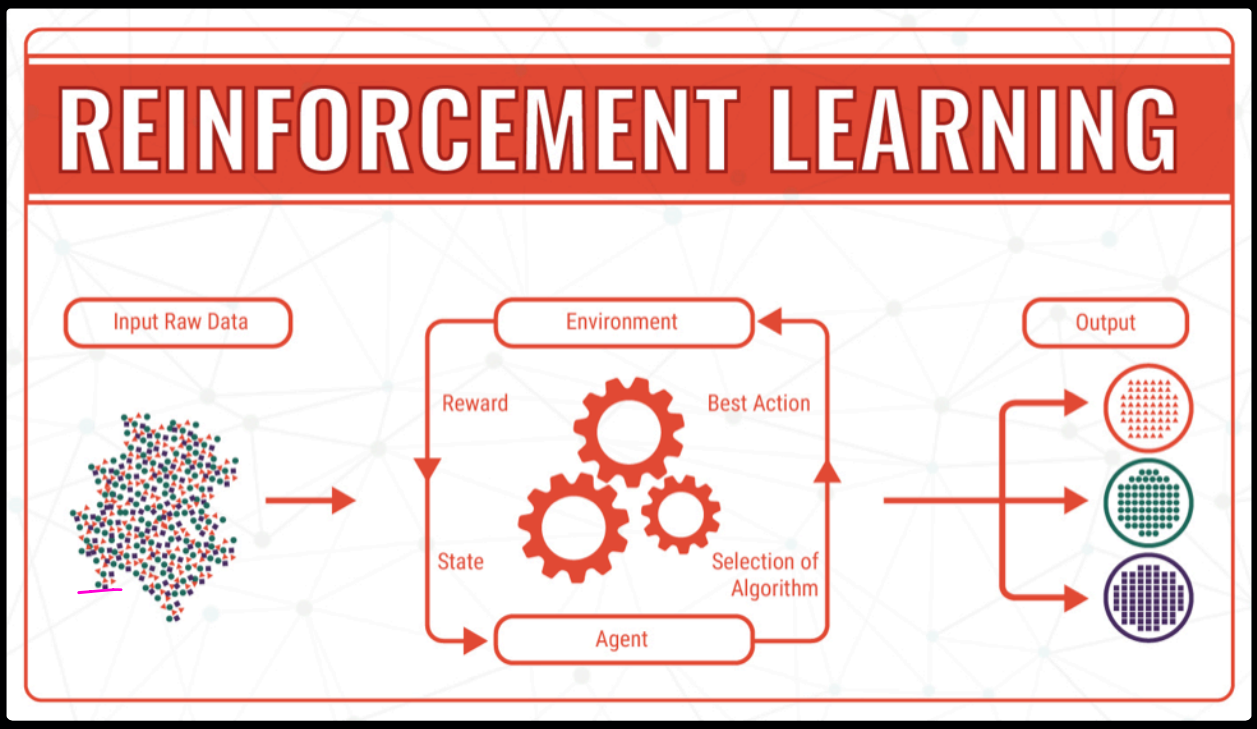
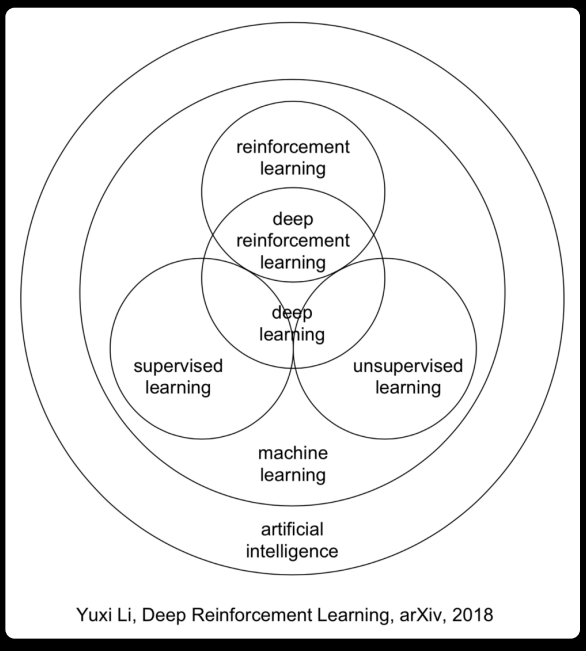
This lecture is completely adapted from Stanford lecture series: CS-231n by Serena Yeung





Yuxi Li, Deep Reinforcement Learning, arXiv, 2018

(A) → Introduction to learning paradigms

There are (3) major paradigms of m/c learning

(I) Supervised Learning

(i) Data : (x, y)
- x → data
- y → label

(ii) Goal : Learn a fn that can map $(x \rightarrow y)$

Examples : Classification (Cat Vs Dog), Object detection, Semantic segmentation, Regression $[f(x) : y]$, image Caption.

Anything in which target is known/ given atleast for training

(II) Unsupervised Learning

(i) Data : (x) w/o labels

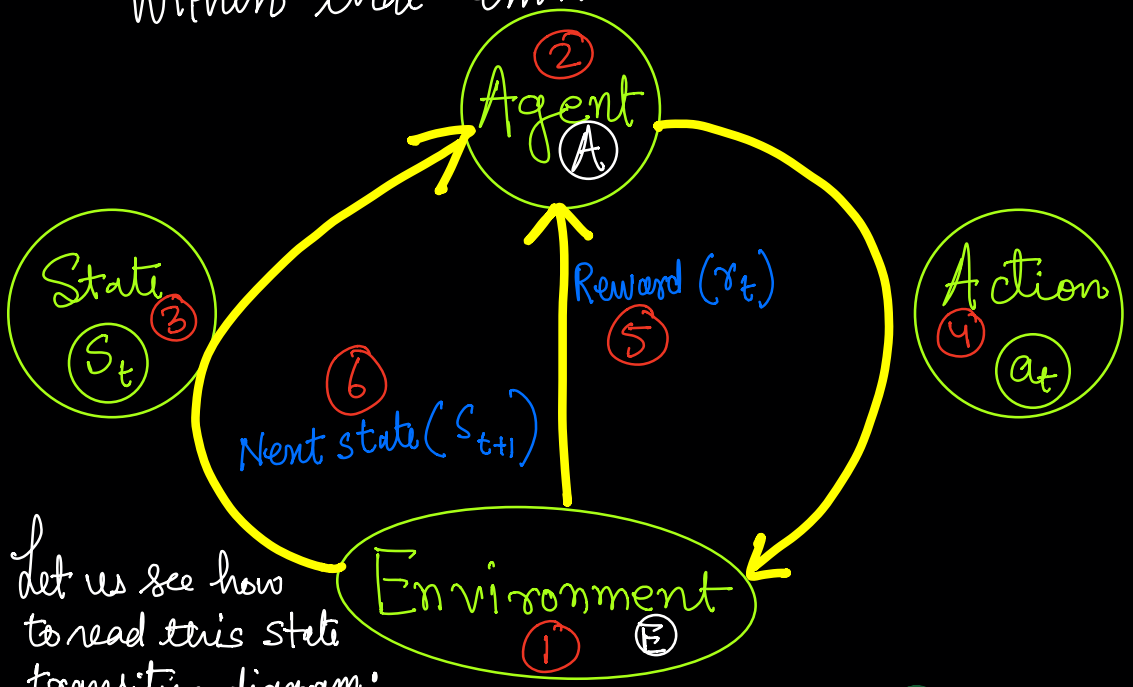
(ii) Goal : learn some underlying hidden relationship between the data that can be used to explore the structure of data

Examples : Clustering, dimensionality reduction, feature learning, density estimation.

In the absence of target underlying data probability distributions can be estimated and used for various purposes. Such Generative models.

(III) **Reinforcement learning** : It is a learning paradigm in which learning happens by exploration. So agent interact with the environment repeatedly w/o any prior knowledge and labels wrt the environmt.

— Completely relying on hit and trial strategy to learn how to behave optimally within that environment.



Let us see how to read this state transition diagram:

(*) The agent (A) is at some state (S_t) at time (t).

↳ It is interacting with Environment (E), via. few defined set of actions (A).

Action at any state is choosen only by utilizing the state (S_t). It assumes that (S_t) completely characterises the Environmental state. (Req. No History) Markov

(*) At any state s_t, taking a particular action say a_t is going to fetch a reward r_t.

(*) The final Goal of the agent is :

To learn how to take an appropriate (optimal) action so as to maximize the reward.

(Future discounted reward maximization)

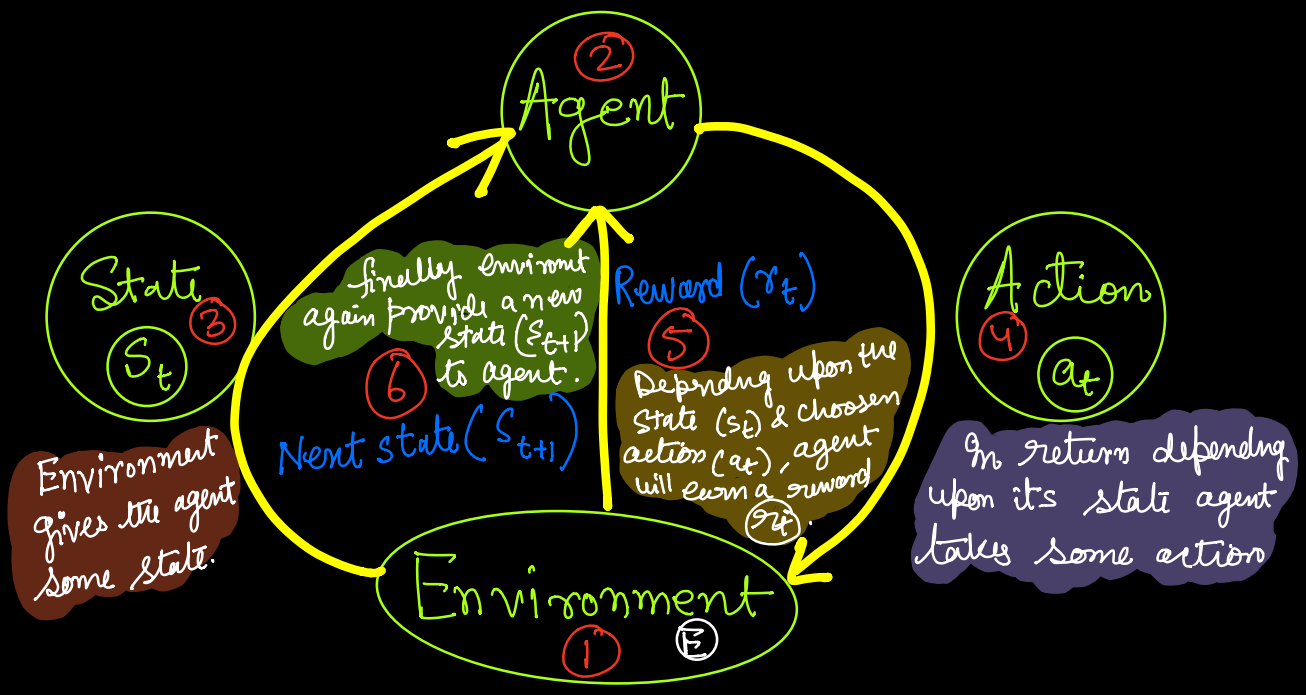
(*) Re-inforcement learning can be formulated as Markov decision process ⇒ MDP.

(*) There are (2) major classes of RL- algorithms

Q-Learning

Policy gradients

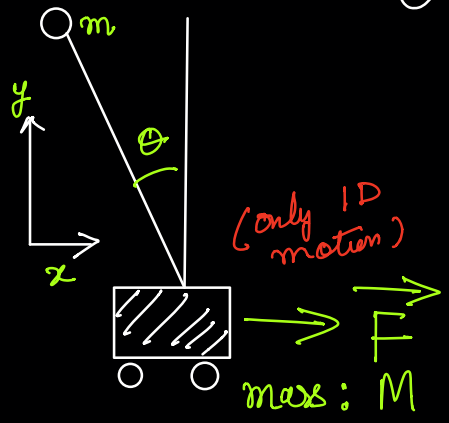
(B) → Basic RL-setup/framework



Episodic training : These six steps keeps on repeating in a loop and considered as an episode. The episode continues untill the environment provides a terminal state (final) to the RL-Agent, after which the current episode get terminated.

(*) Hence one need to generate data (episodes) by allowing agent to explore the environment in episodes. Later use these episodic data for training the agent to learn how to behave optimally wrt the environment.

Example-01 : Cart Pole problem.



(A) Objective : To balance the pole on the top of a movable Cart.

(B) State : The physical state can be encoded as : angle (θ), angular speed (ω), position (x, y), horizontal velocity (v_x) (may be more)

(C) Action : Horizontal force vector applied on the Cart.

(D) Reward : (+1) each time, if the pole is upright.

Example-02 : Atari game playing agent.

(A) Objective : Finish the game with the highest score.

(B) State : Game state can be raw pixel input of the game screens. (Images of game screen)

(C) Action : Game controlling actions, (L / R / U / D).

(D) Reward : At the end of the episode if win then all <state, action> pair will get a (+1) reward else they will get (-1) reward.

C → Mathematical formulation of RL.

(*) RL can be mathematically formulated using Markov Decision processes (MDP).

C.1 Markov property :

The future is independent of the past given the present.

Def : A state (S_t) is Markov iff

$$P[S_{t+1} | S_t] = P[S_{t+1} | S_1, S_2 \cdots S_t]$$

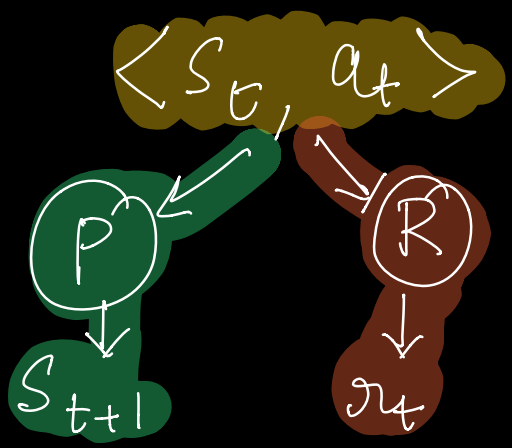
(*) State captures all relevant informations from the history. State is a sufficient statistic of the future.

C.2 Markov Decision Process:

MDP is a decision process, choosing action at each state. It is an environment in which all states are Markov. The MDP can be characterized by tuple:

$\langle S, A, P, R, \gamma \rangle$

- **S**: Set of all possible states: $s_1, s_2 \dots s_t \dots s_N$.
- **A**: Set of all possible actions: $a_1, a_2 \dots a_t \dots a_K$
- **P**: State transition Matrix: $\langle s_t, a_t \rangle \rightarrow s_{t+1}$
- **R**: Reward fn: $\langle s_t, a_t \rangle \rightarrow r_t$
- **γ**: Discount factor $(0-1)$ (How much current rewards are important as compared to future.)



C.2.1 State transition Matrix (P)

Markov process/chain is a memory less random process of sampling iteratively as per the given STM (P), staring from some given (seed) random starting state and following Markov property.

Full game dynamics can be encoded within (P). Robot can be interacting with with the gaming environment following (P) $\Rightarrow$ The game rules.

$$P^{a}_{s\,s'} = P\left[S_{t+1} = s' \,\middle|\, S_t = s,\ a_t = a\right]$$

C.2.2 Immediate reward

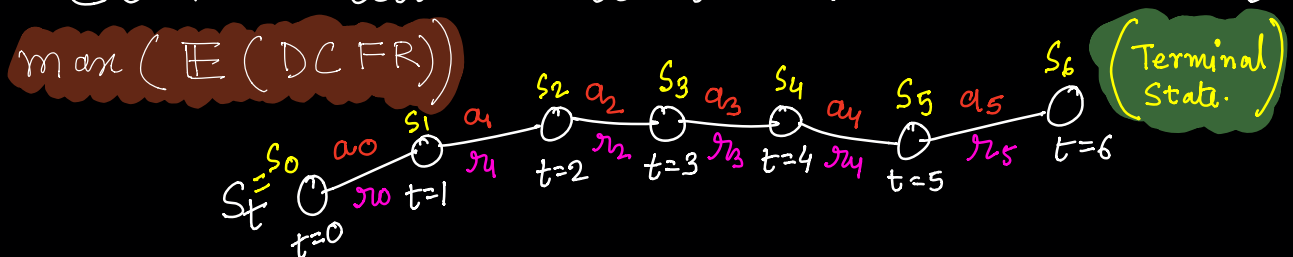
Amount of reward an agent is going to get from the environment when at state S_t agent takes an action a_t.

$$r_{s_t}^{a_t} = r(s_t, a_t)$$

Greedy approach

* But in MDP, we are not intersted in immediate rewards. An optimal behaviour must need to maximize the Average Discounted Cumulative Future Reward (DCFR)

$\max(\mathbb{E}(DCFR))$



How MDP operates and can be used to obtain Episodes.

MDP – Execution

(1) At time ($t=0$), environment may sample randomly an initial state $s_0 \sim p(s_0)$

Initial state probability distribution (ISPD)

(2) from $t=0$ (until terminates)

(2.1) An agent can select an action (a_t) [HOW!!!]

using some policy that we need to figure out so as to max ($\mathbb{E}$(DCFR))

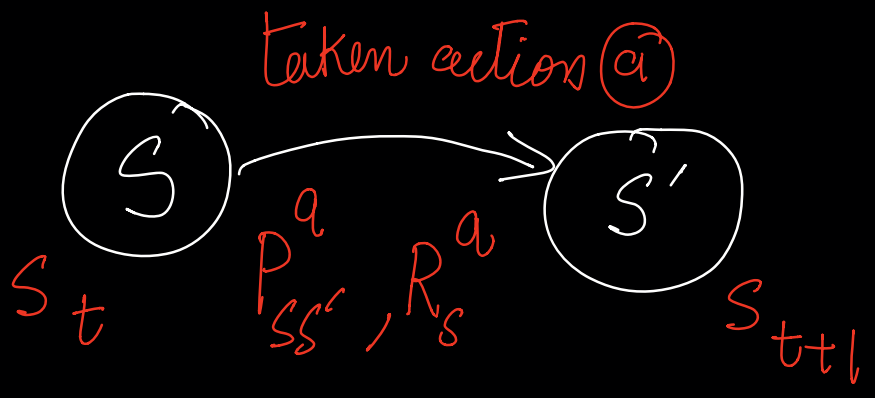
(2.2) Depending upon the agent's state s_t and the action taken a_t environment will return : r_t the immediate reward as well as next state s_{t+1} using R & P.

∴ for a MDP P & R governs.

$$\Rightarrow P^{a}_{ss'} = P\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$$

$$\Rightarrow R^{a}_{s} = \mathbb{E}\left[r_t \mid S_t = s, A_t = a\right]$$

given a discount factor $\gamma \in [0,1]$



C.2.3 But how to take an action ⇒ Policy (π)

A policy (π) is just a mapping of states (S_t) to actions (a_t) enabling any agent to take action

$$\mathbb{R}^{\text{State}} \longrightarrow \mathbb{R}^{\text{action}}$$

(*) It can be deterministic or stochastic (Random). Defined

as $\pi(a_t | s_t) = P[a_t | s_t]$

C.2.4 Goal of MDP

The objective of MDP is to find an optimal policy (π^*), that can maximizes the DCFR.

(*) The actual returns are always in the form of DCFR.

G_t = Discounted cummulative future Reward from time step (t). [future rewards are geometrically weighted]

$= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$

for all upcoming future time step

$$= \sum_{K=0}^{\infty} \gamma^K R_{t+K} \quad (\gamma \in (0,1))$$

{Gamma}

γ → close to (0) "Myopic" evaluation.

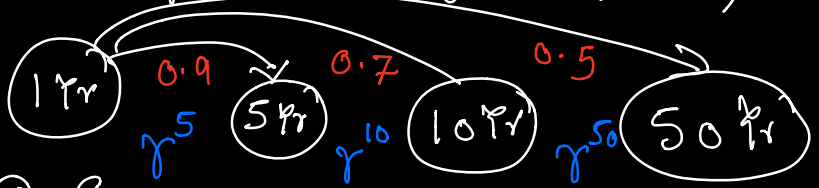
More important to immediate returns.

→ Close to (1) "far-sighted"

Care about all future rewards

⊛ Requirement of discount :

(a) future is uncertain, and we care about :
If I invest 1K-INR (Today) ⟹ then how do I need to care about my return after (1 Yr), (5 Yr), (10 Yr) ---.



$\gamma \in \{0-1\}$

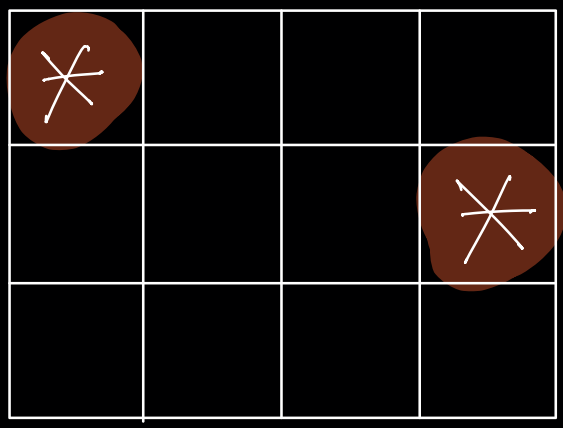
(b) Since we are predicting using an inaccurate model of environment hence the trust over model predictions can decrease exponentially. [Reason]

(c) Mathematically Convenient to discount rewards.

(d) Avoids infinite returns in Cyclic Markov processes.

C.2.5 Example Grid World – MDP

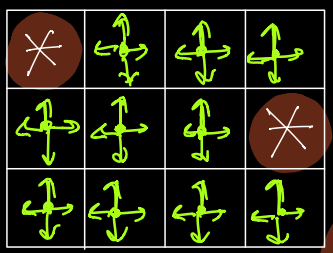
(A) States (2D-locations) (B) Actions = {right, left, up, down}.



(C) Reward = $\{r_t = -1\}$ negative reward for each transition. It ensures to learn how to reach target fast.

(D) Objective: Starting from any randomly choosen block ⟹ Reach to a terminal state (*) in the least number of actions.

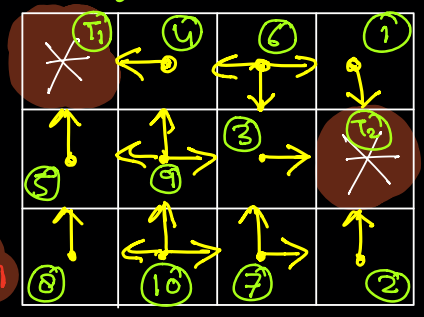
(E) Given the above 2D-Grid world MDP we need to figure out some optimal policy that can maximize the DCFR. (Out of all the policies π)



Random policy (A) (π_1)

Randomly choosing any direction.

Visual policy realization



(B) (π_2)

Policy philosophy: (a) For immediate neighbours:

One should have to move in a direction that can take you to the terminal state.

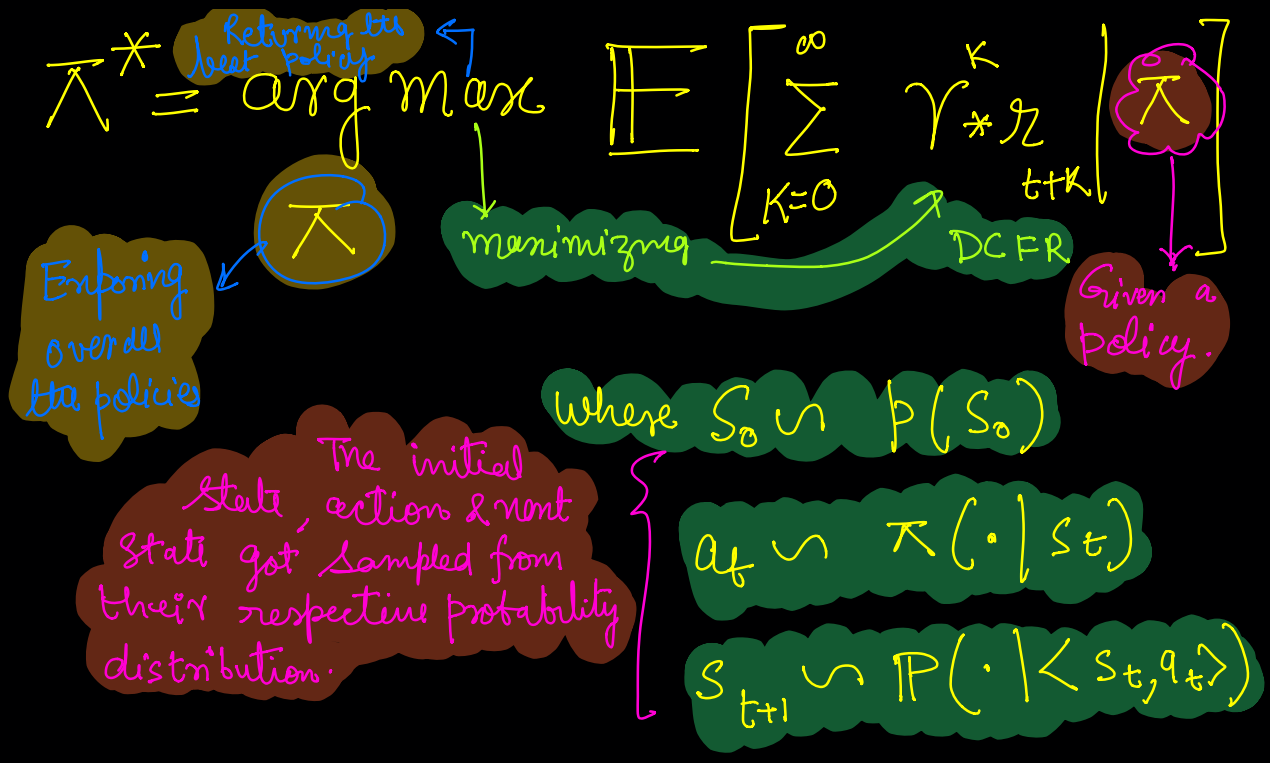
(b) Other states: Move in the direction that can take you closest to one of the immediate neighbouring state. $\pi(a_t|S_t)$ [Mapping from $S_t \rightarrow a_t$]

(D) → Solution of MDP: Optimal Policy (π^*)

Policy ≋> Planning

It do not matter that in whatever state (say S_t) agent is at time t, (π^*) is going to suggest that "what actions (a_t) agent needs to take in order to get the DCFR Maximized. $\forall_{a_i} \pi(a_i|S_t)$

(*) Such MDP's are realized as stochastic framework with initial probability distribution or transition probability distribution. Hence in order to handle the randomness we always talk about Expected value of the reward maximization.



(*) There are few attributes associated to a policy:

$\pi(a_t|s_t)$

(A) Value fn - Wrt a given policy (π), How good is a given state (S), i.e if agent just follow (π) How much DCFR is expected is the value of that state (S) for a given policy (π).

(B) Q-value fn - Wrt a given policy (π), How good is a state action pair. i.e instead of directly following the policy (π) at state (S), agent first takes an action (a) and then follow the policy (π). Q value of $\langle S, a \rangle$ is the expected DCFR.

D.1 → Value function $[V^{\pi}(s)]$

Defined as the expected DCFR if agent follow π at state s until the episode terminates (reaching terminal state)

↳ Given a policy and a initial state $s_0 \sim p(s_0)$; Episode is $(s_0, a_0, r_0) \rightarrow (s_1, a_1, r_1)$

$\leftarrow (s_2, a_2, r_2) \leftarrow (s_3, a_3, r_3) \leftarrow \cdots$ Episode carry on till terminal state is reached

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^{k} R_{t+k} \,\middle|\, S_t = s, \pi\right]$$

Value of state s wrt a given policy π

Expected DCFR from state s following the policy π to choose future actions.

D.2 → Q-value function $[Q^{\pi}(s,a)]$

Let us assume that we have

$S = \{s_1, s_2, s_3\}$ $A = \{a_1, a_2\}$

we can have (6) state action pairs

s_1, a_1	s_2, a_1	s_3, a_1
s_1, a_2	s_2, a_2	s_3, a_2

for some(π) and state (s_2)

① $V^{\pi}(s_2)$

② $Q^{\pi}(s_2, a_1)$ / $Q^{\pi}(s_2, a_2)$

③ $\pi(a_1|s_2)$ & $\pi(a_2|s_2)$

④ $\pi^*(a_1|s_2)$ & $\pi^*(a_2|s_2)$

$Q^{\pi}(s,a)$ ⟹ How good it is to take an action (a) at state (s) & then following (π).

Basically ⟸ the expected DCFR, if at state (s), (a) action has been choosen.

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

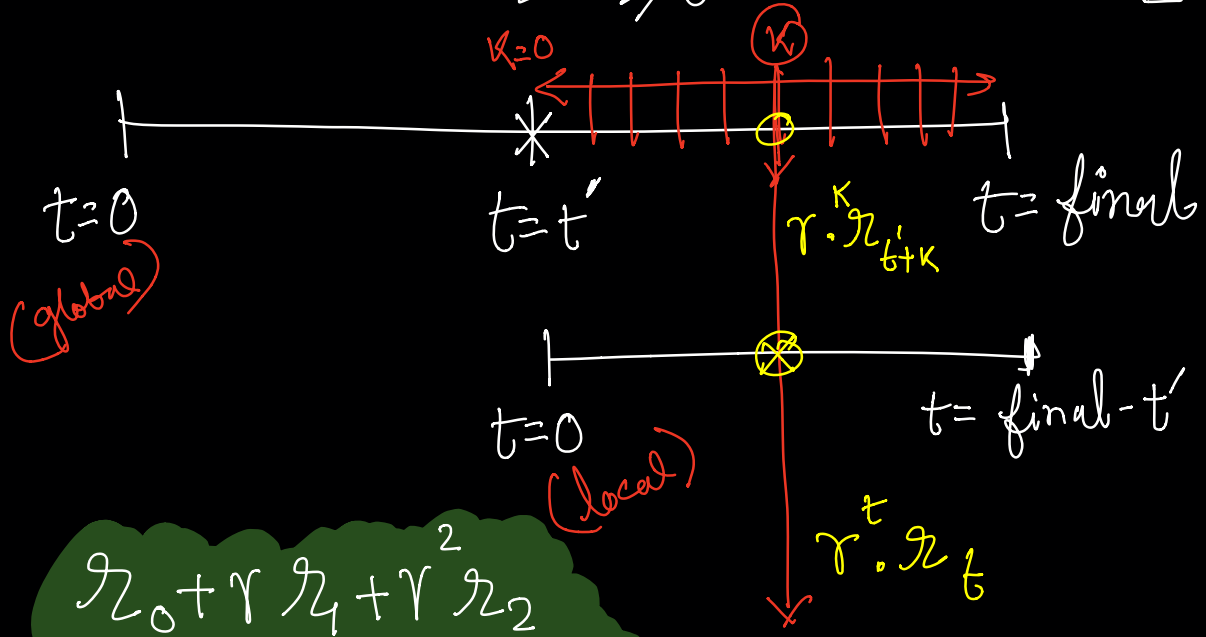
DCFR

Used for policy optimization

at state (s) agent took action (a) then follow (π)

Equivalently

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \;\middle|\; \begin{matrix} s_0 = s, \\ a_0 = a, \\ \pi \end{matrix}\right]$$



$$r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + - - - - -$$

D.3 → Optimal value function and Q-value fn.

(*) Optimal state value fn $V^*(s)$

↳ maximum value function over all the policies

$$V^*(s) = \max_{\pi} V_\pi(s)$$

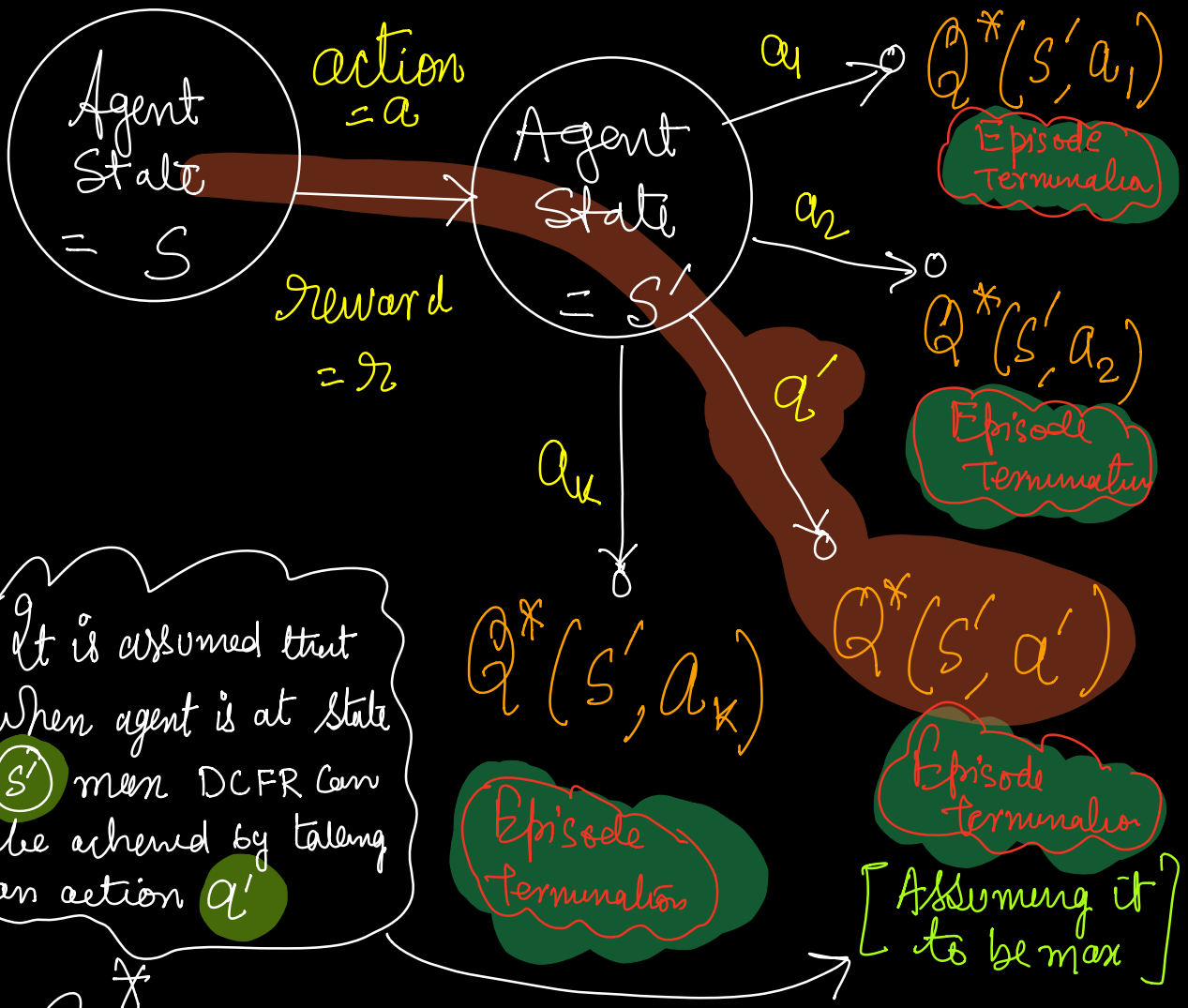
(*) Optimal action-value of Q-value fn $Q^*(s,a)$

$$Q^*(s,a) = \max_{\pi} Q^\pi(s,a)$$

Maximizing over all policies

$$= \max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \,\middle|\, \begin{matrix} s_0 = s \\ a_0 = a \\ \pi \end{matrix}\right]$$

Maximum DCFR one can achieve from any given (state, action) pair.



$$Q^*(s,a) = r + \gamma * \max(Q^*(s',a_1), Q^*(s',a_2), \dots\dots Q^*(s',a_k))$$

Best Q-value over all actions $\Longleftarrow$

$$= r + \gamma * Q^*(s',a')$$

Since at state (s) after taken an action (a) agent will land into state (s') with some probability (p).

$\Rightarrow$ $s, a, s_1 = 0.3$; $s, a, s_2 = 0.4$; $s, a, s_3 = 0.3$

$\therefore Q^*(s,a) = \mathbb{E}_{s' \in S} \left[R + \gamma * \max_{a'} Q^*(s', a') \mid s, a \right]$

it may reach to a set of states ($\in$)

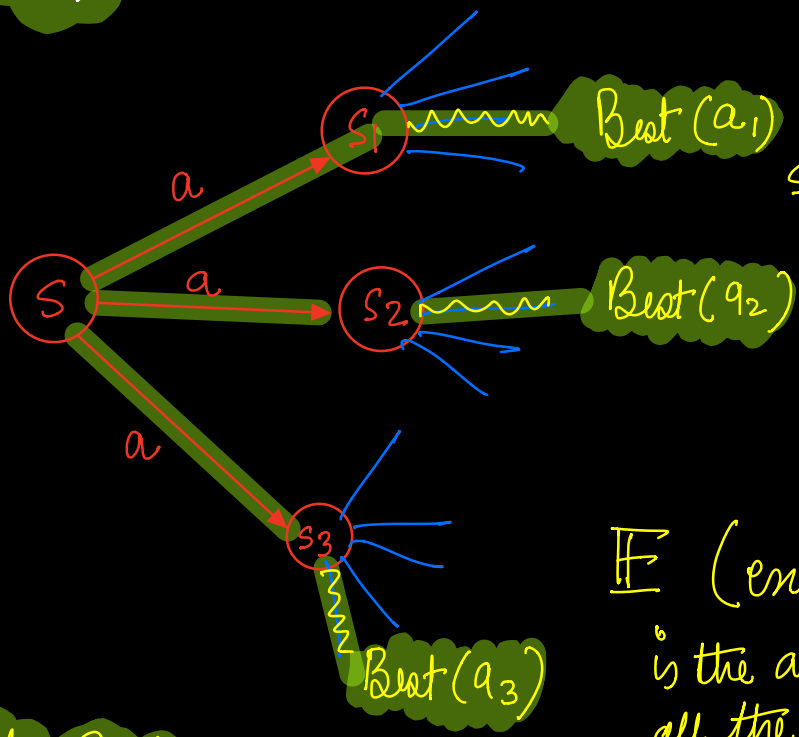
Q^* as well as V^* both functions satisfies Bellman equation and are defined recursively.

(*) Expected values are considered in order to address the randomness in the process.

Just see it once more as we are going to use it for Deep Q Learning (Q-fn = Quality of $\langle s, a \rangle$ wrt π)

$$Q^*(s,a) = \mathbb{E}\left[r + \gamma * \max_{a'} Q^*(s', a') \mid s, a \right]$$

$s' \in$ it may reach to a set of states ($\in$)



$s' \Rightarrow (s_1, s_2, s_3)$

$\in = \{s_1, s_2, s_3\}$

$a' =$ ← as shown

$\mathbb{E}$ (expected value) is the average of all the (3)

basically whatever we observe after taking a decision (a) at state (s)

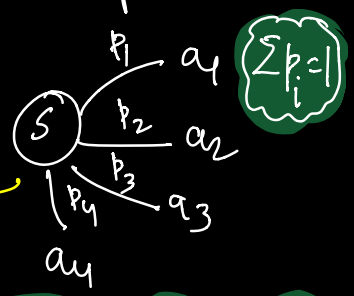
D.4 → Optimal Policy (π^*)

Policy: Def:

(*) MDP policies depends only over Current state. (Not on History)

(*) Hence policies are stationary (time-independent).

$$\pi(a|s) = \mathbb{P}[A_t = a \mid S_t = s]$$



Stochastic policies ensures suitable state, action space exploration

Here p_i's we are deciding under policy assumption to be given. Later we need to compute them optimally using DP ⇒ Not scalable hence approximate it using policy Network.

(*) Policy governs the dynamics of agent.

(*) Best/optimal policy need to ensure maximum future reward.

(*) In Markov Random Processes, we don't care about the historical reward accumulated so far. We only Consider DCFR (Discounted Cumulative future reward.)

This is how we behave optimally

(E) How to solve for optimal policy (using DP)

(*) firstly we need to figure out how to evaluate a policy (Policy Evaluation) This will tell us how good is that policy.

(*) Later using Bellman equation policy iteration or value iterations can be done inorder to update the current policy. (Iterative update)

(*) Basically assuming policy or value function at (i^{th}) step are optimal and will be used to compute the policy/value for $(i+1^{th})$ step. [Using Dynamic Programming]

D.P (Review)

Bellman Ford Shortest Path Algorithm

(*) -ve edges are allowed.

-ve cycles are NOT allowed.

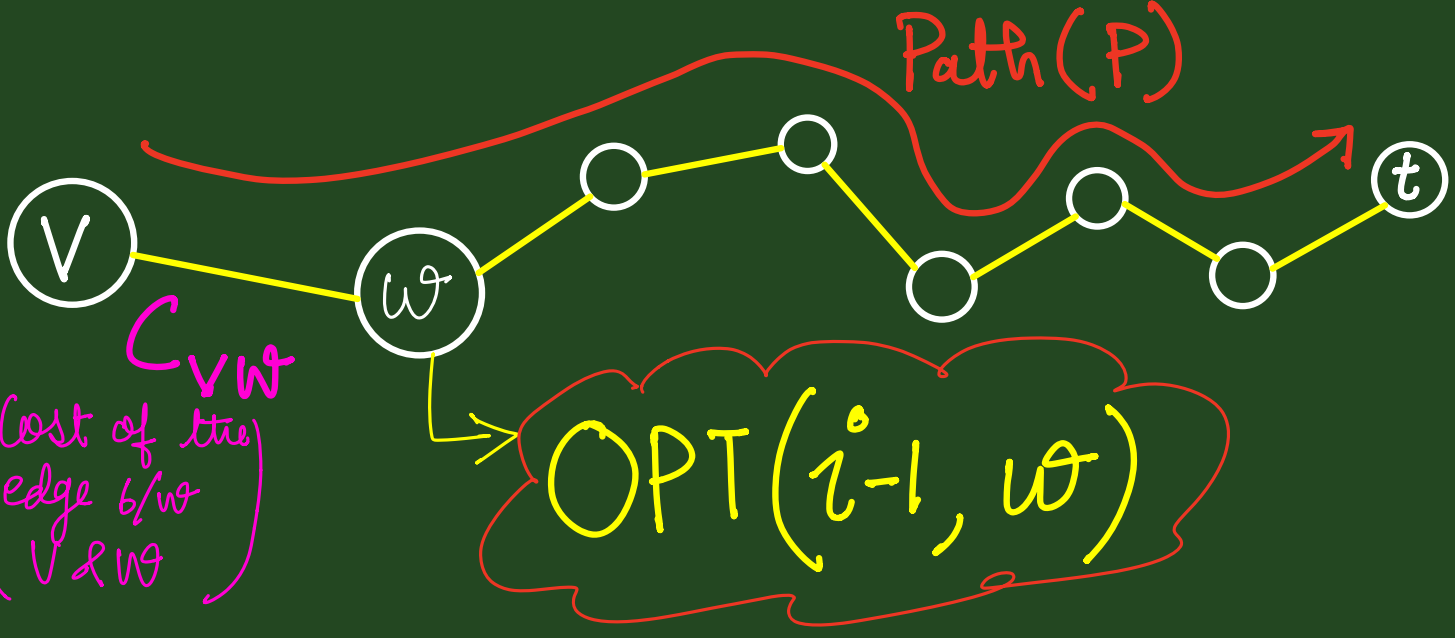
(*) Observation : for a given graph $G=(V,E)$

If $G=(V,E)$ has NO -ve cycle, then there exist a shortest path from say node (s) to (t) that is

(a) Simple [No repetion of nodes]

(b) with atmost $|V|-1 = (n-1)$ edges.

⊛ Setting up the recurrance :



⊛ OPT(i, v) ⇒ Represent the minimum Cost of say path(P) from (v) to (t) using atmost (i) edges.

Since -ve edges are present allowing more edges may reduce the cost.

$OPT(5, v) \geq OPT(6, v)$

Hence under such setting our originally defined problems solution will be :

$OPT(n-1, s)$ ⇒ Minimum Cost of the path (P) that can take you from (s) to (t), Using atmost (n-1) edges.

How to derive a recurrance relation in terms of smaller subproblems.

$OPT(i, v)$

Let us assume an optimal path (P) from v to t using atmost $(n-1)$ edges.

OR ignoring this new edge.

Either utilizing this new edge

(P) either uses atmost $(i-1)$ edges

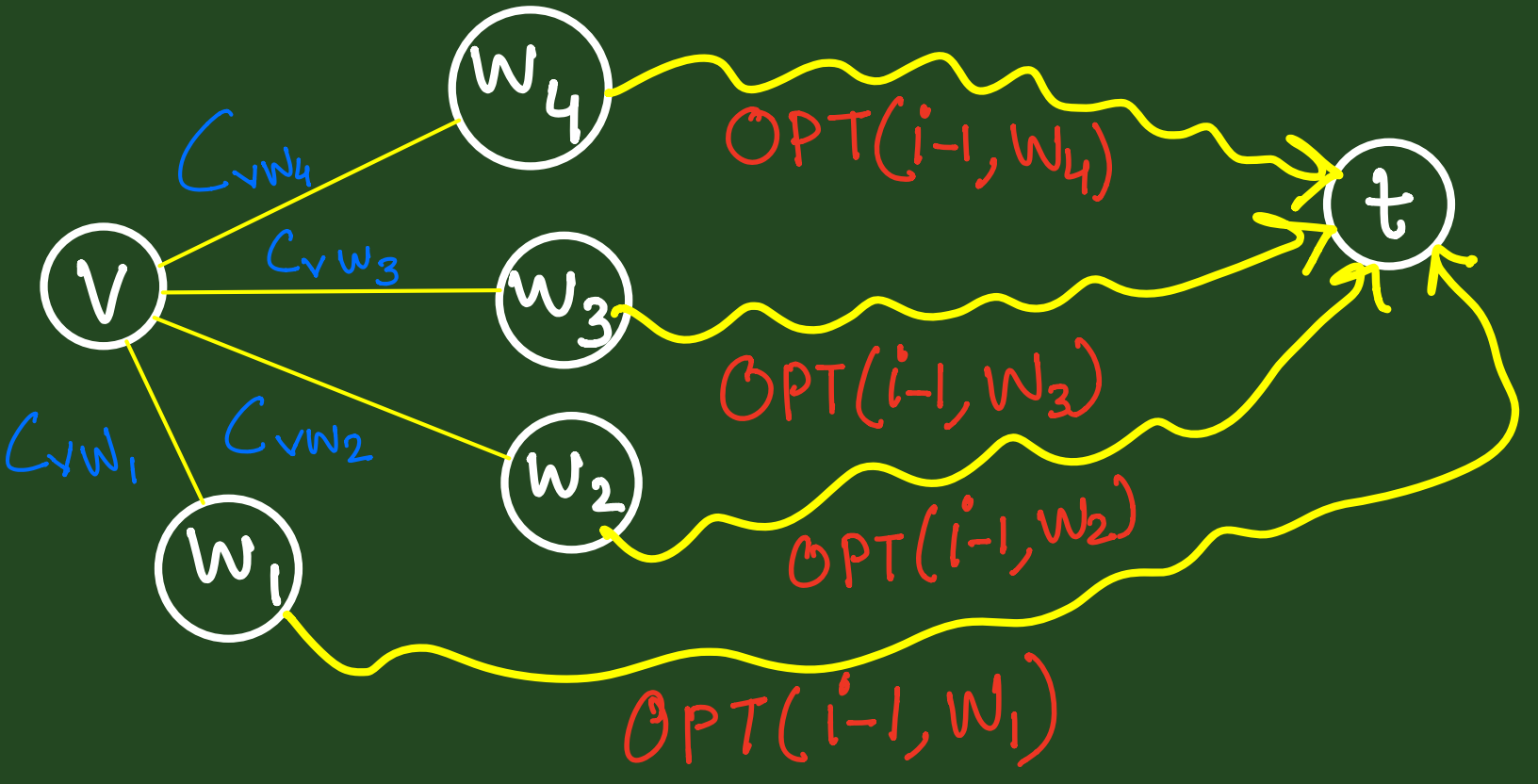
$\Downarrow$

$OPT(i, v) = OPT(i-1, v)$

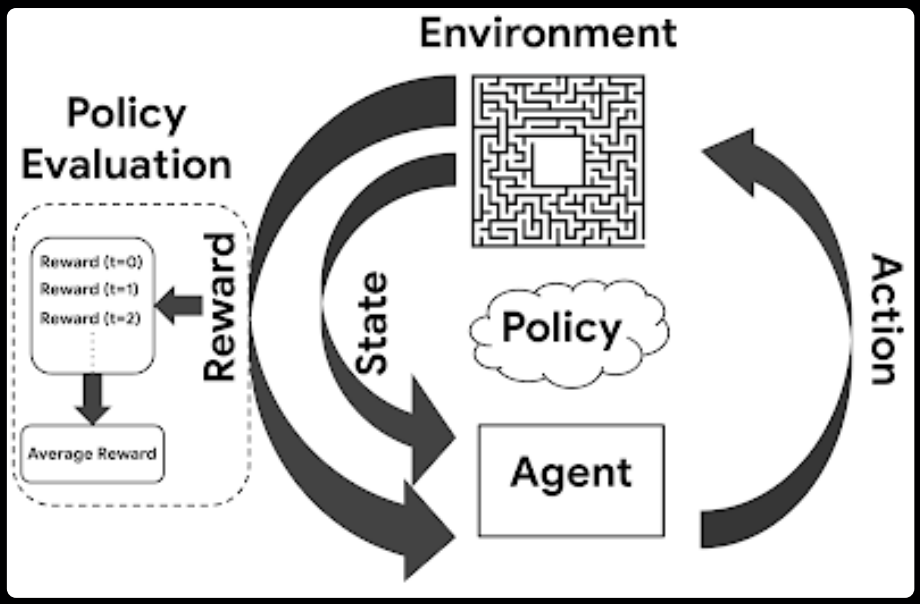
OR (P) uses atmost (i) edges

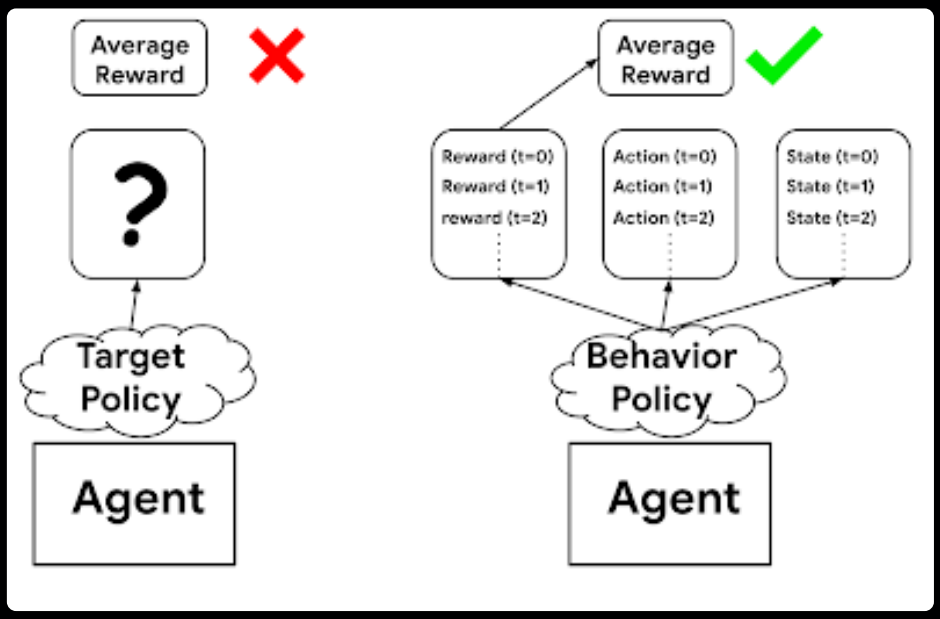
$\Downarrow$

$OPT(i, v) = OPT(i-1, w) + C_{vw}$

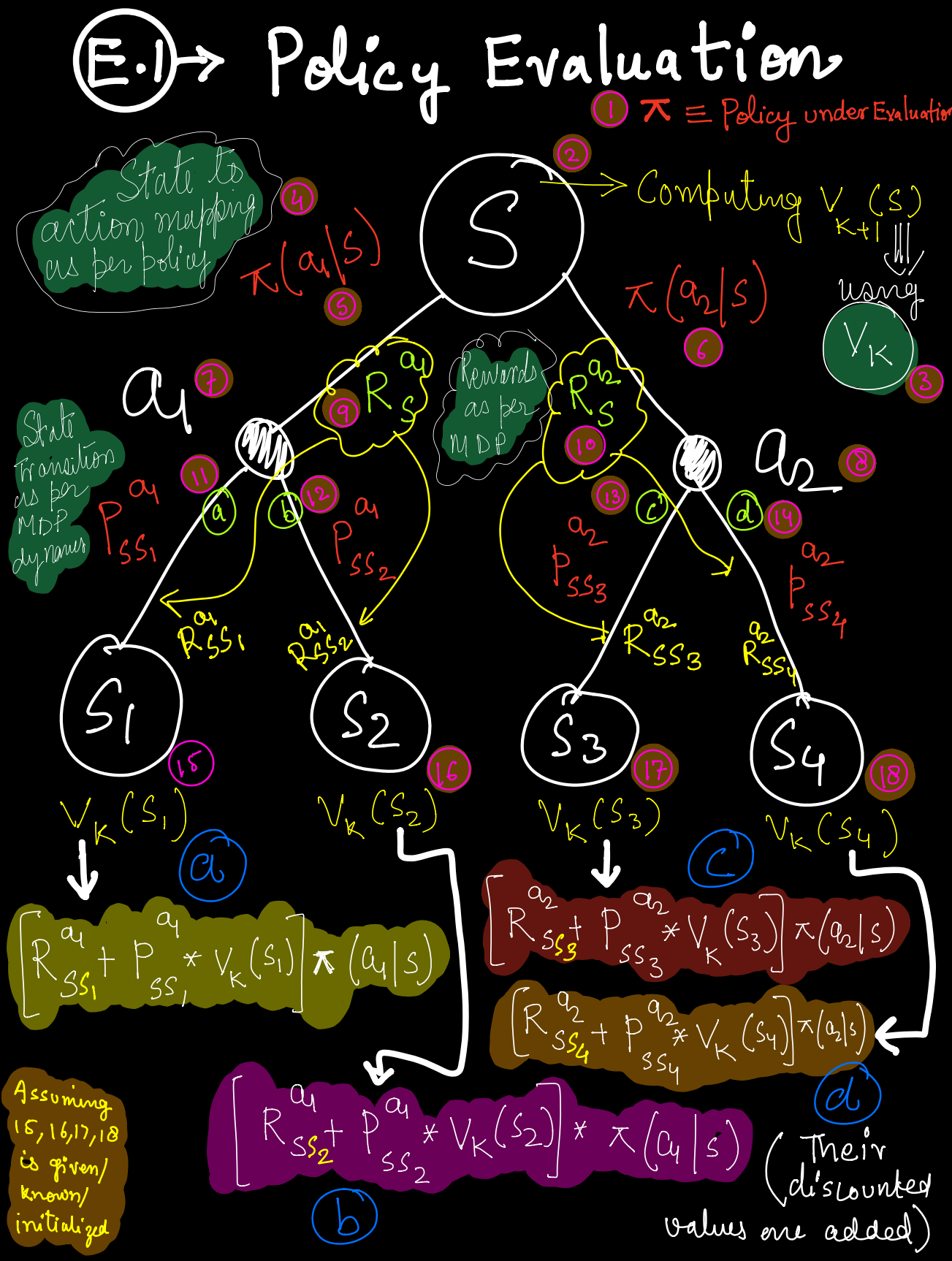


$$OPT(i, v) = Min\left[OPT(i-1, v) \;,\; \underset{w \in V}{Min}\left[OPT(i-1, w) + c_{vw} \right] \right]$$


Environment
Policy Evaluation
Reward (t=0)
Reward (t=1)
Reward (t=2)
Average Reward
Reward
State
Policy
Agent
Action


Average Reward
?
Target Policy
Agent
Average Reward
Reward (t=0)
Reward (t=1)
reward (t=2)
Action (t=0)
Action (t=1)
Action (t=2)
State (t=0)
State (t=1)
State (t=2)
Behavior Policy
Agent

E.1 → Policy Evaluation



$$V_{k+1}(s) = a + b + c + d$$

(Expected DCFR of (s) following and using π and V_k's

$$= \sum_{a \in A} \pi(a|s) * \left[R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a * V_k(s') \right]$$

This probability is doing the expectation.

In closed form:

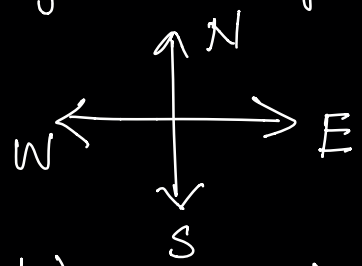
$$V_{k+1} = \vec{R}^{\pi} + \gamma \vec{P}^{\pi} V_k$$

- V_{k+1}: Value vector for all states at $(k+1)^{th}$ iterations
- $\vec{R}^{\pi}$: Reward function as per the MDP
- $\vec{P}^{\pi}$: State Transition Probability
- V_k: Value fn at k^{th} iteration

Let us evaluate a random policy in small grid world.

E.2 Policy Evalution : Example

Example: Policy evaluation for small grid world having four actions:
moving in N | S | E | W
with equal probability: $\pi(N|\cdot) = \pi(E|\cdot) = \pi(S|\cdot) = \pi(W|\cdot) = 0.25$



Uniformly random policy.

(2) Terminal states.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

(*) Reward $r = -1$ for any transitions to ensure step minimization

(*) Undiscounted episodic MDP ($\gamma = 1$)

(*) How long does it take to reach to some terminal state from any Grid point. [Trajectory length] as per some policy π.

(*) Compute the optimal policy π: How to reach as soon as possible.

(*) At terminal state episode is finished and looping with zero reward.

(*) Non-terminal state ($r = -1$) loop.

V_K for Random Policy (K=0)

V_0

	1 0.0	2 0.0	3 0.0
4 0.0	5 0.0	6 0.0	7 0.0
8 0.0	9 0.0	10 0.0	11 0.0
12 0.0	13 0.0	14 0.0	

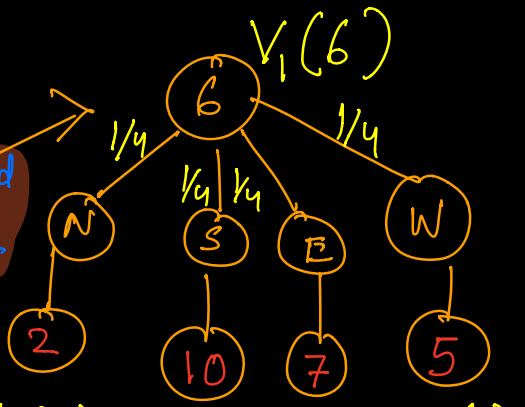
$\Rightarrow$ $V_0(i)$ denoting the no. of steps required to move from state/grid (i) to a Terminal state (T) & $V_K(T) = 0$ for all (K's)

For random policy V_0 donot Know anything. Later after exploration (V_1) is Computed using (V_0) which slowly Converge to optimality.

K=1 (V_1)

	1 -1.0	2 -1.0	3 -1.0
4 -1.0	5 -1.0	6 -1.0	7 -1.0
8 -1.0	9 -1.0	10 -1.0	11 -1.0
12 -1.0	13 -1.0	14 -1.0	

Reward = -1
r = 1



$$V_1(6) = [-1 + V_0(2)]\,1/4 + [-1 + V_0(10)]\,1/4 + [-1 + V_0(7)]\,1/4 + [-1 + V_0(5)]\,1/4 = [-1 + 0]\,\tfrac{1}{4} * 4 = -1$$

(0) (0) (0) (0)

K=2 (V_2)

	1 -1.7	2 -2.0	3 -2.0
4 -1.7	5 -2.0	6 -2.0	7 -2.0
8 -2.0	9 -2.0	10 -2.0	11 -1.7
12 -2.0	13 -2.0	14 -1.7	

looping back into the same state.

why values are not reaching to the correct values.

$$V_2(6) = [-1 + V_1(2) \; -1 + V_1(10) \; -1 + V_1(7) \; -1 + V_1(5)] * \frac{1}{4} = (-8) \times \frac{1}{4} = -2$$

$$V_2(14) = \frac{1}{4} * [-1 + V_1(13) \; -1 + V_1(10) \; -1 + V_1(7) \; -1 + V_1(14)]$$

$$= \frac{1}{4} * 7 = 1.75$$

K=3 (V_3)

	1 -2.4	2 -2.9	3 -3.0
4 -2.4	5 -2.9	6 -3.0	7 -2.9
8 -2.9	9 -3.0	10 -2.9	11 -2.4
12 -3.0	13 -2.9	14 -2.4	

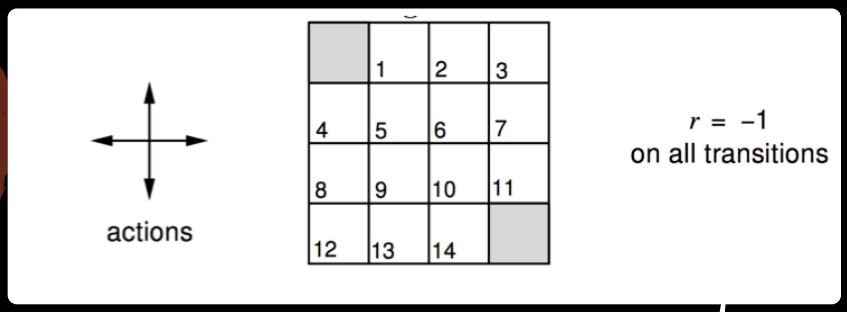
$$V_3(2) = \frac{1}{4}[-1 + V_2(2) \; -1 + V_2(1) \; -1 + V_2(3) \; -1 + V_2(6)]$$

$$V_3(2) = \frac{1}{4}[-11.7] = -2.925$$

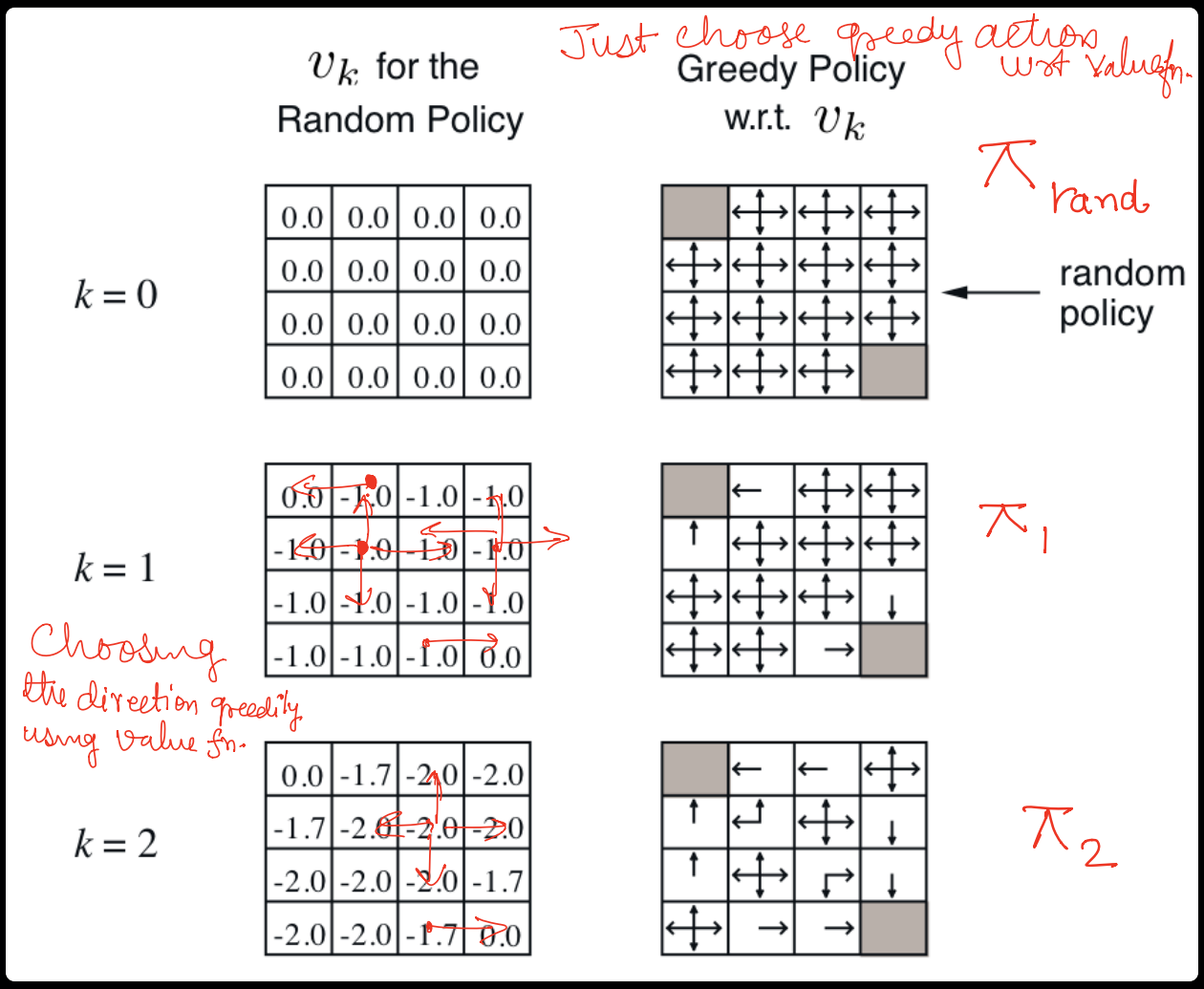
$$V_3(14) = \frac{1}{4}[-1 + V_2(14) \; -1 + V_2(13) \; -1 + V_2(7) \; -1 + V_2(10)]$$

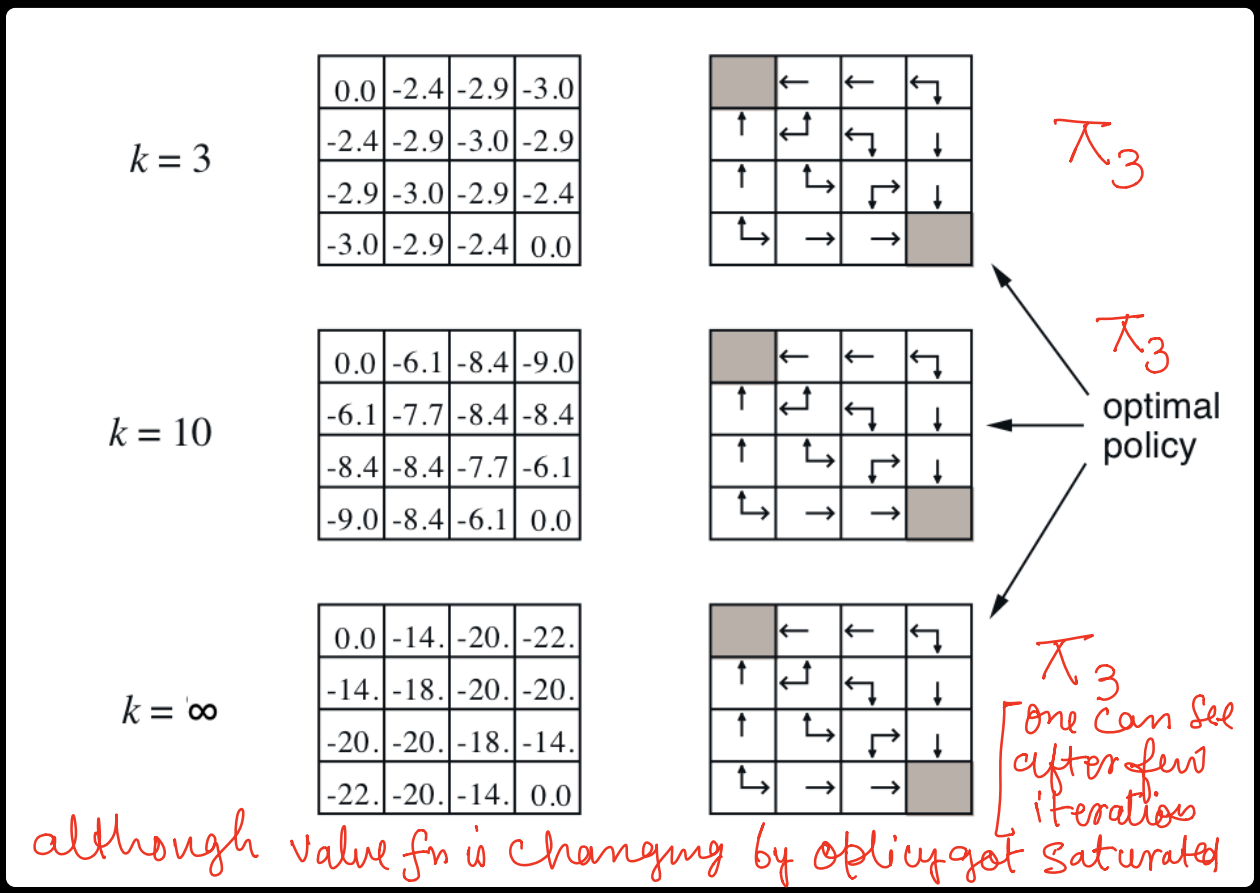
$$V_3(14) = \frac{1}{4}[-9.7] = -2.425$$

E.3 Policy Improvement



For this MDP, we have seen how to evaluate any policy (say π = Random). Given any (V_k) one take action greedily inorder to get refined policy (say π').

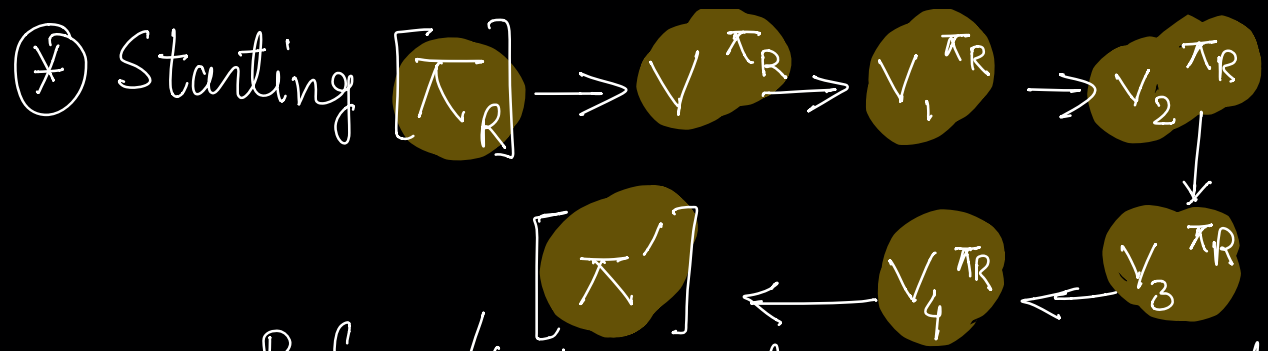




⊛ Hence we understand how to evaluate a policy. [Policy (π) ⇒ Value fn V^{π}]

⊛ Later given a value fn act greedily as per it and get a refined policy

[$V^{\pi} \longrightarrow \pi' = \text{greedy}(V_{\pi})$ (ref.)]



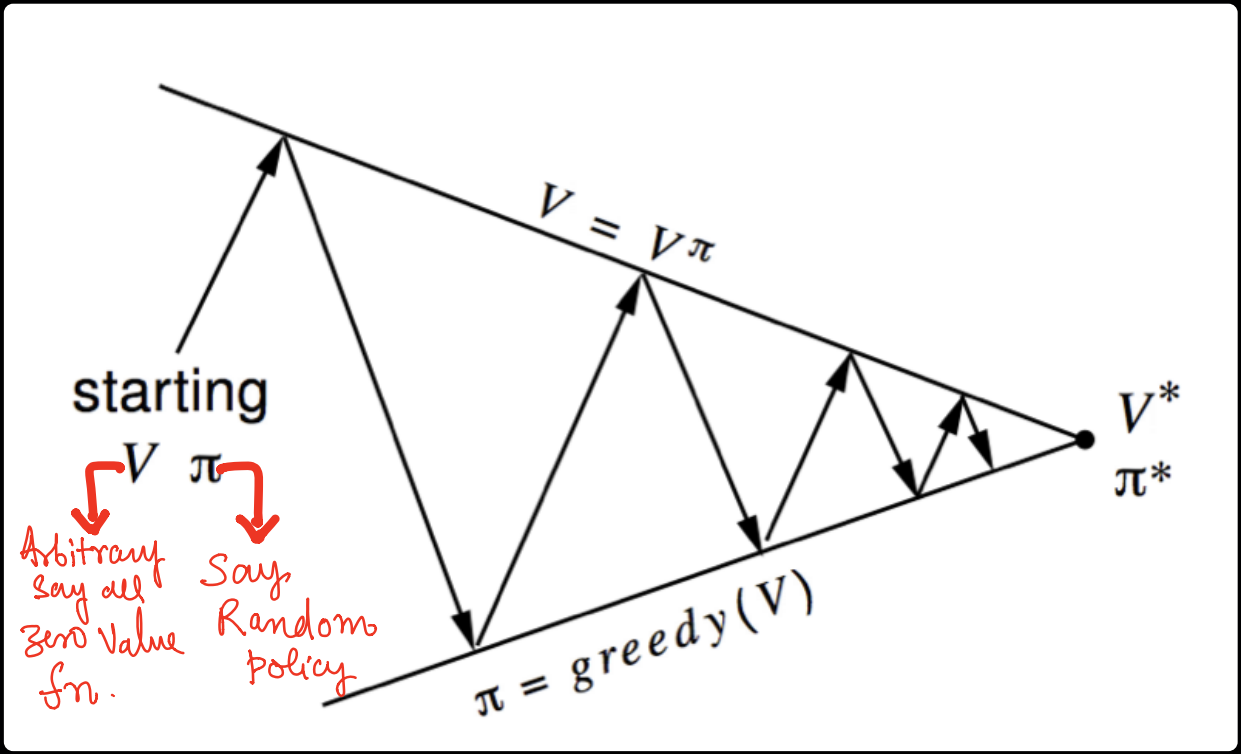
Refined/improved policy & keep on repeating.

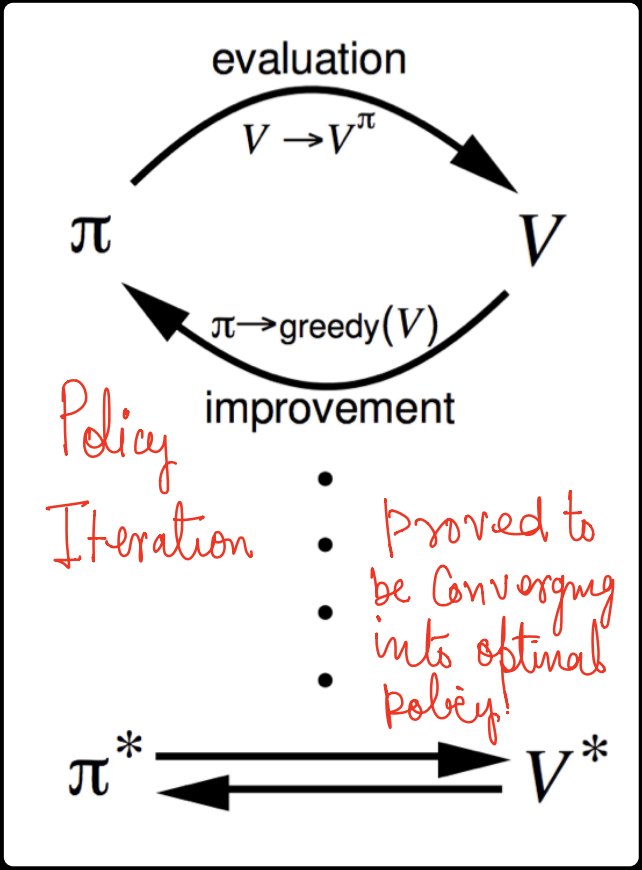
looping

1. Given any policy (π)
2. Evaluate the policy (π) by computing value fn.
3. Keep on iterating over value fn so that its corresponding Greedy policy(π) got saturated.
4. Improve (π) $\xrightarrow{to}$ (π')

Policy evaluation Estimate v_π
Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement





Policy Iteration

- proved to be converging into optimal policy!

This has been proved that this process will always converge into the optimal policy π^*.

(*) No matter from where one is starting.
(Initialization just effect the rate of convergence).

(F) Solving for Optimal policy (Deep Q-learning)

Basically learning the Q fn

Coming from above

(Already discussed)

$$Q^*(s,a) = \mathbb{E}_{s' \in \epsilon}\left[r + \gamma \max_{a'} Q^*(s',a') \mid s,a \right]$$

$s' \in \epsilon$ → it may reach to a set of states (ϵ)

* Q_i will Converge to Q^* when $(i \to \infty)$.

* Basically at each iteration optimality has been enforced so as to conform to the Bellman eqn

Q^* as well as V^* both functions satisfies Bellman equation and are defined recursively.

(F.1) Problems with DP Solution

* As shown above Bellman equation can be used to solve for the optimal Q-value fn. (Evaluate & Improve).

* This optimal solution is proved to be Converging and easy to compute.

* But is simply not scalable. for any typical modern day problem Say Atari gaming agent (S) = State (Raw pixels of a frame) (a) Can limited but (S) = Can be exponentially many. Hence one need to Compute Qvalue for full $\langle s, a \rangle$ i.e $\langle$State, action$\rangle$ space $\Rightarrow$ Simply Computationally in feasible.

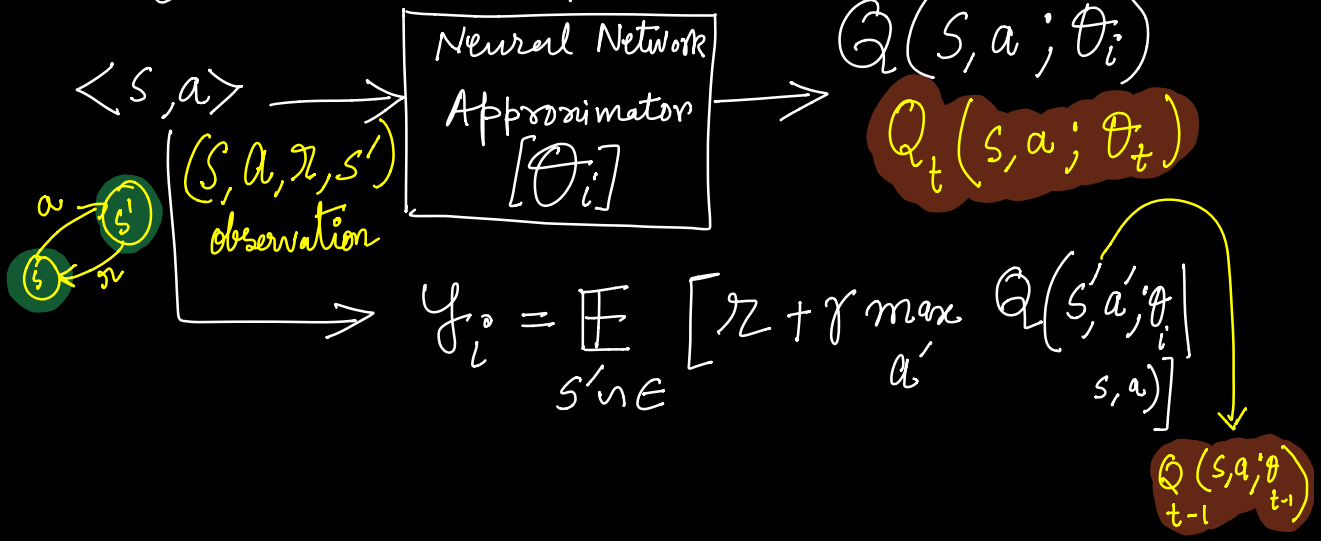
F.2 Deep Q-learning for Qfn approximation

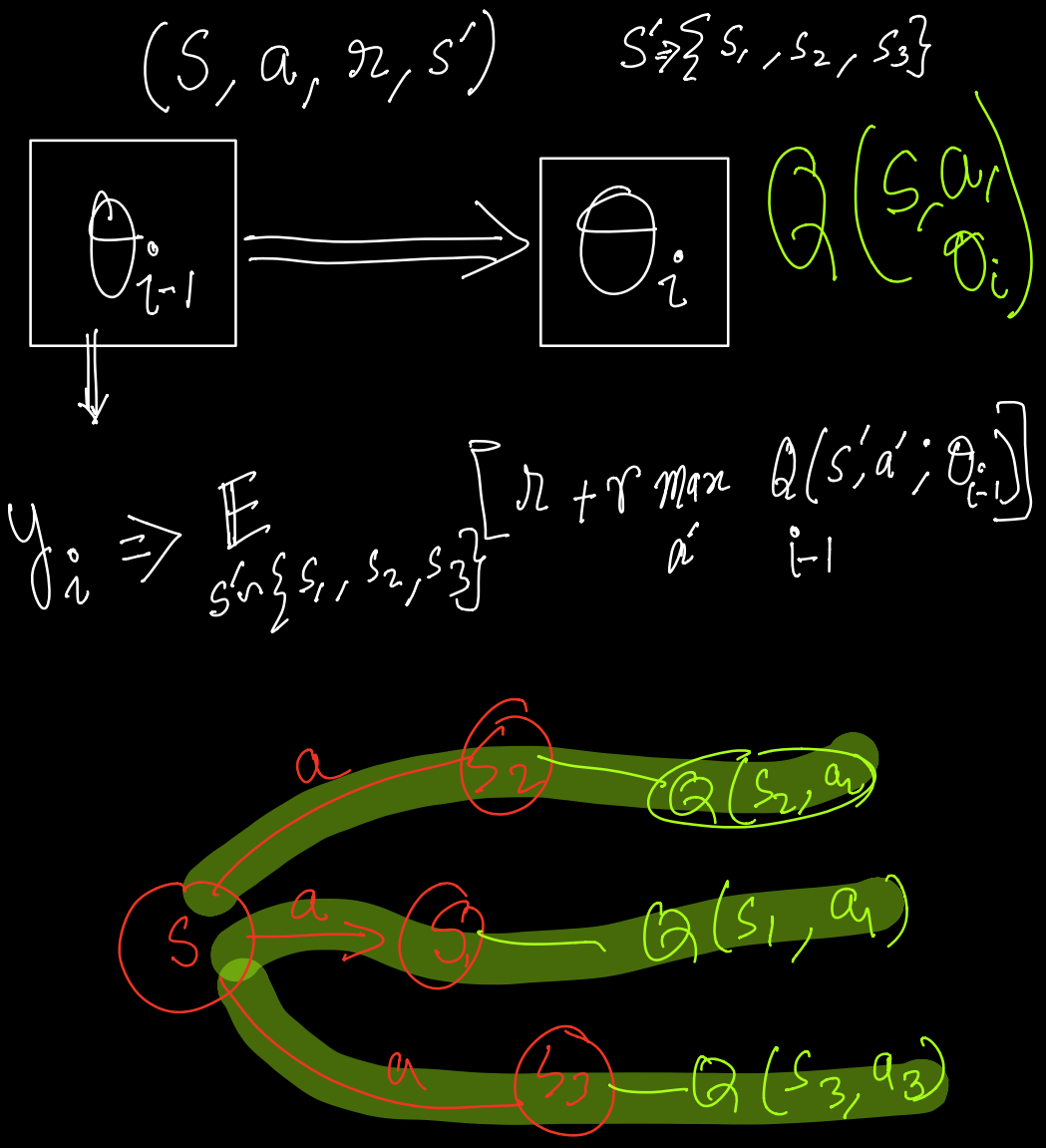
* Instead of computing $Q(s, a)$, we need to use some function approximator (say neural network) to approximate/estimate $\{Q(s, a) \ \forall \ s, a\}$

$$Q(s, a; \theta) \rightsquigarrow Q^*(s, a)$$

Using DNN parameters to estimate the Q value for (s, a) pair. { Deep Q-Learning }

* We need a Qfn approximator that satisfies Bellman equation by enforcing Bellman optimality at each iterative step. (Intuition)





F.3 Forward Pass : Loss function

$$\therefore L_i(\theta_i) = \mathbb{E}_{s,a \sim P(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$$

(Forward Pass)

↳ Current network o/p. (n/w parameters at i^{th} step)

$$y_i = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma * \max_{a'} Q(s',a';\theta_{i-1} | s,a)\right]$$

↳ Previous n/w o/p (old parameters at $(i-1)^{th}$ iteration)

for addressing the scalability issue, we donot need to Compute Q fn for each & every <State, action> pair.

* Using the environment & agent machinery along with the full MDP dynamics, play few game episodes. [Play game]

* Instead of Computing the Q-fn for full <State, action> pair space, explore only few of them.

That we have experienced during our episodic Simulations

(Episodic learning)

Therefore play games ⟹ Generate Episodes

Use these episodes for training ⟸ only consider those <state, action> pairs that we observe in some state instead of full space

F.4 Game Play (given MDP dynamics OR N/W parameter(θ))

Generating training data. [Q] approximating N/W(θ)

(1) Saving the game states into (T) [Training data]

$\langle S_t \text{ (or } \phi_t), a_t, r_t, S_{t+1} (\phi_{t+1}) \rangle$

Raw State

$[\phi_t = \text{Processed}(S_t)]$

Or may be processed Raw State

Action performed

Reward

Next state.

(2) for $t = 1$ to (M) do → Generating (M) full game data : M episode training data

(2.1) ⊙ Initilize starting state $S_1 = \{x_1\}$ → May be some starting game screen.

(2.2) ⊙ Pre-processed sequence $\phi_1 = \phi(S_1)$ → Some preprocessing fuction (say edges of differences)

(2.3) for $t = 1$ to (T) do → for each time step (t) of the game episode

Select random action or Select greedy action as per the current policy

(2.3.1) Select a random action (say a_t) with some small probability (ϵ) ⇒ for sufficient state action space exploration. Small probability to explore new things.

(2.3.2) Otherwise select $a_t = \max_a Q^*(\phi(S_t), a; \theta)$

$(\phi(S_t), a)$ → [θ N/W parameter] → a_t (next action need to be taken)

Approximating Q^*

(2.3.3) Execute action (a_t) in the game emulator and observe the reward (r_t) and next image (x_{t+1}) ≈ {new next state (S_{t+1})}

(2.3.4) Store the transition $\langle \phi_t, a_t, r_t, \phi_{t+1} \rangle$ into (T) ≈ as traing data.

(3) Return the training data (T).

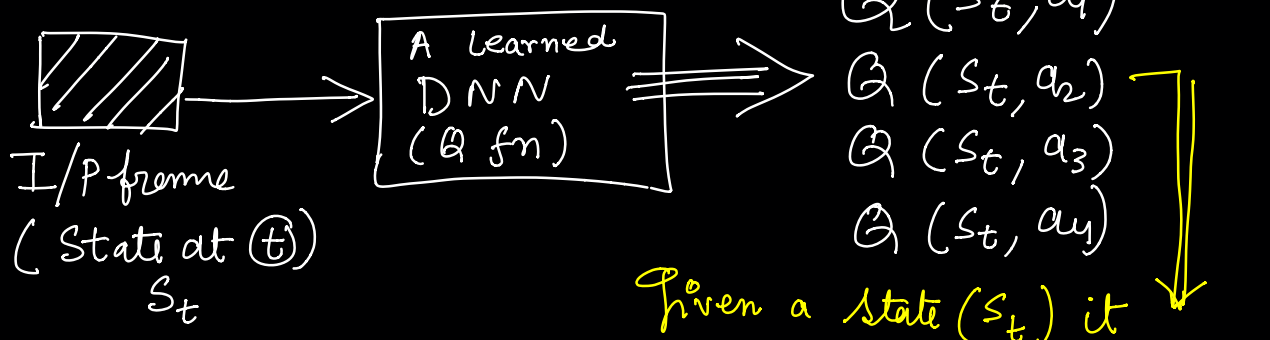
👉 This is just Gameplay algorithm in case actual Q^* function is known or an approximating N/W is given.

👉 It can be used to generated more traing data in (T) and incrementally train the n/w more and get better and better N/W parameters $(\theta^t \Longrightarrow \theta^*)$ at Convergence.

(*) What does it mean when we say an approximator deep neural network is learning / learned the required {Q^* function}. (Intution)

In most of the games (for example Attari Games) actions are always finite in number and fixed.

(Say UP | DOWN | LEFT | RIGHT)
a_1 a_2 a_3 a_4



Given a state (S_t) it need to o/p these Q-distribution.

(*) Attari games have around (4-18) actions

Hence $\forall_{a_i} Q(S_t, a_i) \Rightarrow$ one scalar value per action and per state.

[Efficiently Computing full $Q(S_t)$ distribution for all action in one single forward pass]

F.5 Training issues and Experience Replay

Problem: Learning from consecutive samples coming from same episode (T) is problematic.

↳ Samples will be highly correlated hence easily learn some trivial dynamics but fails to generalize.

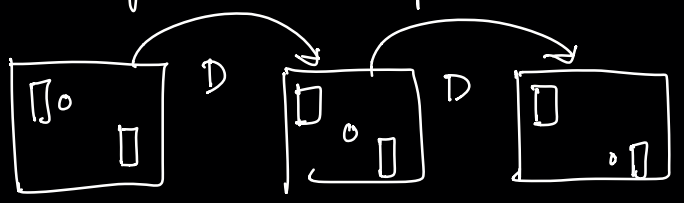
↳ May become biased towards the current o/p.

↳ If the current action maximizing Q-value is (left) ⟹ Training samples are dominated by left hand side samples (Down Side)

Down

Due to this ↙

the current Q-Network parameter (θ_{t-1}) may lead to bad bias towards (DOWN) & may even keep on oscillating.



⟹ Correlated frames & not a very meaningful sampling

Experience Replay :

Better idea is to make random mini batches out of Training data (T) ≈ Experience & train n/w over that ≈ Replay

Replay Memory (RM)

OR

Training data (T)

≈≈≈⟹ $\langle s_t, a_t, r_t, s_{t+1} \rangle$

F.6 Algorithm : Deep Q-learning with Experience Replay.

(1) Initilize the Deep Q-learning N/w parameters (θ_0) randomly. (N/w learning the Q-function)

(2) T = Gameplay (θ_0)

(3) Till Convergence (Working for some θ_i)

(3.1) Sample a random minibatch of transtion $\langle \phi_j, a_j, r_j, \phi_{j+1} \rangle$ from (T).

Sampling some (j) transitions

(3.2) Set $y_j = \begin{cases} r_j & \text{for terminal } (\phi_{j+1}) \\ r_j + \gamma * \max_{a'} Q[\phi_{j+1}, a' | \theta] & \text{for non-terminal } (\phi_{j+1}) \end{cases}$

for each (j'th) transition

(3.3) $L_j(\theta_j) = (y_j - Q(\phi_j, a_j | \theta))^2$

(3.4) $L^{batch} = L + L_j(\theta_j)$

New T = GamePlay(θ_i)

Forward Pass :

$$L_i(\theta_i) = \mathop{\mathbb{E}}_{s,a \sim P(\cdot)} \left[(y_i - Q(s, a | \theta_i))^2 \right]$$

$$\Downarrow$$

$$y_i = \mathop{\mathbb{E}}_{s' \sim \mathcal{E}} \left[r + \gamma * \max_{a'} Q(s', a' | \theta_{i-1}) \,\middle|\, s, a \right]$$

Backward Pass : Differentiating loss wrt N/w parameters (Gradient update)

$$\nabla_{\theta_i} L_i(\theta_i) = -2 * \mathop{\mathbb{E}}_{s,a \sim P(\cdot)} \left[y_i - Q(s, a | \theta_i) \right] * \nabla_{\theta_i} \left[Q(s, a | \theta_i) \right]$$

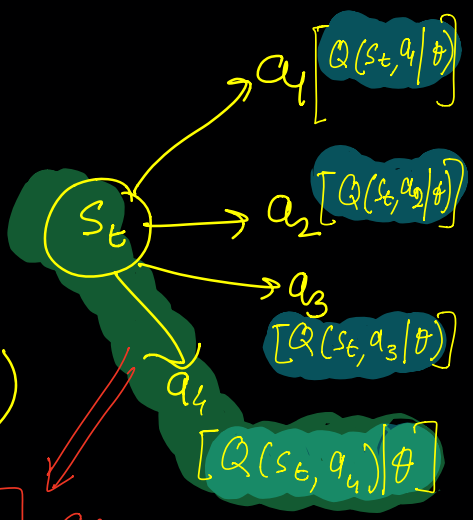
Bellman's estimate

Q-fn estimate using Current N/w.

(*) Hence, we have seen that Deep Q-Learning Network tries to estimate the optimal Q-function starting from some random estimate and keep on refining it iteratively by imposing Bellman optimality. That is ensured by the loss function.

(*) Given any MDP its corresponding optimal Q fn is its solution. Given Q^* fn for any MDP, one can plan to define the policy by moving greedily using $[Q^*]$.

MDP $\xrightarrow[\text{Learning it using Deep Q-N/W}]{\cancel{Q(S,a|\theta)}}$ Q^* fn



$$a_t = \max_{\forall i} Q(S_t, a_i|\theta)$$

$[a_t = a_4]$ assuming this is maximizing.

Hence, Deep Q-learning is indirectly learning the policy.

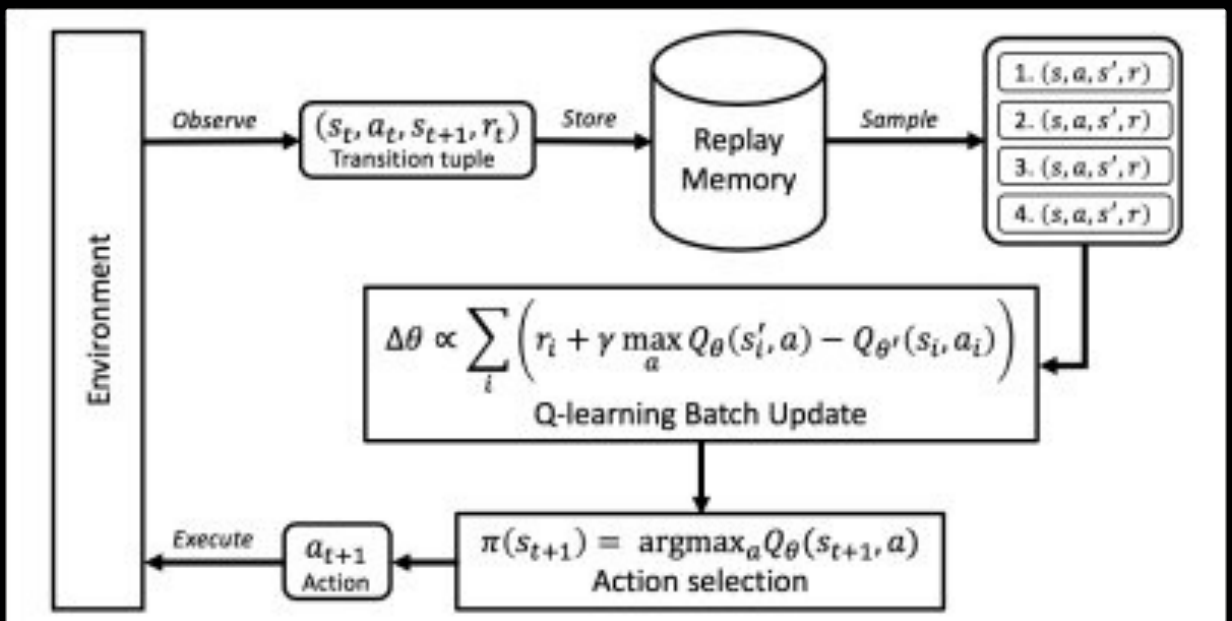
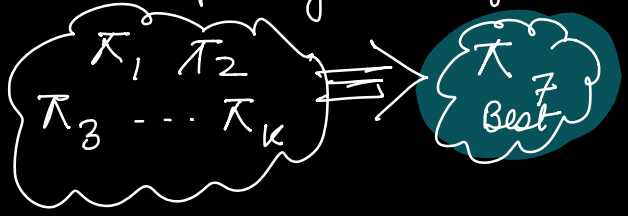


Figure 1. Lifelong Reinforcement Learning using Replay memory to learn tasks.

(*) Deep Q-learning N/w learns the optimal Q-function and using it greedily provides us the suitable policy $\Longrightarrow$ Hence it gives us an implicit handle over the policy (π).

(*) Can we find out the best policy out of the collection of policies



How can we learn the policy explicitly w/o learning an Q-value estimate

G Solving for Optimal Policy (Directly)

G.1 Introduction to Policy Gradient

(*) Instead of using DNN to approximate Q-Value function, why can't directly learn the suitable policy (π), parameterized by θ.

Class of parameterized policies : $\pi = \{\pi_\theta \mid \theta \in \mathbb{R}^m\}$

for any policy (π), define it value as : $J(\theta) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \mid \pi_\theta\right]$

- $J(\theta)$: Value of the policy (π) Parameterized by (θ)
- $\mathbb{E}$: Expected
- $\sum_{t \geq 0} \gamma^t r_t$: DCFR
- π_θ : while following policy (π_θ)

(*) Given any policy (π_θ), how much DCFR on an average one can extract $\Rightarrow$ value of π_θ i.e. $J(\theta)$

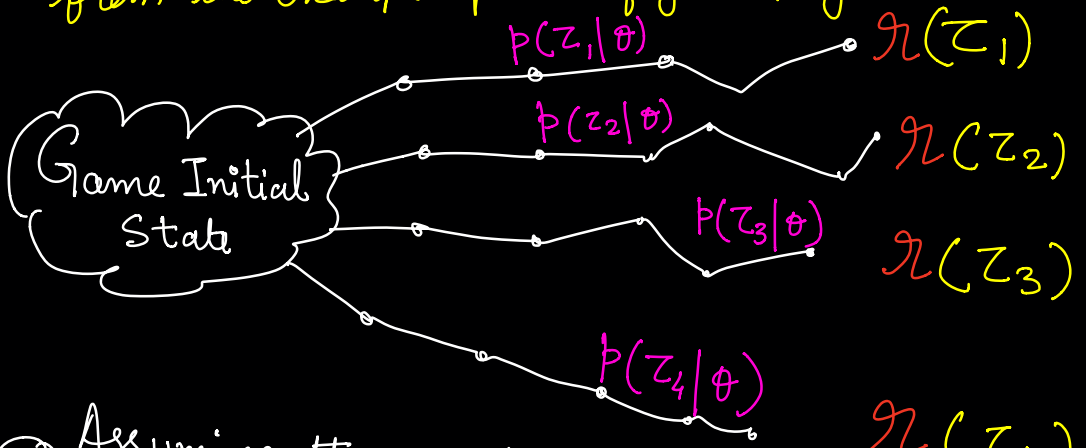
GOAL : DNN need to optimize for the parameter θ that can realize the optimal policy π^*.

$\theta^* = \arg\max_\theta J(\theta)$ Gradient Ascent over θ $\Rightarrow$ REINFORCE also.

G.2 REINFORCE algorithm Loss function

(*) Trajectory (τ): $\tau = (\langle s_0, a_0, r_0 \rangle, \langle s_1, a_1, r_1 \rangle, \langle s_2, a_2, r_2 \rangle, - - - - - -)$

Sample trajectory from the example episode of game play. (Sampled randomly).



(*) Assuming the agent to be starting from some initial game state and following (π_θ) policy parameter over (θ).

→ (a) Assuming all trajectories to be equally probable: $J(\theta) = \frac{1}{4} \sum_{i=1}^{4} r(\tau_i)$

→ (b) Assuming trajectories follow $p(\tau|\theta)$ distribution

$$J(\theta) = \sum_{i=1}^{4} p(\tau_i|\theta) * r(\tau_i)$$

Value associated to a policy (π_θ), parameterized over $[\theta]$. $= \underset{\forall i}{\mathbb{E}}\left[r(\tau_i)\right]$

Expected Reward $= \mathbb{E}_{\tau \sim p(\tau|\theta)}[r(\tau)]$

$$= \int_{\tau} (r(\tau) * p(\tau|\theta))\, d\tau$$

Considering all trajectories that one can encounter while agent start from some initial state & following (π_θ) policy.

$$\therefore \quad \theta^* = \arg\max_{\theta} [J(\theta)]$$

(*) REINFORCE algorithm uses $J(\theta)$ for its forward pass as a Gain fn [Need to be maximized]

G.3 REINFORCE algo Backward pass

(*) Forward Pass : $J(\theta) = \int_{\tau} r(\tau) * p(\tau|\theta)\, d\tau$ — (1)

(*) Backward pass: $\nabla_\theta J(\theta) = \int_{\tau} r(\tau) * \nabla_\theta p(\tau|\theta)\, d\tau$ — (2)

*** This is intractable. Gradient Computation of an expectation is problematic when p depends upon θ. [Gradient of expectation]

Trick: $\nabla_\theta p(\tau|\theta) = p(\tau|\theta) * \left[\frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)}\right]$ — (3)

Putting eq (3) into eq (2) $= p(\tau|\theta) * \nabla_\theta \log\left[p(\tau|\theta)\right]$

$\therefore \nabla_\theta J(\theta) = \int_{\tau} r(\tau) * \nabla_\theta \log\left[p(\tau|\theta)\right] * p(\tau|\theta)\, d\tau$

$\therefore \nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau|\theta)}\left[r(\tau) * \nabla_\theta \log\left[p(\tau|\theta)\right]\right]$

Expectations of Gradient

G.4 Issue with Gradient & its Computation

(*) There can be infinitly many trajectories possible for any parameterized policy (π_θ) estimated by DNN.

(*) The reward function [$J(\theta)$] has been defined as an Expectation over all such trajectories. Hence $J(\theta)$ Can be computed essentially by Monte-Carlo Sampling of trajectories.

(*) All trajectories ($\tau_1, \tau_2 \dots \tau_\infty$) for some given ($\pi_\theta$) are not equiprobable, instead following a distribution. Say $p(\tau|\theta)$

(*) Hence, for a some trajectory say [$\tau_1 = \langle (s_0, a_0, r_0), (s_1, a_1, r_1), (s_2, a_2, r_2) \rangle$]

Let us compute the probability of (τ_1) given $p(\tau|\theta)$

$$p(\tau_1|\theta) = [T(s_1|s_0, a_0) * \pi_\theta(a_0|s_0)] * [T(s_2|s_1, a_1) * \pi_\theta(a_1|s_1)] * [T(s_2|s_2, a_2) * \pi_\theta(a_2|s_2)]$$

Given the policy (π_θ) parameterized by (θ) & learned by some Deep policy N/w maximizing the reward fn [$J(\theta)$].

Transition Probabilities

As per Parameterized π_θ Policy.

(*) $$P(\tau|\theta) = \prod_{t \geq 0} T(s_{t+1}|s_t, a_t) * \pi_\theta(a_t|s_t)$$ — Eq-A

Estimating the probability of randomly selected trajectory (τ), while following policy(π_θ)

The transition probability of going to (s_{t+1}) state after taken an action (a_t) at state (s_t)

Probability of taking an action (a_t) at state (s_t) under the policy (π_θ)

(*) The major problem is the $J(\theta)$ requires $P(\tau|\theta)$ which inturn requires transition probability (T) which is unknown. Our policy network is estimating (π_θ) by maximizing $J(\theta)$.

(*) Now the question is: Can we compute $J(\theta)$ without transition probabilities (T).

(*) But for backpropagation $\nabla_\theta[J(\theta)]$ only gradient of $J(\theta)$ is required and it is not depending upon (T).

As already shown

$$\therefore \nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p(\tau|\theta)} \left[r(\tau) * \nabla_\theta \log [p(\tau|\theta)] \right] \quad \text{(Eq- B)}$$

Expectation of Gradient

The log liklihood of such a sampled trajectory (τ) is using (Eq- A)

$$\log [p(\tau|\theta)] = \sum_{t \geq 0} \log T(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \quad \text{(Eq- C)}$$

Just differentiating it wrt [θ]:

Independent of [θ]

Dependent over [θ]

$$\nabla_\theta \log [p(\tau|\theta)] = \sum_{t \geq 0} \nabla_\theta \log [\pi_\theta(a_t|s_t)]$$

Independent of transition probability

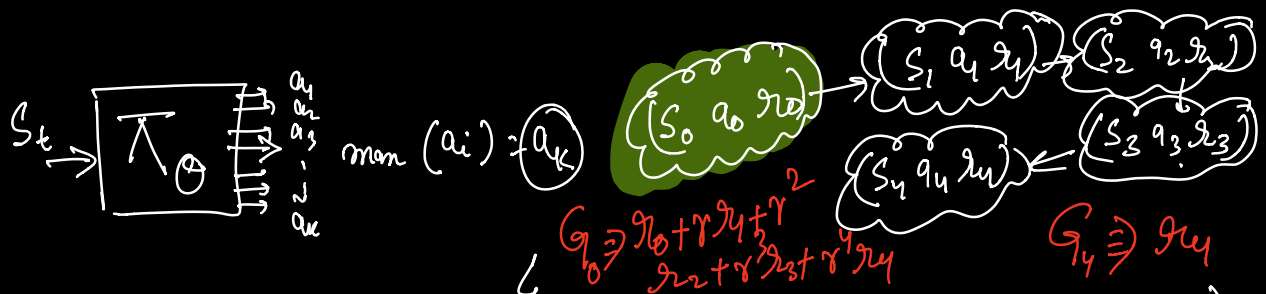
(Eq- D)

(*) Hence the derivati of log liklihood of the trajectory (τ), which is required for the Computation of the Gradient of the Reward function i.e $\nabla_\theta(J(\theta))$ is independent of Transition Probability [T].

$\Longrightarrow$ This also implies that $\nabla_\theta(J(\theta))$ is also independent of [T]

Hence putting (Eq-D) into (Eq-B)

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim p(\tau|\theta)} \left[r(\tau) * \sum_{t \geq 0} \nabla_\theta \log \left[\pi_\theta(a_t|s_t) \right] \right]$$



$$\nabla_\theta (J(\theta)) = \sum_{t=0}^{4} \nabla_\theta \log \left(\pi_\theta(a_t|s_t) \right) * G_t$$

$$\Rightarrow \underline{\underline{G_0}} * \nabla_\theta \log \left(\underline{\underline{\pi_\theta(a_0|s_0)}} \right) +$$

$$G_1 * \nabla_\theta \log \left(\pi_\theta(a_1|s_1) \right) +$$

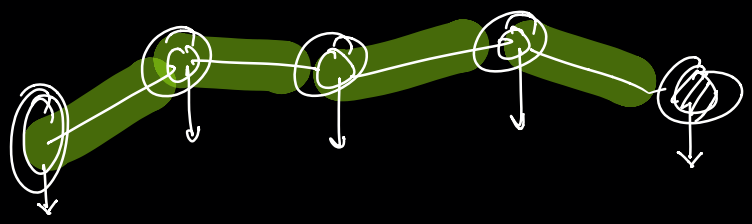
$$G_2 * \nabla_\theta \log \left(\pi_\theta(a_2|s_2) \right) +$$

$$G_3 * \nabla_\theta \log \left(\pi_\theta(a_3|s_3) \right) +$$

$$G_4 * \nabla_\theta \log \left(\pi_\theta(a_4|s_4) \right)$$

$R(z)$ Good — $R(z)$ (Gradient Estimation)

Credit-Assignment — Unbiased.

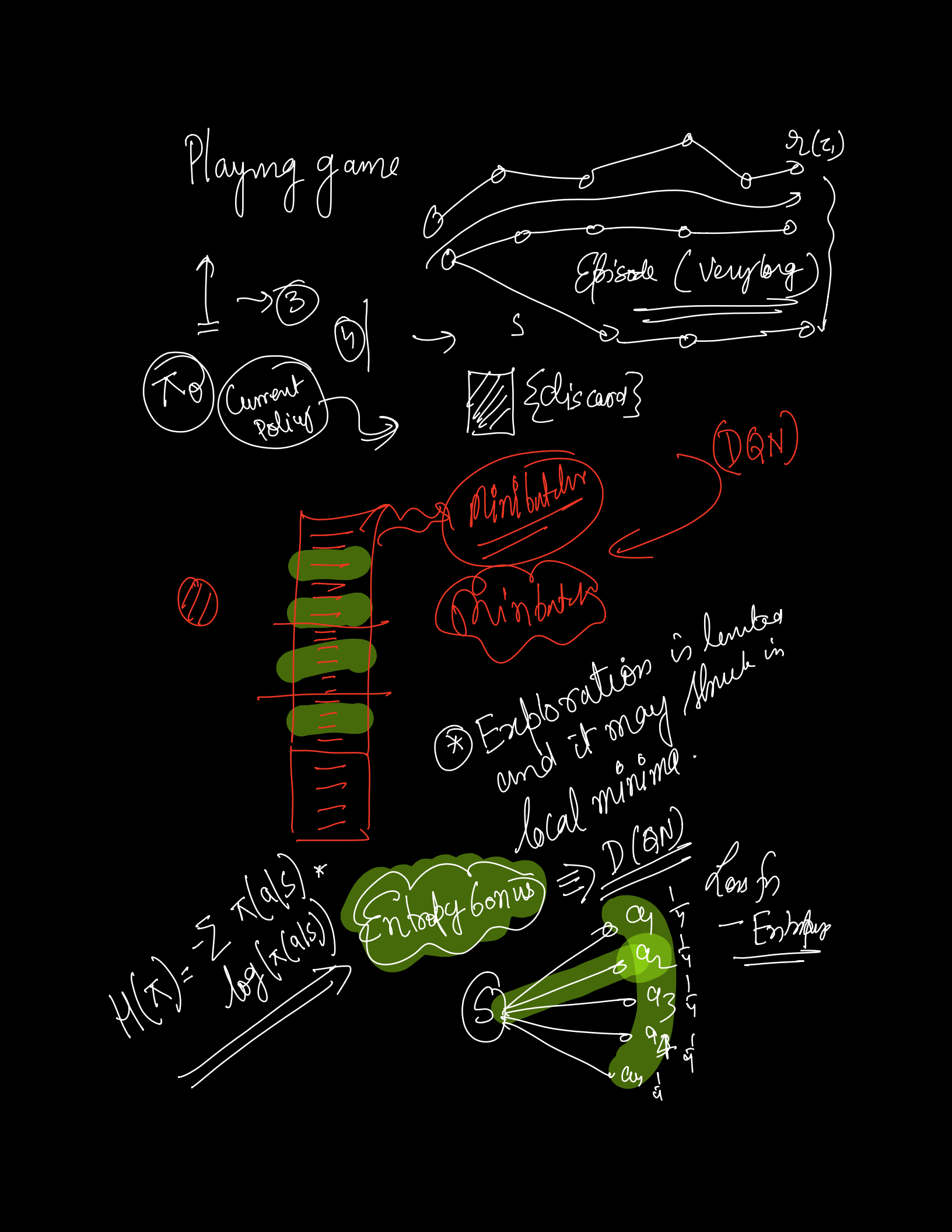


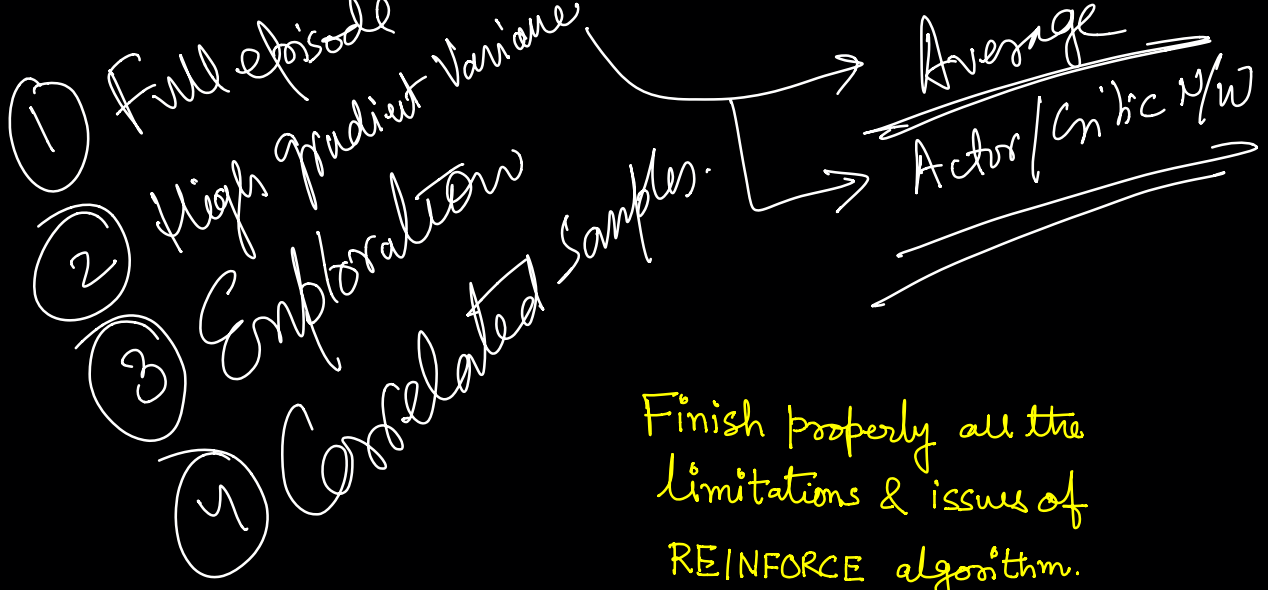
$$\nabla_\theta J(\theta) \propto \sum_{t \geq 0} r(z)\, \nabla_\theta \log(\pi_\theta(a_t|s_t))$$

$$\sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma_{t'} \right) \nabla_\theta \log(\pi_\theta(a_t|s_t))$$

$$\sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} \gamma_{t'} \right) * (\text{Same})$$

$r(z_1)$
$r(z_2)$
$r(z_3)$


Playing game
Episode (Very long)
Current Policy
{discard}
(DQN)
Minibatch
Minibatch
Entropy bonus
D(QN)
loss fn
Entropy



Finish properly all the limitations & issues of REINFORCE algorithm.

(1) Algorithm for Policy Gradient (REINFORCE)

(1) Initialize the network parameters θ.

looping

(2) Generate (N) trajectories $\{\tau^i\}_{i=1}^{N}$ $(\tau^1, \tau^2, \tau^3 \ldots \tau^N)$ following the policy π_θ.

(3) Compute the return of the trajectory $R(\tau)$

(4) Compute the gradient:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \right] * R(\tau)$$

(5) Update the network parameter

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

(6) Repeat steps (2) to (5) for several iterations.

(θ) getting updated after each iteration. Hence policy got changed (become better) and thus generate new data /episode/ trajectories for training.

It is an On-policy method.

How to manage high variance in the gradient

⊛ Actually we are using policy to generate the trajectory and then computing $\nabla_\theta J(\theta)$ to update the policy itself.

↳ This will in turn improve the policy after each iteration.

↳ Hence returns vary greatly introducing high variance in the gradient updates.

(ii) Policy gradient with reward-to-Go

for Vanila PG

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) * R(\tau) \right]$$

Proper Credit Assignment

where $R(\tau) = \sum_{t=0}^{T-1} r_t$

But we say Reward-to-Go defined as (R_t)

Sum of the rewards of the trajectory starting from the state (s_t).

$$\therefore R_t = \sum_{t'=t}^{T-1} r(s_{t'}, a_{t'})$$

Instead of $R(z)$ use R_t

An action is only as good as the DCFR it can generated

(iii) Policy gradient with the baseline

→ Trajectory lengths are different.

→ Favoring early rewards

Can we Normalize the Reward to G_0

↳ In order to reduce the variance one can subtract some baseline (b) from the (R_t) (Just like DC-Component/Average)

Can we develop the notion of "Expected"

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) * (R_t - b) \right]$$

Baseline (b): It is the value that can give us the expected return from the state the agent is in. Simplest baseline can be: $b = \frac{1}{N} \sum_{i=1}^{N} R(z)$ {or even moving average}

baseline can be any function but should not dependent upon (θ). (Does not affect N/W para)

Other obvious choices can be

↳ Value function ($V(s_t)$)

Value of a state is the expected return an agent would obtain starting from the state following (π).

↳ Other options are Q fn & Advantage fn.

How can we learn the baseline fuctions.

↳ Just like approximating policy using (θ) network

↳ One can use value network (ϕ)

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) * (R_t - V(s_t))$$

Now since the value of a state is floating point number, ϕ can be trained by minimizing M.S.E.

$(R_t) \Rightarrow$ Actual Return

$V(S_t) \Rightarrow$ Predicted Return

$$\therefore J(\phi) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \left(R_t - V_\phi(S_t) \right)^2$$

We need to minimize $J(\phi)$ hence

$$\phi = \phi - \beta \nabla_\phi J(\phi)$$

Algorithm – REINFORCE with baseline

The algorithm of the policy gradient method with the baseline function (REINFORCE with baseline) is shown here:

1. Initialize the policy network parameter θ and value network parameter ϕ
2. Generate N number of trajectories $\{\tau^i\}_{i=1}^{N}$ following the policy π_θ
3. Compute the return (reward-to-go) R_t
4. Compute the policy gradient:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \; (R_t - V_\phi(s_t)) \right]$$

5. Update the policy network parameter θ using gradient ascent as $\theta = \theta + \alpha \nabla_\theta J(\theta)$
6. Compute the MSE of the value network:

$$J(\phi) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \left(R_t - V_\phi(s_t) \right)^2$$

7. Compute gradients $\nabla_\phi J(\phi)$ and update the value network parameter ϕ using gradient descent as $\phi = \phi - \alpha \nabla_\phi J(\phi)$
8. Repeat *steps* 2 to 7 for several iterations

(A) Advantage fn (A_t^i) :- t^{th} step of i^{th} episode.

→ How "good" it is to take an action "a", at state "s".

when $Q(s, a) \geq Value(s)$

(Action (a) is better than expected/ average.)

$$A_t^i \Rightarrow Q(s_t, a_t) - V(s_t)$$ for some policy (π).

$$\nabla_\theta J(\theta) = \sum_{t \geq 0} \left[Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t) \right] * \nabla_\theta \log \pi_\theta(a_t | s_t)$$

Scaling fn for log likelihood

(*) Minimization of the advantage function will also enforces Bellman optimality.

(*) Computation of Advantage fn needs (2) neural networks (a) for Q fn (b) for value fn [This is not optimal]

This is computationally inefficient & expensive.

Q fn, Can be

↳ (a) Approximated via. Monte Carlo based Random Sampling or [Reward-as-Go].

I

$Q(S, a)$ ⇒ All future reward (R_t)

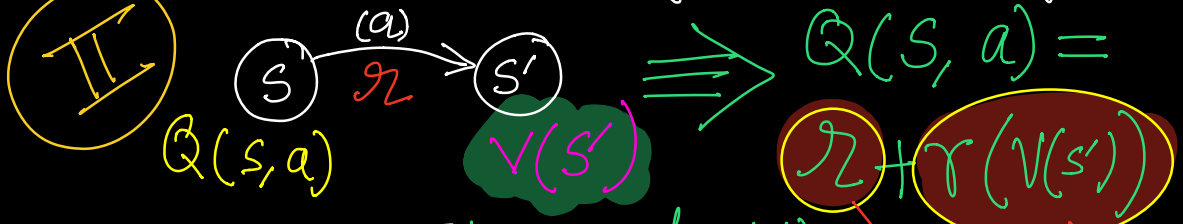
policy itself says take action (a) at (S)

(Doing it over the Experience)

(Major Concern is that, we need to have the full trajectory or trajectories)

(b) It Can be approximated by using the definition of Q-fn and using Bellman optimality.

II



This we can get from (Ø)

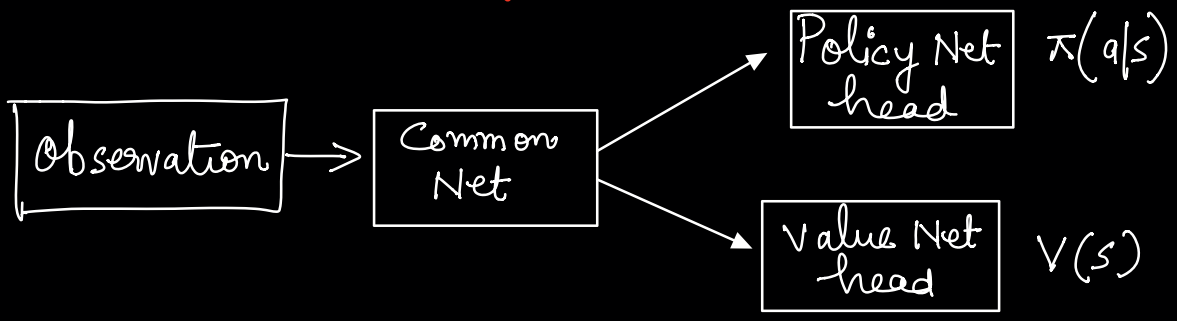
Q Value Can be Computed using Value fn N/w.

Now Q fn Can be Computed for any step, instantly using, immediate reward & O/P of Value fn.

Hence

$$A_t = A(S, a) = r + \gamma V(S') - V(S)$$

Actor Critic Algorithm



Observation :

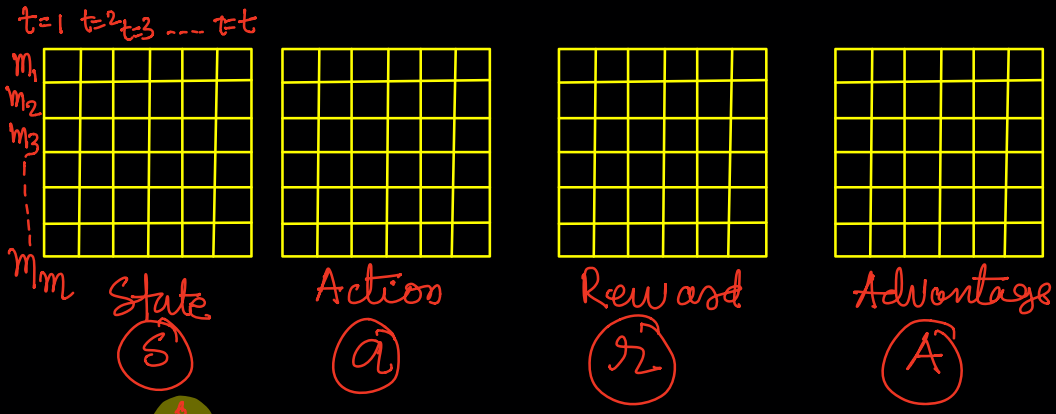
Sample (m) trajectories under the current policy (π).

m_1 $\langle s_0^1, a_0^1, r_0^1 \rangle$ $\langle s_1^1, a_1^1, r_1^1 \rangle$ ---- $\langle s_t^1, a_t^1, r_t^1 \rangle$

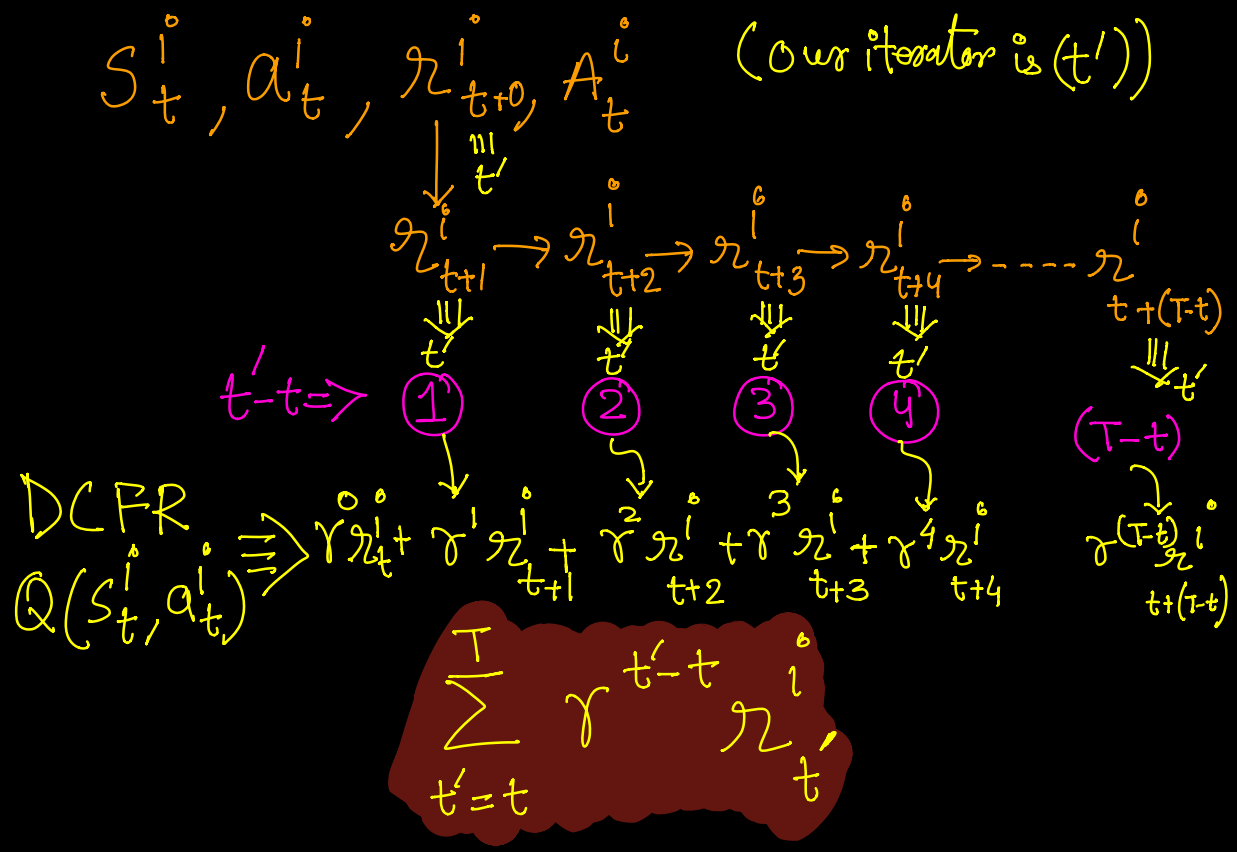
m_2 $\langle s_0^2, a_0^2, r_0^2 \rangle$

m_3 $\langle s_0^3, a_0^3, r_0^3 \rangle$

m_m $\langle s_0^m, a_0^m, r_0^m \rangle$ $\langle s_t^m, a_t^m, r_t^m \rangle$



S_t^i $\Rightarrow$ State of the agent at t^{th} step in the (i^{th}) episode.



ALGORITHM (Actor/Critic)

(1) Initialize the
- Policy N/w [θ] (Actor)
- Q learning N/c [ϕ] (Critic)

(2) For each training iteration = (1, 2, - -) do

After every iteration policy will be updated. New data need to be generated.

(2.1) Sample (m) trajectories under the current policy (π). (S, a, r, t)

(2.2) $\Delta\theta = 0$ — Gradient accumulator for Actor N/W. for a given policy (π), before training reset.

(2.3) for $i = 1, 2, \ldots (m)$ do — for each episode / trajectory

(2.3.1) for $t = 1, 2, \ldots (T)$ do — for each step (t)

Advantage fn for the (t^{th}) step of (i^{th}) episode

(2.3.1.1) $$A_t^{(i)} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^{i} - V_{\phi}(s_t^i)$$

A_t doesn't depends upon the Actor N/W (θ)

all future rewards (R_t)

Critic network

Minimization of A_t, enforces Bellman Eq.

Getting the updation from the i^{th} episode's t^{th} time step. scaled by (A_t^i)

(2.3.1.2) $$\Delta\theta = \Delta\theta + A_t^i \nabla_{\theta} \log(a_t^i | s_t^i)$$

Accumulating all the policy updates for (Actor N/W).

Gradient ascent

Gradient estimator step.

(2.4) $$\Delta\phi = \sum_i \sum_t \nabla_{\phi} \| A_t^i \|^2$$

Minimizing the advantage fn, Discounted accumulated future rewards getting close to the predicted value function.

for all episodes and for all time steps, just accumulate them all.

Critic N/W updation

A_t minimization enforces Bellman Constraint.

(2.5) $\theta = \theta + \alpha \Delta\theta$

(2.6) $\phi = \phi - \beta \Delta\phi$

old problem of deep Q-learning:
* Learn Q-values for all <state, action> pairs.

(3) End for

Also incorporate standard Experience replay trick too.

Critic only learns the Q values for generated/observed $\langle s, a \rangle$ pairs by the policy under consideration.

Are these algorithms Sample efficient (as they are On-policy)

* Instead of generating the full trajectory and then only Compute the Return/Reward, Can we do Something efficient.

* we Can approximate (R_t) or $Q(s,a)$ as:

$$R \approx r + \gamma(V(s'))$$

* Now, we don't need to wait till the end of episode, to Compute reward.

* One Can Compute gradient at each step & update the policy network parameters. [At each step]

Let us see the flow : π fn, V fn

1. Initialize, θ & ϕ
2. Get a random starting state (s_0)
3. Get a action $\pi(a_0|s_0)$
4. Get a reward (r_0) and next state (s_1)
5. Get the Value of (s_1) as $V_\phi(s_1)$
6. Get $Q(s_0, a_0) = r_0 + \gamma * V_\phi(s_1)$
7. Get Advantage fn, $A(s_0, a_0)$ as
$$A(s_0, a_0) = r_0 + \gamma * V_\phi(s_1) - V_\phi(s_0)$$
8. Get Policy Gradient : $\nabla_\theta J(\theta)$
$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(a_0|s_0)\left(r_0 + \gamma * V_\phi(s_1) - V_\phi(s_0)\right)$$
9. Update Actor N/W (θ) : $\theta = \theta + \alpha \nabla_\theta J(\theta)$
(See NO $(\Delta\theta)$ accumulation) (Gradient Ascent)
10. Get Critic N/W loss : $J(\phi) = A(s_0, a_0) = r_0 + \gamma * V_\phi(s_1) - V_\phi(s_0)$
(ϕ)
11. Get $\nabla_\phi J(\phi)$, Value Gradient and update Critic N/W (using gradient descent) : $\phi = \phi - \alpha \nabla_\phi J(\phi)$

The actor-critic algorithm

The steps for the actor-critic algorithm are:

1. Initialize the actor network parameter θ and the critic network parameter ϕ
2. For N number of episodes, repeat *step 3*
3. For each step in the episode, that is, for $t = 0, \ldots, T\text{-}1$:
    1. Select an action using the policy, $a_t \sim \pi_\theta(s_t)$
    2. Take the action a_t in the state s_t, observe the reward r, and move to the next state s'_t
    3. Compute the policy gradients:

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta (a_t|s_t)\big(r + \gamma V_\phi(s'_t) - V_\phi(s_t)\big)$$

    4. Update the actor network parameter θ using gradient ascent:

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

    5. Compute the loss of the critic network:

$$J(\phi) = r + \gamma V_\phi(s'_t) - V_\phi(s_t)$$

    6. Compute gradients $\nabla_\phi J(\phi)$ and update the critic network parameter ϕ using gradient descent:

$$\phi = \phi - \alpha \nabla_\phi J(\phi)$$

Both updates are done at each & every step of the episode

⇓

Sample efficient.