NPTEL CS52 - Reinforcement Learning

Spring 2024

Prof. Balaraman Ravindran (CSE, IIT Madras)

PMRF TA - Sandarbh Yadav (CSE, IIT Bombay)

Reinforcement Learning - Course NPTEL
Course Details Swayam
https://onlinecourses.nptel.ac.in/noc24_cs52/announcements?force=true#registration_confirmation
PMRF - Reinforcement Learning - (noc24-cs52)
NPTEL CS52 Reinforcement Learning - Live Sessions - Spring 2024 - YouTube

Content

Reinforcement Learning | Balaraman Ravindran **(contains reading materials)**
Read Sutton Barto (and be ahead of lectures)
Tom Mitchell chapter on RL
Russel Norvig RL intro
Markov Decision Processes Puterman
Csaba Szepesvari videos
Neurodynamic programming book of Bertsekas and Tsitsiklis

Programs

1. Coin
2. Greedy with mean, UCB
3. Thompson, CTR (Kaggle)
4. Value Iteration
5. Policy Iteration
6. QL SARSA
7. MLE, MAP, Full Bayesian (Thompson), TD(lambda), SARSA(lambda)
8. FQI
9. DQN
10. PG, AC
11. MAXQ
12. POMDP

Week 1

Introduction to RL

Behavioral psychology
Pavlov Dog
Delayed rewards
Noisy and Stochastic environment
Branching factor is huge in Backgammon Go
Neural Backgammon
TD Gammon
Self play: freeze one and learn other so gradually opponent also improves
Online learning: No priori feedback
Computational advertising

RL Framework and Applications

Sequence of decisions
No target unlike Supervised Learning
We know reward is 3 but not 3 is out of how much, can we do better than 3
Trial and error
In USL, primary goal is pattern detection
RL is all about making prediction of outcome (how much reward we will get if take this action or probability of winning)
Prediction at t+1 is more accurate compared to t
Go to t+1 better knowledge then make change at t for action at that state like prob of winning (Maybe take hit in that episode, do better next time)
Animals use TD learning
Tic tac toe: imperfect opponent coz need to have episodes where we learn, else no learning just random actions
1 win 0 otherwise : probabilistic interpretation -> prob of winning
Value function: move to state that has more prob of winning
Drawback: Need to wait till end to know how good we did
TD learning can learn on the way (no need to wait till end to update) Here use next state to update previous
Repeated states as can reach many ways
Explore coz if always play current best move then no real learning as no chance of getting better
Should exploratory moves be used to update previous states? Algos for both
When to stop exploring and start exploiting?
Try every action at least once to know which is best (once for deterministic more for stochastic)
DP: repeated structure n to n+1

Online Approximate DP
Atari was not given reward, inferred it from scores on screen as raw pixels
Sea quest: RL did not solve well

Introduction to Immediate RL

Immediate rewards
Stateless (or same state every time)
Outcome can be stochastic: repeat multiple times before being sure of outcome
Exploration exploitation dilemma
Reward (from psychology) / payoff (from economics) / evaluation (from optimization) / cost(from control theory)
Reward sampled from a distribution (Bernoulli here) ~ stochastic env (think like coin toss 1 if head else 0)
Different coin for each action
Repeat same coin for same action
For Gaussian: x axis has reward and y have probability of reward
Sample from distribution: random num between 0 and 1 use CDF to generate sample
Stationary distribution
Pick arm with highest expected payoff but the challenge is that we do not know the distribution generating payoff of arms so estimate it

Bandit Optimalities

Bandit problems (bandit coz it steals money)
Single arm
Multi arm (MAB)
Solution concepts:

1. Asymptotic correctness: Guarantee that **eventually** you'll selecting arm with highest payoff **Correctness perspective**
2. Regret optimality: total reward (**even during learning** want optimality)
   How steeply, you can become optimal
   If limit exploration, might miss out on optimality (tradeoff)
   No algo can guarantee that regret better than log T
   Rate at which regret grows is at least log T Rate of convergence perspective
3. PAC optimality: probably (in sense either right or wrong) approximately (close to being right)
   Final arm returned approximately close to actual best arm
   PAC: Probably it is approximately correct
   Epsilon-delta (similar to Hoeffding)
   P(q*(a) >= q*(a*) - epsilon) >= 1 - delta
   PAC optimal if can give above guarantee
   Can always do it if infinite samples
   Interested in minimizing sample complexity to give PAC guarantee

Sample complexity perspective

Value Function Based Methods

V: value function
Estimate Q* (sum of reward received by taking that action / # that action was taken) although do not know it
Q_t
Greedy: argmax (but explore too)
Epsilon-greedy: prob epsilon explore exploit with 1 - epsilon
Prob of taking current best action is 1 - epsilon + (epsilon/n)
Prob of exploration is (epsilon - 1)/n
Greedy gets bulk of probability and all others equal (some might be good even not best and some might be bad)
Also even if best is better by just 0.000000000001 it still gets bulk of probability
Need to decrease epsilon so that eventually take best action only
But do not decrease too fast else lose guarantee over correctness
Can fall proportional to 1/t but very slow convergence
So in practice set high epsilon and run it many times initially then decrease in steps
Softmax: proportional to current estimates (softer version of max)
Expo to avoid prob being negative
Beta: temperature parameter (hot means randomness)
High beta: behaving at uniform random as all goes close to zero
Beta close to zero: it becomes max

Running average instead of remembering whole history
Number of times you took action a (maintain that)
Can rearrange it in terms of alpha
Stochastic averaging equations
New estimate = Old estimate + Step_size * Error term
Error term = Target - Old estimate
Real target is expected value of R_t
But do not have that so using unbiased samples (hence called stochastic averaging)
Till now stationary distributions
Non stationary problems (coin might wears off)
Need ways to track changes over time
As n_a becomes large step size becomes very small
In stationary, it is fine coz already drawn many times
In non-stationary, need more help of newer samples than older ones
Use **constant** alpha < 1 as step size
Gradually alpha^2, alpha^3 for older rewards
So more attention to newer rewards
Tune alpha accordingly

Problem with constant alpha in case of stationary env: it will not converge coz paying more attention to recent rewards (Oscillation near true reward)
In non stationary, we can track it
Alpha decay for stationary

Week 2

UCB 1

K arms
n for discrete time
T for continuous time
Q*(a) : expected payoff for pulling arm a
q*(a) is actual
Assumption: reward bounded between 0 and 1

UCB Algo:
Init: pull each arm once
Loop: play arm j that maximizes
$Q(j) + \sqrt{2 \ln(n)/n_j}$

Track n_j, n and Q values

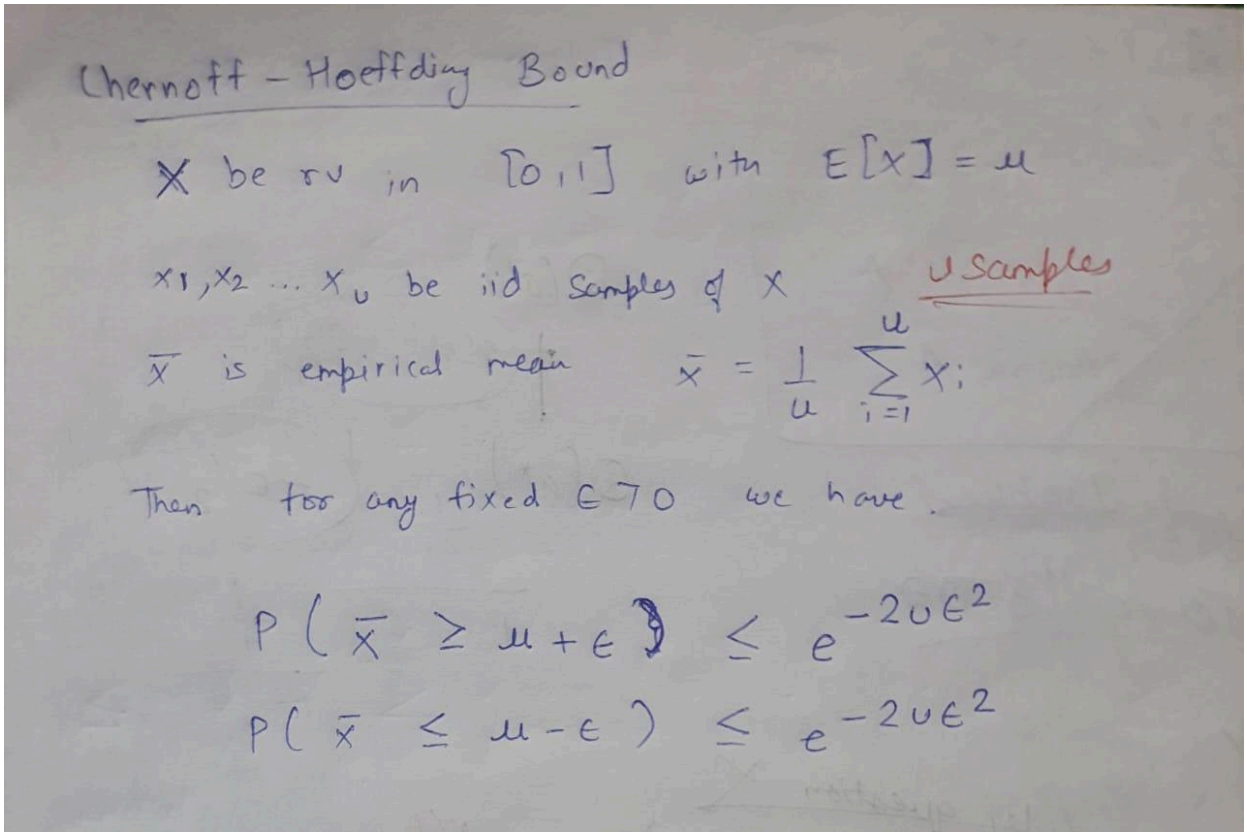
Confidence with which we can say that this is the expectation
Some kind of confidence interval
Lots of samples more confidence: interval of uncertainty becomes narrow
Larger n_j means second term becomes small
Numerator of second term is some kind of normalizing term
Deterministic algo: No random number involved

Concentration Bounds

Normal UCB, UCB revisited, UCB improved, UCB 2 etc. give better theoretical results than UCB 1 but exponentially harder analysis
UCB 1 uses arbitrary rewards, others might assume Bernoulli etc.
Default UCB is UCB 1

Concentration bounds/Large deviation bounds: relate true expectations (or variance or other statistic) of distributions with estimated values of expectations (or variance or other statistic)

Chernoff Hoeffding Bound



Small epsilon larger prob of making error
Need more samples to be confident enough

UCB 1 Theorem

Theorem: For all K>1, if UCB 1 is run on K arms having arbitrary reward distribution, with support in [0,1] then, its expected regret after any number of plays is at most

Regret = Sum over arms (Expected number of pulls of an arm * Delta)
Delta = Expected reward by pulling optimal arm - Expected reward by pulling that arm

Theorem : For all $K > 1$, if UCB 1 is run on K arms having arbitrary reward distribution, with support in $[0, 1]$, then its expected regret after any number of plays is atmost

$$8 \sum_{i : q_*(i) < q_*(a^*)} \left(\frac{\ln n}{\Delta_i} \right) + \left(1 + \frac{\pi^2}{3}\right) \left(\sum_{j=1}^{K} \Delta_j \right)$$

summation over non optimal — here optimal coz either way zero

where $\Delta_i = q_*(a^*) - q_*(i)$ it is calculated by true values

$T_i(n)$: No. of times played arm i after n trails (rv)

$$\text{Regret} = \sum_i E[T_i(n)] \Delta_i$$

$X_{i,n}$: (rv) denoting reward obtained by playing i at n^{th} step

$$E[X_{i,n}] = q_*(i)$$

Proof Show $E[T_i(n)]$ is bounded.

$$E[T_i(n)] \leq \frac{8}{\Delta_i^2} \ln(n) + C$$

Let $C_{n,s} = \sqrt{\frac{2 \ln(n)}{s}}$

$\{I_n = i\}$: 1 if $I_n = i$, 0 if $I_n \neq i$ — at time n whether arm i is played or not

This is 1 for a specific value of i & for others it's zero

$$T_i(n) = \underset{\text{initialize}}{1} + \sum_{m=k+1}^{n} \{I_m = i\}$$

Let l be an arbitrary positive integer

$$\leq l + \sum_{m=k+1}^{n} \{I_m = i, \ T_i(m-1) \geq l\}$$

If this holds then only count or m pull

coz already counting from l.

Basically we have pulled atleast l times

Now find l

Basically arm i has higher value

$$\leq l + \sum_{m=k+1}^{n} \left\{ Q_{m-1}(a^*) + C_{m-1, T_{a^*}(m-1)} \leq Q_{m-1}(i) + C_{m-1, T_i(m-1)} \right\}$$

Longer set so $\leq$ coz it has to be higher than all arms but it is compared only to optimal arm

and $T_i(m-1) \geq l$ } curly bracket is indicator

$$\leq l + \sum_{m=k+1}^{n} \left\{ \min_{0<s<m} \left(Q_s(a^*) + C_{m-1, T_{a^*}(s)} \right) \leq \max_{1 \leq s_i \leq m} \left(Q_{s_i}(i) + C_{m-1, T_i(s_i)} \right) \right\}$$

over count so $\leq$

$\rightarrow$ l coz atleast l times cond^n

either way this counts above

$$\leq l + \sum_{m=1}^{\infty} \sum_{s=1}^{m-1} \sum_{s_i=l}^{m-1} \left\{ Q_s(a^*) + C_{m, T_{a^*}(s)} \leq Q_{s_i}(i) + C_{m, T_i(s_i)} \right\}$$

min will occur at some index say p_1

max will ... say p_2

summing over all indices so that p_1, p_2 combo occurs somewhere

Now Chernoff Hoeffding bound

Above $\{\}$ can be true if one of several condition holds

(1) $Q_s(a^*) \leq q_*(a^*) - C_{m, T_{a^*}(s)}$ — grossly underestimating my optimal action's value

estimate — true value

(2) $Q_{s_i}(i) \geq q_*(i) + C_{m, T_i(s_i)}$ — grossly overestimating value of i^{th} action

(3) $q_*(a_*) < q_*(i) + 2\, C_{m, T_i(s_i)}$ — True values are close than not grossly overestimate or underestimate can be making mistakes.

can make a mistake. both ways

First 2 cond^n outside boundy

3rd one ~~or~~ mistake with in bound

close but due to bound mistake

(1) $P\left(Q_s(a^*) \leq q_*(a^*) - (m, T_{a^*}(s)) \right)$

empirical actual

$\leq \quad m^{-4} \qquad e^{-2 * \frac{2\ln(m)}{I_{a^*}(s)} \; I_{a^*}(s)}$

$e^{-4\ln m}$

$= e^{\ln(m^{-4})}$

$= m^{-4}$

(2) $P(Q_{s_i}(i) \geq q_*(i) + (m, T_i(s_i))$

$\leq m^{-4}$

(3) use l $\qquad T_i(s_i)$ is atleast l

For $l = \left\lceil \frac{8\ln(m)}{\Delta_i^2} \right\rceil$, (3) is false.

if sufficient no of pulls, 3rd event will never happen

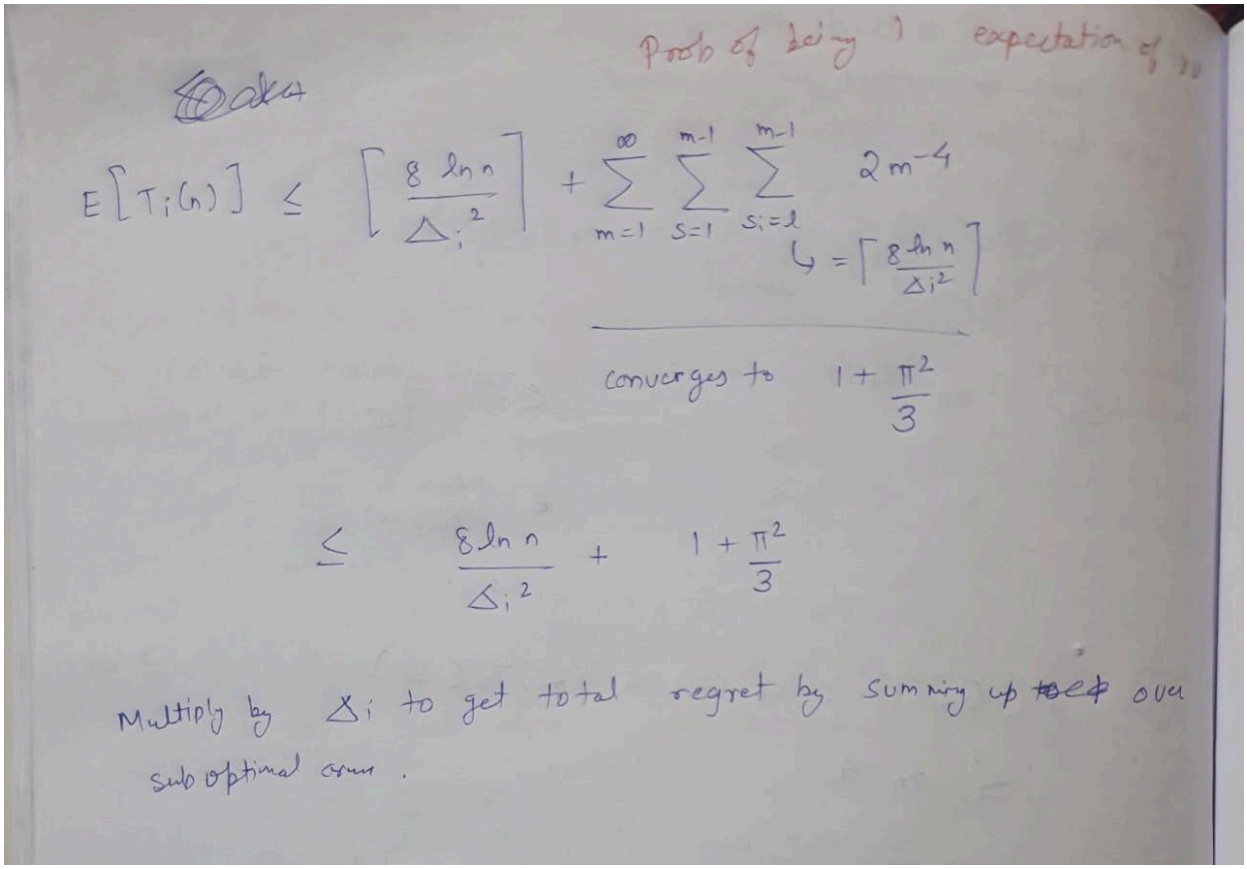
$q_*(a^*) - q_*(i) - 2\,(m, T_i(s_i))$ Just put $T_i(s_i)$ in $(m, T_i(s_i))$

$q_*(a^*) - q_*(i) - \Delta_i = 0$

This inequality will never hold.

$$\sqrt{\frac{2\ln(m^{-1})}{8\ln(m^{-1})}\Delta_i^2} = \frac{\Delta_i}{2}$$

If $l > \left\lceil \frac{8\ln(m)}{\Delta_i^2} \right\rceil$ prob of (3) is zero



ML algos can be analyzed in a similar way to randomized algos
a* true best action not empirical

Regret grows as log n
Result from 80s Lai Robbins can not do better than
Can improve on 8

PAC Bounds

Markov inequality: Pr(x>=a) <= E[x]/a where a is such that LHS is probability
Union bound: Probability of union of events <= Sum of probabilities of individual events

Input eps > 0, delta > 0
Give guarantee that returned arm is eps close to optimal arm with prob 1 - delta

Naive algo
For each arm a do
Sample a l times
l = (2/(eps*eps)) * ln(2k/delta) ; k denotes number of arms
Let Q(a) be avg reward of arm a
Return argmax of Q over a

Basically getting good enough estimate of each arm and playing best one accordingly

Theorem: Naive algo is (eps, delta) PAC algo with arm sample complexity O((k/(eps*eps))*log(k/delta))

Theorem Naive (ϵ, δ) is an (ϵ, δ) - PAC algo with arm sample complexity $O\left(\frac{k}{\epsilon^2} \log\left(\frac{k}{\delta}\right)\right)$

$\epsilon = q_*(a^*) - q_*(a')$

Proof Let a' be an arm s.t. $q_*(a') < q_*(a^*) - \epsilon$

a' is not ϵ - optimal.

$$Q(a') > Q(a^*)$$

if this happens then algo o/p a' instead of a^*

$$P\left(Q(a') > Q(a^*)\right)$$

Some subevent coz in reality need to be best better than others too.

$\leq P\left(Q(a') > q_*(a') + \frac{\epsilon}{2}\right)$ OR ← over estimate

$Q(a^*) < q_*(a^*) - \frac{\epsilon}{2}$ ← underestimate

using Union bound.

$$\leq P\left(Q(a') > q_*(a') + \frac{\epsilon}{2}\right) + P\left(Q(a^*) < q_*(a^*) - \frac{\epsilon}{2}\right)$$

Now Chernoff

$$\leq 2e^{-2\frac{\epsilon^2}{4} * l} \qquad l = \frac{2}{\epsilon^2} * \ln\left(\frac{2k}{\delta}\right)$$

$$= 2e^{-2\frac{\epsilon^2}{4} * \frac{2}{\epsilon^2} \ln \frac{2k}{\delta}}$$

$$= \frac{2\delta}{2k} = \frac{\delta}{k} \quad (k \text{ arms})$$

or

$$\frac{\delta}{k-1}$$

$$P\left(Q(a') > Q(a^*)\right) \leq \frac{\delta}{k}$$ Sum over all a

Prob of failure $\leq (k-1)\frac{\delta}{k} \leq \delta$

Hard to get rid of 1/(eps*eps) and log(1/delta)
Can not get rid of k either as need to try atleast once
Try to get rid of log k or constants

Median Elimination Algo

Input: eps>0, delta>0
Set S_l = A, eps_l = eps/4, delta_l = delta/2, l = 1
Repeat
    Sample each arm belonging to S_l for (1/((eps_l/2)*(eps_l/2)))*log(3/delta_l)
    Let Q_l denote its value
    Find the median of Q_l denoted by m_l
    Eliminate all arms whose value is below median
    S_(l+1) = S_l \ {a: Q_l(a) < m_l} ; Set difference
    eps_(l+1) = (3/4)*eps_l
    delta_(l+1) = delta_l/2
    l = l+1
Until |S_l| = 1

Log k rounds (base 2)

Show that at every round the probability of eliminating all eps optimal arms is very small
So small that even after summing across rounds it is still below delta

Theorem: Median elimination algo is (eps, delta) PAC algo with sample complexity O((k/(eps*eps))*log(1/delta))

Lemma:

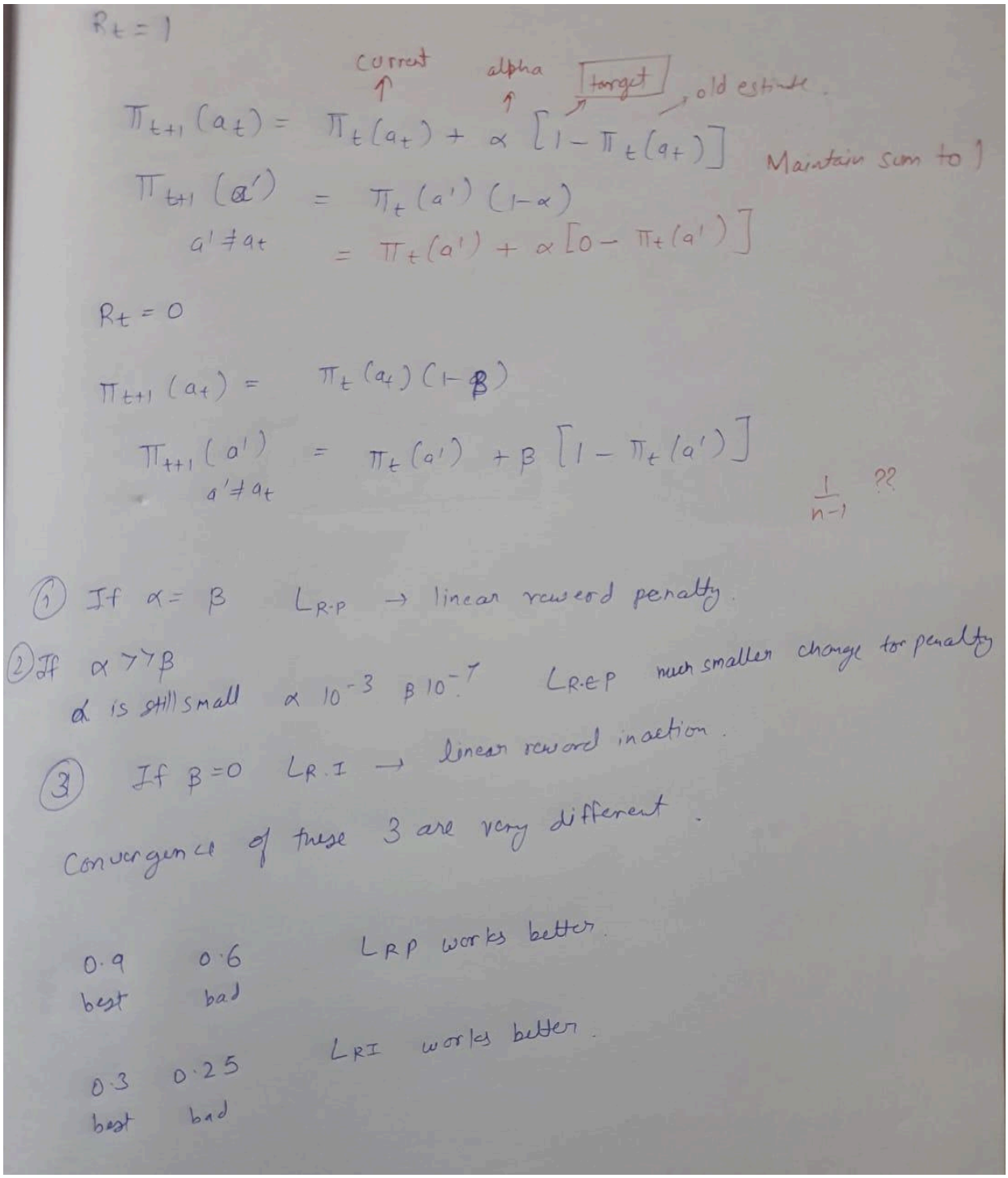
Thomson Sampling

Also called posterior sampling
Bayesian perspective
Some assumptions about parameters of unknown problems
Assume true Q* come from some distribution
Beta distribution (limited to 0-1) as prior
Belief
Gives better regret bounds than UCB
Analysis is complex
Learning automatons
Variable structured FSMs to solve Bandit problems
They do similar to posterior sampling

Week 3

Policy Search

Stochastic policy maps states to prob distribution over actions
Pi_t(a): prob of pulling arm a
Policy evolves over time
Eventually the prob of optimal arm becomes one
Binary bandit: 2 outcomes for reward 0 and 1



Variable structure finite automata
Policy gradient approaches (parameterized policy approaches)
Policy depends on set of parameters
Parameters used to generate values getting plugged into exponents of softmax
Can simply specify 25 for action 1 and 13 for action 2 with these numbers not necessarily representing value function

Preferences
Weights of an ANN
Alpha Go
Deep RL
Binomial Multinomial
Bernoulli Categorical
Modify policy params directly instead of value function

REINFORCE

Different params denote different policy
Performance measure of params: eta (theta)
One choice can be expected payoff: probability weighted q values of actions
Gradient ascent: move in direction of performance improvement
Grad asc coz we do not know function at all forget about closed form
Finding gradient by sampling (might not be correct so small steps instead of huge steps)
Stochastic gradient: estimate gradient
In expectation we move in right direction
How to estimate gradient when we do not know the function itself

$\theta \leftarrow \theta + \alpha \nabla \eta(\theta)$ $\quad \eta(\theta) = E[R_t] = \sum_a q_{\pi}(a)\ \pi(a,\theta)$

q_{π} does not depend upon θ

$$\nabla \eta(\theta) = \sum_a q_{\pi}(a)\ \nabla_\theta \pi(a;\theta)$$

$$= \sum_a \left(q_{\pi}(a)\ \frac{\nabla_\theta \pi(a;\theta)}{\pi(a,\theta)} \right) \pi(a;\theta)$$

π should not be zero for any action

$$= E_{\pi(\cdot;\theta)} \left[q_{\pi}(a)\ \frac{\nabla_\theta \pi(a;\theta)}{\pi(a,\theta)} \right]$$

Expectations can be estimated by sampling.

Also expectation over process generating reward q_{π}. So sample consists of R_t. $q_{\pi}(a)$ is also expectation over R_t.

$$\left(R_t\ \frac{\nabla_\theta \pi(a,\theta)}{\pi(a,\theta)} \right) \rightarrow \text{This is one sample}$$

Sum over $\qquad$ n coz discrete time

$$\approx \frac{1}{N} \sum_{n=1}^{N} R_n\ \frac{\nabla_\theta \pi(a,\theta)}{\pi(a,\theta)} \rightarrow \text{This is computable coz we chose } \pi\text{'s functional form.}$$

Fix θ & pull arm multiple times (N)

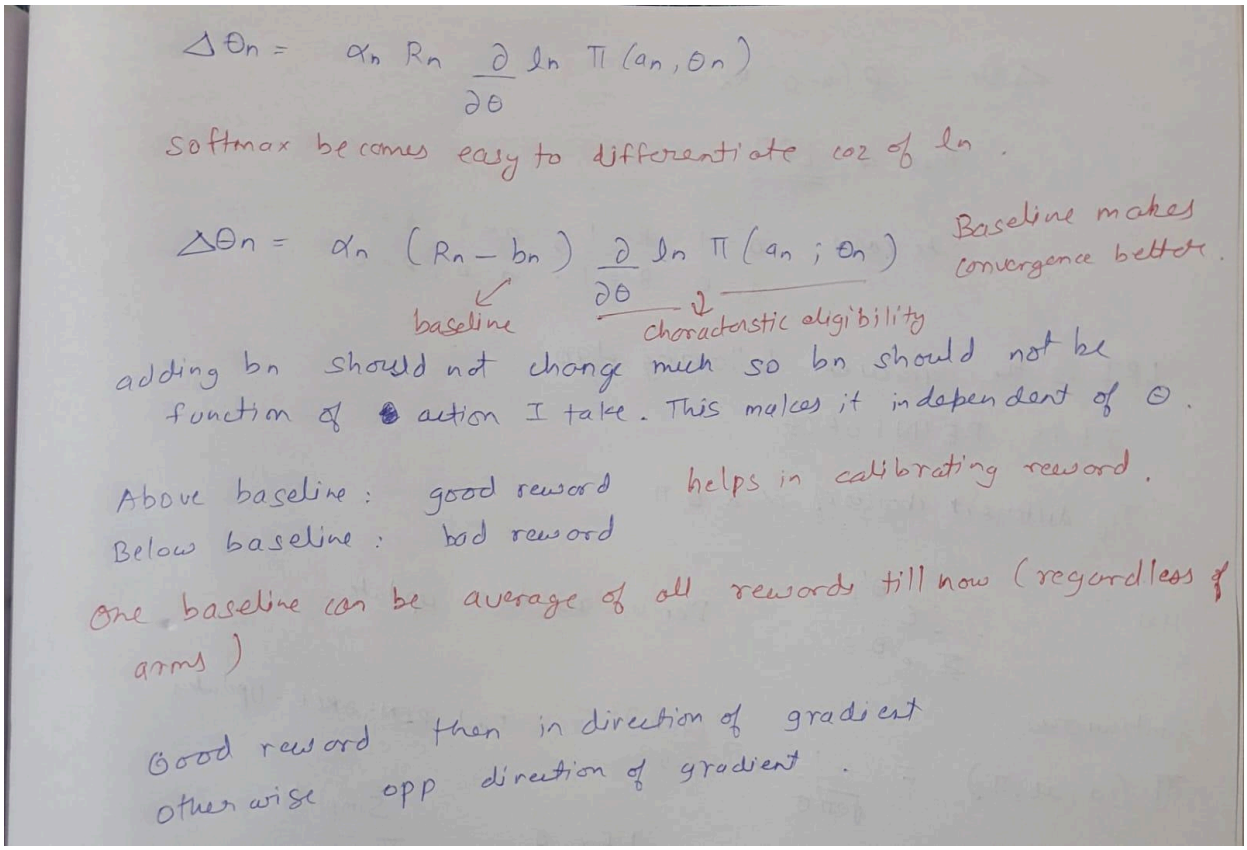
Incremental version : at every step change parameters

kind of like SGD

(REINFORCE)

$$\Delta\theta_n = \alpha_n R_n \frac{\nabla \pi(a_n,\theta_n)}{\pi(a_n,\theta_n)}$$

$$\theta_{n+1} \leftarrow \theta_n + \Delta\theta_n$$



Q value can turn out to be very complex function of state space, policy as direct representation is much simple

PG generalizes quite easily to **continuous** actions

When action spaces are continuous, value function based approaches are not that well behaved compared to policy based approaches

Reinforcement baseline

Characteristic eligibility: basically quantifies moving in which direction of theta gives max change

Becomes AC if use Q instead of R

AC addresses some problems of PG approaches

REINFORCE is the incremental version

In an expected sense, the gradient will be in the right direction

REINFORCE works very well but very slow due to high variance

Special cases of REINFORCE

Bandit with 2 actions but arbitrary rewards

$$\pi(a,\theta) = \begin{cases} \theta & \text{if } a=1 \\ 1-\theta & \text{if } a=0 \end{cases}$$

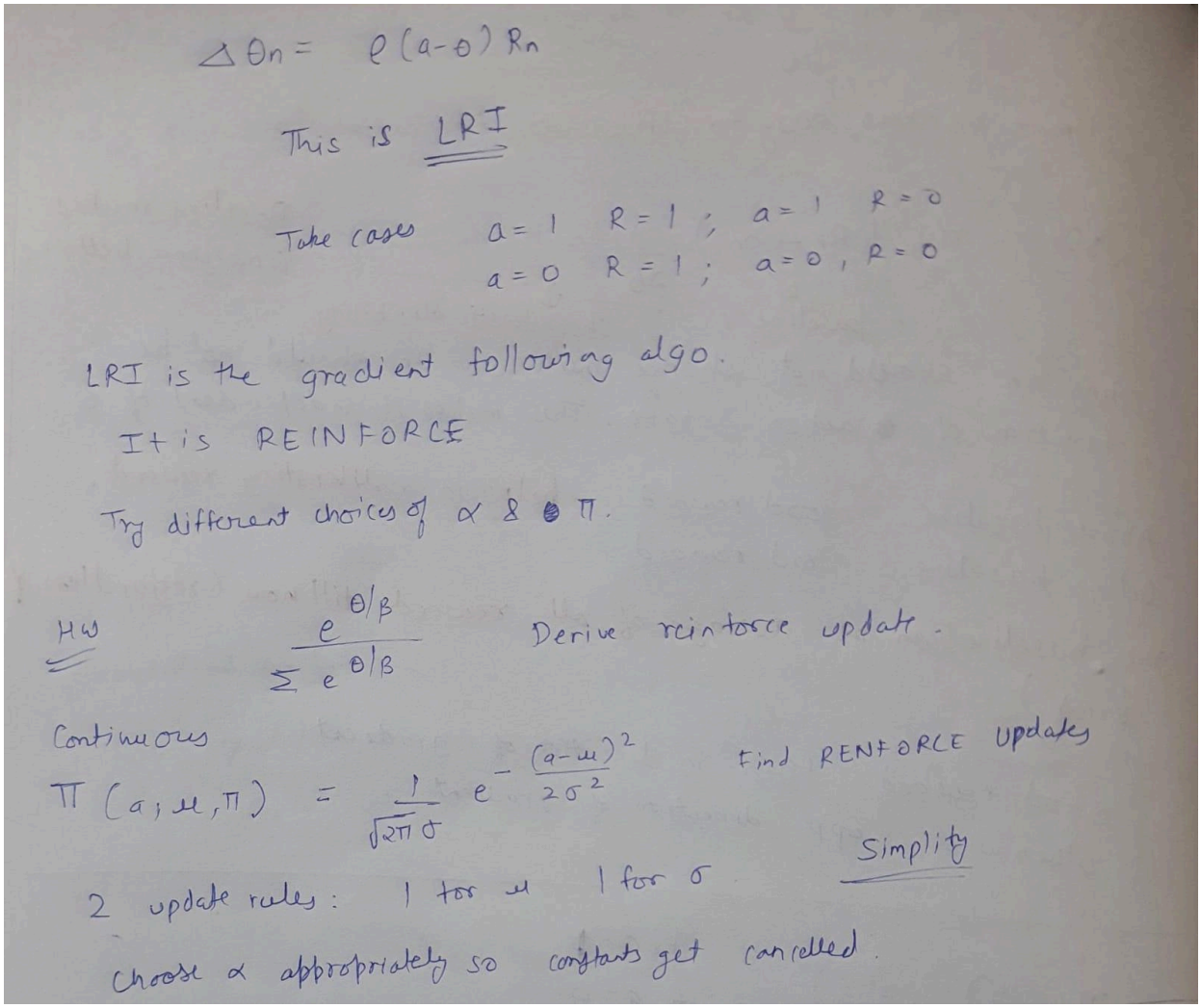
$$\frac{\partial \ln \pi}{\partial \theta} = \begin{cases} \frac{1}{\theta} & \text{if } a=1 \\ \frac{-1}{1-\theta} & \text{if } a=0 \end{cases}$$

$$= \frac{a-\theta}{\theta(1-\theta)}$$

α Should not depend on actual reward & actual acti
can depend on θ.
(FOR CONVERGENCE)

$$\alpha = \rho\,\theta(1-\theta)$$

$$b = 0$$



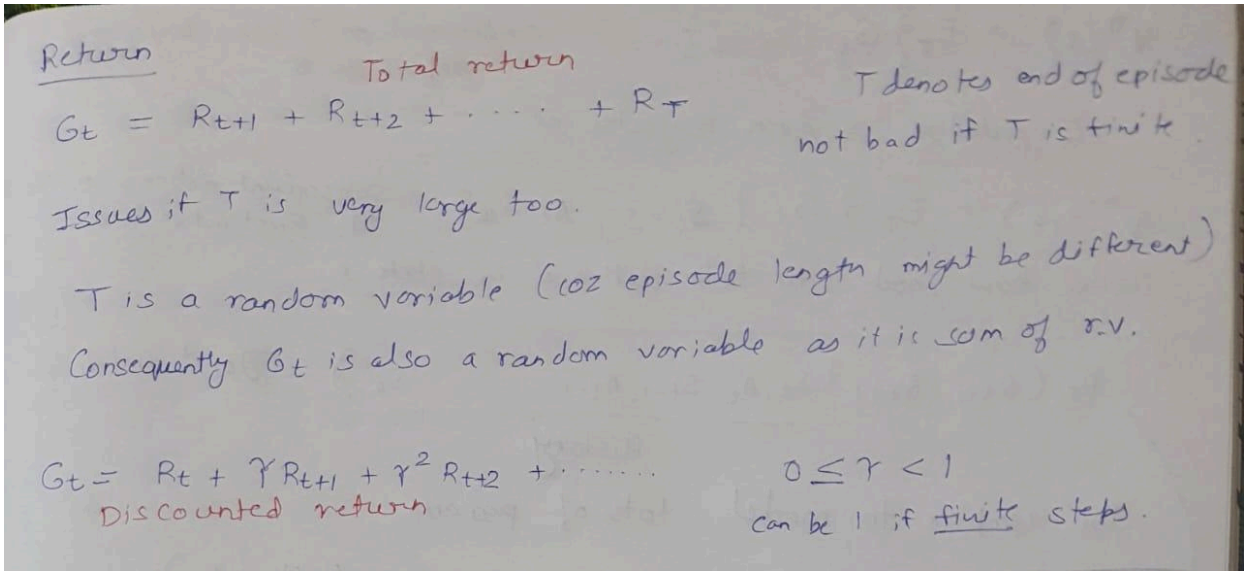
Contextual Bandits

Relax bandit assumption of being stateless
State dependence: diff rewards in diff states
Sequential dependence is missing
Set of bandit problems instead of just one
Maintain 10 different policies/values but this won't work if very large set
News story selection. Ad selection
Grouping users?
One way to solve contextual bandits: **parameterization of value** depending upon state as well as action (action can have params like story selected can have params like political, India etc. people's attributes form state params)
Parameterized bandits by default mean values
Linear bandits if linear in params
LinUCB: Q is linear in params

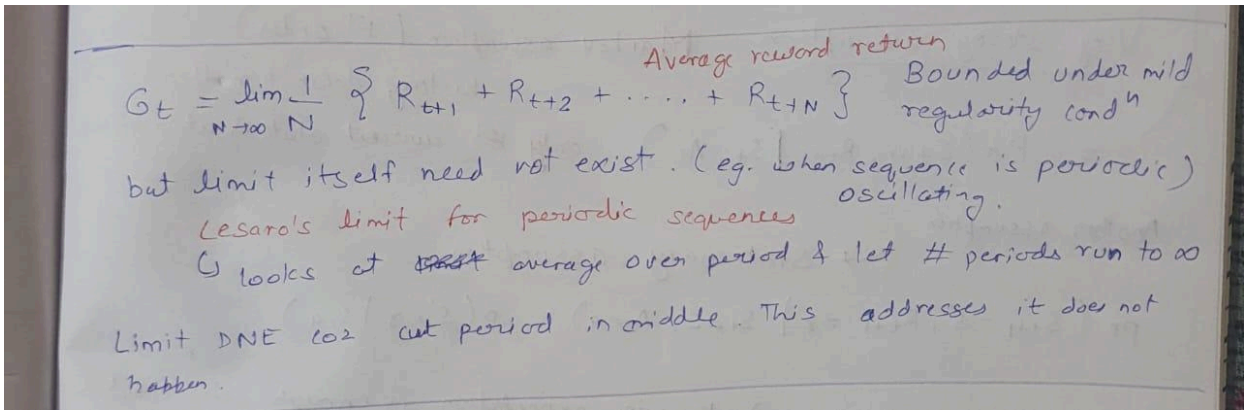
Full RL Introduction

Till now immediate RL and Bandits
Introduce states and sequential dependence
Starting states (can be a set) and terminal states (can be a set)
Can use a dummy state by grouping and use it as a sink
Agent does not control reward
In biology, sense through organs but rewards generated in brain: in real world, it is complex
Reward can come from anywhere: take environment one with a pinch of salt
Boundaries between agent and environment are not that clear
You will have to figure out the abstraction yourself
Book labels R_t as R_t+1 as they say next state and reward are determined at same time
From now on t for discrete time too
t need not be time but decision steps (later it will be discussed where we do not need to take decisions at every step)
Hierarchical RL: how long it takes to take an action
S+ is terminal state
Classical RL assumes reward to be a **scalar** and outside agent's control (agent cannot set reward)
Negative reward so as to reduce wandering around: like getting rid of constant pain
Will go faster toward final goal
Additional pain if unintended tasks
Need to set magnitude carefully: problem dependent
Scalar rewards formulation is quite powerful and we humans also do same type of decision making
Scalar values help in optimization too
Rewards are **bounded** mathematically
Frequent (non-zero) rewards are desirable not like once in million timesteps else poor learning coz zero changes nothing
Some people have tried vector rewards: each component corresponds to one of the desirables
Multi-criteria RL
Multi-agent RL
Qualitative feedback
Fuzzy RL: feedback is fuzzy
Instead of reward, look at perception of reward
Policy
Pi_t(a|s) = Prob(a_t = a|s_t = s) #stochastic policy (non-stationary)
Focus on stationary policy: Pi_1 = Pi_2 =...= Pi (basically the mappings from state to prob distributions over actions do not change over time)
End goal is a stationary policy although while learning policy might change as we strive towards optimal stationary policies
Goal: Learn a policy that maximizes **long-term** cumulative discounted reward
Focus should not be on short term rewards because they might be deceiving

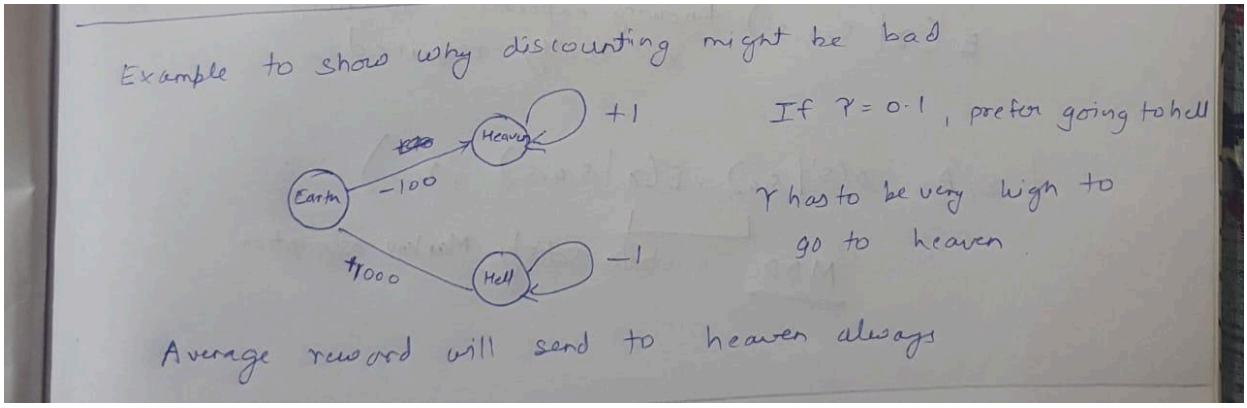
Returns, Value Functions and MDPs



Gamma not just for discounting but also for bounding sums at infinity
Near-sighted (gamma close to zero) or far-sighted (gamma close to one)
Discounting works for negative rewards also because
Instead of delivering pain (-1 reward), discounting can be used: sooner you reach final goal bigger the reward else it gets discounted many times
Tricky part is to figure out value of gamma
RL community thinks bandits as immediate RL
Bandit guys think sequence of pulls as learning process but do not use discount factor
Mismatch in perspectives
Optimal policy means a policy that from wherever I start following it will result in maximum expected reward
If horizon T is known then non-stationary problem (end horizon effects)
Here, T is known to be finite but exact value is not known
Payoff cost reward



Computationally easier but not from RL theory perspective: average reward

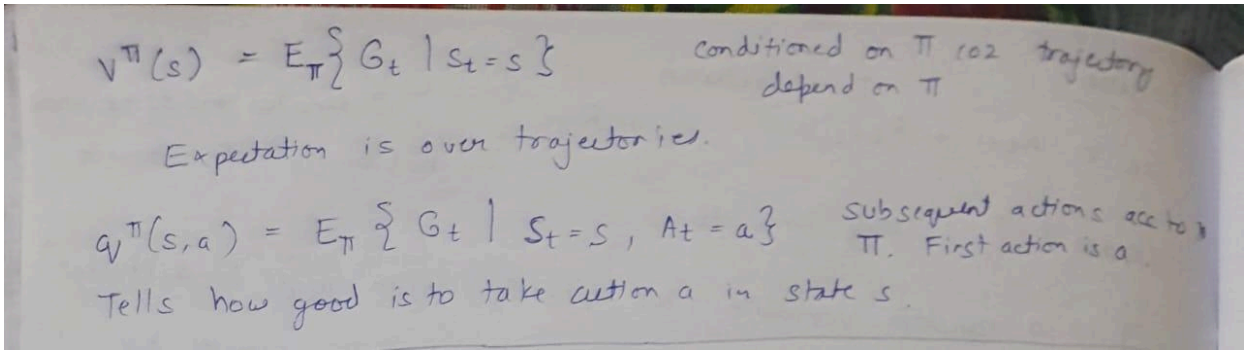


Gamma becomes part of problem definition

Optimal policy changes according to value of gamma

Discounted return is more realistic

Value function



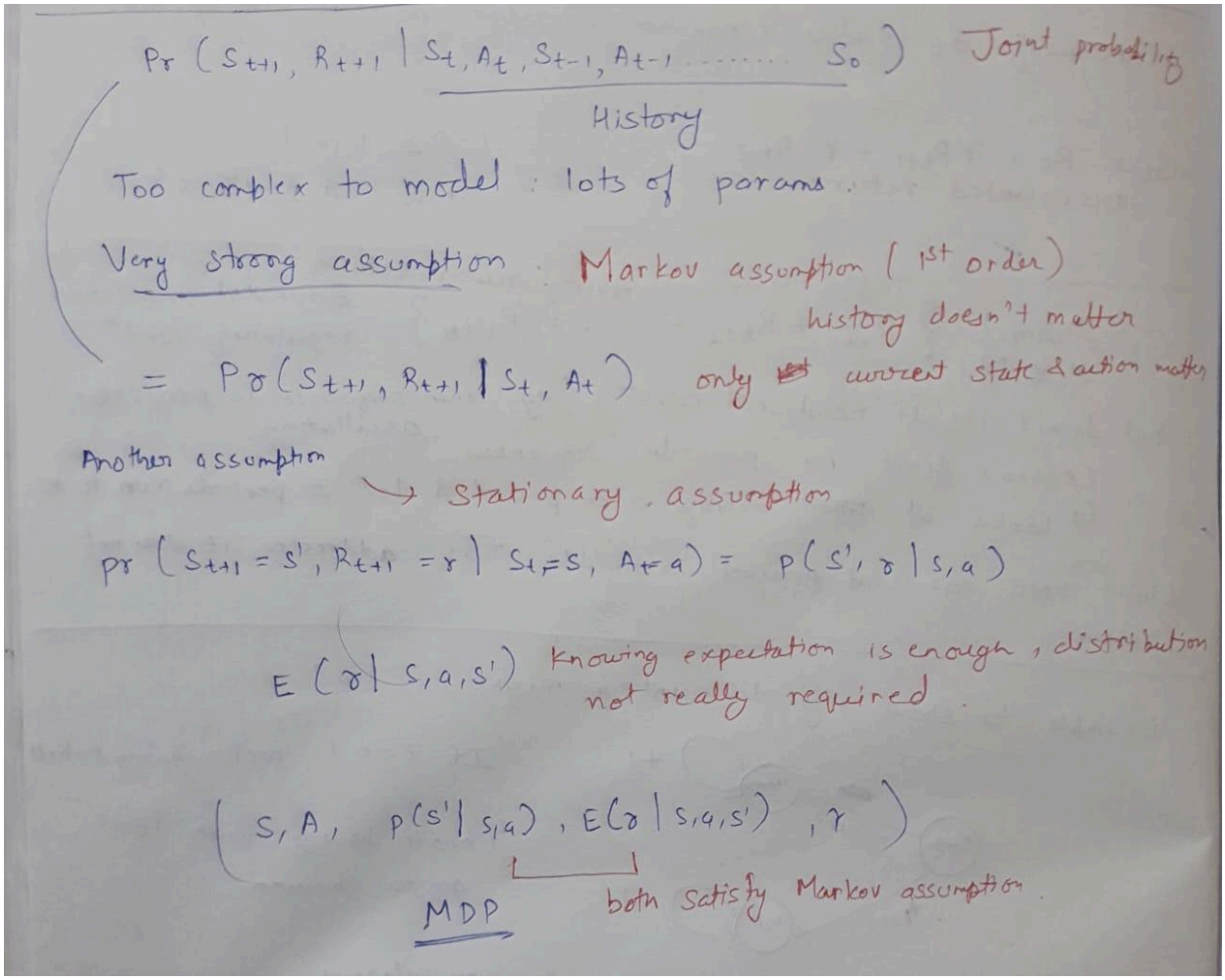
In bandits, expectation of reward

But here, expectation of return

Trajectory generation

2 levels of stochasticity: stochastic policy and then transition function

Trajectory depends on policy as well as transition function



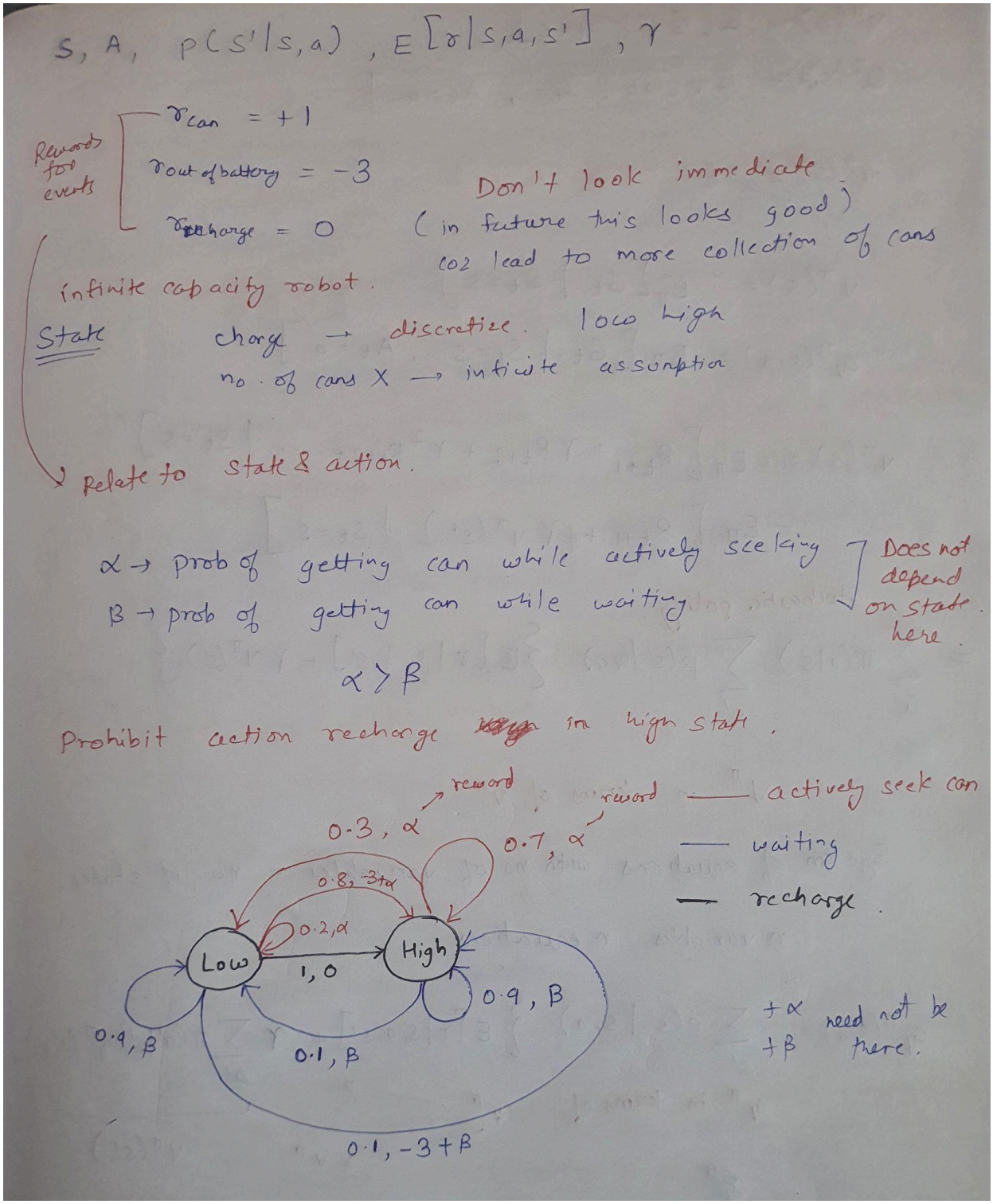
Markov and stationary assumption
Assume that the RL problems we are going to consider can be formulated using MDP
Value function makes sense only if we have Markov property
If it is not satisfied, then how I got to that state will influence future
Another way of thinking is value function is marginalizing over history
Quite often we apply RL blindly to non-Markov problems and value function is also used.
In that case this marginalizing idea gives sense.

Week 4

MDP Modelling

RL books leave out gamma but OR books keep it in definition of MDP (for them gamma is inherent part of problem definition)
Recycling robot example from book

Robot wants to gather empty cans and does not want to run out of battery. If it does then either has to be manually carried for recharge (painful process) or sit down expecting people to drop cans into it.

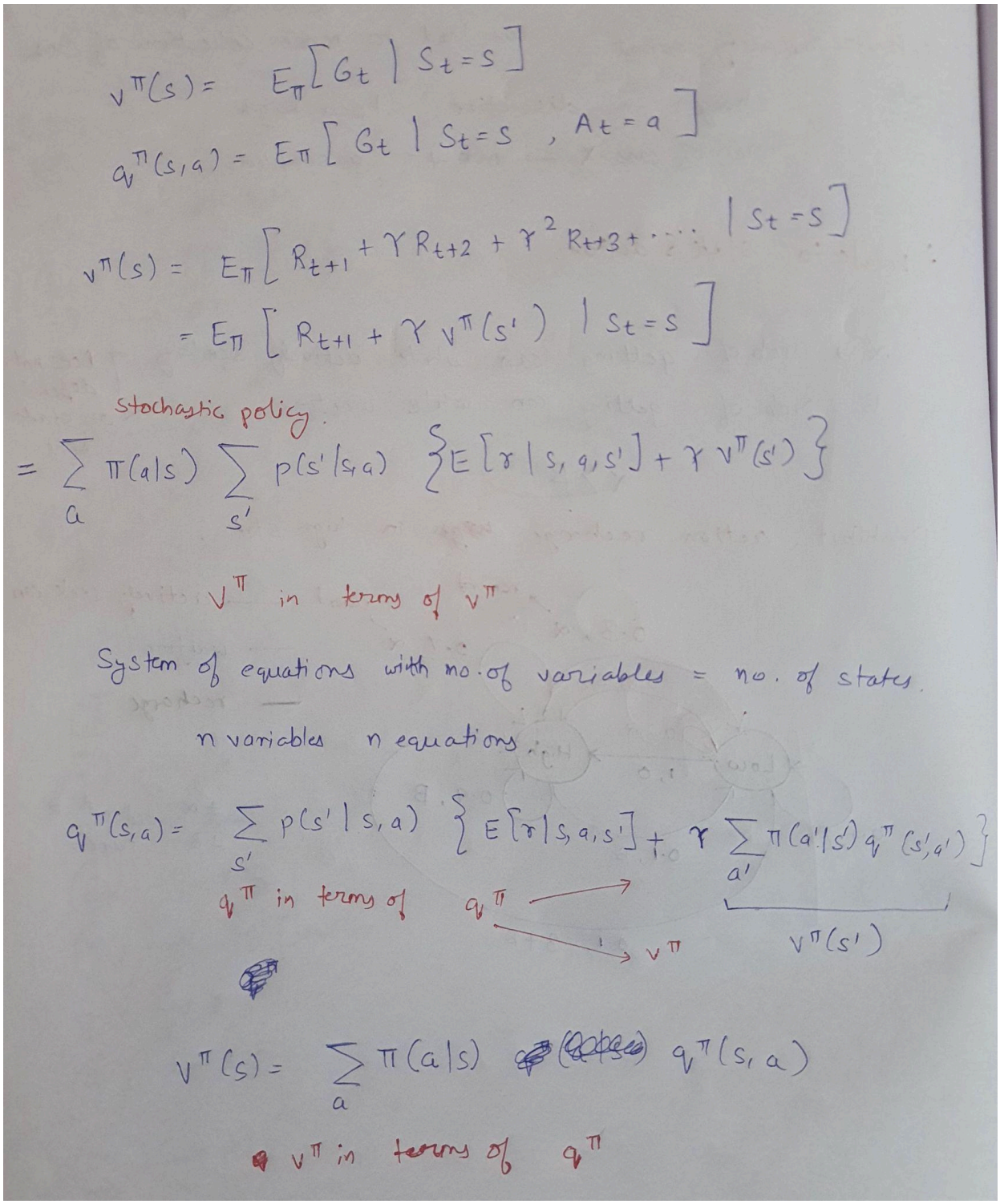


Actions: actively seek cans, wait for cans, recharge
Real life modeling is much complex

Going from problem specification to MDP formulation requires many design choices (need transition in planning but not in RL)
Although transition and reward depend on both state and action sometimes it might be same for some states and actions
Do not worry about solution while formulating problem

Bellman Equation

Bellman equations always have unique solution (by virtue of fact that p is stochastic: sum to 1)



Bellman Optimality Equation

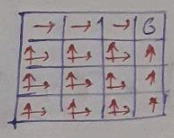
Getting T and R exactly is incredibly hard
Transition probabilities computation involves combinatorial sum

RL algo can work without T

$\pi^* = \arg\max_{\pi} v^{\pi}(s) \quad \forall s$ pointwise

need not be unique.

~~for~~ policy achieving $\max v^{\pi}$ for every state.

Gridworld 

reaching G positive reward
for everything else -1

basically reach G as quickly as possible.

2^9 optimal policies in deterministic setting. (infinite if stochastic)

For all these policies, net reward is going to be same.

So basically value will be same for state regardless of policy chosen to be executed.

optimal policy. $v^{\pi^*} = v^*$ ← optimal value fn

π^* need not be unique

v^* is unique.

Q How will some π reach $\max^m$ at all states??

Later.

Optimal action value function

$$q_v^*(s,a) = \max_{\pi} \sum_{s'} p(s'|s,a)\left[E[r|s,a,s'] + \gamma v^{\pi}(s') \right]$$

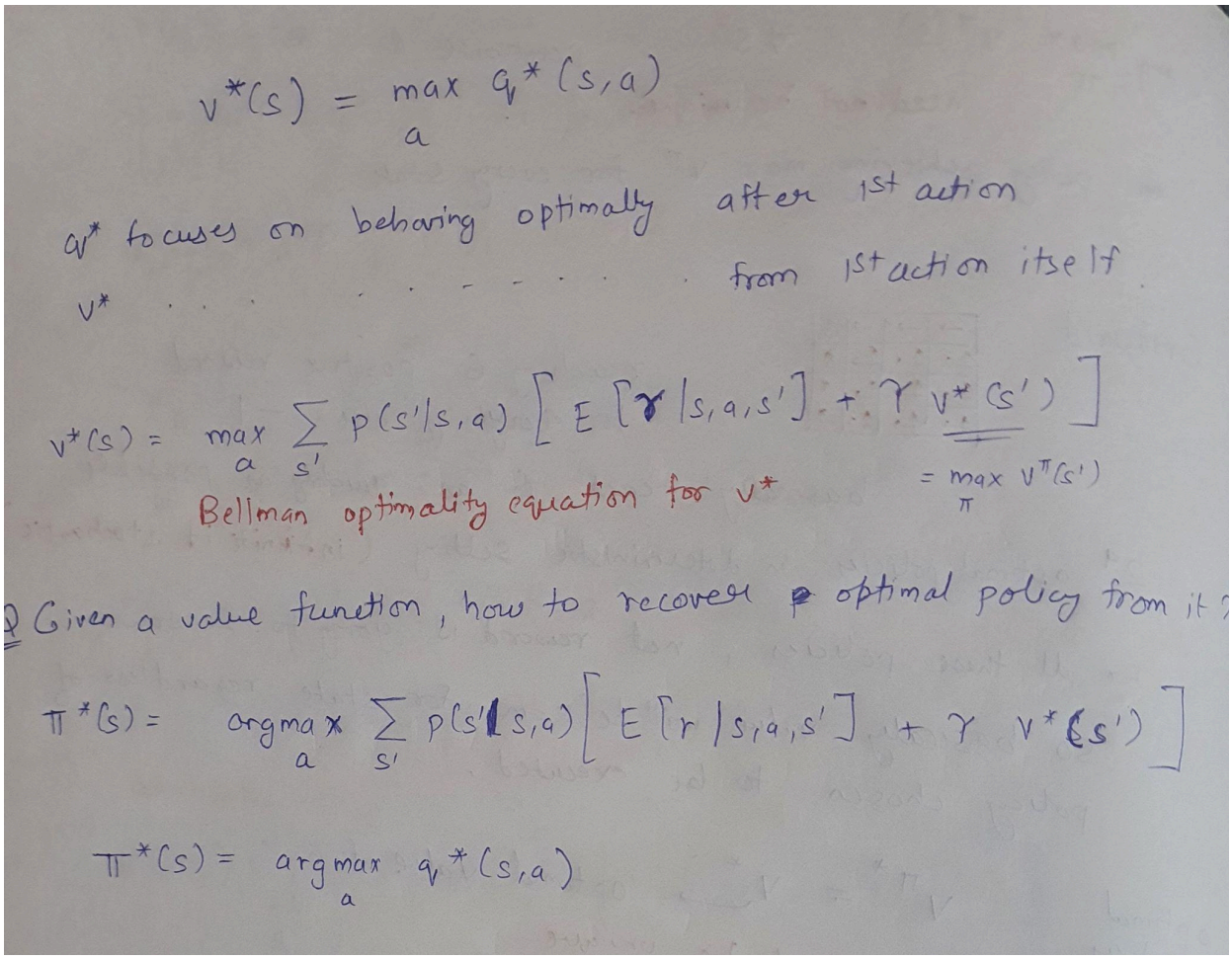
even though optimal policy but still first action is a

Behaving optimally after taking a.

$$q_v^*(s,a) = \sum_{s'} p(s',s,a)\left[E[r|s,a,s'] + \gamma \max_{\pi} v^{\pi}(s') \right]$$

$\left(= v^*(s')\right)$

q_v^* is defined even for suboptimal actions.



A policy assigning non zero probabilities only to optimal actions is also optimal
Easier to solve Bellman equations (as they are linear) compared to Bellman optimality equation (non-linear due to max)
V^pi is unique
V* is unique

Cauchy Sequence and Green's Equation

Assumption: Optimal policy will have max value for each state
Assumption: Optimal policy is deterministic
Vector spaces: closed under addition and scalar multiplication
Finite MDPs: n (|S|) and k (|A|) are finite
Think of value function as a vector with |S| components
Max-norm
Normed vector space: vector space with a norm defined on its vectors
Determinant = product of eigenvalues
This means a matrix is invertible if no eigenvalue is zero

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} p(s'|s,a) \left[E[r|s,a,s'] + \gamma V^{\pi}(s') \right]$$

$$V^{*}(s) = \max_{a} \sum_{s'} p(s'|s,a) \left[E[r|s,a,s'] + \gamma V^{*}(s') \right]$$

$\pi(a|s) \longrightarrow \pi(s)$ for deterministic policy.

$\pi(a|s)$: returns prob of a given s.

$\pi(s)$: output action for which prob is 1

If the MDP has an optimal policy, then $\exists$ a deterministic optpolicy

Norm

$\|x\| = 0$ iff $x = 0$

$\|\alpha x\| = \alpha \|x\|$ α is scalar.

$\|x+y\| \leq \|x\| + \|y\|$ triangle inequality.

Cauchy sequence For every +ve real $\epsilon > 0$, $\exists$ $N \in Z^{+}$ s.t. (any epsilon)

$\forall\ m,n > N$, $\|x_m - x_n\| < \epsilon$ ($x_1, x_2, x_3 \ldots$)

we can find some point in sequence beyond which all points are within ϵ of each other.

successive diff get smaller does not talk about limit

Complete normed vector space: If every Cauchy sequence in a normed vector space, converges to a point in the vector space then vector space is called complete normed vector space. Here, it says there is a limit & it is within the space.

$$r_{\pi}(s) = \sum_{a} \pi(a|s) \cdot \sum_{s'} p(s'|s,a) E[r|s,a,s']$$

reward by starting in s & following policy π — one step reward

$$P_{\pi}(j|s) = \sum_{a} \pi(a|s) \cdot p(j|s,a)$$

one step transition prob of ending in state j starting from s following π

For deterministic policy

$$r_\pi(s) = \sum_{s'} p(s'|s,\pi(s)) \, E[r \mid s, \pi(s), s']$$

$$P_\pi(j|s) = p(j|s,\pi(s))$$

r_π is $|s|$ dimensional vector.

P_π is a $|s| \times |s|$ dim stochastic matrix

all rows non negative elements summing to 1

$0 \le \gamma < 1$

$r_\pi + \gamma P_\pi V$ → this vector contains reward of landing in a particular state.

terminal cost interpretation

want V^π satisfying

$$V^\pi = r_\pi + \gamma P_\pi V^\pi$$

$$\Rightarrow V^\pi = (I - \gamma P_\pi)^{-1} r_\pi$$ Green's equation.

$\Rightarrow V^\pi$ is unique

Eigen values of I are all 1

Eigen value for stochastic matrix is 1

Multiply by γ so < 1

Invertible coz can't have zero eigen value [1 − something less than 1]

2nd way start with arbitrary V^π & keep using Bellman equation on V^π again & again

Claims: ① going to converge

② converge to V^π

③ unique.

Banach Fixed Point Theorem

Every point in vector space need not correspond to a policy whose value function corresponds to that point

There is a family of fixed point theorems depending on which space you are considering: Banach is just one of them (simpler)

$L_\pi : V \to V \equiv L_\pi(V) = r_\pi + \gamma P_\pi V$

↓ complete normed vector space of value function

Bellman eqn tells $L_\pi V = V$

$\Rightarrow V^\pi$ is a fixed point of L_π

Banach Fixed point theorem

Suppose you have a Banach space U (complete normed vector space) and $T: U \to U$ is a contraction mapping.

[$T(u)$ & $T(v)$ are closer to each other than u & v]

Then, $\exists$ an unique v^* in U s.t. $Tv^* = v^*$ & for arbitrary unique fixed point ① ②

v^0 in U, the sequence $\{v^n\}$ defined by

$v^{n+1} = Tv^n = T^{n+1} v^0$ converges to v^*.

unique fixed pt $\not\Rightarrow$ all will converge to it

① $\not\Rightarrow$ ②

But here both happen.

Need to show L_π is a contraction mapping.

Contraction mapping: T is a contraction mapping if

$$\|Tu - Tv\| \leq \lambda \|u - v\|, \quad 0 \leq \lambda < 1, \quad \forall u, v \in U$$

can swap u & v

Smaller λ, more intense contraction.

Larger λ, less intense contraction

$\lambda = 1$, no contraction. (not guaranteed)

Proof (of Banach's fixed point theorem)

Triangle inequality.

$$\|v^{n+m} - v^n\| \leq \|v^{n+m} - v^{n+m/2}\| + \|v^{n+\frac{m}{2}} - v^n\|$$

Just 3 points in vector space.

Split further : repeatedly apply Δ inequality

$$\|v^{n+m} - v^n\| \leq \sum_{k=0}^{m-1} \|v^{n+k+1} - v^{n+k}\|$$

$$= \sum_{k=0}^{m-1} \|T^{n+k} v^1 - T^{n+k} v^0\|$$

$$\leq \sum_{k=0}^{m-1} \lambda^{n+k} \|v^1 - v^0\|$$

$$= \|v^1 - v^0\| \frac{\lambda^n (1-\lambda^m)}{1-\lambda}$$

As n & m become longer & longer this equence will become smaller & smaller.

$\Rightarrow$ sequence $\{v_n\}$ is Cauchy.

Convergence Proof

Triangle inequality

$$0 \le \|Tv^* - v^*\| \le \|Tv^* - v^n\| + \|v^n - v^*\|$$

$$= \|Tv^* - Tv^{n-1}\| + \|v^n - v^*\|$$

$$\le \lambda \|v^* - v^{n-1}\| + \|v^n - v^*\|$$

As $n \to \infty$, $v^{n-1} \approx v^*$ & $v^n \approx v^*$ so both zero at ∞

$\{v^n\}$ is Cauchy. $\Rightarrow v^* - v^{n-1}$ is going to go to zero

$v^n - v^*$ will also go to zero.

Since $\|v^* - v^n\| \to 0$ as $n \to \infty$

$$0 \le \|Tv^* - v^*\| \le 0$$

$$\Rightarrow Tv^* = v^*$$

Uniqueness

Let u^* & v^* be 2 fixed points

$$\|Tu^* - Tv^*\| \le \lambda \|u^* - v^*\|$$

Since fixed points

$$\|u^* - v^*\| \le \lambda \|u^* - v^*\|$$

~~Case 2 but~~ $\lambda < 1$

This can only happen if $u^* = v^*$

Till now showed Banach fixed point theorem.

Now need to show that L_π is contraction mapping.

Let u & v be in V

$$L_\pi\, u(s) = r_\pi(s) + \sum_{j \in S} \gamma\, P_\pi(j|s)\, u(j)$$

$$L_\pi\, v(s) = r_\pi(s) + \sum_{j \in S} \gamma\, P_\pi(j|s)\, v(j)$$

Let $L_\pi\, v(s) > L_\pi\, u(s)$

$$0 \le L_\pi\, v(s) - L_\pi\, u(s)$$

$$\le r_\pi(s) + \gamma \sum P_\pi(j|s)\, v(j) - r_\pi(s) - \gamma \sum P_\pi(j|s)\, u(j)$$

$$= \gamma \sum_j P_\pi(j|s) [v(j) - u(j)]$$

$$\le \gamma \|v - u\|_\infty \sum_j P(j|s)$$

$$= \gamma \|v - u\|$$

Do on your own for $L_\pi\, v(s) < L_\pi\, u(s)$.

Together

$$\|L_\pi\, v(s) - L_\pi\, u(s)\| \le \gamma \|v - u\| \quad \forall s.$$

point wise (component) its drawn closer so obviously max wise also

$\Rightarrow$ L_π is a contraction mapping.

Week 5

Lpi Convergence

For **finite MDPs**, there will **always** be a deterministic optimal policy. So searching for a deterministic policy is sufficient.

$v \in V$ V is space of bounded functions. (component wise bounded)

$v(s)$ one component. How to show V is Banach space??

Vectors are getting ϵ-close to each other & component wise they are bounded so the limit also has to be a bounded function. If limit is bounded then it will exist in same space.

Some bounded f:n have unbounded limits.

R^n is bounded under any norm. Tricky part is to show for subset of R^n (since all components are bounded so subset)

Sequence of bounded f:n with difference becoming smaller should be bounded. co2 can't have an unbounded value which is intinitely close to a bounded value.

~~Stat~~
Intuition type proof ↑

Optimality

vector form of Bellman optimality eqn

$$v^* = \max_{\pi} \{ r_\pi + \gamma P^\pi v^* \}$$

Component wise maximization

|||

vector notation.

$$v^*(s) = \max_a \{ E\{ r | s,a \} + \gamma \sum_j p(j|s,a)\, v^*(j) \}$$

marginalising over s'

$$L v = \max_\pi \{ r_\pi + \gamma P^\pi v \}$$

L is operator.

L is not linear f:n of v although denoted as Lv.

Claim: v^* is fixed point.

Need to show L is contraction (Banach space one is intuitive)

Let $a_s^* \in \arg\max_a \{ E[r|s,a] + \gamma \sum_j p(j|s,a)\, v(j) \}$

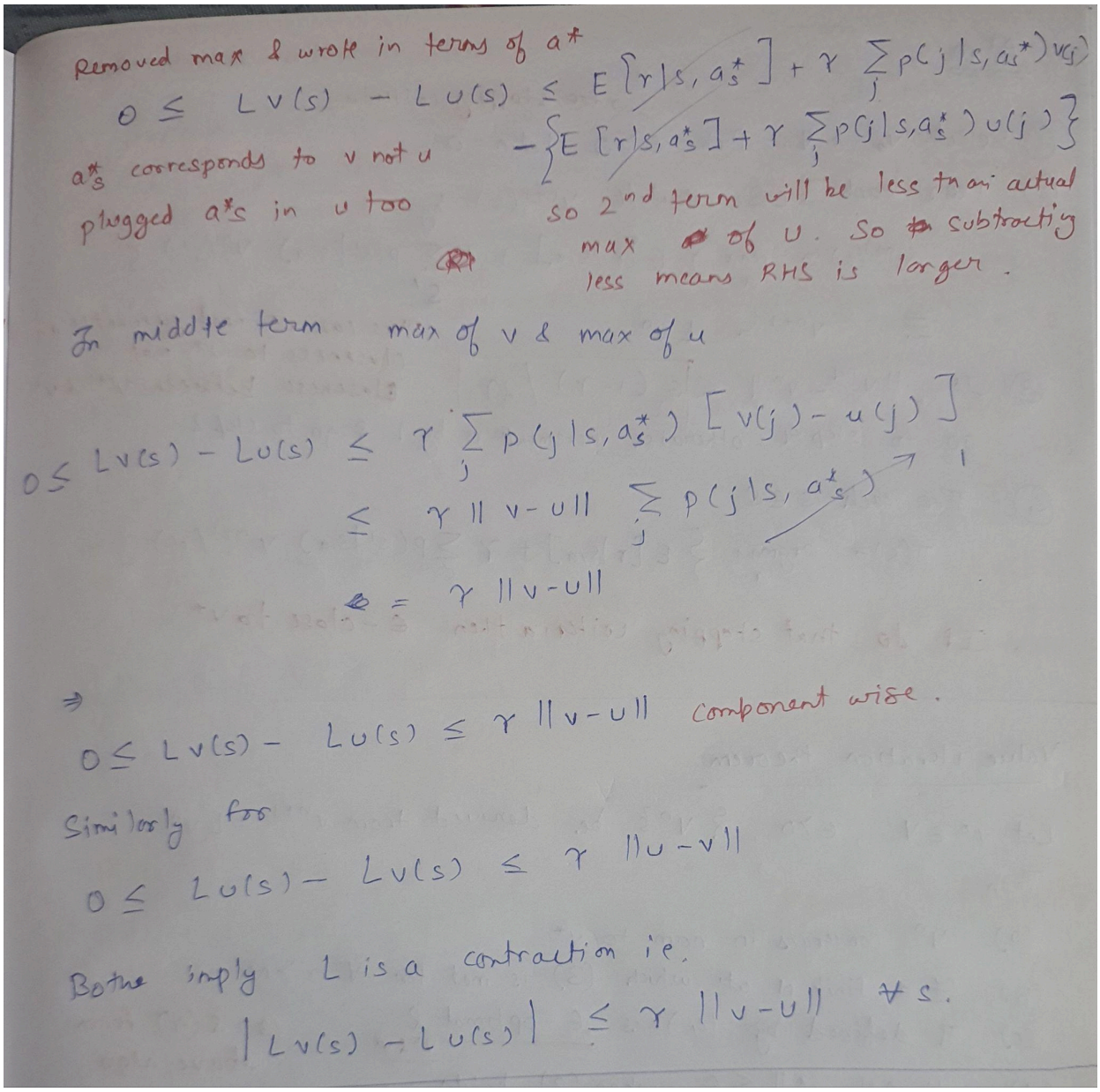
can be many.

one of the optimal action for state s.

Assum $0 \le L v(s) - L u(s)$

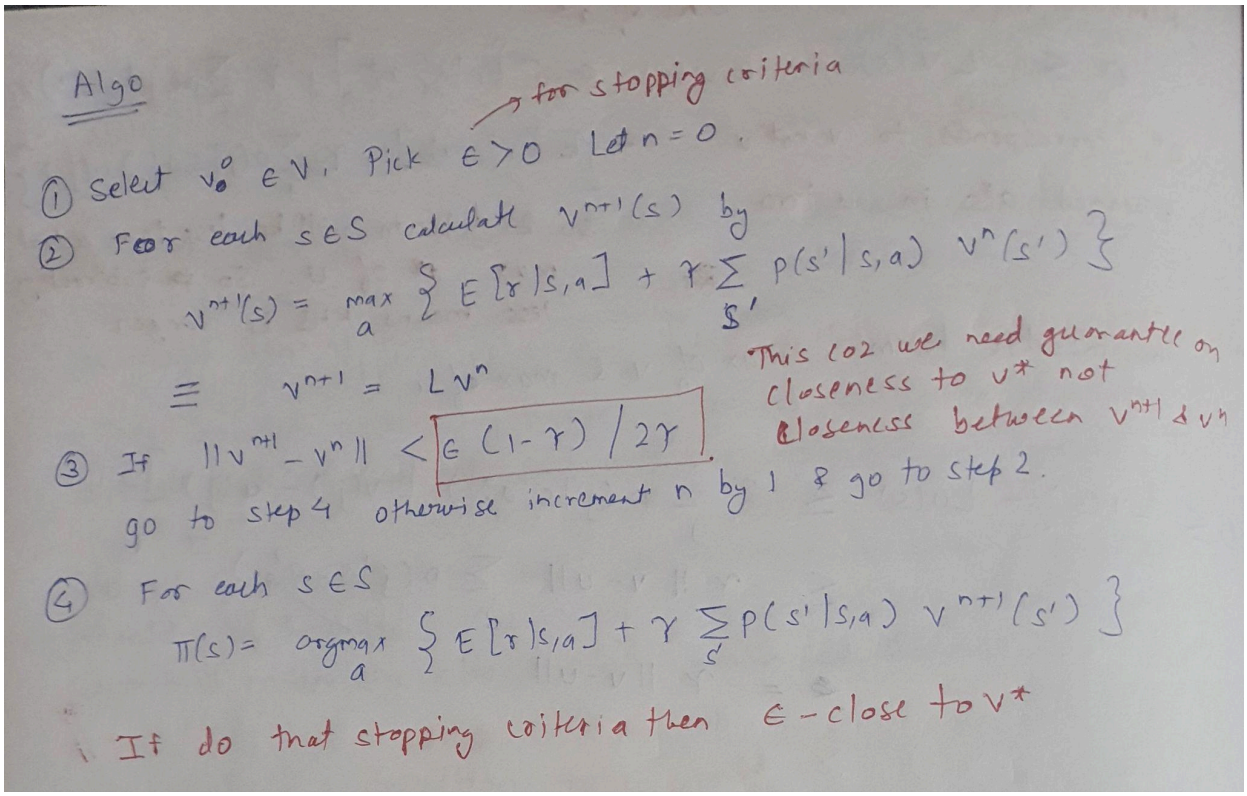
Lv is a function operating on v not v(s)

Lv(s) is one component of that function



Value Iteration

If you want to give certain guarantees about what will hold when things converge, have a bound on value function.
Stopping criteria: successive iterates are very close (same might take forever)



Value iteration theorem

Value iteration theorem

Let $v^0 \in V$, $\epsilon > 0$, $\{v^n\}$ be derived from $v^{n+1} = Lv^n$

Then

(a) v^n converges in norm to v^*

(b) $\exists$ a finite N at which (3) is met for $\forall\, n > N$

(c) π defined by (4) is ϵ-optimal & v^π & v^* — (3) & (4) from above algo

(d) $\|v^{n+1} - v^*\| \leq \frac{\epsilon}{2}$ when (3) holds

$\frac{\epsilon}{2}$ close — Banach theorem gives (a) & (b)

V does not necessarily contain ~~all~~ only value functions. It's a space of functions. — V is space of functions. Not necessarily all points correspond to value functions.

V^π is ϵ-optimal

v^{n+1} is $\frac{\epsilon}{2}$-optimal.

Proof (a) & (b) by Banach fixed point theorem.

(c) Suppose (3) is met for some n & π satisfies (4)

Then $\|v^\pi - v^*\| \le \|v^\pi - v^{n+1}\| + \|v^{n+1} - v^*\|$ — ① Triangle inequality.

$$L_\pi v^{n+1} = E[r|s,\pi(s)] + \gamma \sum_{s'} p(s'|s,\pi(s))\, v^{n+1}(s')$$

marginalised over s'

we chose $\pi(s)$ to be $\arg\max$ action acc to (4) so L_π & L are same.

$$Lv^{n+1} = L_\pi v^{n+1}$$

$\rightarrow v^\pi$ is fixed pt. triangle ineq

$$\|v^\pi - v^{n+1}\| = \|L_\pi v^\pi - v^{n+1}\| \le \|L_\pi v^\pi - Lv^{n+1}\| + \|Lv^{n+1} - v^{n+1}\|$$

1st term of RHS of ①

$$= \|L_\pi v^\pi - L_\pi v^{n+1}\| + \|Lv^{n+1} - Lv^n\|$$

$v^{n+1} = Lv^n$

$Lv^{n+1} = L_\pi v^{n+1}$

$$\|v^\pi - v^{n+1}\| \le \gamma \|v^\pi - v^{n+1}\| + \gamma \|v^{n+1} - v^n\|$$ using contraction.

$$\Rightarrow \|v^\pi - v^{n+1}\| \le \frac{\gamma}{1-\gamma} \|v^{n+1} - v^n\|$$

$\|v^* - v^{n+1}\|$

$\|v^{n+1} - v^*\| \le \sum_{k=0}^{\infty} \|v^{n+k+2} - v^{n+k+1}\|$ Repeated Δ inequality

2nd term of RHS of ①

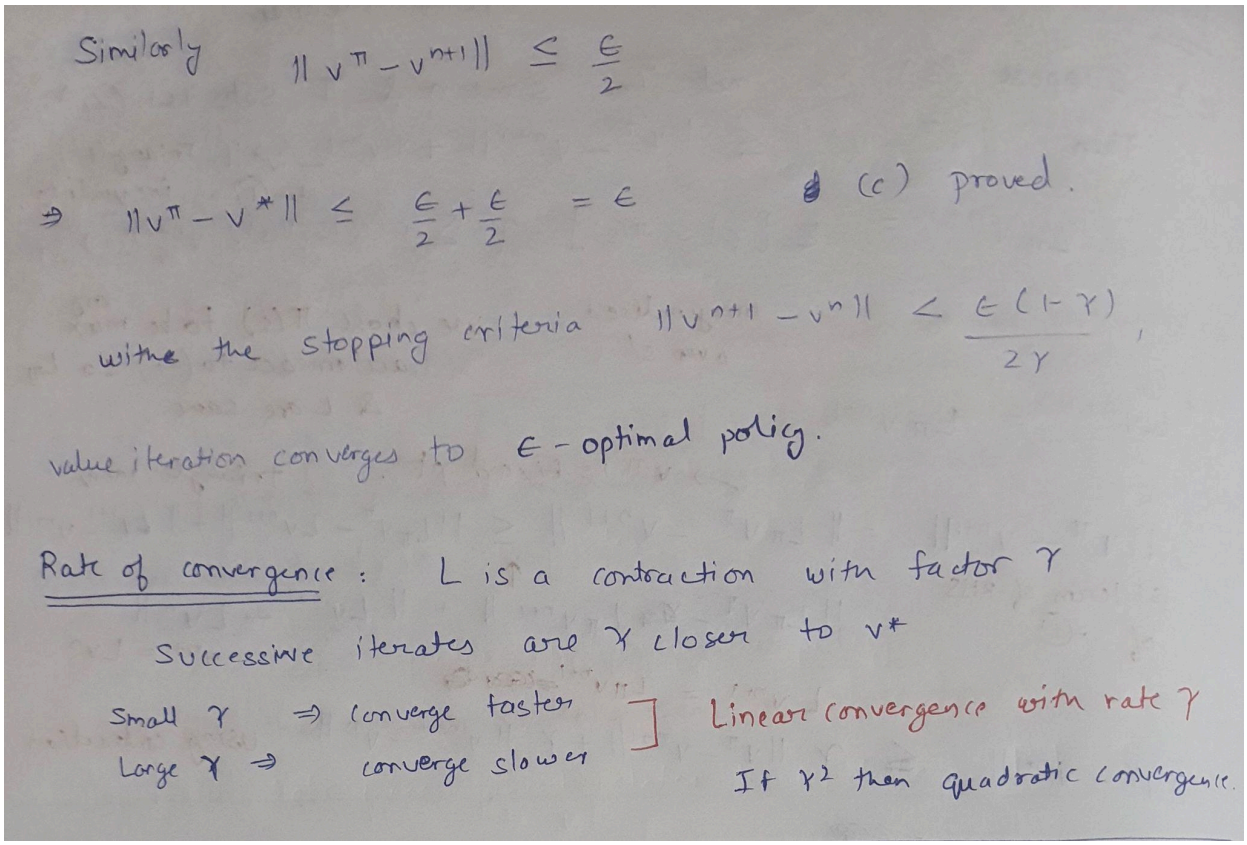
$$= \sum_{k=0}^{\infty} \|L^{k+1} v^{n+1} - L^{k+1} v^n\|$$

$$\le \sum_{k=0}^{\infty} \gamma^{k+1} \|v^{n+1} - v^n\|$$

$$= \frac{\gamma}{1-\gamma} \|v^{n+1} - v^n\|$$

when ③ holds of algo $\|v^{n+1} - v^n\| < \frac{\epsilon(1-\gamma)}{2\gamma}$

$$\Rightarrow \|v^{n+1} - v^*\| \le \frac{\epsilon}{2}$$ (d) proved.



Policy Iteration

Convergence: if stay with same policy then converge
Rate of convergence is much faster in policy iteration compared to value iteration but still VI used by a lot of OR people

Policy iteration

1. Select π_0, set $n=0$
2. Policy evaluation : $V^{\pi_n} = \left(I - \gamma P_{\pi_n}\right)^{-1} \gamma_{\pi_n}$ — Green's can or do iterative like VI using L_π
3. Policy improvement : choose π_{n+1}

$$\pi_{n+1} \in \underset{\pi}{\operatorname{argmax}} \{ \gamma_\pi + \gamma P_\pi V^{\pi_n} \}$$ — component wise argmax

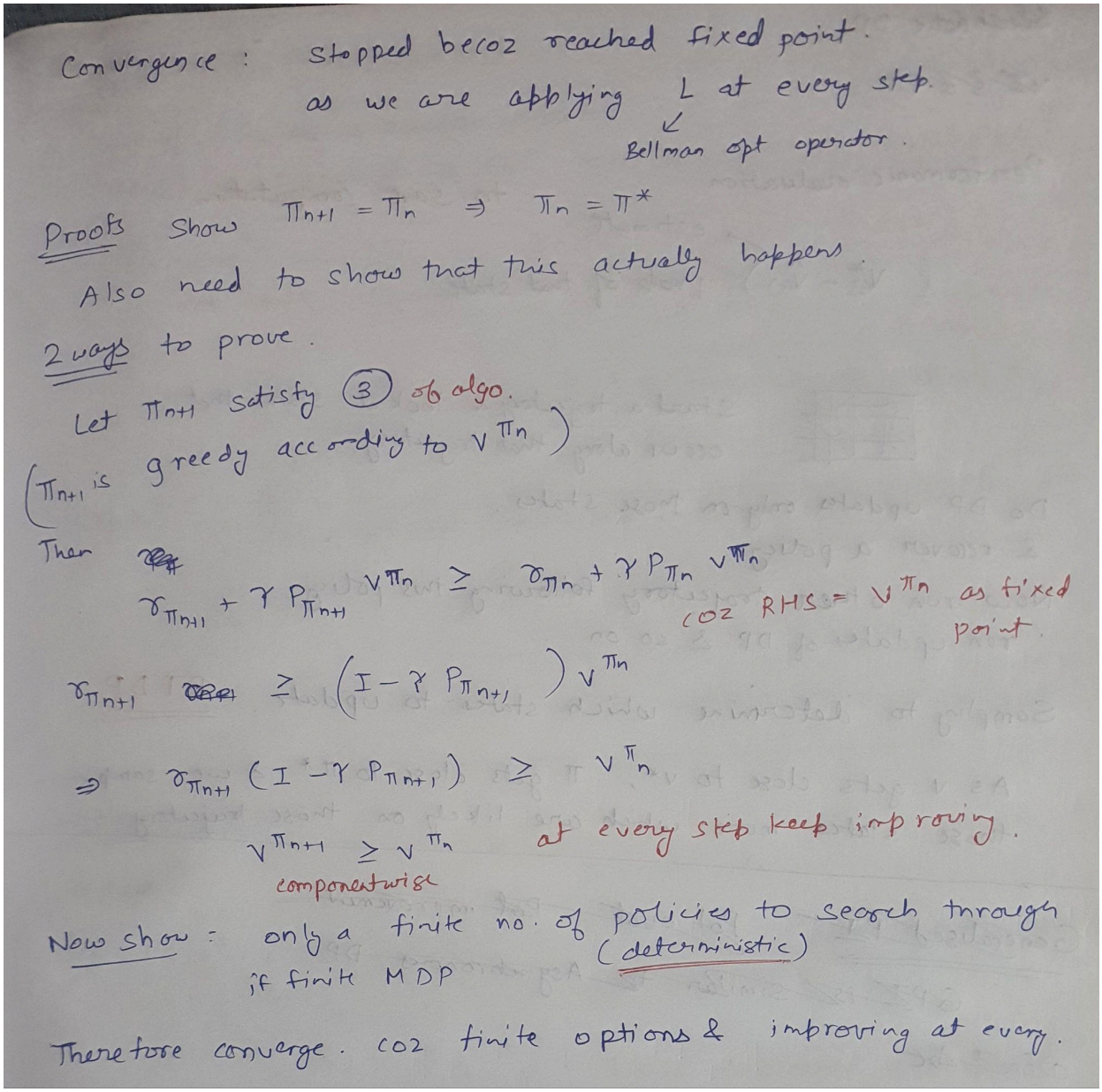
If tie & select one of them & next step also that action is in argmax then choose that action only. otherwise oscillating.

Consistent tie breaker criteria.

4. Stop if $\pi_{n+1} = \pi_n$

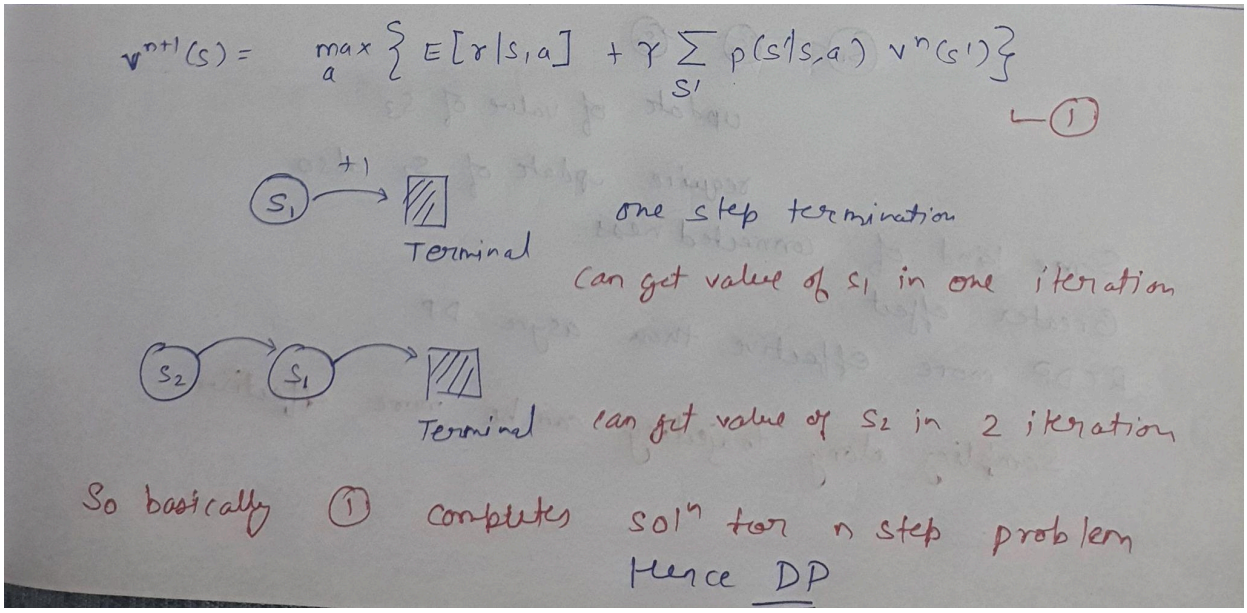
   Declare $\pi_n = \pi^*$

   Otherwise increment n & go to Step 2.



Dynamic Programming

Stochastic DP
VI and PI both are DP approaches



Using V_n to compute V_n+1 in VI: can use just one array instead of two (simply use old values to get new values)

Asynchronous DP:

For just one array ordering over states

When doing for s_1, v_n is used for all states

When doing for s_2, v_n is used for all states except s_1 (v_n+1 is used for that)

When doing for s_3, v_n is used for all states except s_1, s_2 (v_n+1 is used for those)

Further iterates (v_n+1) are getting used instead of v_n

It converges regardless (that too if we do not follow an ordering over states: can randomly pick states too)

For convergence, just need to ensure that you sample every state infinite number of times

All states should be sampled infinitely often

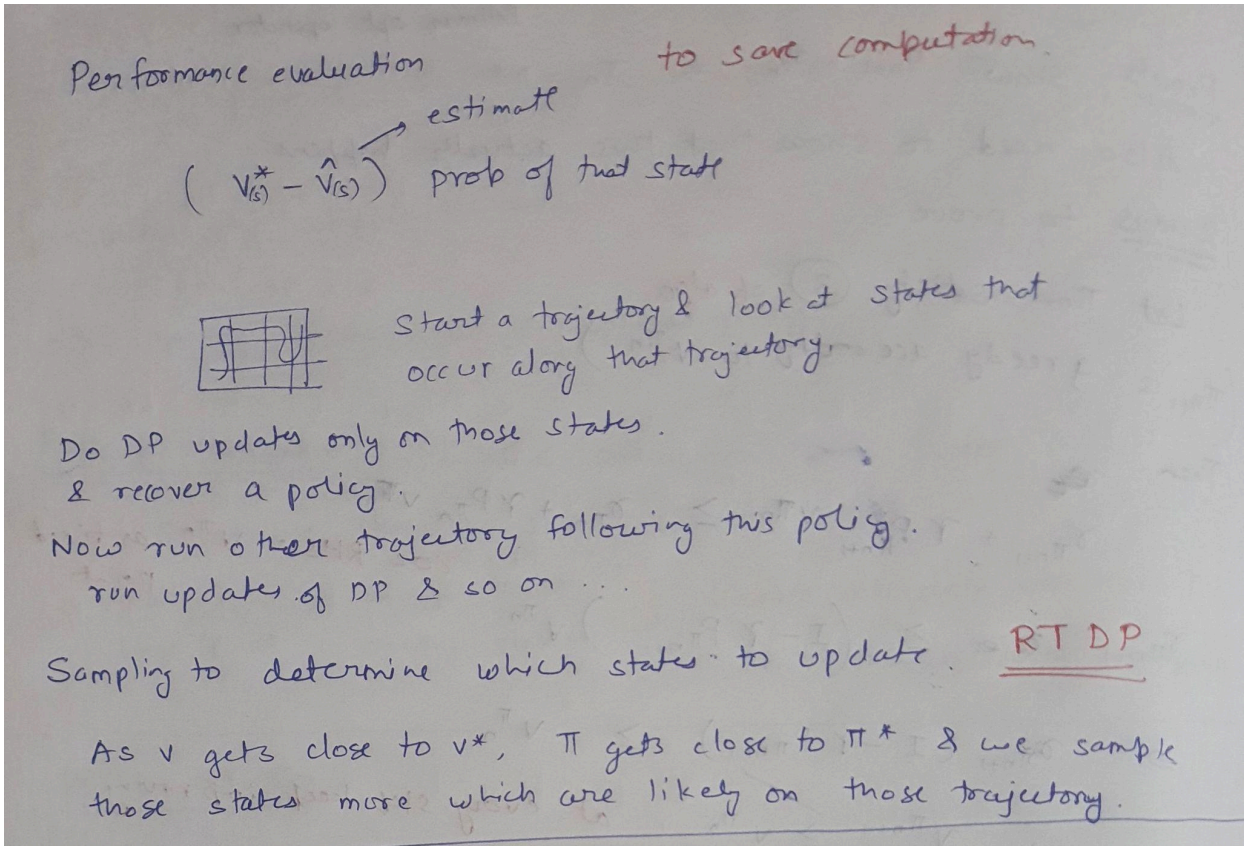
Synchronous DP: use v_n for all states (needs 2 arrays)

Real time DP: almost like RL

Large spaces, methods like VI require visiting all states infinitely often but in real world problems, many states are seldom visited: so can focus on getting to true value function for states that are visited often

Don't mind much about value estimates of states which are visited rarely

How to intelligently use limited computation power: focus on states that we are likely to visit under an optimal policy? Weight by probability of visiting state



Can do epsilon greedy here too: greedy with prob 1 - epsilon

RTDP is very powerful provided we have a model available to focus computation on only a small number of states

Convergence: visit every state infinitely often (infinite computation power)

Sampling in RTDP just to figure out which states to sample more

In full RL. we need sampling not just to figure out states to update but also to determine what is the reward used for updating

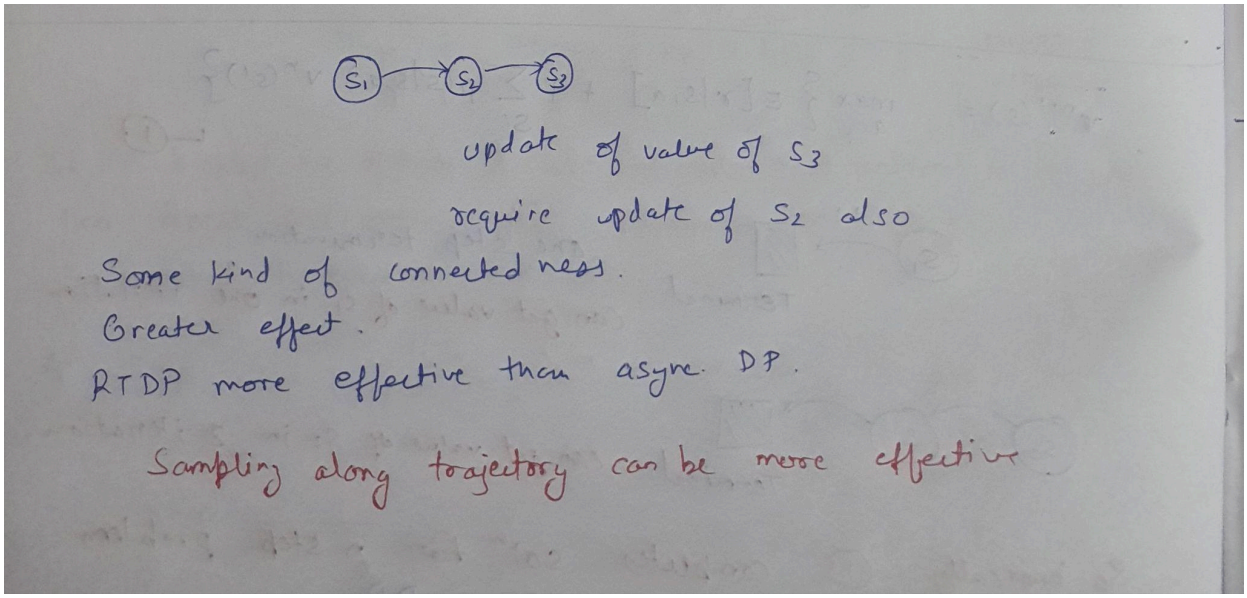
Generalized PI: do not be greedy wrt all states. Just pick some random states and change policy by being greedy on those states only

GPI is similar to Asynchronous DP

Evaluation not needed on all states but the states whose policy gets updated and neighboring states which get affected due to this change in value

Both ways: partial policy evaluation and partial policy improvement

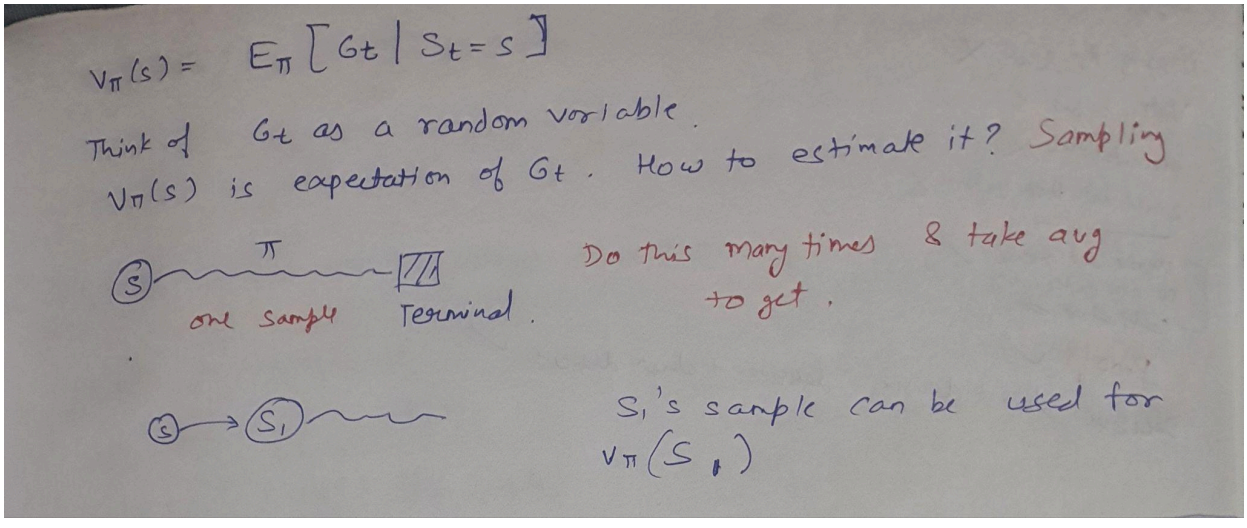
No need to remember states can pick random set of states for both



Convergence: sample infinitely enough (GPI converges)
Extreme case of GPI: sample a state; take one action acc to policy; update the value of state; be greedy wrt this new value and produce a new policy (change action for that state only) **Do this while sweeping through states**
VI can be thought as an extreme form of GPI (eval and improve rolled into one step)
Many full RL algos are some form of GPI
How powerful is the GPI? As soon as we come with a mechanism for policy evaluation, that can be used to solve the MDP **(Many RL algos do this)**

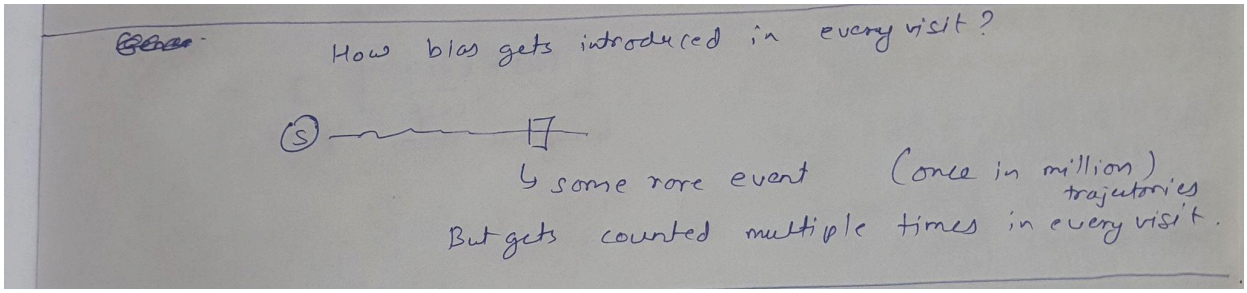
Monte Carlo

In real world transition dynamics are hard to model
When don't have access to MDP



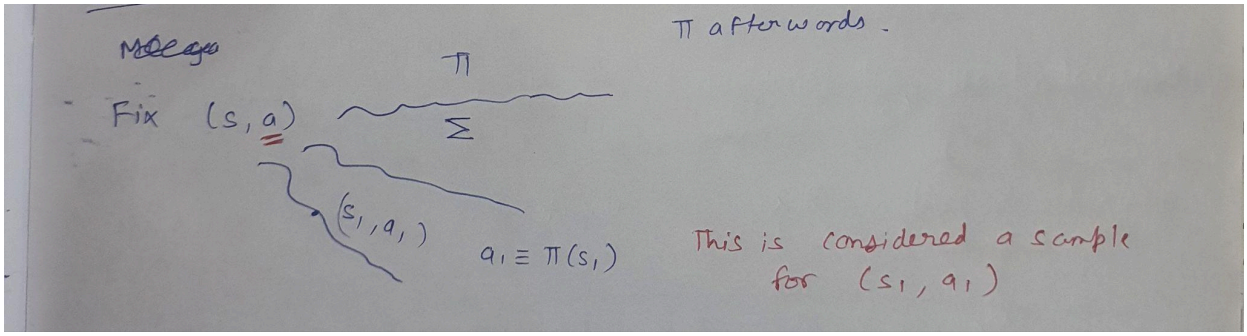
Fix policy
Repeat:

Generate a trajectory; Look at all the occurring states in trajectory; Take sum of rewards; Use that as a sample
Use samples for estimating values
Monte Carlo method for policy evaluation
Do not need model: all is needed is a way of sampling from that system
Can also get by without a real model: through a simulation model
Blackjack: Easier to write program to play it rather than computing probabilities and all
Such models are called simulation models (sample models)
MC methods can be used with sample models
Sample (trajectory) not only determine the states which will be evaluated but also supply the value too
In RTDP, sampling just to pick states and value update according to Bellman equation
Exploring starts:
For deterministic policy, if want value for all states then need to make sure that start from each state
Even with stochastic policy, no guarantee that we will hit every state (unless completely random policy)
Entire state should be reachable for above
What if we start from s and it reappears in trajectory: will it be one trajectory or 2 trajectories? Both
First visit MC uses only first occurrence
Second visit MC
Every visit MC uses all occurrences
In every visit, one sample trajectory but using it multiple times
Issue: using twice changes the distribution (we are kind of sampling from different distribution from the one which we are computing expectation with, hence a little bias can get introduced but in the limit it also converges to same value function as first visit MC)



Control in Monte Carlo

MC gives us a mechanism to perform policy evaluation: no guarantees but **approximation**
Control using GPI
How to be greedy with respect to value function? Bellman eqn requires transition prob
In control use q function not v function



Issue: Draw disproportionate number of actions corresponding to policy pi compared to actions not from pi

So convergence cannot be guaranteed (although in practice it works but no theoretical guarantee)

Way out for ensuring convergence: if want sample for (s_1, a_1) then start from s_1 take a_1 and then follow pi afterwards

So need to do many times for every state-action pair

One way of avoiding exploring starts is to use **eps-soft policy** (mostly greedy but some prob to other actions too; highest prob still to optimal action)

Every action has at least an eps prob of being taken from every state

Uniform random policy with n actions is (1/n)-soft policy

Eps-greedy is (eps/n)-soft policy

Being eps-soft wrt prev value converges to an optimal eps-soft policy (optimal in the sense that highest prob to optimal action)

Importance sampling:

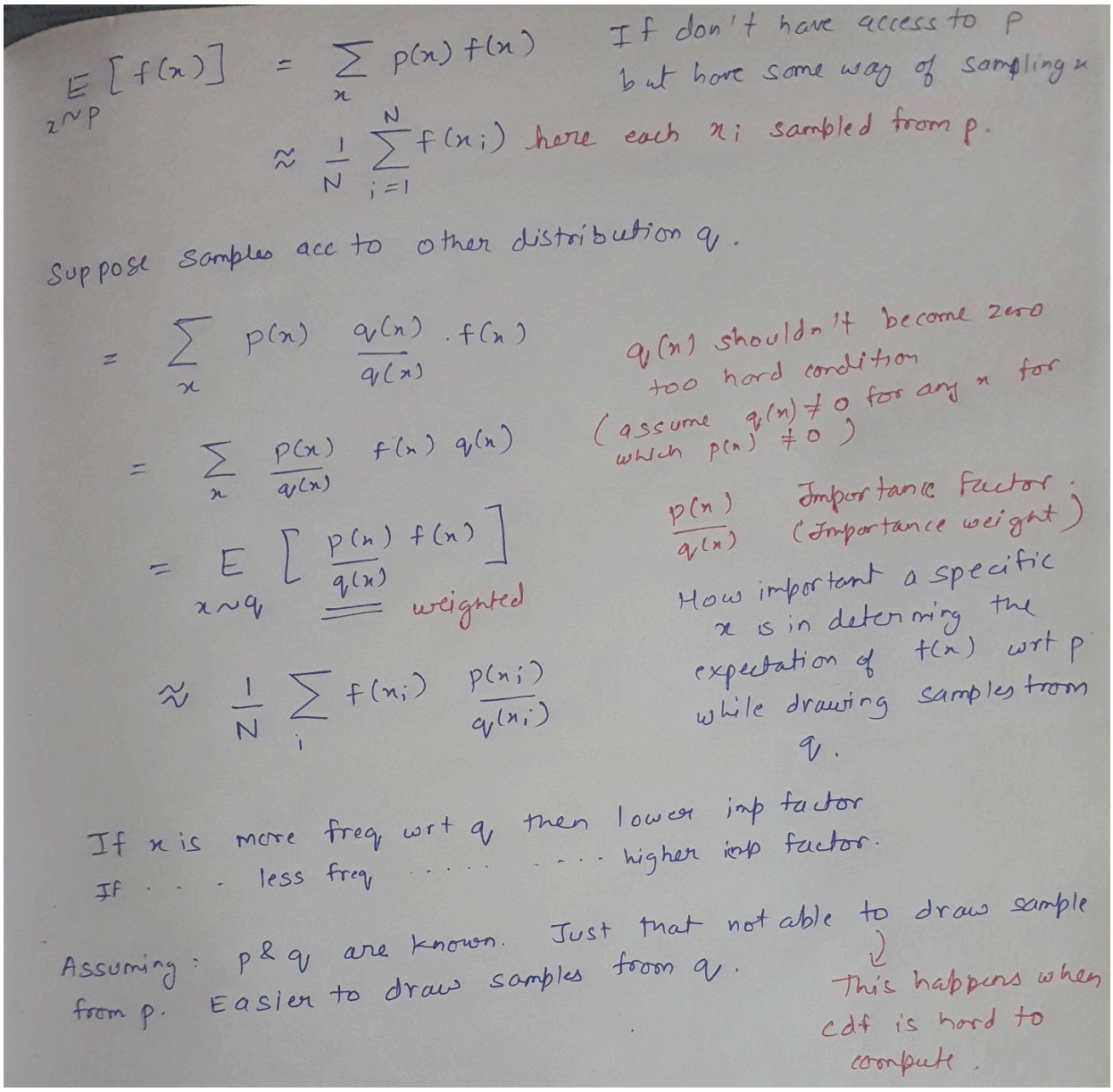
Exploring starts needed for deterministic policy

Ideally we want to behave any way we want in the state space but it should give the value of any policy we want

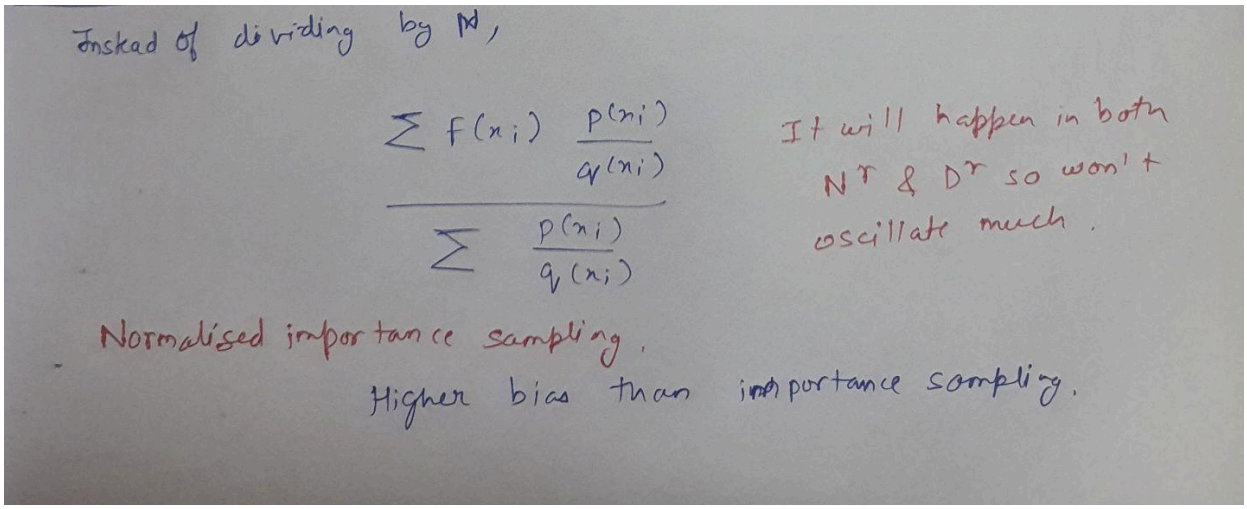
Can we decouple policy used for behaving and policy that is to be evaluated: **off-policy methods**

Till now on-policy methods (samples generated from policy that we are evaluating)

In off-policy, samples generated by a policy different from the policy that we are evaluating



Can use random behavior policy to generate trajectories
Easy way to ensure q(x) is non-zero is eps-soft
Softmax is another alternative
Importance sampling might suffer from numerical instabilities (due to cases where p(x) is non-zero but q(x) is close to zero, importance weights become very very large resulting in oscillations)
Way out: **Normalized importance sampling (weighted importance sampling)**

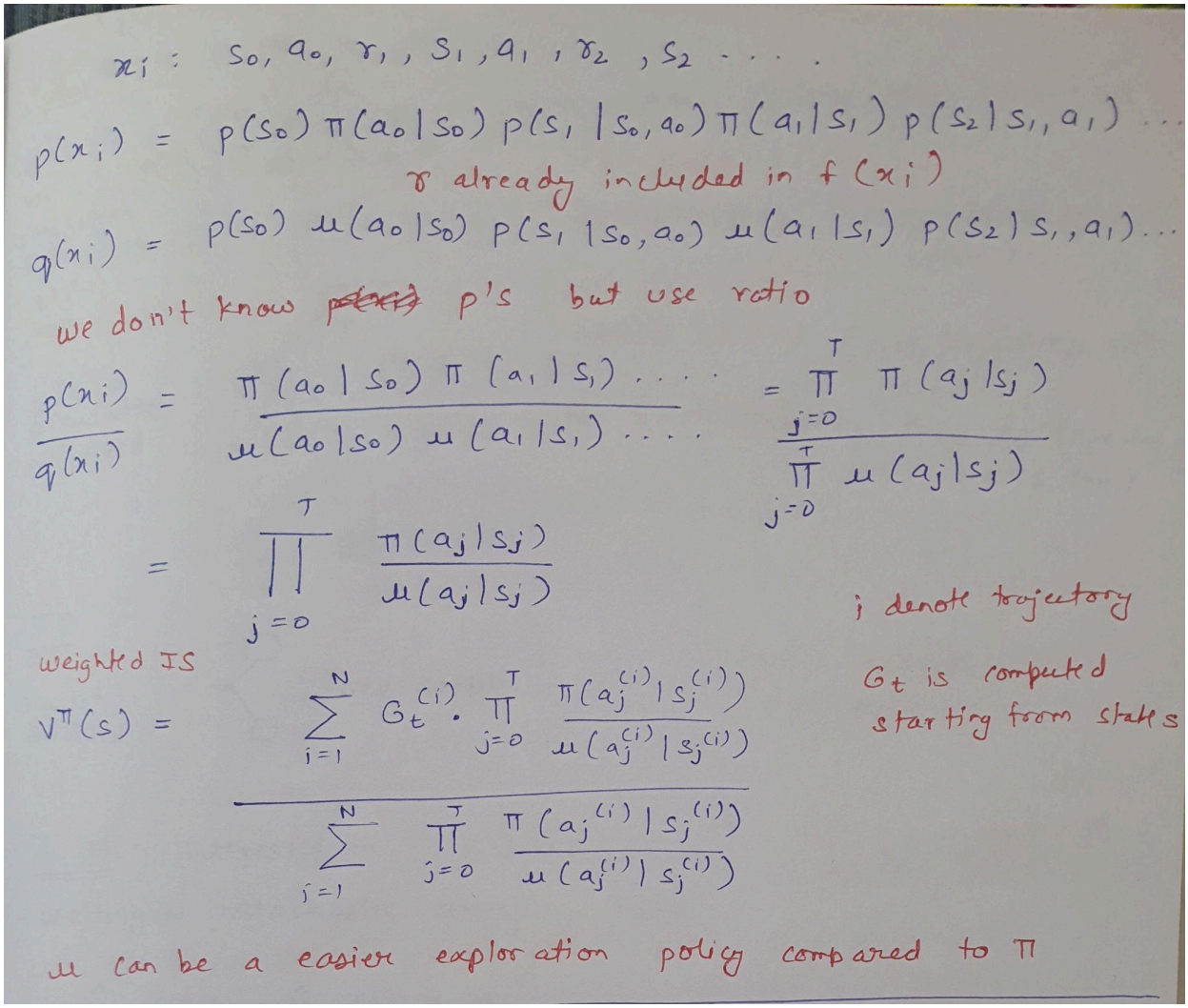


Importance sampling higher variance
Normalized importance sampling higher bias
Normalized don't fluctuate much so better in practice
In limit, both converge to same answer

Week 6

Off Policy MC

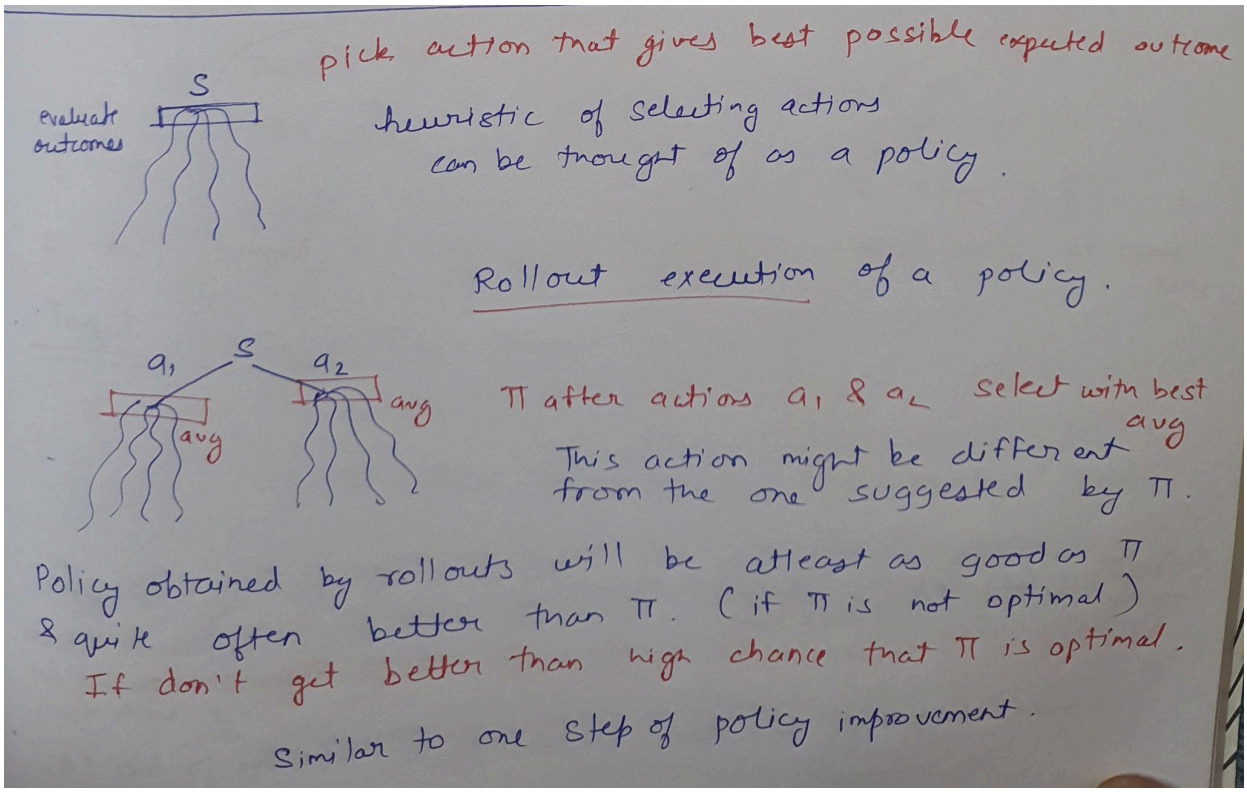
f(x_i): return
p(x_i): prob of trajectory generated by policy to be evaluated **Estimation policy** (Pi)
q(x_i): prob of trajectory generated by policy that i am actually following **Behavior policy** (Mu)



Incremental method: No need to wait till end of trajectory for computation of importance weight but keep updating as we go along
Implementation in incremental manner
All this for evaluation
For control, we need action values (read **off policy MC Control** from book)

UCT

Rollouts are general. Can be used in games, adversarial settings etc. Predate value function, return etc.
If system is deterministic, one trajectory is enough
If policy is stochastic or system is stochastic, do many times
If variance in samples is very small, it's fine to run small number of times



Longer the trajectory, more variance so need to generate more trajectories (**based on computational power how much you can afford**)

Don't go all the way till end, stop somewhere in between

3 factors: variability in system, computation power, time to spare

If have a heuristic to tell how much it is going to cost if go till end then can use it as terminal cost

Value function is a good enough heuristic of terminal costs

Planning: when some model of environment is available even if it is a simulation model and draw sample from it

Learning: when sample are drawn from a real system

MCTS: simulation model needed to do rollouts

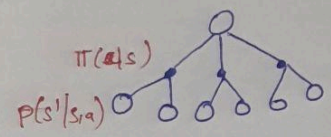
```
search (state, depth):
        if (terminal state) then return 0 # more like terminal reward, here 0
        if leaf (state, depth) then return eval(state) # leaf depending on our search depth
        action = selectAction(state, depth) # action acc to heuristic
        next state, reward = simulate (state, action) # basically passing rewards back up (recursion)
        g = reward + gamma * search (next state, depth+1) # eventually this will have adjusted return (as terminated at max depth using value fn)
        updateValue(state, action, g, depth) # depth needed coz no Markovian assumption (finite horizon effects, if Markov then depth not needed here)
        return g
```

Initial call with depth 0
Can call multiple times for multiple trajectories
Can retrieve policy from value function
Can modify algo to use q function instead of value function
UCT: rollouts + value function + UCB 1
It is one of the most powerful planning algorithm
Most popular MCTS algo
selectAction uses a UCB like formulation: expected reward + confidence bound term
c_p * sqrt((ln t)/n) # here t is depth
C_p depends on problem
Larger variability larger c_p (coz want to explore more)
UCB coz want to explore more promising actions more (instead of exploring uniformly)
Bandit solution for solving full RL problem
If **simulation model** can do UCT like planning
UCT is very robust planning algo
Can have parameterized value function too: then update those parameters
Long trajectory and small gamma then contribution in return dies down very quickly: so instead of maintaining full product (of importance weights) can do truncated

TD(0)

Soul of RL
Difference in predictions over time that's why TD

DP



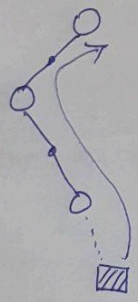
No simulation

• → action Stochastic transitions

Full backups

Bootstrapping (using values of next state)

MC

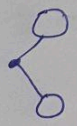


Simulation involved

Sample backups (one trajectory not all)

No bootstrapping (can do like in MCTS tho)

TD



Simulation involved

one sample

Sample backups

Bootstrap.

In MC interested in $E[G_t \mid S_t = s]$ had some sample not expected value.

writing as stochastic averaging method.

$$\hat{V}_{new}(s) = \hat{V}_{old}(s) + \alpha \left[G_t - \hat{V}_{old}(s) \right]$$

classical stochastic averaging rule.

This is for MC

$\hat{V}_{old}(s)$ → "current"

G_t → "Target"

$\hat{V}_{old}(s)$ → "current"

"error" TD error (δ)

In MC, target is one entire trajectory which acts as sample here.

Target

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

$$= R_{t+1} + \gamma G_{t+1}$$

$$\approx R_{t+1} + \gamma \hat{v}(S_{t+1})$$ Bootstrapping

$$\hat{V}_{new}(S_t) = \hat{V}_{old}(S_t) + \alpha \left[R_{t+1} + \gamma \hat{V}_{old}(S_{t+1}) - \hat{V}_{old}(S_t) \right]$$ TD(0)

new estimate of $\hat{v}$ for state encountered at time t

$\hat{V}_{old}(S_{t+1})$ → next state's old estimate.

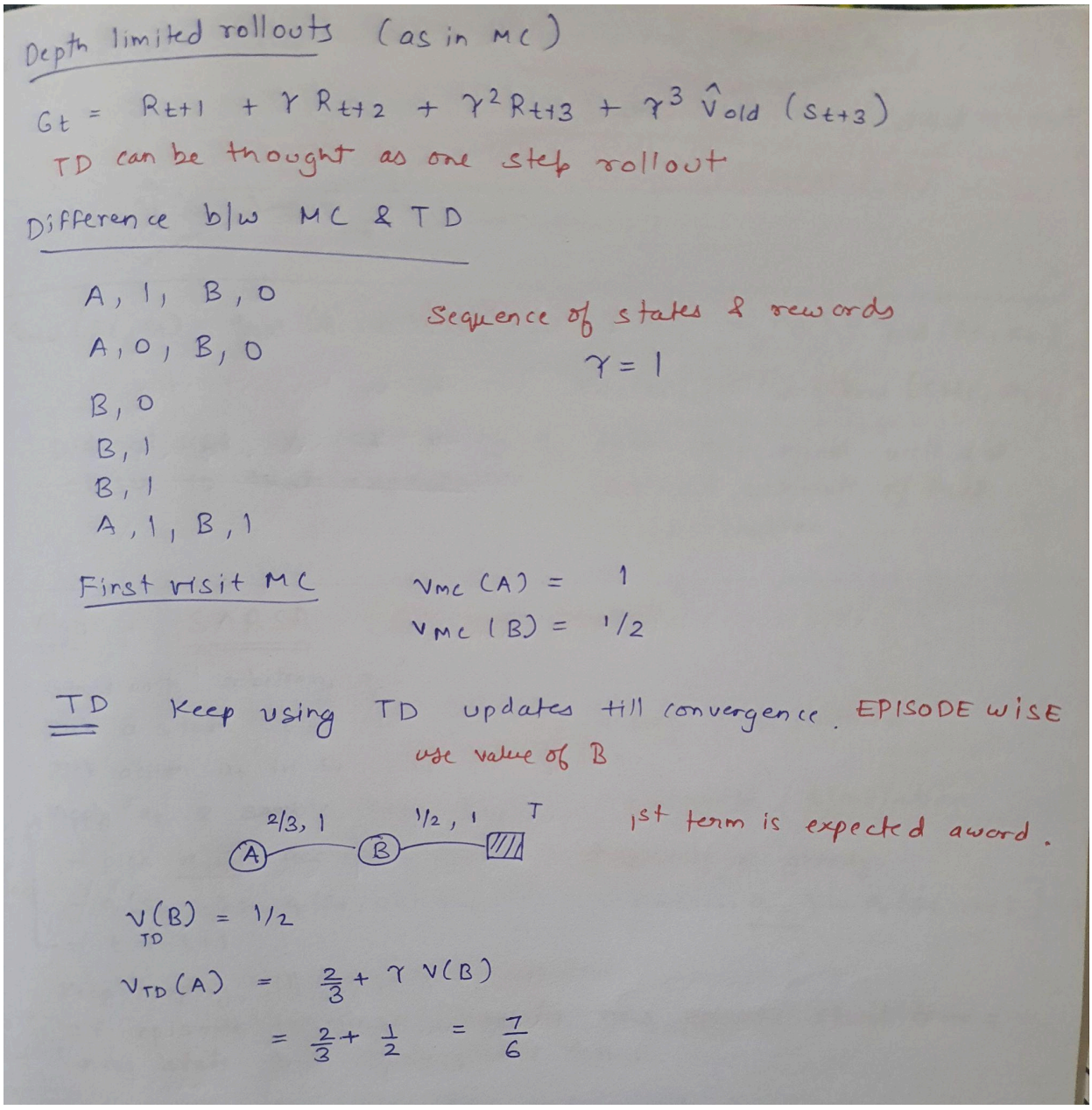
At end of episode, reset state but not value function.

Optimistic initial value

Exploration on top of base policy: **do not make TD updates corresponding to exploratory actions**

TD(lambda) family of algorithms

s_t is time indexed state not necessarily indexing based on number of states



Although TD rule is applied incrementally, it computes value function by **implicitly constructing a Markov process**

TD converges to value function supplied by above MDP

TD and MC gave different answers

B has been much rewarding (2/3 times gave 1) when alone but if A occurs before it rewards less (2/3 times gave 0)

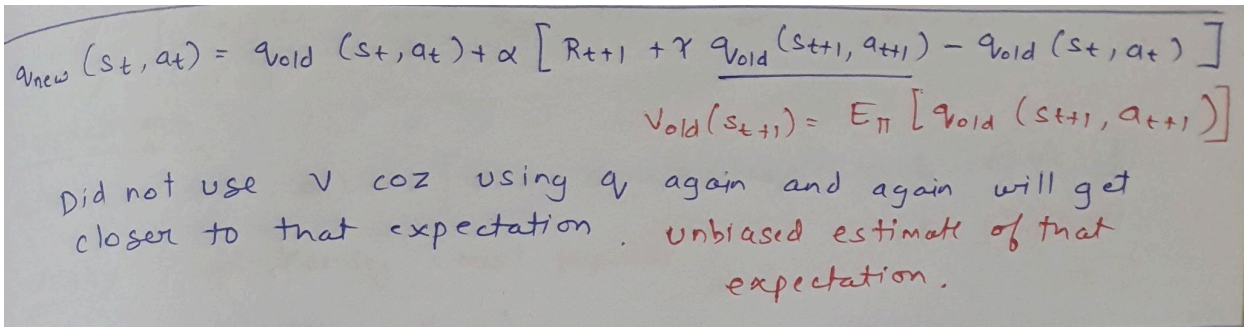
TD methods implicitly assume whole system is Markov
TD does not take into account the structure just use Markov property (**strong assumption**)
Which is better? **Depends on problem**
MC methods try to minimize error (squared) wrt fixed set of samples drawn
TD methods form **certainty equivalence model** (it best explains the data seen so far under certain assumptions) and make predictions according to it
TD and MC are optimizing different objective functions

1. Given an **infinite amount of data drawn from a truly Markov system**, both converge to same answer
2. Given a **finite amount of data**, MC converges to minimizer of squared data and TD converges to whatever Markov model you can form with that limited amount of data (need not be the right model)
   In finite data regime, which is better depends on application
3. If the **system is not very Markov**, prefer MC methods as they don't make any assumptions. Don't use TD methods in this case.

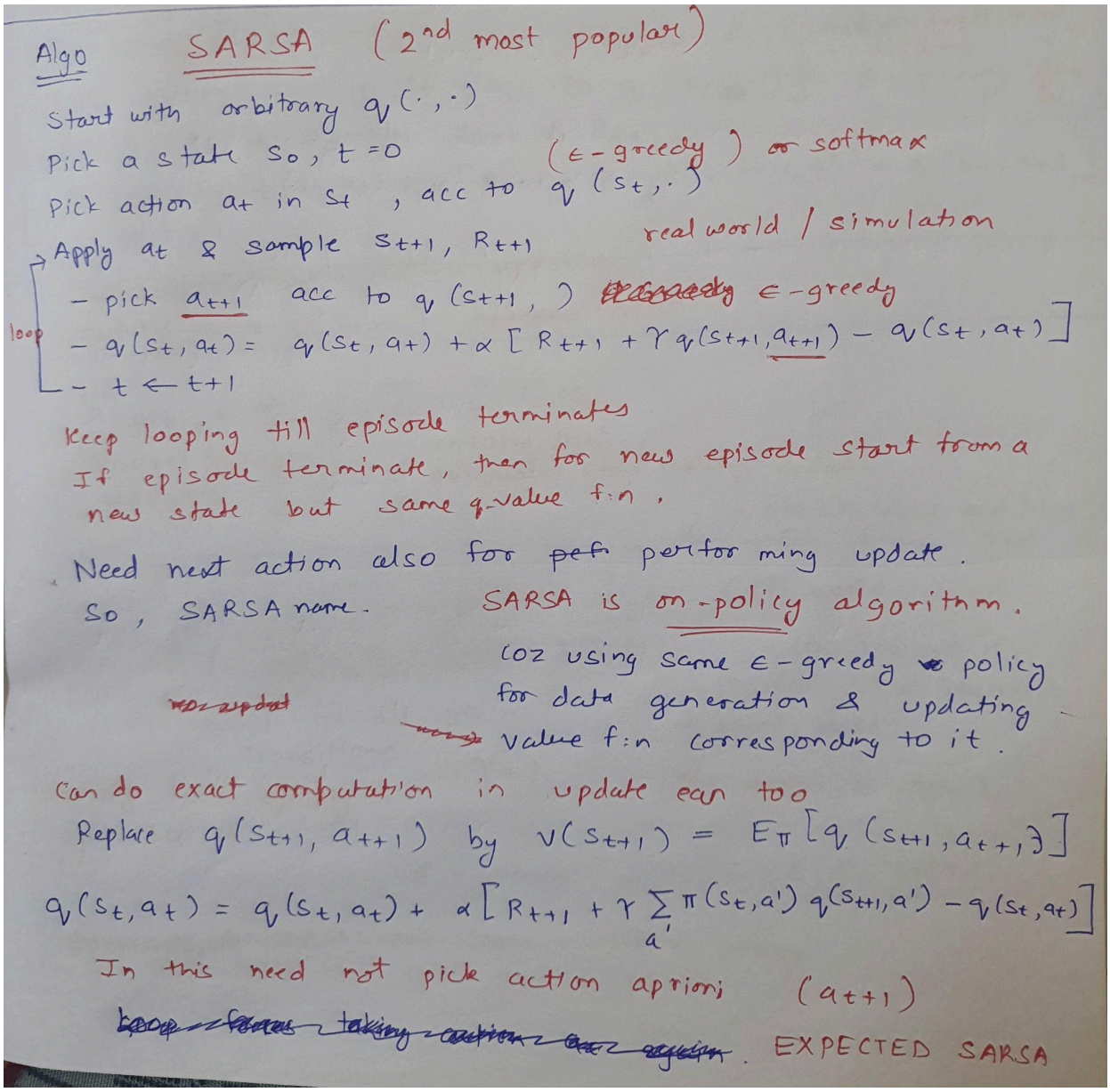
But remember that if you are using the notion of value function then already assuming Markovian otherwise would have used histories but that's ok. Further rewards will take care of non-Markovian nature.

TD(0) Control

TD(0) does evaluation using sample backup and bootstrap
For control, use q function instead of v function



One way of control: Run few iterations of above eqn, learn q-fn, be greedy with it and update policy and repeat
Another way: Instead of many iteration, just one iteration and update (similar to extreme GPI)
Alternating between being greedy and updating



No explicit evaluation stage no explicit greedification stage

For convergence, decrease alpha and epsilon over time so that eventually we stop updating but that eventuality should happen long long time away in future (Explore a lot so that each pair is sampled enough)

Alpha should decay at a slower rate than epsilon coz if alpha goes to zero then no point of sampling

In SARSA, loop forces taking same action a_t again (to ensure on-policy)

In expected SARSA, can take different action (so it's kind of off-policy: gray region between on and off)

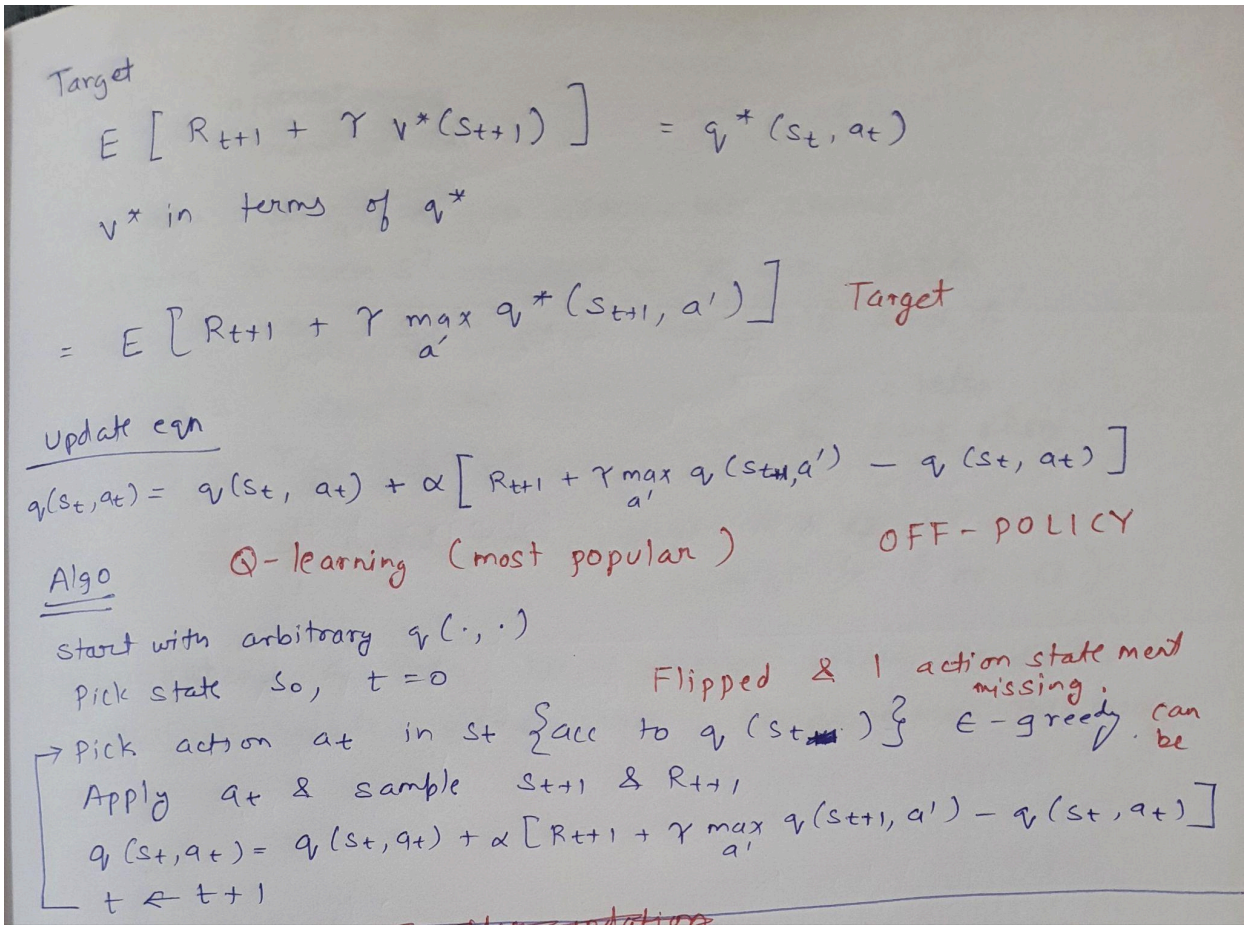
No explicit repr of policy in SARSA: implicitly defined by q function

When q function changes, policy automatically changes

Bayesian exploration is another technique of exploration apart from eps-greedy and softmax

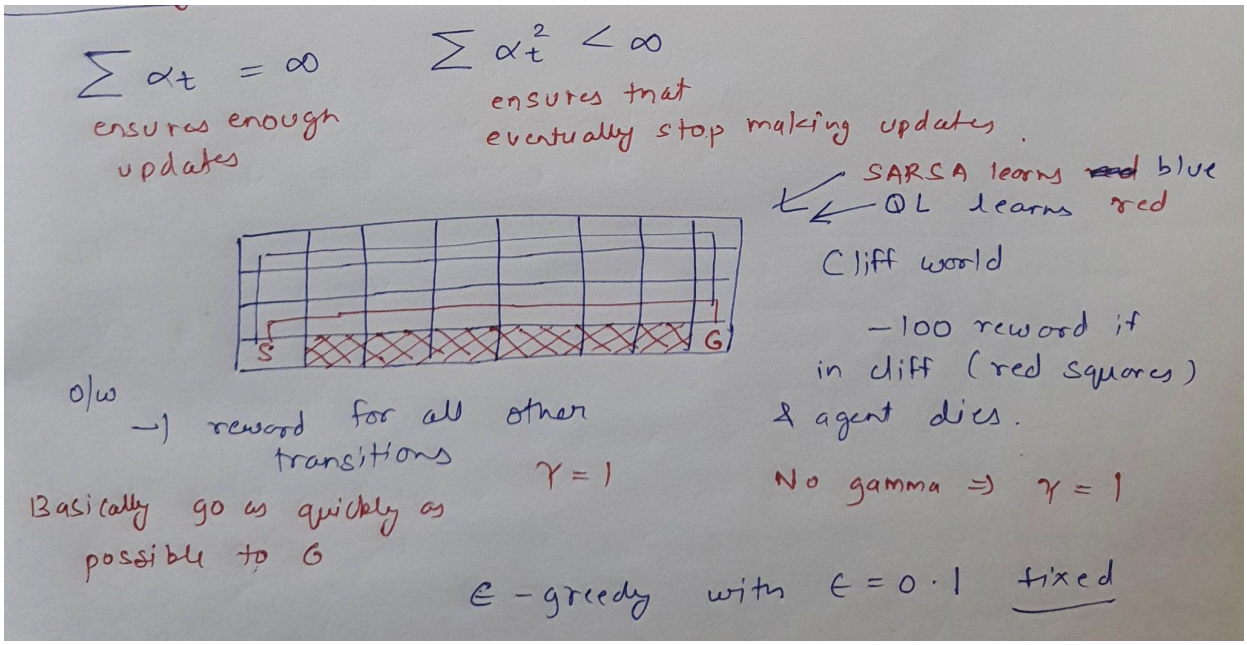
Q-Learning

Another way of control using value iteration kind of approach



Just like Expected SARSA, no need to pick action beforehand
If initialize q function as all zero that means uniform random policy
In SARSA, we pick action acc to q (s_t+1, .) before making update
In QL, we pick action acc to q (s_t, .) after making update
Basically, q_old getting used in SARSA and q_new in QL
If cool alpha and epsilon properly, SARSA converges to optimal policy
If cool alpha properly, QL converges to optimal policy
QL is off-policy
In QL not even required to have s_t+1 (if simulation model can arbitrarily reset to new state)
This is similar to proper asynchronous GPI
But more meaningful to update along trajectories as values are more likely to have changed along those trajectories (similar to RTDP)
QL need not be eps-greedy if sample action arbitrarily. No need to follow any rule. Can do it randomly.
Eps-greedy helps in quickly converging to most likely trajectory
So some merit in using eps-greedy for QL too

SARSA: can continue to explore and while exploring you learn something that is taking into account the exploration
QL: can behave arbitrarily with QL but if choose to behave according to q function that is being learned it does not take into account the fact that you are doing exploration

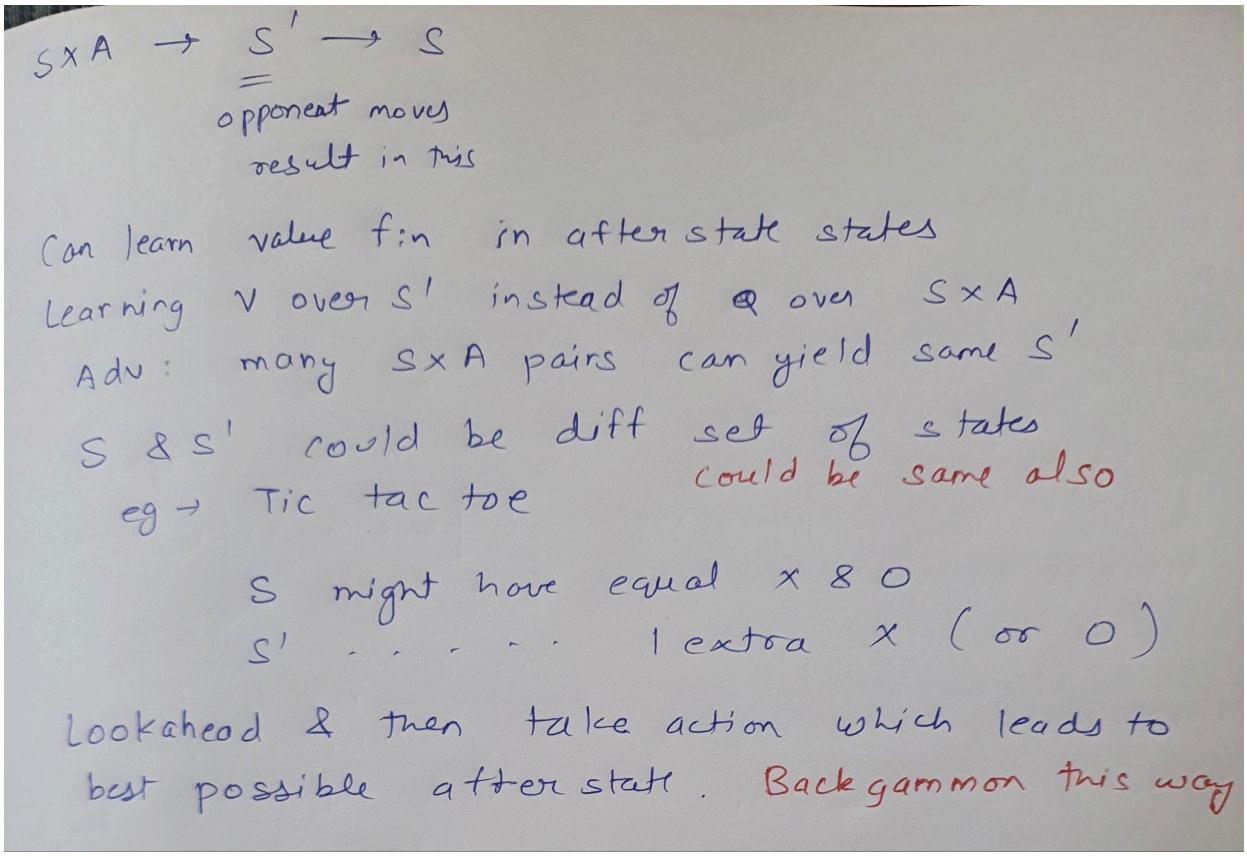


QL will find optimal policy but not SARSA as epsilon fixed at 0.1
SARSA will learn safer policy as it will need 2 exploratory moves to fall off the cliff
To avoid falling it will learn to stay as far as possible
SARSA takes into account the fact that we are doing exploration while QL might learn the optimal policy it may end up killing you more often while learning the optimal policy
If continue to explore after finishing learning, better to use SARSA than QL
QL assumes that eventually you will execute only the optimal policy
Reason for not executing the optimal policy greedily forever even after sufficiently learning: **non-stationarity**
SARSA preferable in non-stationary environments
If keep following eps-greedy with 0.1 then expected reward in case of QL will be far less (compared to what agent might believe it should be getting as agent is ignoring exploration) as we might keep falling off the cliff
QL can explore much more than SARSA while learning as can choose agent to behave completely at random and still learn the optimal policy
QL can be used in large state spaces as it is off-policy and we can behave arbitrarily
In general, QL converges faster than SARSA
Hybrid algos: 2 steps acc to current policy (on-policy) and then max like QL (off-policy)
Here, using more of the actual samples (2 steps) & less bootstrapping (due to discounting by gamma)
Sometimes hybrid give better results (like 3 step and then q)
This is tied to the stochasticity of the system and how far ahead is the system predictable (in above example system was predictable up to 3 steps)

Not easy to figure out what should be lookahead

Afterstate

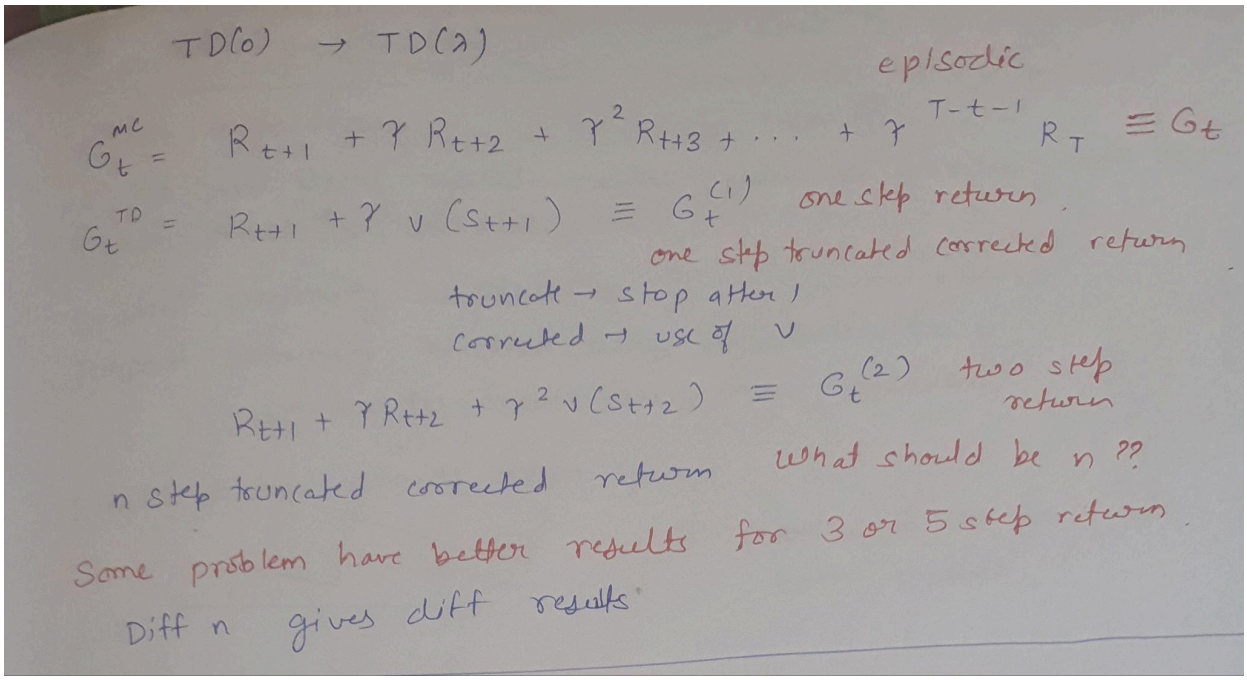
Whole stochasticity in environment is introduced through moves by nature/opponent
Effects of our actions on system are deterministic
Deterministic component then stochastic component



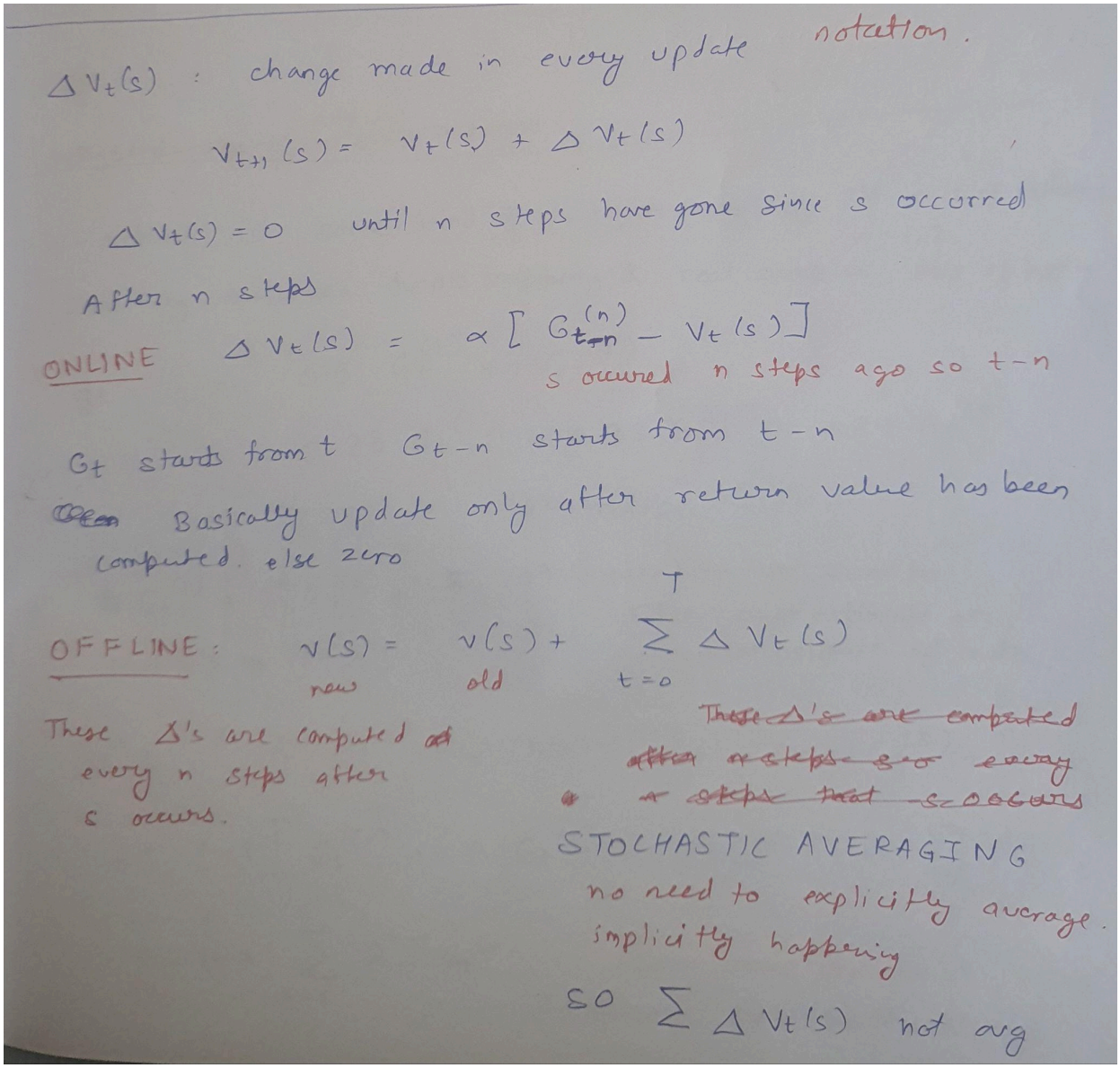
Week 7

Eligibility Traces

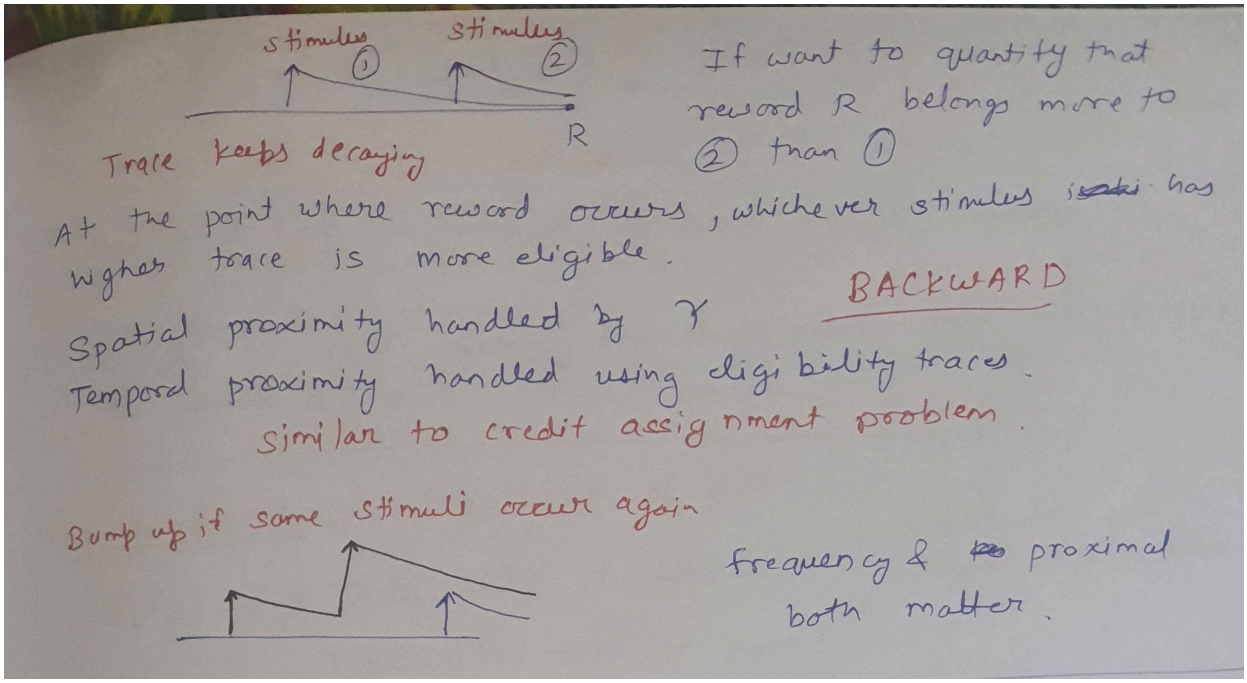
QL, SARSA, UCT very popular
Sample complexity: how many samples are needed to learn
Algos reducing sample complexity
Speed up convergence of TD algos using eligibility traces
TD(lambda)



Online: Updates at end of n steps as soon as you can compute G_t
Offline: Update at end of episode only regardless of when you compute G_t (if visit the same state many times then use avg)
If never visit any state more than once: online and offline are same
If visit more than once then will be using updated value function every time for bootstrapping in online
In offline same value function is used in bootstrapping step
In online there can be many updates for a state while in online at max one update for each state
Sometimes online works better sometimes offline
If trajectories too long then waiting long time in offline
Online updates work better in control
Bias variance tradeoff
Online: variance is high
Batch mode update: not at end of each episode but end of each batch (batch might have many episodes)
If trajectories are short batch mode is better in terms of variance reduction
Online: 3 and 5 step gave least error (for some example)
Offline: 6 and 8 step gave least error (for some example)
This example of 19 states with starting in mid and 9 states on both sides and +1 for reaching one end and -1 for other (uniformly random walk)
Trajectory is expected to be much larger than 9 (as some rights might cancel out left) but above it gave 3,5,6,8 etc. as best n
What is the best value of n?



Concept of eligibility traces predates RL
It came from behavioral psychology
What stimuli is causing a particular change in behavior?
This stimuli is eligible for any changes in behavior
Things more proximal to the outcome thought to be more eligible for getting payoff from the outcome
Frequency also matters
Trace mechanism



Eligibility traces for modeling temporal proximity

Whenever a reward occurs not just update the immediately previous state but also states before that

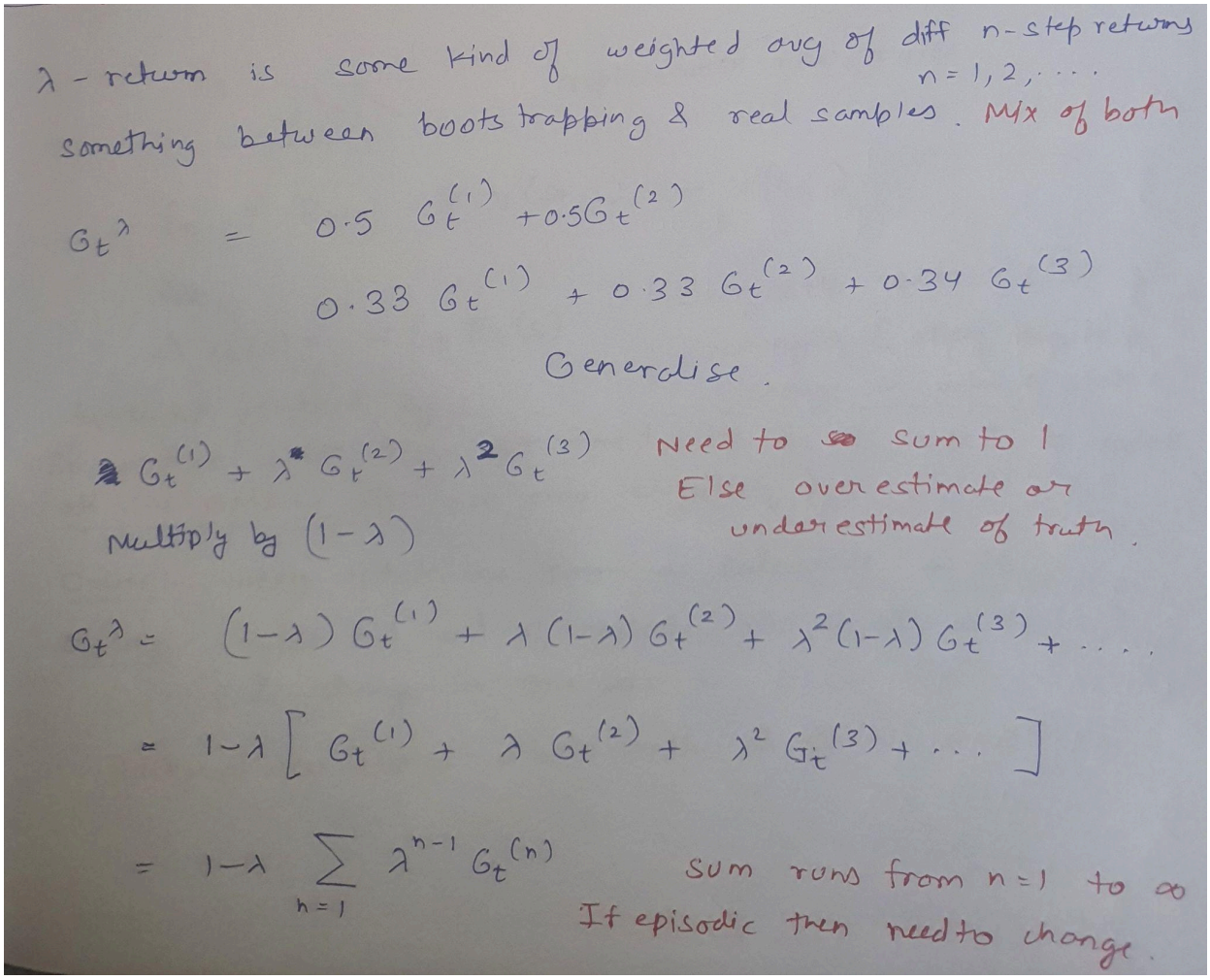
Probably most immediate one is most responsible but not completely responsible

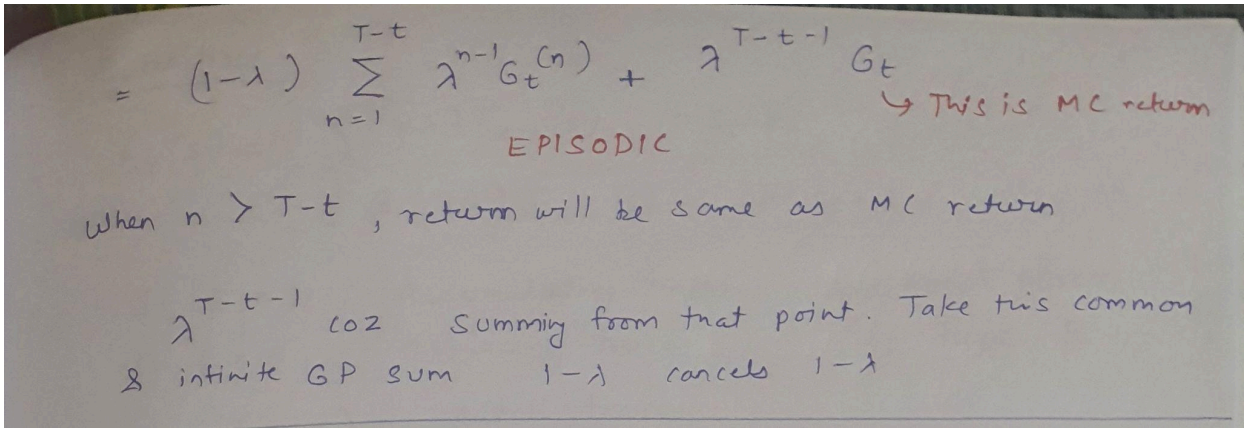
Backward view of TD(lambda): when reward happens, lookback at states (or have computed TD error and now deciding what all visited states should be updated using this TD error)

Forward view of TD(lambda): suppose at s then what should be the notion of return that should be used for updating value of s

Backward view came first but forward has nice math so it is presented first in book

Forward view might look arbitrary but it is done to match backward view

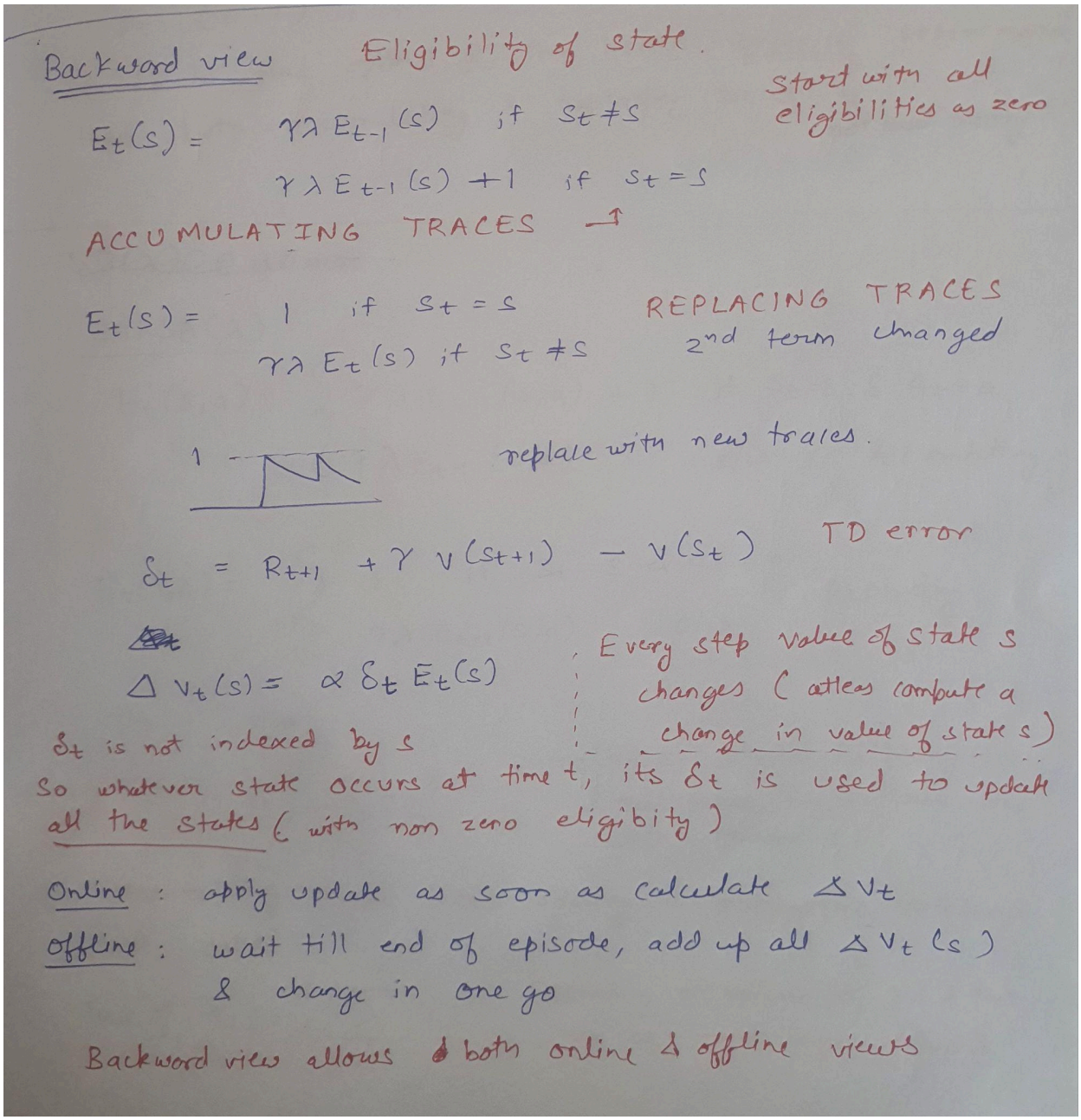




No question of online here as either way have to wait till end of episode to calculate lambda-return so only **offline** in forward view
Actually online offline does not make sense in forward view
Backward view allows us to implement TD(lambda) in **online** manner
Take graph from book

Backward View of Eligibility Traces

When a reward occurs, update states based on their eligibility



In TD(0) update value of just the current state

In TD(lambda) update eligibility trace for all states and then update value function of all eligible states (online) in offline accumulate updates till end of episode and then make it once

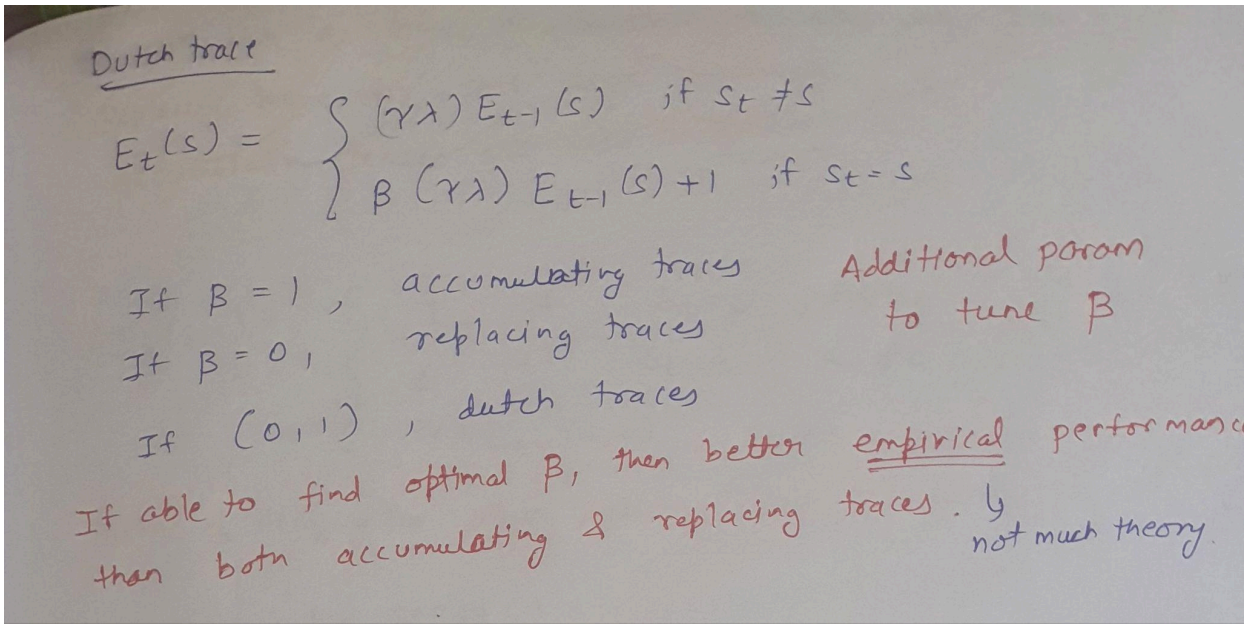
Accumulating traces might have to be bounded as they can keep increasing. If make sure that eligibility decays sufficiently before returning in expectation to the state then no need to bound. Can use small lambda too.

Makes sense to use lambda = 0 to avoid huge computation in very large spaces where need to update every state in every step

Need mechanism to track non zero eligibility traces: update only those
If below certain threshold, can make eligibility zero

1. lambda = 0: TD(0)
2. lambda = 1: MC

TD(lambda) allows incremental way of implementing MC: earlier version of MC had to wait till end to get full return
For infinitely long trajectories (non-episodic), make sure gamma is not very large else doomed
In episodic tasks, can implement MC with gamma = 1
In TD(lambda), if using lambda = 1, make sure gamma is not 1 (else rewards blow up)
Read section 7.4 from 1st edition (equivalence of forward and backward view: offline equivalence)
Student question: In backward view zeroing out eligibility traces after a while (forward view equivalence is not trivial)
Suppose zero out eligibility trace after 3 steps (backward view): figure out forward view by reading equivalence from book
Variants of MC: first visit and every visit
True (or exact? check) TD(lambda): backward view is modified so that forward view stays neat
Accumulating traces implements every visit MC (check 7.4 1st edition book)
Replacing traces implements first visit MC (check 7.4 1st edition book)
Some telescopic terms cancel out
Dutch traces: something between accumulating and replacing traces

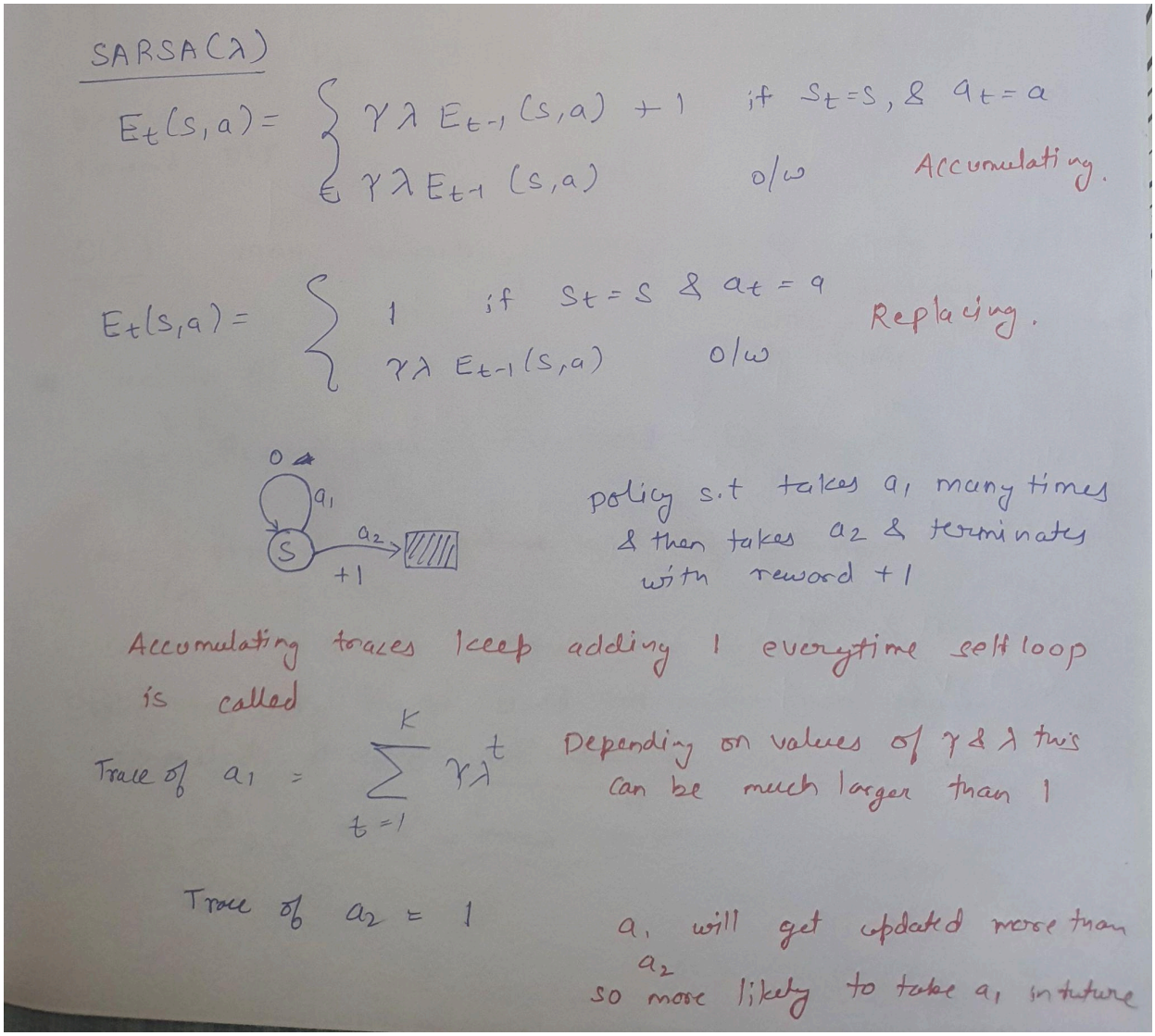


Gamma is a **parameter of the problem**: changing gamma changes the right answer (optimal policy or value function)
Lambda is a **parameter of solution method**: changing lambda does not change the right answer (optimal policy and value function still remain same, lambda just gives a different technique for estimating value function)
All TD(lambda) algos converge to the same value in expectation. The curves in the book are due to finite sample sizes. True expectation, all converge to the same.

Alpha is also a parameter of the learning process.
Any param of the learning process can affect the solution as it can influence the trajectory through solution space, especially in finite sample sizes. However, the optimal one remains the same regardless.

Eligibility Trace Control

TD(lambda) is estimation method
Now control
SARSA(lambda)
Backward view is intuitive
Eligibility trace for (s,a) pair



Rate at which a2 comes to beat a1 is very slow
Replacing traces fix this issue eventually
Accumulating traces are bad in control

Replacing traces are better

Aggressive version of replacing traces: when come to a state and make trace zero of all other actions so that reward goes to that action only. Reward will change the value of only that action which caused it.

$$E_t(s,a) = \begin{cases} 1 & \text{if } s_t = s,\ A_t = a \\ 0 & \text{if } s_t = s,\ A_t \neq a \\ \gamma\lambda E_{t-1}(s,a) & s_t \neq s \end{cases}$$

instead of decaying for other actions, making it zero.

Forward view for SARSA(λ) similar to TD(λ)

generalise from TD(λ) just $Q(s,a)$ instead of $V(s)$

$$\Delta Q_t(s,a) = \alpha \delta_t E_t(s,a)$$

$$\delta_t = R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t)$$

Bad

Forward DIY

Q(λ) many variants. peng's, naive, watkin's Q(λ)

justifiable although does not work well in practice

max in QL

$$\delta_t = R_{t+1} + \gamma \max_{a'} Q_t(S_{t+1}, a') - Q_t(S_t, A_t)$$

For 2 step return, behave optimally subsequently. (or n step return)

Can't do ϵ-greedy. Have to be greedy.

Entire trajectory can be used for return as long as it is generated greedily subsequently.

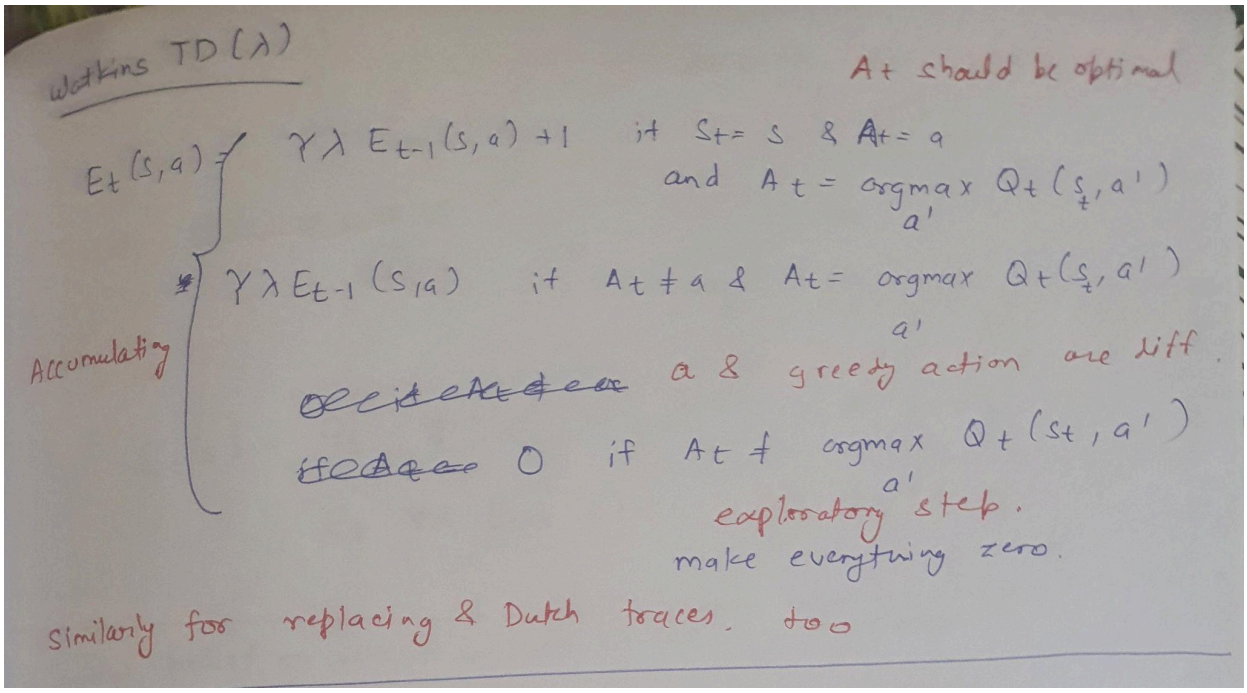
$$S \xrightarrow{a} S' \xrightarrow{\max a'} S'' \xrightarrow{\max a''} S''' \xrightarrow{\max a'''} \cdots$$

Greedy (not ϵ-greedy)

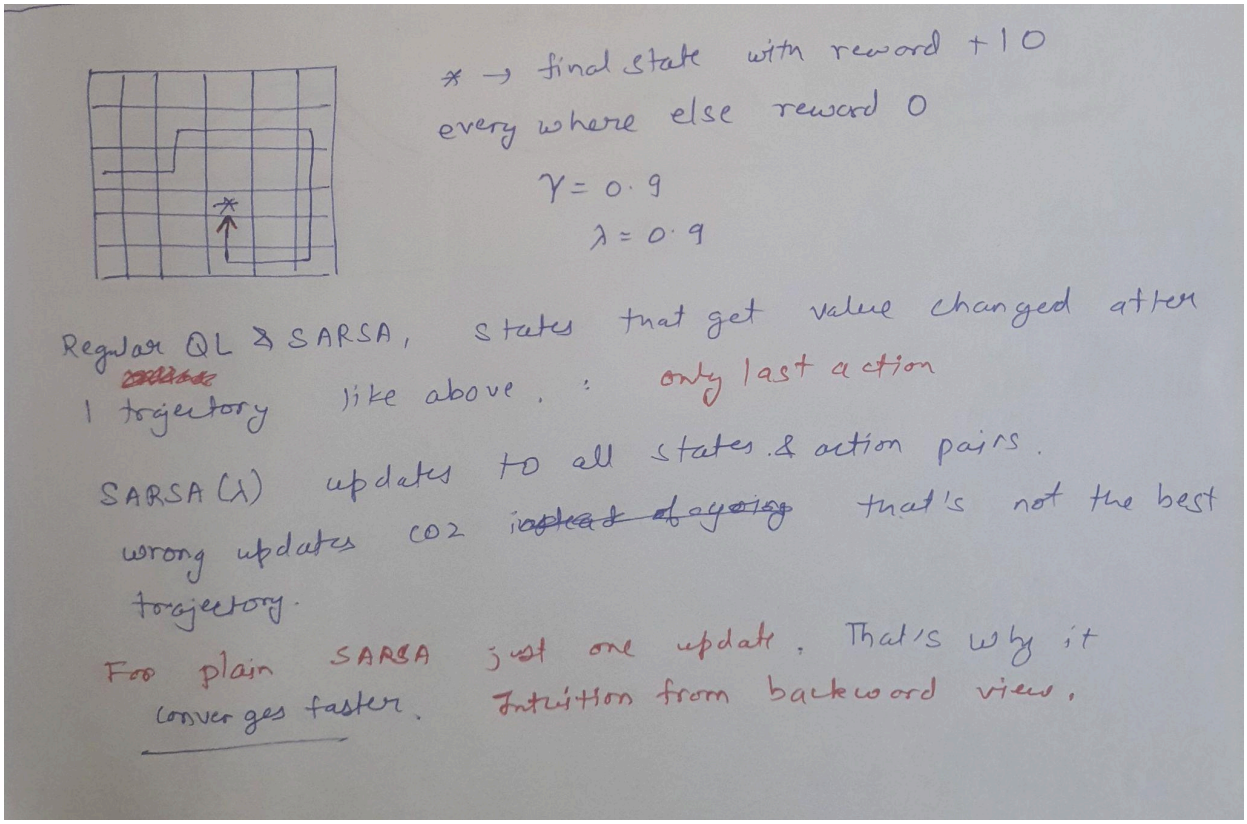
We have to be greedy. Will deal with exploration later.

Watkins Q(lambda): eligibility trace will not run till end of trajectory (only for short periods of time until exploratory step) if use exploration **(issue with Watkins Q(lambda))**

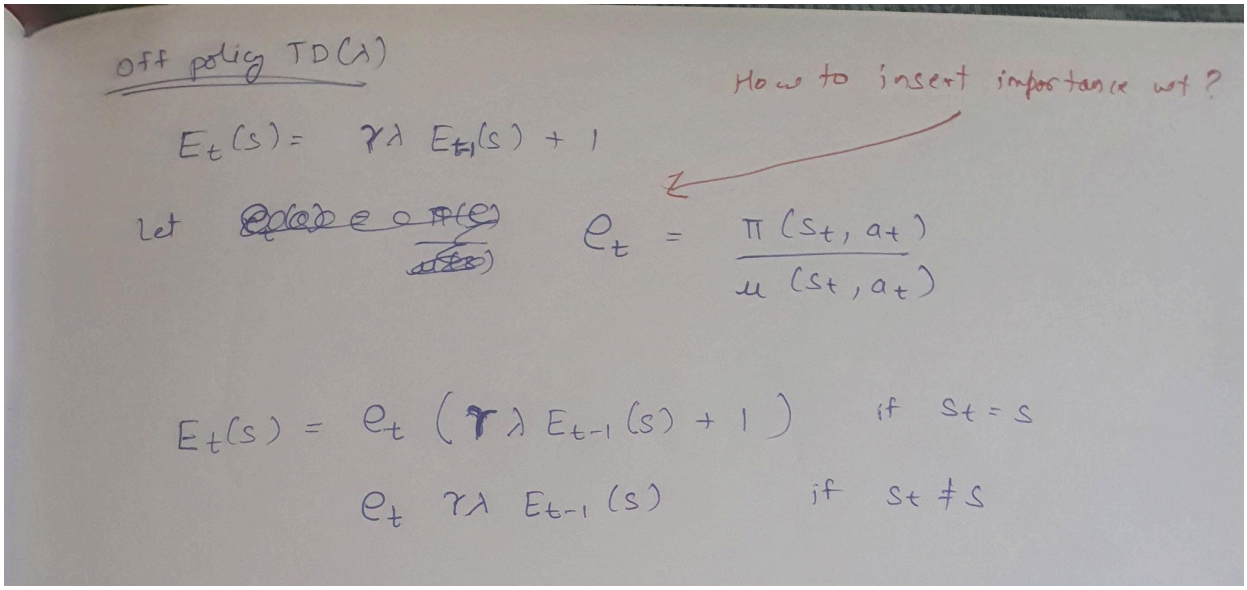
Exploration: reset all traces and start over
Forward view of Watkins Q(lambda) is very messy



Main problem with Watkins Q(lambda): all traces are very short unless decrease exploration by significant amount

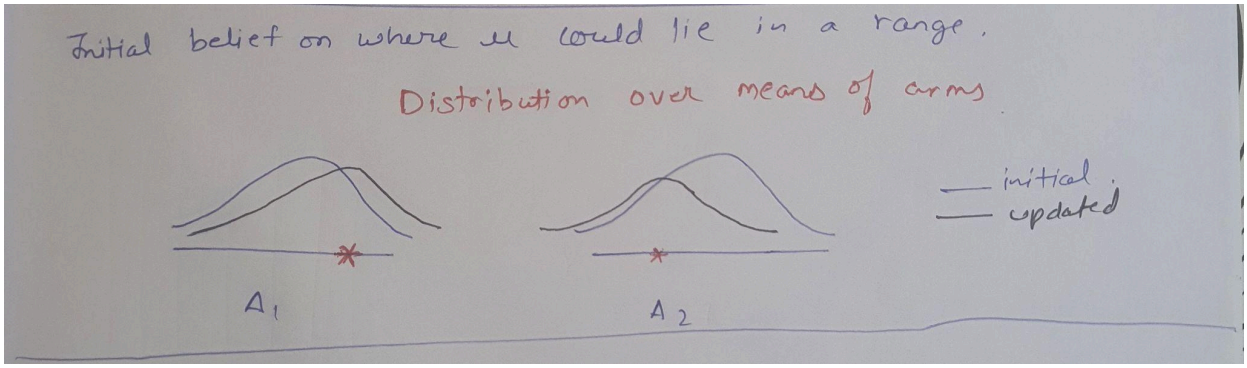


Showing equivalence of SARSA(lambda) forward and backward view has to be done offline (tricky coz never run SARSA offline): recent versions of SARSA(lambda) which use online version to show equivalence of fwd and bwd view
QL does not really fit into off policy setup as discussed in MC: estimation policy, behavior policy
There we took lots of shortcuts as estimation policy was the optimal policy
Here, we don't know the estimation policy pi here
QL really does not fall into off policy setup
So off policy version (diff from online and offline concept) of TD(lambda)



Thompson Sampling Recap

MLE: based on given data figure out the parameters (trying to figure out what process generated the data) and then solve the problem posed by the parameters
In value function based approaches, we make MLE guess (average of rewards accumulated by arms)
The alternative is the **MAP** version: assume something about priors of means, get some samples and based on those samples modify the prior to get posterior. Then pick the most likely outcome (mean of posterior).
2 probabilities here: probability of parameter (means) and probability of reward for that parameter



MLE does not make any assumptions on distribution. It simply uses the average of data it has seen so far.

Full Bayesian version: Instead of picking mean of the posterior, sample from posterior. This is because everytime we pull the arm, we do not get mean reward. Hence, it makes sense to sample from posterior instead of just using the mean.

Distribution over means as above

For each value of the mean that we could sample from that distribution, I am going to find the expected reward and average it by the probability of that mean being correct.

2 integrals: outer over means and inner one corresponding to each mean and running over rewards for that mean

This expectation can be approximated by simply drawing sample from posterior and then updating posterior

Instead of computing full expectation, using one sample (assuming that they are the means)

In bandit case easy: pull arm whose drawn sample is highest

Thompson sampling is also called posterior sampling: it is in effect the full Bayesian version

Week 8

Function Approximation

Till now value function based approaches

Lookup tables

Tabular representation drawbacks:

1. Memory
2. Handling continuous states
3. Can not have an estimate unless visited that state often enough (can have values only for visited states) Need of **generalization**

Learning using parameters: given data points learn some mapping which can generalize to data points it has not seen

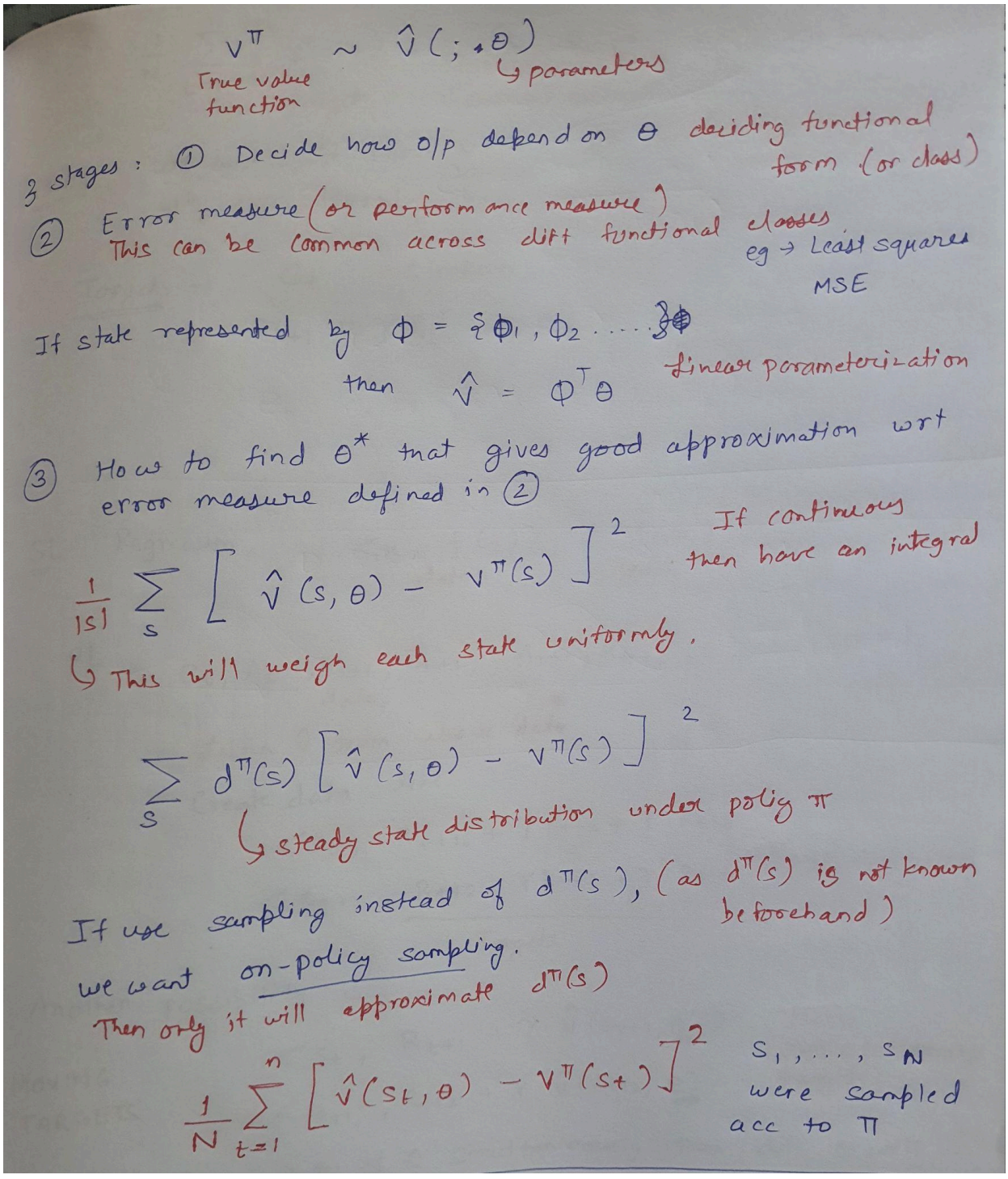
Represent value functions as parameterized function instead of a lookup table

Parameters of NN: weights

Cons of parameterized representations:

1. Bias in terms of approximation: If use so powerful parameterization that can represent every value function then what's even the purpose, could have gone with lookup table (parameters needed in such a complex representation will be equal to number of entries in lookup table itself): Lookup table is also a kind of parameterized representation (value of each state can be treated as the parameter) Want a **simpler representation** (with lot fewer parameters than the number of states)

Function approximation ~ Parameterized representation



Regression trees have been used for representing value function
Some functional forms: NN, linear parameterization, state aggregation etc.
Assume state is represented by some features
If data is coming to us from probability distribution then can use 1/N for mean under the the assumption that more likely datapoint will appear more times
As we are moving in MDP using a fixed policy, the trajectory is actually a Markov chain
In Supervised Learning, training data has labels but in RL we do not have data in that form. Here, data is in form of set of (s, a, r, s')

$$\frac{1}{N}\sum_{t=1}^{N}\left[\underbrace{v^{\pi}(s_t)}_{\text{target}} - \underbrace{\hat{v}(s_t,\theta)}_{\text{current estimate}}\right]^2$$

Issue: we ~~kno~~ don't know $v^{\pi}(s)$ If known already then why even learning

Targets → G_t MC return.

$R_{t+1} + \gamma\,\hat{v}(s_{t+1};\theta)$

SL Regression $\langle x, f(x)\rangle \rightarrow \hat{f}(x)$

data learn

RL $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, \ldots$

data

Learn $\hat{v}$ from above data.

Create data like

$\langle S_t, R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \cdots \rangle$

$\langle S_{t+1}, \underbrace{R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} \cdots}_{\text{Targets}} \rangle$

Another possibility is

MOVING TARGETS

$\langle S_t, R_{t+1} + \gamma\,\hat{v}(S_{t+1},\theta)\rangle$

$\langle S_{t+1}, R_{t+2} + \gamma\,\hat{v}(S_{t+2},\theta)\rangle$

Here non stationary target

If ~~use~~ do updates of θ simultaneously then diff θ will give diff data points.

Targets ~~kep~~ keep changing after every updation of params.

So not like converge to some fixed target.

θ_t & S_{t+1} not θ_{t+1}

$$\theta_{t+1} \leftarrow \theta_t + \alpha\,\nabla_{\theta_t}\left[\underbrace{R_{t+1} + \gamma\,\hat{v}(S_{t+1},\theta_t)}_{\text{can use other target too.}} - \hat{v}(S_t,\theta_t)\right]^2$$

~~Alwaysup~~

(0.2 error)

Many people doubted RL due to moving targets but it has worked in practice
With linear parameterization, convergence can be showed even with moving targets (provided sampling on policy)
Recently, results in off policy domain too
Even with approximation, it becomes contraction and that is used in showing convergence
Gradient ascent/descent: Maximize performance and minimize error
Ascent in direction of gradient
Descent in opposite direction of gradient
Stochastic GD/GA: online so update every time step

Linear Parameterization

X axis: params
Y axis: error/performance
Some mathematical desirable properties of features:

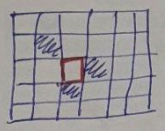
1. Linearly independent: vector of all values (over states) of phi_1 and phi_2 should be independent

In regression trees, linearly dependent features are not that catastrophic compared to linear regression (just adds redundancy in regression trees not like issues related to stability, correctness etc)
Linear independence is very important in methods using gradient computation or extensive numerical computation

Linear parameterization $\Phi(s)$ vector Φ_{1*} → First column

$s \rightarrow \langle \phi_1(s), \phi_2(s), \ldots \phi_k(s) \rangle$ features / attributes

k is 192 in TD-Gammon k can be large.



$k = 2$
x, y coordinate as features
or in terms of an obstacle whether r, l, u, or down
$k = 4$

(2, 2) x, y coordinate.

(0, 1, 1, 0) clockwise from above. obstacles.

Diff ways of coming up with features.

dimension of $\Phi(s) = k$

. . . . . $\Phi_1 = m$ (no. of ~~dimen~~ states)

$$\hat{v}(s, \theta) = \theta^T \Phi(s)$$

In parameter update, just take gradient of current estimate, not target (although non stationary)
assume target is coming from somewhere else like G_t

$$\theta_{t+1} \leftarrow \theta_t + \alpha \left(R_{t+1} + \gamma \hat{v}(s_{t+1}, \theta_t) - \hat{v}(s_t, \theta_t) \right) \Phi(s_t)$$

plus coz − of gradient chain rule

VECTOR equation (set of equations for each $\theta_1, \ldots \theta_t$)

Hadamard pdt in 2nd term.

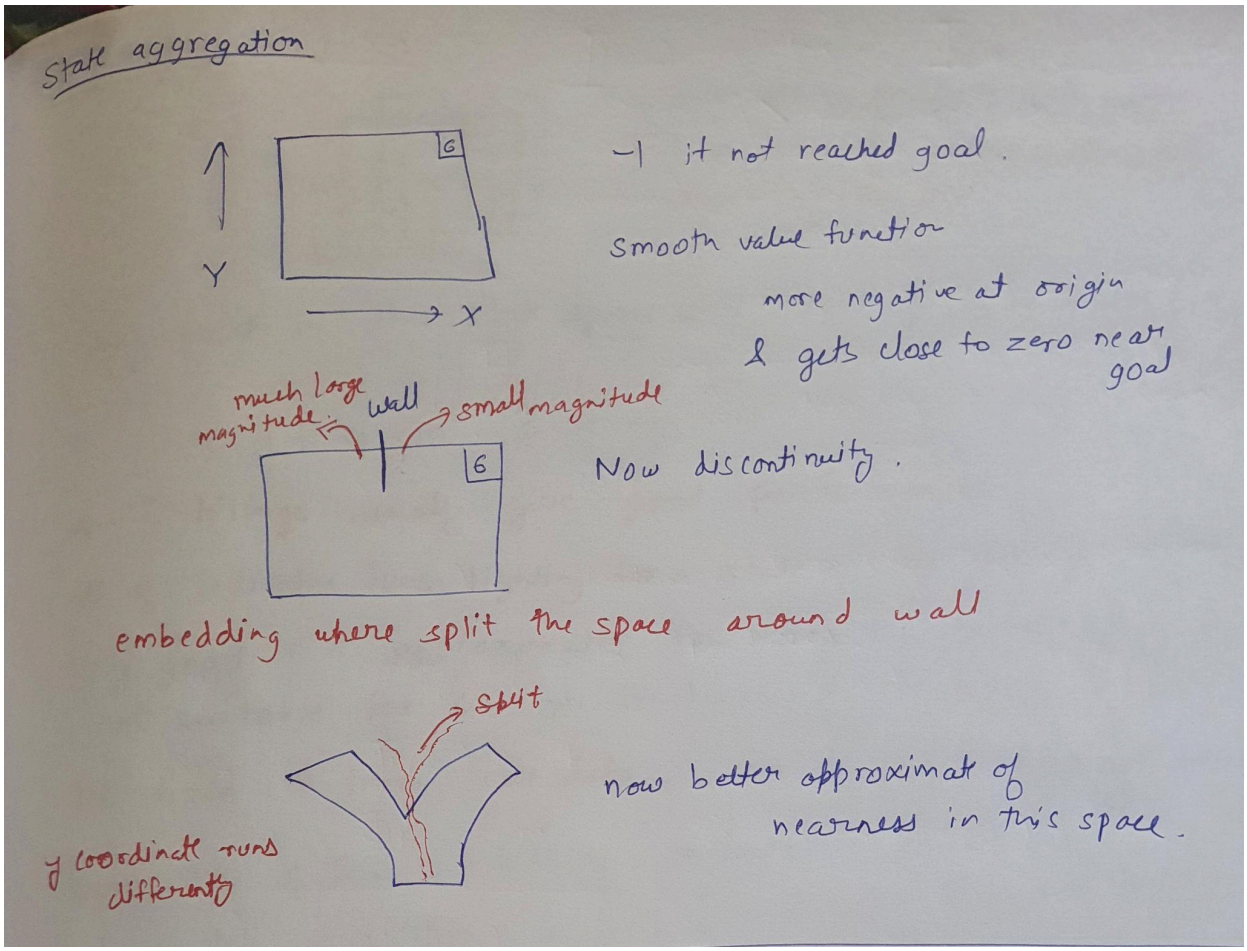
$$\theta_{t+1} \leftarrow \theta_t + \alpha \bullet \delta_t \Phi(s_t)$$

↳ TD error.

Different forms of linear parameterizations.

State Aggregation Methods

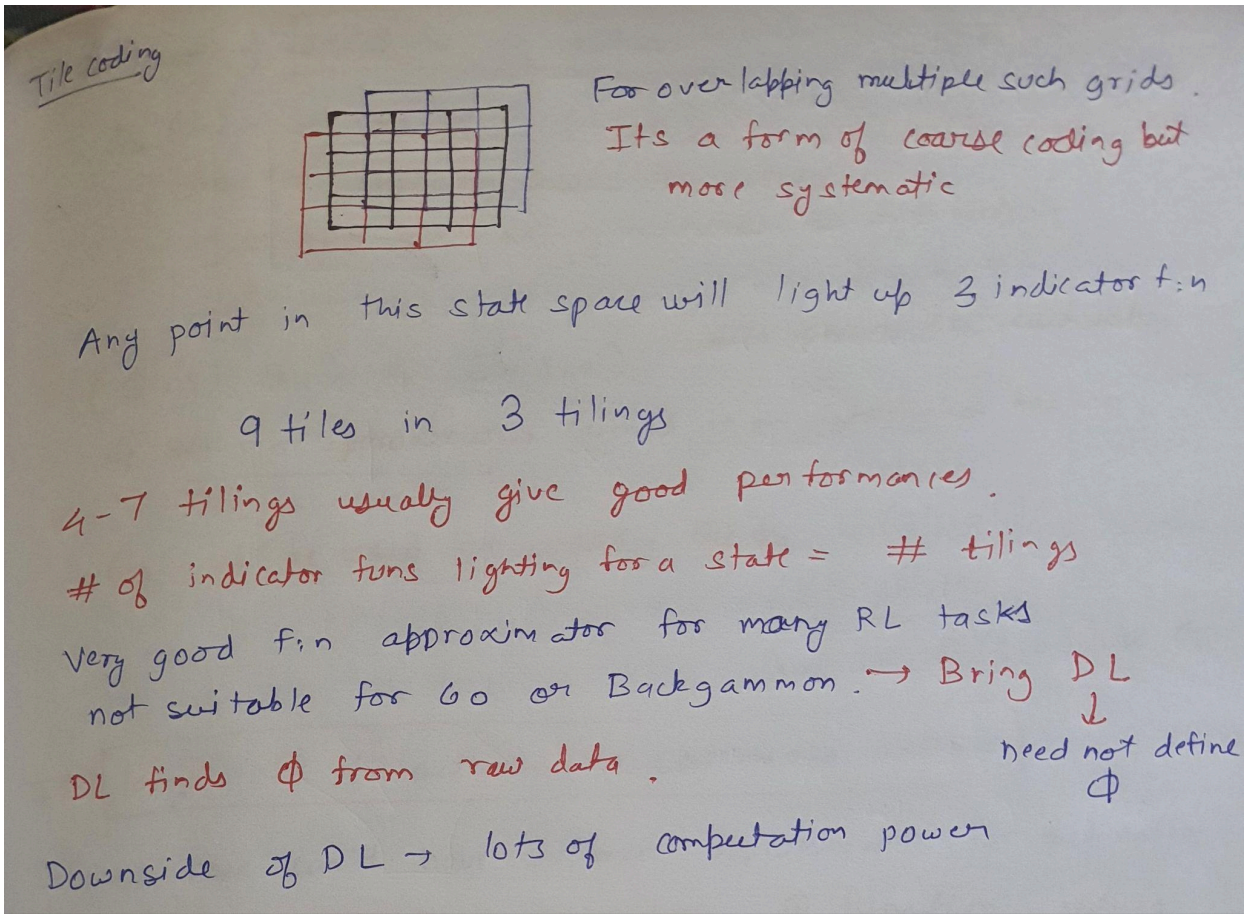
Lookup table is a linear function approximator: phi will be indicator function in term of state
phi_i(s) = 1 if s = i (Here features and parameters equal to number of states)
Another form of indicator function approximator: state aggregation
Assume that closeby states have same value (trying to reduce k: number of features/parameters): some information is lost
Now indicator function on that set of states
Sampling is still on-policy
For continuous state spaces, use membership concepts: can have overlapping to cover entire state space
Overlapping regions actually leads to better generalization as multiple indicator functions will come up due to overlaps
Will learn average value over that set as each update will be applied to all members of the sets
Drastic change in value at boundary of sets in discrete case
Overlapped regions will get multiple updates: this results in smoother shift instead of abrupt
It is ok to have overlaps actually one can argue that it is desirable
Till now nearness in terms of distance
But can actually transform into embeddings in such a space where nearness means nearness in state values
Basis functions: phi_1 to phi_k



Coarse coding: coarse representation of states (the continuous case described above)
Problem with coarse coding: no uniformity in the number of ‘on’ bits used for representing a state
Diff states can have diff bits

Tile coding:

Each tile has one parameter associated with it (total params = #tiles * #tilings)



CMAC (Cerebellar Model Arithmetic Computer): specific type of tile coding
Much older and derived from biological models
Classical CMAC uses hashing based tricks to compress tilings so that need not store entire vector theta
No issue with regard to storing tile as regularity there so can have some kind of function that computes tile membership
Also, some kind of lookup
Disadv: unintended (destructive) generalization as 2 different tiles (might be far away) can be hashed to same weight
In some large spaces, hashing helps learn faster

Function Approximation and Eligibility Traces

In linear function approximation, can track either phi or theta for eligibility (interchangeable)
In nonlinear case, need to compute the gradient wrt theta
Gradient tells how much is this particular theta responsible for output
For every theta keep track of the gradient

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

$$\vec{e} = \gamma\lambda \vec{e} + \nabla_{\theta_t} V(S_t)$$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \delta_t e_t$$

If linear then gradient is ϕ

If lookup table then gradient of indicator f:n (1 for that visited state 0 for others)

ACCUMULATING TRACES

~~(Place replacing just use gradient)~~ ~~add term~~

GENERAL

Control with function approximation

SARSA with f:n approximation

Need encoding for actions also as Q is used & not V

$Q_t(s, a)$ $\phi(s, a)$

Number of actions typically tend to be small

In case of **small no. of actions**

$\phi_a(s)$ instead of $\phi(s,a)$

don't explicitly encode the actions

just have diff set of features for diff actions.

OR

$(\phi^T(s)\ \theta_a) = Q(s,a)$ diff params for each action.

↳ diff f:n approximators trained in an interlinked fashion,

$\theta_{a'}$ will be used for updating θ_a $Q(s',a')$

use $\theta_{a'}$

but for $Q(s,a)$ use θ_a

If **large no. of actions / continuous actions**

In this case can't have separate approximator for each action.

Values of afterstates are in reality Q function values. (some form)

In navigation, North & South are diff so have diff θ

but In Go, actions are similar so better generalise across action domain too : $\phi(s,a)$

→ afterstate values are actually value corresponding to state action pairs. & afterstate is used as an encoding of state-action pair. implicitly doing that.

Adv (afterstate) (1) generalise across related actions.

similar afterstate will have same value (not same)

Actions from diff states which land in similar ~~same state~~ afterstate, then have same value for that.

Allows generalisation across actions

Adv (2) Fewer parameters to estimate.

Afterstates can be used in case of continuous actions too as long as you have a deterministic forward model to describe things (with stochasticity things are not straightforward)
Difference between after state and state: assuming that there are deterministic moves and nature's moves (transition dynamics can be split into deterministic component and nature component)
What if we have access to the opponent model (nature component)? It is ok to use it if add opponent state into own state representation

Joint features — $\phi(s,a)$ in contextual bandits, joint encoding of state & actions.

SARSA (λ)

$|\vec{e}_t| = |\vec{\theta}| \neq |S|$
components components size of set

$\vec{e}_{t+1} = \gamma\lambda \vec{e}_t + \nabla_{\theta_t} Q(s_t, a_t)$

ACCUMULATING VERSION

In general can't define a replacing version.

In case of binary features, can use replacing traces

If $\phi = 1$, replace by gradient

If $\phi = 0$, decay.

$S \rightarrow \phi(S) = \langle 0\,0\,1\,1\,0\,1 \rangle$

$e_t(1) = \gamma\lambda\, e_{t-1}(1)$

$e_t(2) = \gamma\lambda\, e_{t-1}(2)$ replacing traces

$e_t(3) = 1$

$e_t(4) = 1$

$e_t(5) = \gamma\lambda\, e_{t-1}(5)$

$e_t(6) = 1$

For accumulating traces, $1 + \gamma\lambda\, e_{t-1}(1)$ instead of 1

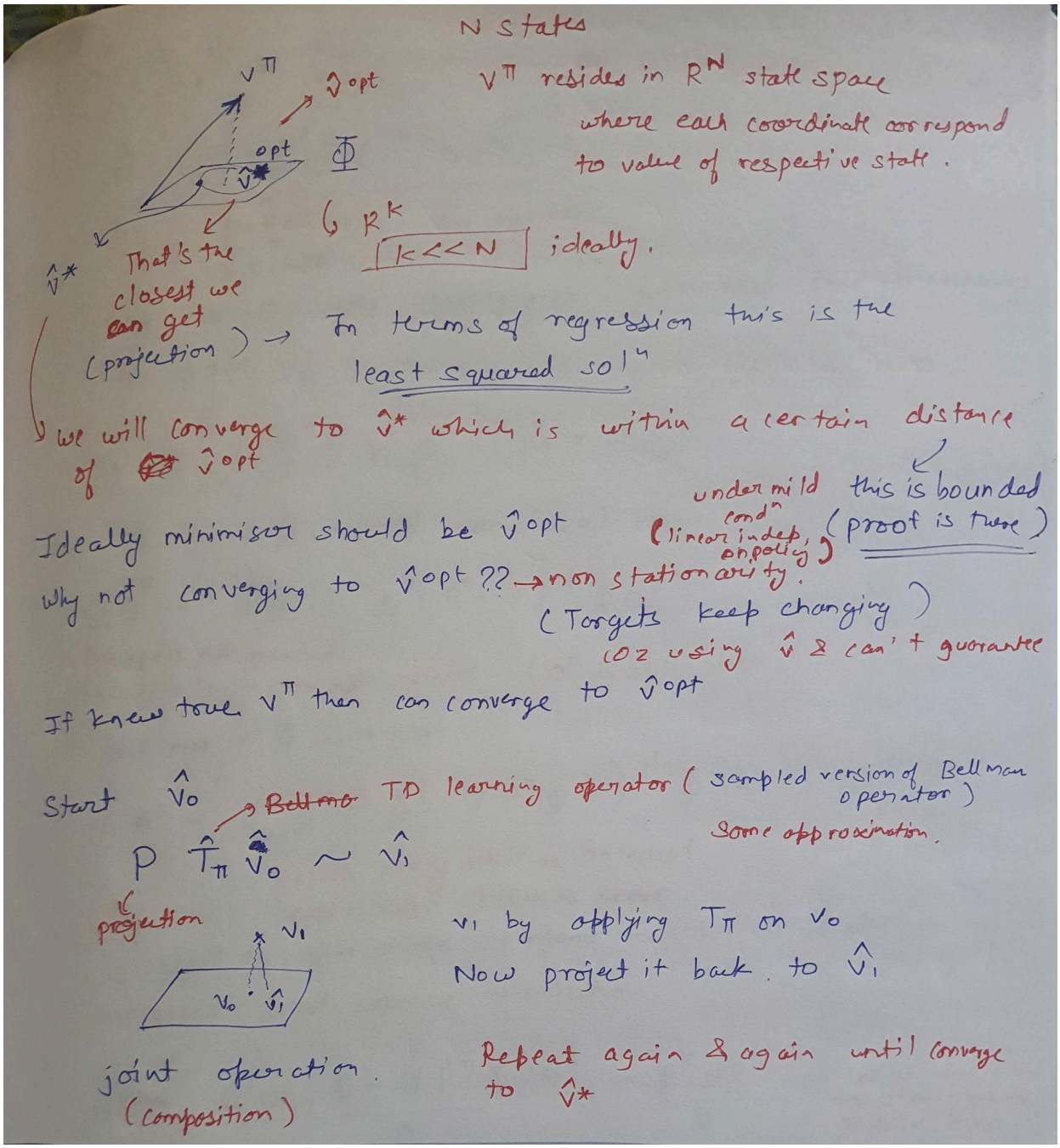
Replacing traces are bad for non binary features coz if even sensitivity of 0.003 then also erase old trace

This corresponds to when for 0 just decay & for non zero, replace it

Coarse coding & tile coding have binary representation (state aggregation)

In case of binary features can use replacing traces but otherwise use accumulating traces

LSTD and LSTDQ



How to set it up as a proper regression problem (labeled data: data matrix or design matrix)? Least squares fit for a given set of samples (no guarantee how good that fit is)

$\langle (x_1, x_2, \ldots, x_k), y \rangle_{i=1}^{n}$

$\langle (\phi_1(s), \phi_2(s), \ldots, \phi_k(s)), v(s) \rangle$

Input — Don't know this

Assume $v(s) = \Phi^T(s)\theta$

MC converges to LS

TD converges to certainty equivalence.

Repeat:
- Make big dataset using some θ
- Use LS to get new θ estimate.

$(P T_\pi)\, \hat{v}^\pi = \hat{v}^\pi$ fixed pt

$(s, \pi(s), s')$

$\hat{v}^\pi = \underbrace{\Phi(\Phi^T\Phi)^{-1}\Phi^T}_{\text{projection}} \underbrace{(R^\pi + \gamma P^\pi \hat{v}^\pi)}_{T_\pi \text{ operator}}$

(LR solution)

Φ is for all states. entire matrix with each row corresponding to state.

$(s', \pi(s'))$

$\hat{Q}^\pi = \Phi(\Phi^T\Phi)^{-1}\Phi^T (R^\pi + \gamma P^\pi \hat{Q}^\pi)$

↳ Φ defined on (s, a) pairs.

$\Phi\theta^\pi = \Phi(\Phi^T\Phi)^{-1}\Phi^T(R^\pi + \gamma P^\pi \Phi\theta^\pi)$

Transpose not needed coz this is matrix not vector. (θ^π is still vector)

each row in Φ is a state.

$\theta^\pi = (\Phi^T(\Phi - \gamma P^\pi \Phi))^{-1}\Phi^T R^\pi$ Least square sol^n

reads every state as an equal contributor towards error.

want to give more importance to important states.

weighted least square minimization.

$\underbrace{\Phi^T\omega(\Phi - \gamma P^\pi \Phi)}_{A}\theta^\pi = \underbrace{\Phi^T\omega R^\pi}_{b}$

coz we dont know transition prob to solve MDPs

ω is weight vector over all states.

eg → steady state prob distribution (we don't know it but can estimate using relative frequencies in the samples drawn already)

could be very far away from actual. if fixed set of samples

$A\theta^\pi = b$

$$A = \Phi^T w (\Phi - \gamma P^\pi \Phi)$$

Comes coz want from matrix notation to individual state notation.

$$= \sum_s \phi(s)\, w(s) \left(\Phi(s) - \gamma \sum_{s'} P(s, \pi(s), s')\, \Phi(s') \right)^T$$

$$= \sum_s w(s) \sum_{s'} P(s, \pi(s), s') \left[\Phi(s) \left(\Phi(s) - \gamma \Phi(s') \right)^T \right]$$

Drop out coz sum over s'

added unnecessary $P(s, \pi(s), s')$ to $\Phi(s) \cdot \Phi(s)$

Above expression is weighted sum of expectation of [ ] qty. Sampling. coz we don't know P^π & R^π (learning setting)

- Samples

$$D = \{ (s_i, a_i, r_i, s_i') \mid i = 1, 2, \ldots L \}$$

i is sample index not time index

$$\tilde{A} = \frac{1}{L} \sum_{i=1}^{L} \left[\phi(s_i) \left(\phi(s_i) - \gamma \phi(s_i') \right)^T \right]$$

weights will factor in implicitly as more imp state will appear more times.

$$b = \Phi^T w R^\pi$$

$$= \sum_s \phi(s)\, w(s) \sum_{s'} \underbrace{P(s, \pi(s), s')\, R(s, \pi(s), s')}_{R^\pi \text{ expected reward.}}$$

$$= \sum_s w(s) \sum_{s'} P(s, \pi(s), s') \left[\phi(s)\, R(s, \pi(s), s') \right]$$

$$\tilde{b} = \frac{1}{L} \sum_{i=1}^{L} \phi(s_i)\, r_i$$

Q versions

$$\tilde{A} = \frac{1}{L} \sum_{i=1}^{L} \left[\phi(s_i, a_i) \left(\phi(s_i, a_i) - \gamma \phi(s_i', \pi(s_i')) \right)^T \right]$$

on policy. might not have in data

$\pi(s_i')$ action in s_i state

$$\tilde{b} = \frac{1}{L} \sum_{i=1}^{L} \phi(s_i, a_i) r_i$$

Approach where we use estimated $\tilde{A}$ & $\tilde{b}$ is called. LSTD (V)

LSTDQ (Q)

LSTD(λ) tricky. $\gamma\lambda$ terms in A matrix

$\tilde{A} \Theta^T = \tilde{b}$ invertible if ϕ_i $i=1$, to k are linearly independent

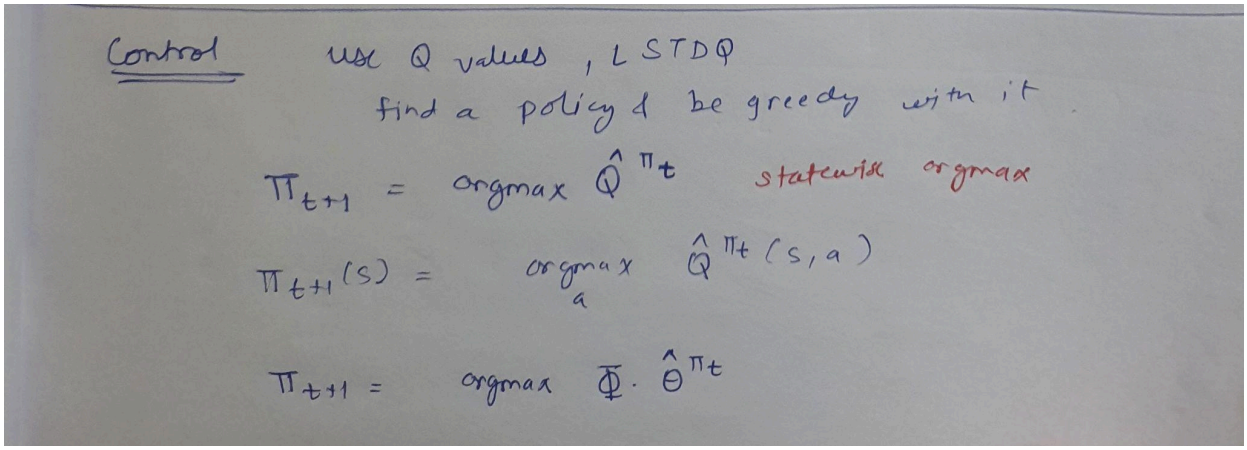
throw away $\frac{1}{L}$ coz cancel both sides.

remaining is summation

very easy to update $\tilde{A}$ incrementally as we keep on getting transitions.

At any time, can take current snapshot of $\tilde{A}$ & solve for V^π or Q^π

LSPI and Fitted Q



Problem depends on how my states & actions are represented

→ For small no. of ~~states~~ actions its fine. but for continuous actions or if $\Phi(s, a)$ parametrization translate into continuous value

→ If state itself is also continuous, what is meant by ~~argmax~~ argmax over all states.

Many clever ways to solve it: people have formulated LP & all

(1) Solve maximization for few states. & get estimate of $\hat{\pi}$ by solving classification problem

Solve for $\begin{pmatrix} s_1, s_2, \ldots s_j \\ a_1, a_2 \ldots a_j \end{pmatrix}$ (j states)

$\pi_{t+1}(s)$

Training data to classifier:

↳ for finite action case else regressor for continuous actions.

$$\begin{matrix} \Phi_1(s_1) \ldots \Phi_k(s_1) & a_1 \\ \vdots \quad\quad \vdots & \vdots \\ \Phi_1(s_j) \ldots \Phi_k(s_j) & a_j \end{matrix}$$

* Don't take argmax every time. Precompute it & store then do lookup.

~~Take~~ The classifier / regressor will learn function f Next time come to a state, pass state to f & it gives action. ~~Take~~ f is representation of policy π.

$\Phi(s) \rightarrow A$

~~$\Phi(s) \rightarrow A$~~

This Φ can be different from Φ used previously as this is completely different function approximator.

LSPI → least sq policy iteration.

Fitted Q iteration

Q-learning

Fixed set of samples

Generate targets $R + \gamma \max_{a'} Q^*(s_{t+1}, a')$ → eg: use LSTDQ

with same data train a function approximator to approximate the targets.

$\hat{Q}$ initial → $\left(\Phi(s_i, a_i),\ r_i + \gamma \max_a \hat{Q}_0(s_i', a) \right)_{i=1}^{L}$

Input (training data) — output (Target)

Feed data to function approximator.

$\hat{Q}_1$ → $\left(\Phi(s_i, a_i),\ r_i + \gamma \max_a \hat{Q}_1(s_i', a) \right)$ New training data.

$\hat{Q}_2$

& so on.

take care of that we can do maximization properly (argmax)

ensure generated sufficiently random samples else tricky.

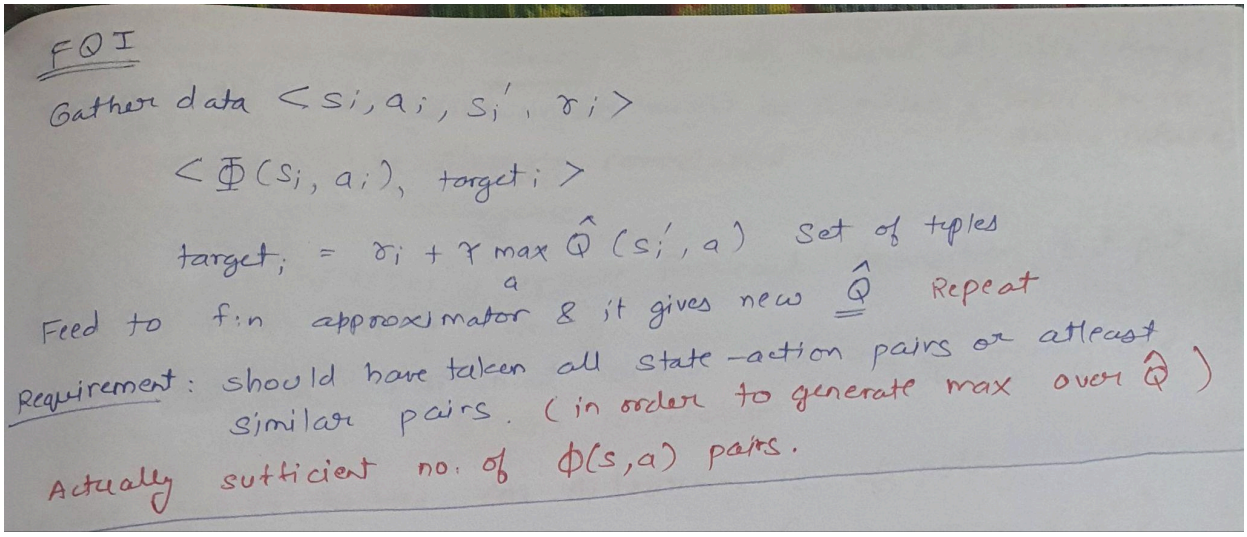
FQI :

If fixed set of samples & no way of generating more samples, FQI works much better Imp

A lot of stuff from FQI setting is used in DQL

Week 9

DQN and Fitted Q-Iteration

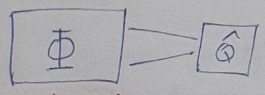


Static training data should be broad enough: representative so as to sample all kind of state-action combinations (in terms of phi)
LSTD and FQI have stationary targets
Neural fitted-Q (NFQ): uses NN as function approximator
Possible Problems:

1. Should have sampled enough pairs
2. Max could be a problem but can be dealt with easily in case of discrete actions by having different NN for each action
3. Coming up with features: Deep NN can learn features on its own

DQN addresses above problems
DQN samples online and does not use offline data
DQN was trained to play Atari games
Original DQN paper had memory of last 100 transitions or so but many variants like preference retention

DQN

Input — 

Deep learning finds features 18 actions in Atari so 18 such n/w

18 o/p take argmax take best action.

feed next state into Φ box & do Q update in one shot

Nothing like target generation in FQI

Doing online updates.

Can run DQN as online off policy algo. (Q-learning is off)

Problem (with online Q-learning):

① not using samples efficiently. use & throw.

Have a memory & add. $\langle s_i, a_i, s_i', r_i \rangle$ into it

Pull them once in a while & replay them.

Replay memory Experience replay.

Old concept but there replay entire trajectory but here just transition. (samples)

exact same experience again.

If go through samples often enough, converge to something similar to certainity equivalence estimate.

Replay memory ensures efficient reuse of samples

(2) states are changing slowly & typically reward will also change slowly. successive inputs might be similar (issue due to online nature)
updates will be strongly correlated
issues with convergence.

Fix: not necessarily update for each transition but put it in replay memory.
pick one transition at random from replay memory and update it. breaks correlation

we are able to do this coz Q learning is off policy.
Why experience replay works ??

(3) Biased sampling of i/p space (issue due to online nature)
coz it depends on actions taken.
eg → left movement means see states on left
increases correlation b/w successive inputs seen
also skews the kind of samples seen. right side might not be seen
some portions of state space get their values updated others don't.

Fix Transition replay handles it too.

Other online methods also have these issues but then we were dealing with small discrete spaces. Here, large continuous spaces.

NN what you do in one part of state space can affect (non-deterministically) what is happening in other part of state space. f.n approximators are complex.

$\hat{Q}$ need not necessarily come from NN. Can be anything as long as you can compute gradient through it.

Deep representation learning.

Policy Gradient Approach

Advantages:

1. Value function can be a lot more complex than policy. Eg. inventory control.
   In parameterized space, policy can be represented using **fewer params** compared to value function
2. Value function based methods are not guaranteed to converge in parameterized space (some results in linear parameterization tho). PG has guarantees of **convergence** to local optima. Stability in policy approximation but might diverge in value function approximation. In large problems go for policy based approaches.
   Note: Although in theory, value function approaches are shadowed by policy based approaches, value function based approaches work very well in practice as demonstrated by many success stories in large scale applications of RL.

PG

$\pi(\cdot; \theta)$

Some defⁿ of performance of system.
Gradient of performance measure wrt θ
Move in direction of gradient. GA not GD.
Under certain conditions GA converges to local optima.

step sizes ↙ ↘ unbiased gradient estimate

Performance measure

$$\eta(\theta) = \sum_a q_*(a)\, \pi(a, \theta)$$

we don't know q_*

From REINFORCE lecture

$$\nabla \hat{\eta}(\theta) = \frac{1}{N} \sum_{i=1}^{N} r_i \frac{\nabla_\theta \pi(a_i; \theta)}{\pi(a_i, \theta)}$$

N samples.

This was for Bandits. Now for full RL.
No state.

There Q fⁿ was avg of rewards. Here, Q fⁿ is average of returns. So sample should be trajectories not ~~samples~~.
transitions.

$$< S_0, a_0, r_1, S_1, a_1, r_2, \ldots\ldots S_T >$$

THAT'S WHY EPISODES ARE USED IN PG METHODS.

$$\nabla \hat{\eta}(\theta) = \frac{1}{N} \sum_{i=1}^{N} G_i(s_0) \frac{\nabla P_i(s_0; \theta)}{P_i(s_0; \theta)}$$

$G_i(s_0)$ ↙ Return of ith sample starting from s_0

↘ prob of seeing ith sample starting from s_0
Basically prob of trajectory when start from s_0 & follow π

Can convert product into sum as derivative of log is there.

$$\frac{\nabla_\theta p_i(s_0;\theta)}{p_i(s_0;\theta)} = \nabla_\theta \ln p_i(s_0;\theta)$$

as transition prob are determined by env.

$p(s_0, a_0, s_1)$, $p(s_1, a_1, s_2)$ etc. will cancel out as they will be constant (independent of θ) & we are taking gradient

Leftout will be be prob of a_0 in s_0, a_1 in s_1 etc.

$$\sum_{j=0}^{T-1} \frac{\nabla \pi(s_j, a_j, \theta)}{\pi(s_j, a_j, \theta)}$$

once gradient is estimated can do update in direction of gradient

Performance measure $\eta(\theta)$ is f:n of θ alone not state or action.

No notion of value f:n associated with state

Still some notion of value is needed to summarise utility of a policy.

s_0 does not depend on i

Assuming an unique start state & using returns from s_0 as evaluation for policy, many cases its acceptable like chess.

If there is a set of start states can introduce an ~~set of~~ imaginary state from which transition uniformly to any of start state.

Another assumption assume MDP satisfy cond^n of avg reward

Avg reward $\lim_{N\to\infty} \frac{1}{N}\left(\sum_{i=1}^{N} R_i\right)$ existence of limit, well defined limit

As $N \to \infty$, all the variance caused due to starting in state 1 or state 2 get wiped out.

As $N \to \infty$, value of policy converge to some value regardless of start state.

$$e^\pi(s) = e^\pi \quad \forall s$$

e^π denotes value of policy.

$$e^\pi = E\left[\lim_{N\to\infty} \frac{R_1 + R_2 + \ldots + R_N}{N}\right]$$

Some cond^n for average reward to exist & be well defined

Average reward RL

Certain convergence results are easier to prove using average reward RL

It removes the need of gamma: for some problems gamma is naturally defined but for others gamma needs to be tuned

Recent results on function approximation better behaved with average reward: for discounted setting sometimes target is not well defined

More robust as slight change in gamma can change values significantly

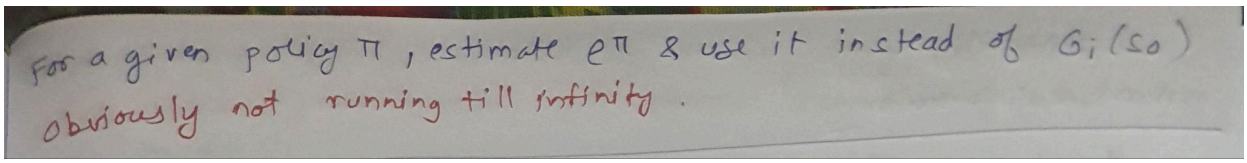
Easier to show results for PG using average reward

Many advantages of average reward RL: most are theoretical

Disadvantages of average reward RL: numerically very hard to get it to work

Also what is definition of average reward if all trajectories are finite length: N can never tend to infinity

Fix: assume N tends to infinity (assume to have recurrent class of states)



Problems (with running MC gradient estimates like PG):

1. High variance if very long trajectories: can keep on running forever unless do something clever to reduce variance
   Fix: truncate trajectories (fine to do that as gamma is going to decay either way and in case of average reward it is fine coz we will have good estimate if done enough circles around recurrent states: designate an unique recurrent state and truncate whenever enter it if still long can designate multiple such states)
   Another method of **reducing variance: Value function** are essentially estimates of returns

Actor critic methods: Critic is used to evaluate the actor

Actor Critic and REINFORCE

REINFORCE

$$\Delta\theta = \alpha_n (r_n - b_n) \frac{\partial}{\partial\theta} \ln(\pi(a_n, \theta_n))$$

b_n → baseline eg → avg of previous rewards.

If $r_n > b_n$: good action.

$r_n < b_n$: bad action.

This for bandits now for full RL

$G_n(s_n) - b_n(s_n)$

$b_n(s_n)$ → avg of returns starting from s_n — essentially value f:n.

Critic ←

$$\Delta\theta = \alpha_n \left(G_n(s_n) - V_n(s_n) \right) \frac{\partial}{\partial\theta} \ln \pi(s_n, a_n, \theta_n)$$

Still solving for policy params but using value f:n for evaluation

Actor

$$\left(R_{n+1} + \gamma V_n(s_{n+1}) - V_n(s_n) \right)$$

TD error

$$\Delta\theta = \alpha_n \delta_n \frac{\partial}{\partial\theta} \ln(\pi(s_n, a_n, \theta_n))$$

Indicator variable like policy parameterization can make gradient disappear. some kind of lookup table.

$$\Delta\theta = \alpha_n \delta_n \phi_{\blacksquare}(s_n, a_n)$$

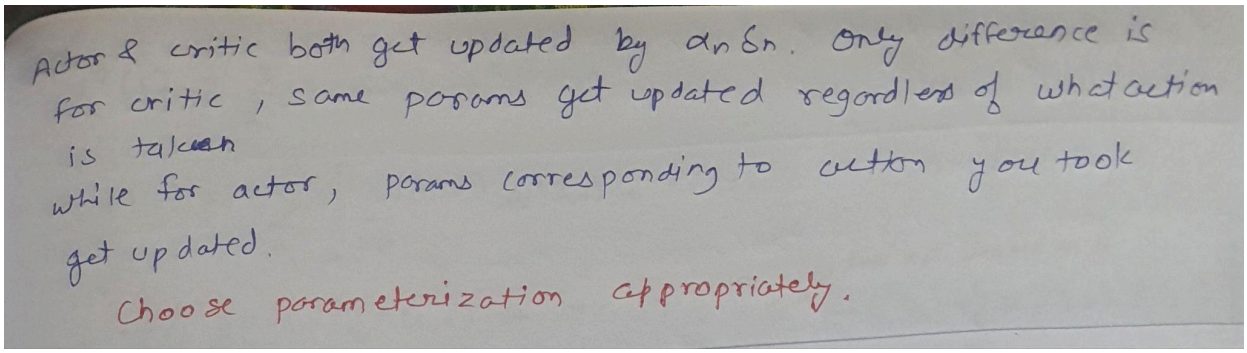
zero for others
1 for this state-action pair

updation for policy param

Updation for value params : TD update rule.

For policy, update corresponding to that action

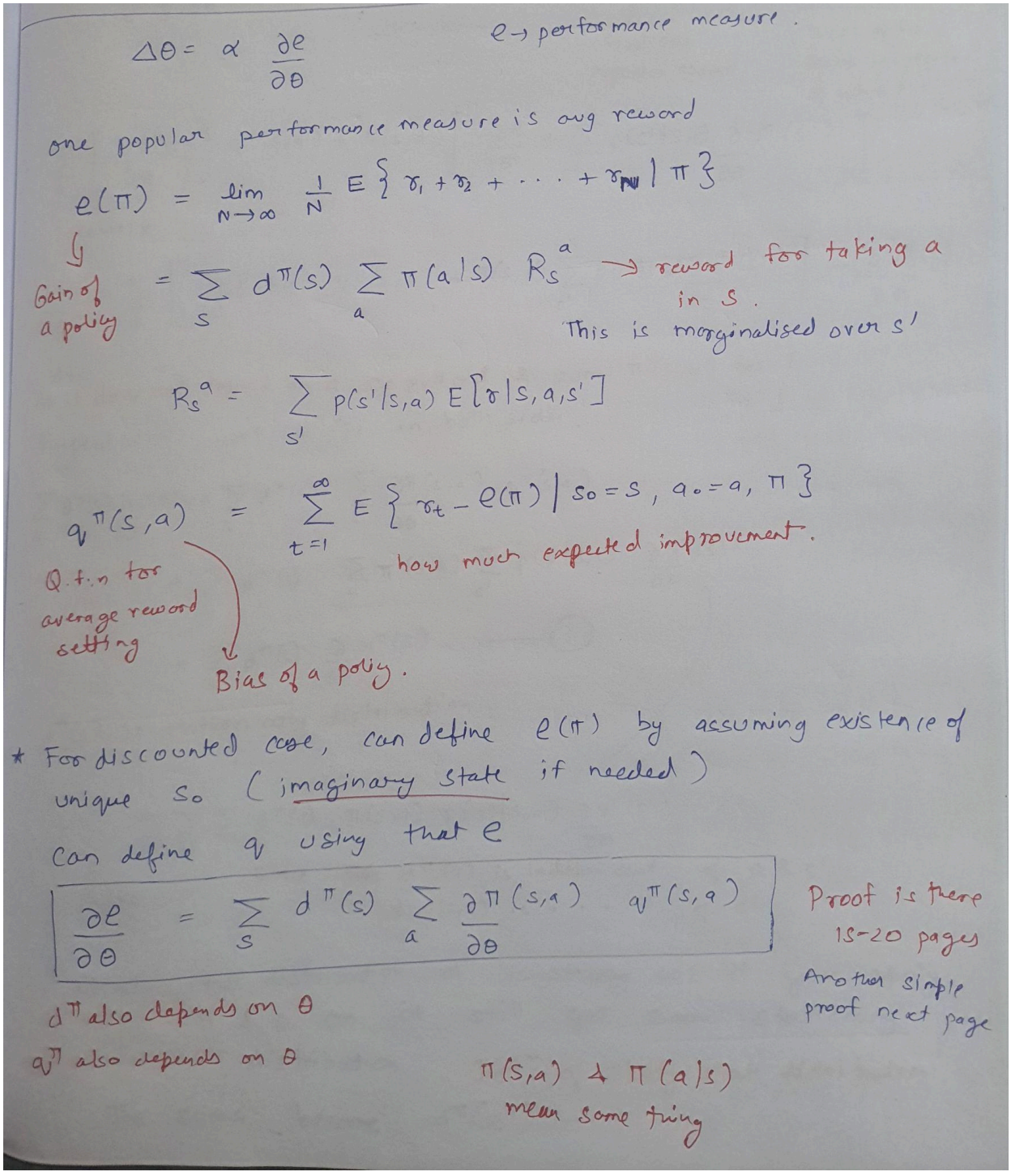
For value, update ~~neg~~ state regardless of action.



Motivation for actor-critic:

1. Generalizing from REINFORCE baseline
2. Reduce variance of PG

REINFORCE (cont'd)



Proof

$$v^\pi(s) = \sum_a \pi(s,a) \cdot q^\pi(s,a)$$

$$\frac{\partial}{\partial\theta} v^\pi(s) = \sum_a \left[\frac{\partial \pi(s,a)}{\partial\theta} q^\pi(s,a) + \pi(s,a) \frac{\partial}{\partial\theta} q^\pi(s,a) \right]$$

$$= \sum_a \left[\frac{\partial \pi(s,a)}{\partial\theta} q^\pi(s,a) + \pi(s,a) \frac{\partial}{\partial\theta} \left[R_s^a - e^\pi(s) + \sum_{s'} p(s'|s,a) v^\pi(s') \right] \right]$$

(R_s^a: does not depend on θ; first step expected reward; rest of the terms in term of value fn)

$$= \sum_a \left[\frac{\partial \pi(s,a)}{\partial\theta} q^\pi(s,a) + \pi(s,a) \left[\frac{-\partial e}{\partial\theta} + \sum_{s'} p(s'|s,a) \frac{\partial v^\pi(s')}{\partial\theta} \right] \right]$$

Rewrite.

$$\frac{\partial e}{\partial\theta} = \sum_a \left[\frac{\partial \pi(s,a)}{\partial\theta} q^\pi(s,a) + \pi(s,a) \sum_{s'} p(s'|s,a) \frac{\partial v^\pi(s')}{\partial\theta} \right] - \frac{\partial v^\pi(s)}{\partial\theta}$$

by as it does not depend on a so pull out & sum $\pi(s,a)$ to 1

Expectation wrt $d^\pi(s)$ on both sides.

$$\sum_s d^\pi(s) \frac{\partial e(\pi)}{\partial\theta} = \sum_s d^\pi(s) \cdot \sum_a \frac{\partial \pi(s,a)}{\partial\theta} q^\pi(s,a)$$

$$+ \sum_s d^\pi(s) \cdot \sum_a \pi(s,a) \sum_{s'} p(s'|s,a) \cdot \frac{\partial v^\pi(s')}{\partial\theta}$$

$$- \sum_s d^\pi(s) \frac{\partial}{\partial\theta} v^\pi(s) \quad \text{—— (1)}$$

$d^\pi(s)$ is stationary distribution.

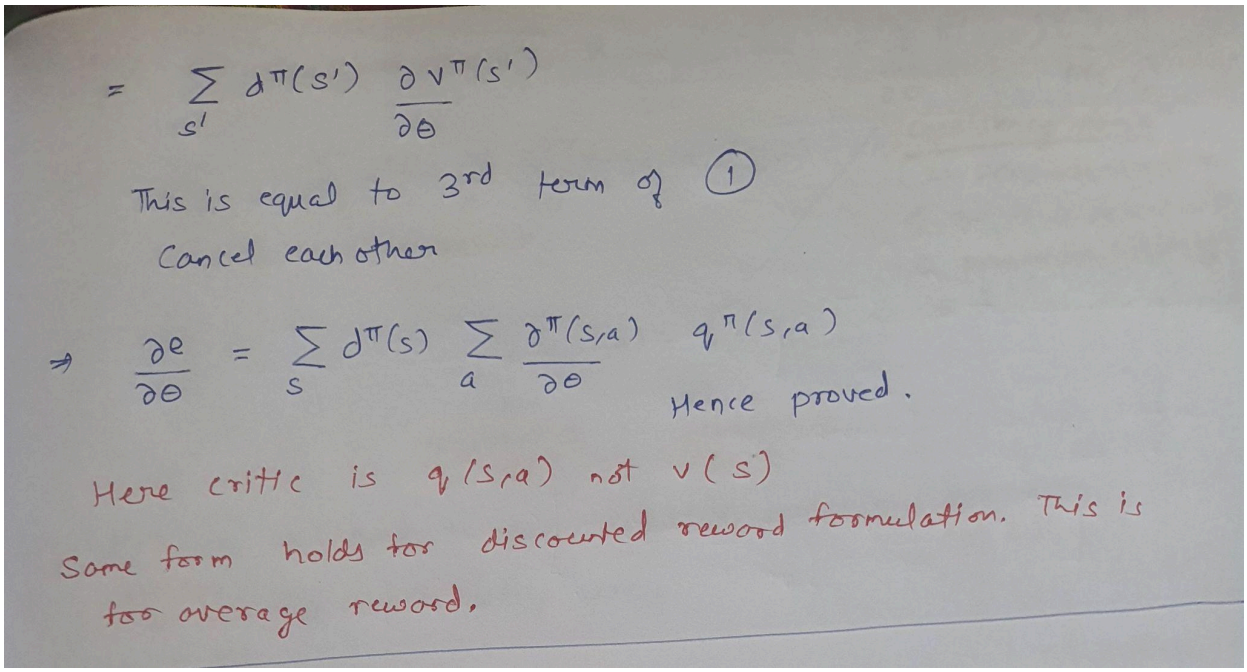
$$\sum_s \sum_a \sum_{s'} d^\pi(s) \cdot \pi(s,a) \, p(s'|s,a) \frac{\partial v^\pi(s')}{\partial\theta}$$

since ~~this~~ $d^\pi(s)$ is independent of a & s'

& $\pi(s,a)$ is ..... of s'

Steady state distribution d^π is steady coz if you apply π & p on it still get some stationary distribution. Defn of steady state distribution.

The sums become $d^\pi(s')$



Connected PG approach to Q function

Policy Gradient with Function Approximation

Following the derived (exact) gradient estimate ensures convergence to local optima
Now approximation for q as we don't know it exactly

θ: params of policy

w: params of q_π

when can we guarantee convergence ?? (while approximating)

$f_w(s,a) = \hat{Q}^\pi(s,a)$ approximation of Q

$\frac{1}{2}\left[q^\pi(s,a) - f_w(s,a)\right]^2$ error squared.

Gradient of error wrt w

$\Delta w \propto \left[q^\pi(s,a) - f_w(s,a)\right] \frac{\partial}{\partial w} f_w(s,a)$

opposite dir of gradient so adjusted that in '-' sign.

How often this change is made ??

as often as we pick (s,a) so introduce $\pi(s,a)$

Running π for long time ends in $d^\pi(s)$

so

$$\sum_s d^\pi(s) \sum_a \pi(s,a)\left[q^\pi(s,a) - f_w(s,a)\right] \frac{\partial}{\partial w} f_w(s,a) = 0 \quad (2)$$

sum to get total change in wts.

At convergence total change goes to zero.

(to local optimum)

Theorem If f_ω satisfies (2) & $\frac{\partial f_\omega(s,a)}{\partial \omega} = \frac{\partial \pi(s,a)}{\partial \theta} \frac{1}{\pi(s,a)}$

then $\frac{\partial \rho}{\partial \theta} = \sum_s d^\pi(s) \sum_a \frac{\partial \pi(s,a)}{\partial \theta} f_\omega(s,a)$

Similar to prev result

consistency cond^n (ω parameterization of q_ν is consistent with θ parameterization of π)

$$\frac{\partial f_\omega(s,a)}{\partial \omega} = \frac{\partial \ln(\pi(s,a))}{\partial \theta}$$

gradient of q_ν — gradient of $\ln \pi$

Conclusion → just need to get relative ordering correct (of actions)

This is similar to Shivaram sir's intuition for PG

f_ω can be very wrong

Eg $$\pi(s,a) = \frac{e^{\theta^T \phi_{s,a}}}{\sum_b e^{\theta^T \phi_{s,b}}}$$

$\phi_{s,a} = \phi(s,a)$ feature vector representing (s,a) pair

$$\frac{\partial f_\omega(s,a)}{\partial \omega} = \phi_{s,a} - \sum_b \pi(s,b)\, \phi_{s,b}$$

$\Rightarrow$

$f_\omega(s,a) = \omega^T \left[\phi_{s,a} - \sum_b \pi(s,b)\, \phi_{s,b}\right]$

make it linear.

representation used for value f:n is some linear combination of ϕ's which are representation of policy

Basically tells that if policy parameterization is poor then Q-parameterization should be poor too

Also, if policy parameterization is rich, q_ν-value f:n representation should be equally nuanced.

mean of ~~repro~~ feature vector for state.

$$\Rightarrow \frac{1}{|A|} \sum_a f_\omega(s,a)\, \pi(s,a) = 0$$ How??

This guarantees that just plug in value f:n (approximate) for value f:n (actual) & still get same result.

Can guarantee convergence to local optima using value function approximation under consistency condition
All this when gradients are available (differentiable)
Hypothesis: can do linearly only while relating consistency condition
Not known for non-linear

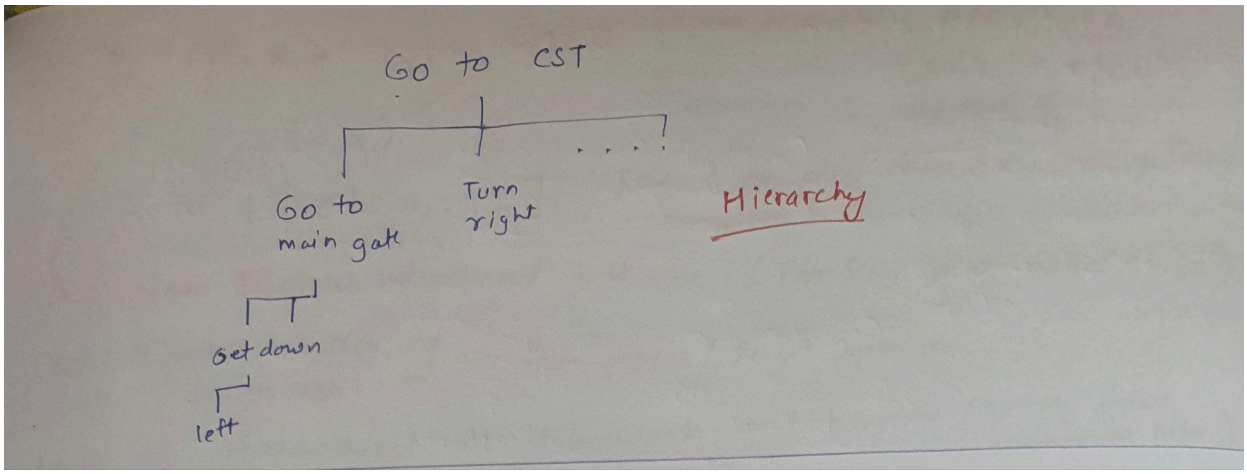
Week 10

Hierarchical Reinforcement Learning

Reasons for function approximation

1. Large spaces
2. Generalization

Another way of scaling up problems: find structure in problems and exploit that (function approximation also does that (in terms of repetition of states/actions))
Higher order structure: subproblems to be solved over and over again (eg. Navigation)
Humans naturally break long horizon problems into subproblems
Temporal abstraction: breaking over time (subproblems might take different amount of time)
Can simply combine solutions of subproblems to get final solution



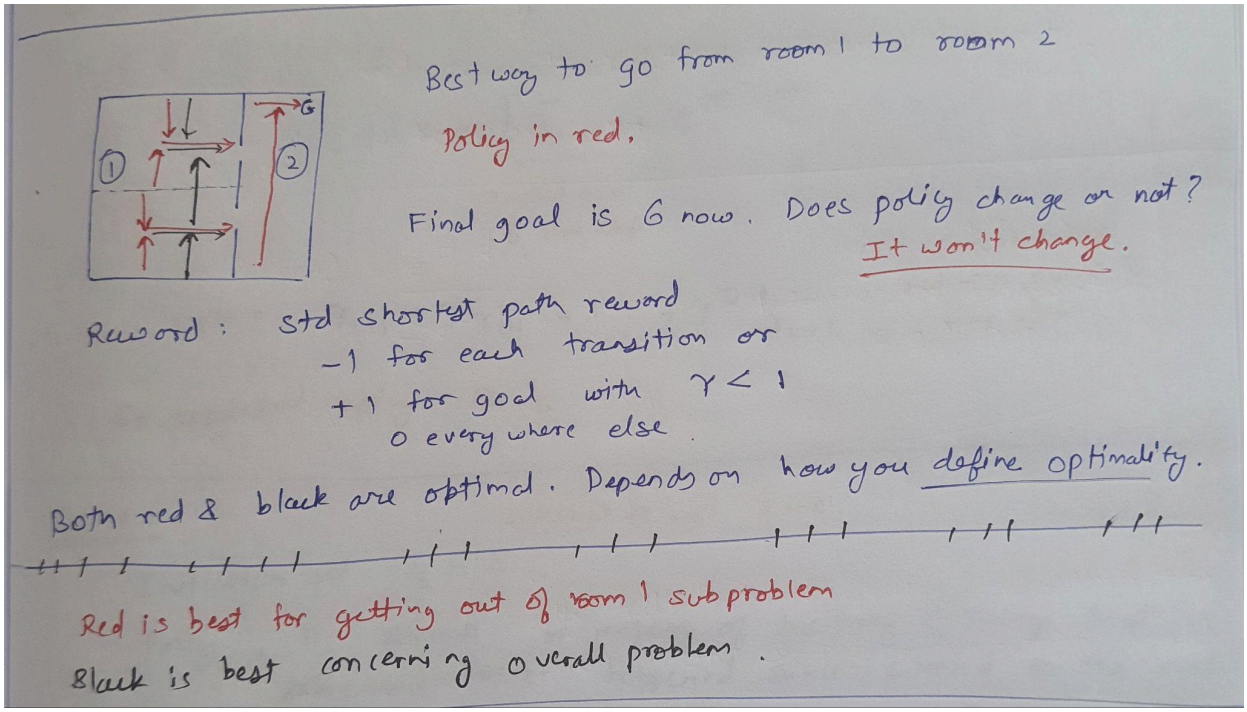
Learn solutions that can be used over and over again (like we learnt walking long back in childhood)
Can **optimize reusable solutions** (need not focus much on optimizing subproblems that are rarely used)
Lifetime value of policy
Cuban cigars, Sportsperson
Humans don't optimize things that aren't that often repeated
Transfer/Reusability: reuse solutions of already solved subproblems
HRL also gives meaningful modules to optimize
Planning comes up with operators to be applied in order to reach a goal state from a start state

These operators have pre-conditions which need to be satisfied in order to apply operator (eg. table should be empty to put laptop)
Post-conditions: thing that hold after operator is applied (eg. table no longer empty after putting laptop)
Post-conditions becoming true might make some other operators feasible
Planning community is happy to find ‘a’ solution (it’s a hard combinatorial problem)
They do hierarchical planning (subproblem discovery) but assume that operators, preconditions, postconditions, goals etc. are given to them. Just have to execute them.
Why **learn** at all? Why not just planning?
Learn in order to optimize
More powerful/”meaningful”
State abstraction: essentially phi’s
Optimal solution of getting out of building (assuming one path) does not depend on where I want to go afterwards
Breaking into subproblems allow focusing only on things needed to solve subproblems
Advantages

1. **Fewer parameters** (going to main gate compared to possible state space of going to station: so many extraneous variables and can be hard to abstract them away): do not need variables corresponding to highway while exiting building
   Sum instead of product (like FML): 2^3 + 2^4 < 2^6
2. Can not **reuse** bigger solution for simply reaching maingate as it will have variables corresponding to highway and all: better to break it down into subproblems and can reuse solution of subproblems

How to split in subproblems? Split in such a way that we can reuse (can ask domain expert or use points of inflection in state space: door is like a funnel state)
Finding such funnels (or bottlenecks) in geometric navigation problems is easy but not so in other problems like Atari
Distribution of subproblems: sample from them and solve them (still many subproblems which are actually not solved)
Real question is can we do it by looking at just one trajectory rather than solving multiple trajectories

Types of Optimality



Hierarchically optimal: search through policies that respect this hierarchy but need not solve each component optimally (some policies might be prohibited: if going through on specific point is defined in hierarchy then can't take shortcuts escaping that specific point)
Not asking what is the best way to solve each subproblem but what is the best way of solving overall problem
Recursively optimal: not only respect the hierarchy but also individual components are solved optimally and then put together to get best combination
Flat optimality: does not respect hierarchy
Usually flat optimality gives more optimal solution than hierarchical optimality which in turns gives better than recursively optimal
If enough computation power, simply use flat optimality (like having big vector of all features of highway and all even when just going outside building)
Flat optimality is **hard to achieve**
Advantages of recursive:

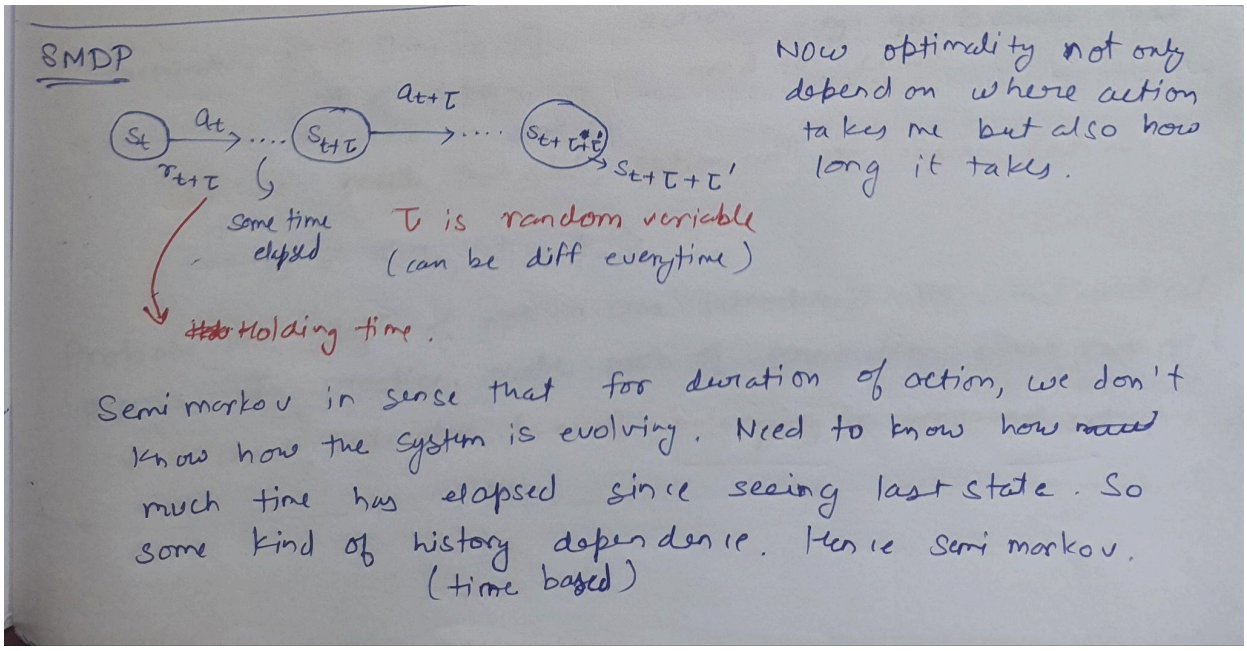
1. Easy to transfer/reuse
2. Don't have to worry about TD error from solving big problem propagating to subproblems

Although a structure is needed to put these individual subsolutions together
When to go for hierarchical optimality?

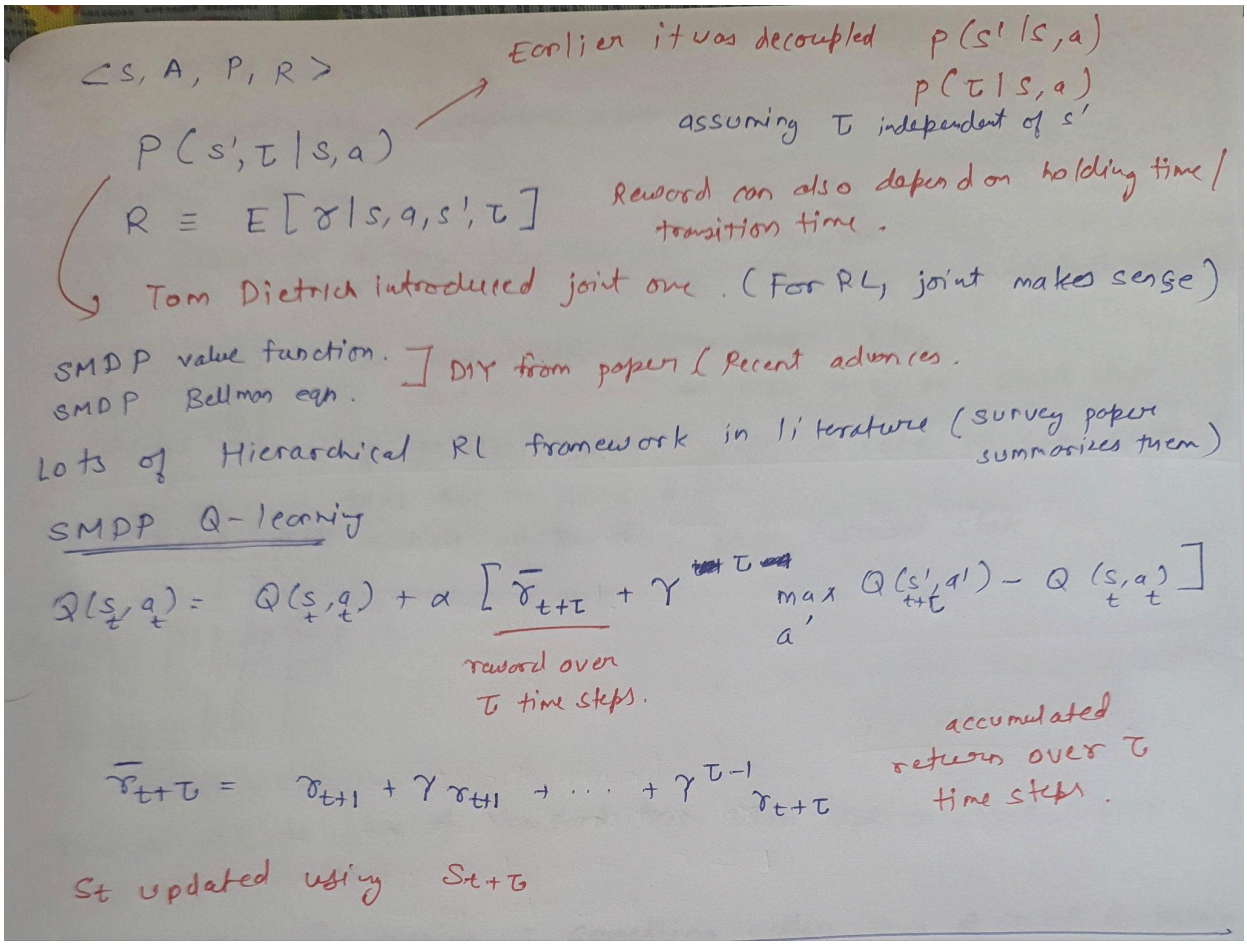
1. Solving same big problem multiple times
2. Recursively optimal policy is really bad (eg. laser in mid of room 2 with extremely negative reward)
   In this case flat optimal would be to go in room 1 if below laser in room 2 then go back to room 2 from above door

Semi Markov Decision Processes

The infrastructure needed for recursively optimal policy is SMDP
In SMDP, actions have **duration.**



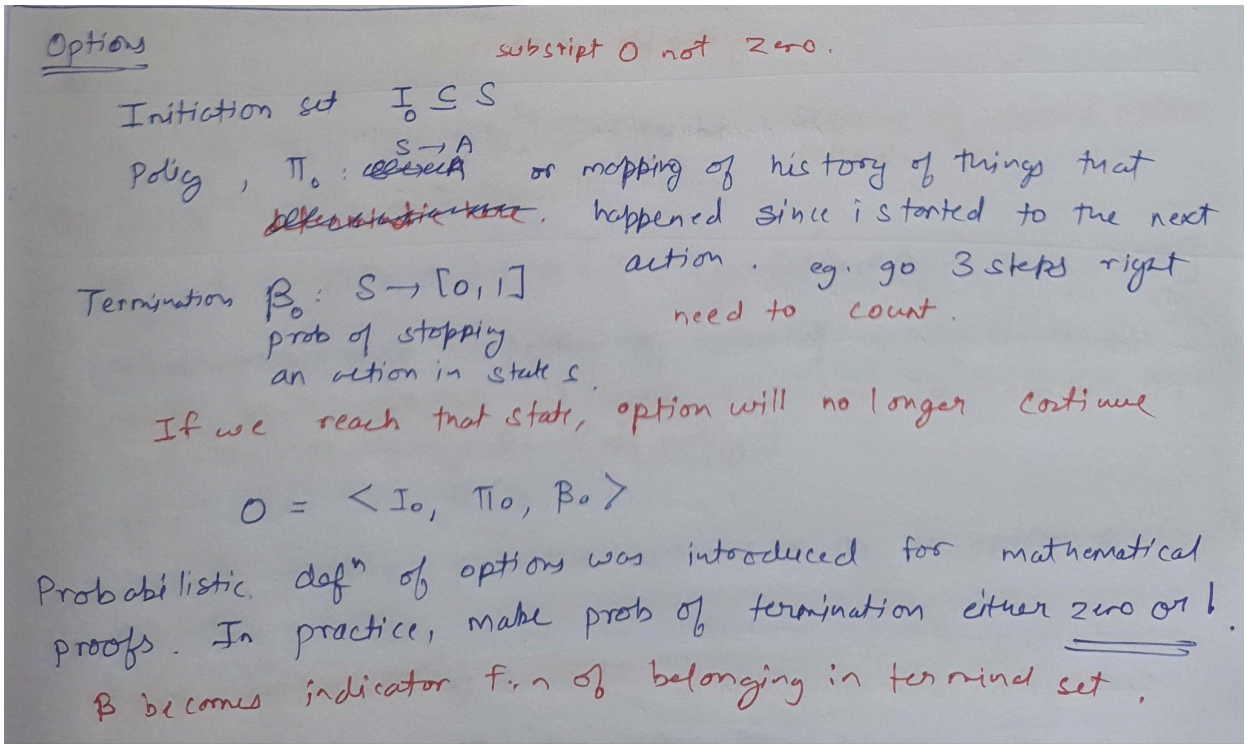
Ideally, tau and s' should have joint distribution but decoupled for simplicity
Delay coz could have sensed market at some time and by the time placed an order, 15 seconds have elapsed
Transition time term more suitable in RL compared to holding time
Holding time (pick action at time t and wait for tau seconds to apply it: holding time is still a function of decision taken) was introduced to get away with how our action affects environment
In RL, we need that.



SMDP Q-learning does not look into the structure of a_t. Assumes that reward comes somehow when a_t is taken. A_t can be go to room 2 (not a simple action hence temporal abstraction) Can take different time from different states to go to room 2 (also can be different from same state if there is stochasticity in environment: move left but wind moves us to right)

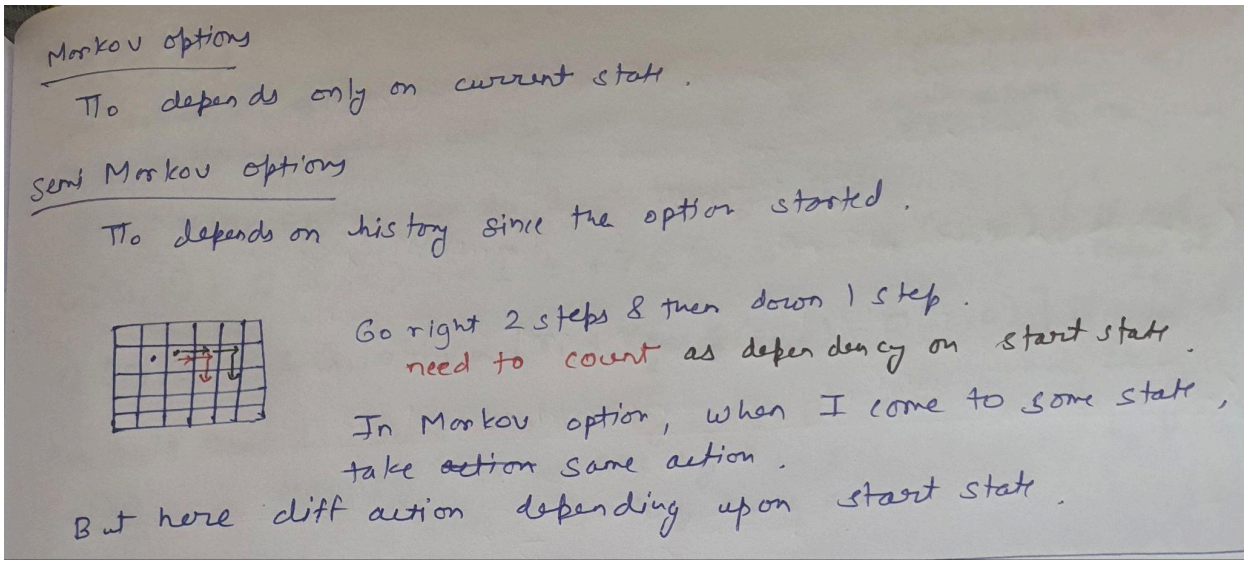
Options

It is a framework for Hierarchical RL: Options is simplest
Encapsulate solutions of subproblems as a single action (idea comes from planning community: macroactions)
In RL, we have stochastic world: things can go wrong using high-level macroactions
What else to specify to get macroaction right? Specify a policy
Option is essentially an **encapsulated policy**
Need to specify where macroactions are applicable (where to start and where to end)
Macroaction needs initiation set, policy to use while executing macroaction and termination



2 kinds of options:

1. **Markov options:** Policy is defined only on current state (no need to worry about where the option was started)
2. **Semi Markov options:** sequence of states since the action was executed

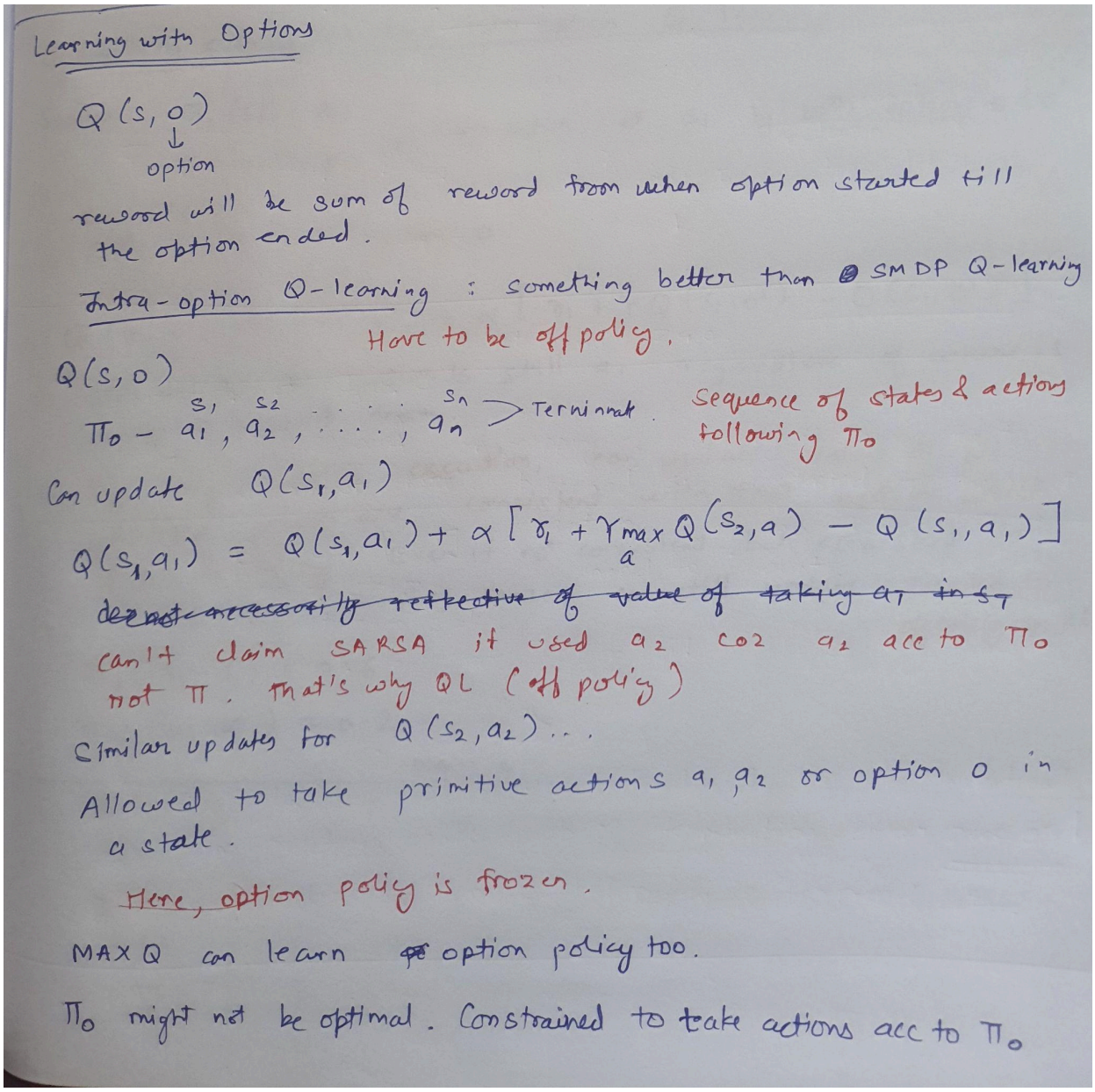


Imagine a 1000*1000 gridworld where an option is defined like go in a specific direction for 15 steps

Define options that make it **more efficient to explore**

Learning with Options

SMDP Q-learning
Defined options on top of MDP
These options are treated as durative actions
MDP + Options give SMDP



$Q(s_2, o)$

can update if option is Markov → SMDP reward should sum from r_2 not r_1

In semi Markov, policy will be diff coz of diff start states.

For Markov options, can do boots trapping

$$Q(s_1, o) = Q(s_1, o) + \alpha [r_1 + \gamma Q(s_2, o) - Q(s_1, o)]$$ if not ended at s_2

$$Q(s_1, o) = Q(s_1, o) + \alpha [r_1 + \gamma \max_a Q(s_2, a) - Q(s_1, o)]$$ if ended at s_2

Can make these 2 updates into 1 by using β

Multiply first ~~eqn~~ RHS by $1 - \beta(s_2)$ & second by $\beta(s_2)$ & sum it up. to get one eqn.

Something even better than intra option Q-learning

Suppose $\pi_o(s_1) = a_1$] same action at s_1 by both options o & o'

$\pi_{o'}(s_1) = a_1$

Can update $Q(s_1, o')$ if taken o.

$$Q(s_1, o') = Q(s_1, o') + \alpha [r_1 + \gamma Q(s_2, o') - Q(s_1, o')]$$

Still do this coz action is still a_1 regardless of where it was suggested from.

By single option execution, can update value of all options which are consistent with that option which I am executing. Even if not consistent but consistent in importance sampling sense, can still update. This led to off policy in RL.

$\pi_o(s_1, a_1) = 0.6$

$\pi_{o'}(s_1, a_1) = 0.2$

Didn't do it

Can implement very efficient option learning algos but need to come with options first (Option discovery is still an active area of research)

Coz of simplicity, options are the most popular hierarchical RL framework

Some caveats: not entirely clear where pi_0 is defined no clear specification of admissible states that the option will visit (sometimes system dynamics may cut terminal states from initiation states)

For an outsider, it might look like non markov or stochastic even if deterministic policy is getting executed (due to different options)
The fact that option that we are executing is part of internal state is never made explicit in options framework: left to implementor's perception

Hierarchical Abstract Machines

Second framework apart from options (Third is MAXQ)
Proposed before the other two frameworks by Parr and Russell
HAM is not adopted that widely although very elegant and full of potential
Use simple Finite state machines to encode policies that we are trying to follow
Hierarchy of FSM

HAM

FSM will have 4 types of states.

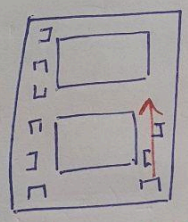
(1) Action state : takes an i/p & emits action
(2) Call state : takes an i/p & call another machine
(3) Choice state : makes non deterministic transition to 2 or more states (in same machine)
(4) Stop state : stops & returns control back to machine that called it.

There is a starting machine (top level machine)
I/p in machine → states of MDP.
O/p primitive actions in action state.
call state has no o/p.

Hierarchy through calls (make sure no loops)
DAG
connection b/w machines.

Eg Problem : very complex gridworld.

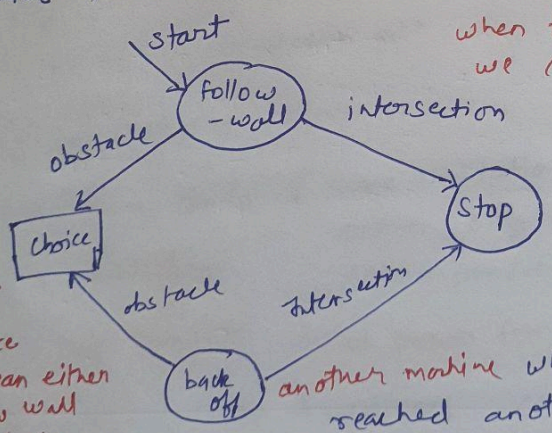
Hierarchy coz once learn to avoid one obstacle can transfer.



navigate one corridor

obstacles in corridor.
follow the wall ~~of obstacle~~ to get out or backup & go back the way I came & turn in arbitrary direction & walk.

Machine for this.



when follow-wall machine returns we can be either in obstacle state or intersection state

go back to previous machine as it decides what to do next

In choice state, can either say follow wall or back off or stop.

another machine when it returns, we either reached another obstacle or reached an intersection.

choice state allows to go to any 3 states (can limit to follow wall or back off)

Primitive actions mostly at lower level machines.
(These machines had no action states)

follow-wall & back-off are call states

Where is SMDP here? What should we learn?

Choice state need not be machine. Could be another action state too. (transitions)

Need to learn what choice I should make?

In options, states of SMDP were same as original MDP. actions of SMDP were actions + options.

In HAM, need to have base MDP states (02 ~~not~~ have to learn a policy that is a f:n of where I am in the world.

But action also depends on what machine I am in & who called it. Stack of calls.

This call stack is also part of state space

States of SMDP = states of base MDP + call stack.

Actions → transitions possible out of choice state. (remember choice state can transition to ~~primitive states~~ action state too which o/p primitive actions)

H_i → call stack M_i → machine state.

S_i → core MDP state.

State space = $H_i \times S_i \times M_i$ ~~(Cartesian pdt with Mi too)~~
cartesian pdt

C_i → o/c of choice states.

Action space = $\bigcup_i C_i$ (some limitations as can take a particular action only if I am in a choice state corresponding to that machine).

~~Mi~~ ~~Ci~~ ~~set of actions~~ ~~choices~~.

Read paper for clarity of notation.

Since, we know states & actions, can use SMDP Q-learning.

Whenever I go to call or choice state, no transition in core MDP state. only Mi & Hi change. Only time core MDP state changes is if I visit action state in any of the machine.

A machine can exist without choice state (suppose policy is known already)
HAM can be used to learn recursively optimal policy as well as hierarchically optimal policy
In options, this question does not arise since hierarchy is frozen: both become equivalent
Have to assume whatever policy is given in options as optimal
Equivalent of already learned policy (option) in HAM: any sequence of action states visited without any choice state in between can refer to a policy (or execution of policy)
Since there is no choice, no learning so kind of fixed and given to us just like options
HAMs can be used in partially observable state problems too
These FSMs can be used to define more complex policies: non Markov (FSM with memory)

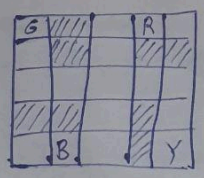
Week 11

MAXQ

Original name was MaxQ Value Function Decomposition
Decompose problem into set of subtasks

MAXQ

Taxi



obstacles.

pickup & put down passengers (only at places R, G, B, Y)

-1 reward for every move

+5 for every passenger successfully delivered. (suppose)

As you drop a passenger, randomly another passenger pops up at another location.

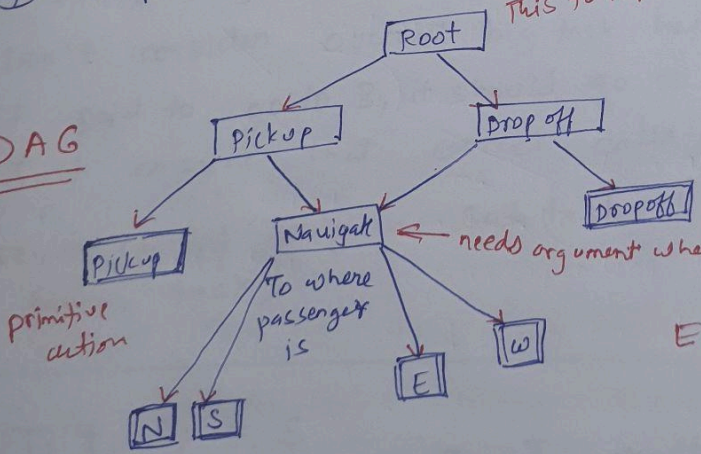
-2 if try to pick up passenger from a location where there is no passenger.

-5 if drop passenger to some other location. (try to as it won't get down)

Repeated structure:

1. Go to passenger location
2. Pick up
3. Go to destination
4. Drop down

DAG



This to repeat again & again

like a call graph.

needs argument where to navigate to

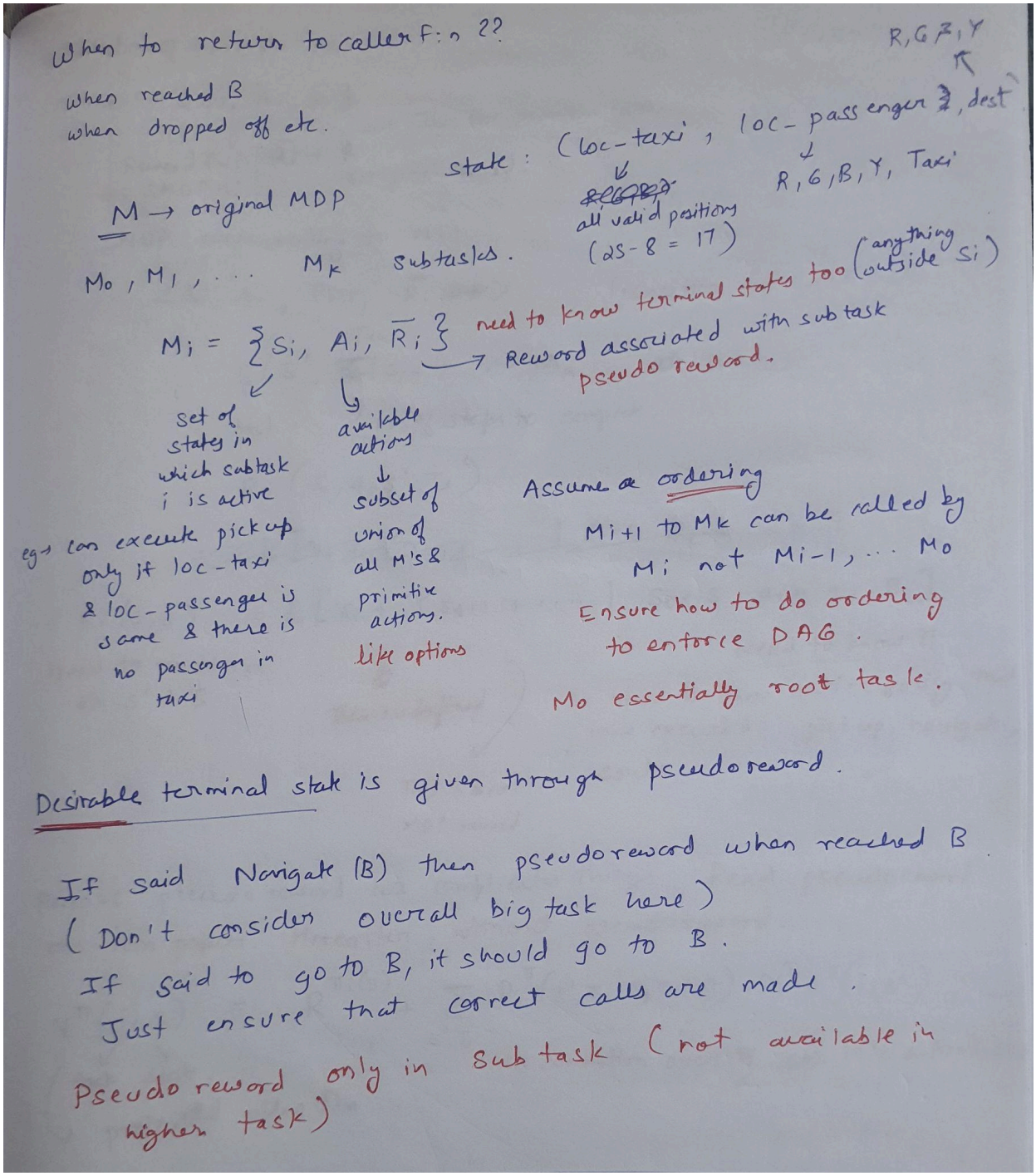
primitive action

To where passenger is

Easier than HAM.

Navigate can have 4 subtasks: Navigate (R) Navigate (G) Navigate (B) Navigate (Y)

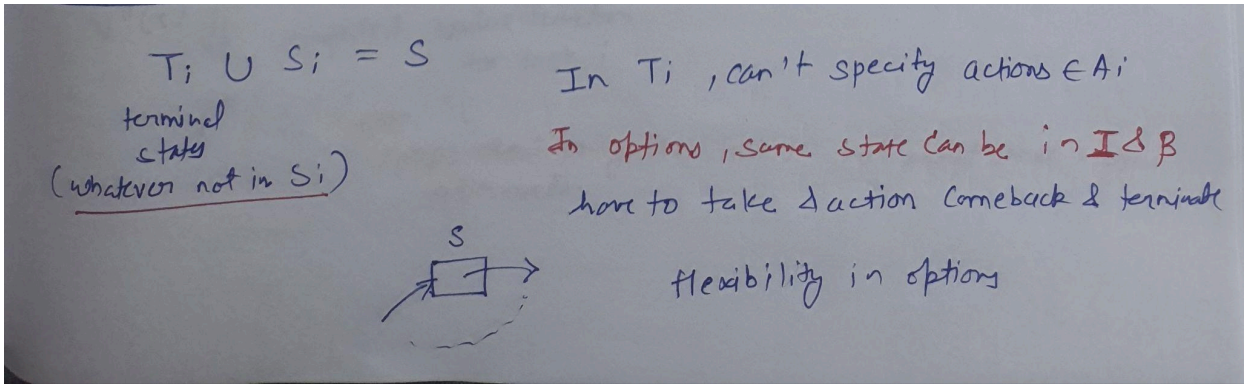
can treat ~~diffco~~ differently as small world so can't do transfer of learning.



In HAM and options, even though hierarchical but learning on a single MDP
In MaxQ, learning on a collection of SMDPs M_0, M_1, … , M_k
Trying to collectively learn a solution for all the SMDPs in one shot
Pseudo reward R_i available only in M_i but there is a reward R (corresponding to original task M) available in all M_i

MAXQ Value Function Decomposition

In options specify, policy of options.
In HAMs, specify the structure of the machine.
In MaxQ, specify S_i, A_i and R_i for each subtask



Collection of SMDP M_i

$M_i = \{S_i, A_i, \bar{R}_i\}$

→ This for primitive action

Reward fn: $\bar{R}_i \to R$ Original reward.
for SMDP M_i

SMDP corresponding to M_i

$\langle S_i, A_i, P_i, \bar{R}_i \rangle$ Transition req joint distr.

$P_i(s', \tau | s, a) \sim M_a$ ath subtask.

task i ↳ no. of steps to complete.

$R_i(s, a, s', \tau)$

$= R_i(s, a) = E[r_t + \gamma r_{t+1} + \cdots | S_t = s, a_t = a, \pi]$

hard to condition on s' & τ

This is defined

This is a core MDP reward.

Need to know π

need to know entire policy that was executed: pickup, navigate, south.

Remove pseudo reward coz complicates things. Read pseudoreward one from paper. Hereafter, without pseudoreward.

$$V^\pi(i, s) = R^{\pi_i(s)} + \sum_{s', \tau} P_i^\pi(s', \tau | s, \pi_i(s)) \cdot \gamma^\tau V^\pi(i, s')$$

No summation over a coz it's deterministic

(task, state. projected value fn)

$V^\pi(s)$ ⇓ projected value function for task i

$V^\pi(i, s)$

cares about only the task i not about what happens afterwards.

$R^{\pi_i(s)}$: reward over π_i's subtask

$$R^{\pi_i(s)} = V^{\pi}(\pi_i(s), s)$$

$$V^{\pi}(i,s) = V^{\pi}(\pi_i(s), s) + \sum_{s',\tau} P_i^{\pi}(s', \tau \mid s, \pi_i(s))\, \gamma^{\tau}\, V^{\pi}(i, s')$$

(subtask)

$$Q^{\pi}(i,s,a) = V^{\pi}(a,s) + \sum_{s',\tau} P_i^{\pi}(s', \tau \mid s, a)\, \gamma^{\tau}\, V^{\pi}(i, s')$$

(subtask; state)

$$= V^{\pi}(a,s) + \underbrace{\sum_{s',\tau} P_i^{\pi}(s', \tau \mid s, a)\, \gamma^{\tau}\, Q^{\pi}(i, s', \pi(s'))}_{C^{\pi}(i,s,a)\ \text{completion function}}$$

accumulated reward in subtask i after completing action a starting from s.

↳ This is c - before is captured by $V^{\pi}(a,s)$

$$Q^{\pi}(i,s,a) = V^{\pi}(a,s) + C^{\pi}(i,s,a)$$

~~$V^{\pi}(a,s) = Q^{\pi}(i,s,a) - C^{\pi}(i,s,a)$~~

$$V^{\pi}(a,s) = \begin{cases} Q^{\pi}(a, s, \pi_a(s)) & \text{if } a \text{ is a "composite"} \\ \sum_{s'} P(s' \mid s, a)\, E[r \mid s, a, s'] & \text{if } a \text{ is a primitive.} \end{cases}$$

one of the subtasks.

recursively. (kind of hierarchical)

$\pi = \{\pi_0, \pi_1, \ldots \pi_k\}$ corresponding to M_0 to M_k

π_0 call some action below it

$$V^{\pi}(0,s) = Q^{\pi}(0, s, a_1)$$
$$= C^{\pi}(0, s, a_1) + V^{\pi}(a_1, s)$$

suppose a_1 ie. $\pi_0(s) = a_1$

Now go to a_1 (composite action) & first action in a_1 is a_2

$\pi_1(s) = a_2$

$\pi_2(s) = a_3$

$V^\pi(a_1, s) = Q^\pi(a_1, s, a_2)$

$= V^\pi(a_2, s) + C^\pi(a_1, s, a_2)$

Recursion till wit a primitive action

$V^\pi(0, s) = C^\pi(0, s, a_1) + C^\pi(a_1, s, a_2) + V^\pi(a_2, s)$

Further down the hierarchy, can reuse completion function.

Breaks down value f:n into chunks of rewards which I am likely to see again & again. Each chunk corresponds to executing a subtask.

No pseudo rewards.

~~Completion f:n only gives rewards after~~

When go to sublevel, completion function says. $C^\pi(a_1, s, a_2)$ says only too completing a_1. Don't care what happens after a_1, But in reality, if want to learn Something hierarchically optimal, need to know what is happening after

need to know what's happening next in order to decide what door to use.

Modification → 2nd completion f:n too whole problem.

Tricky to maintain this for entire structure.

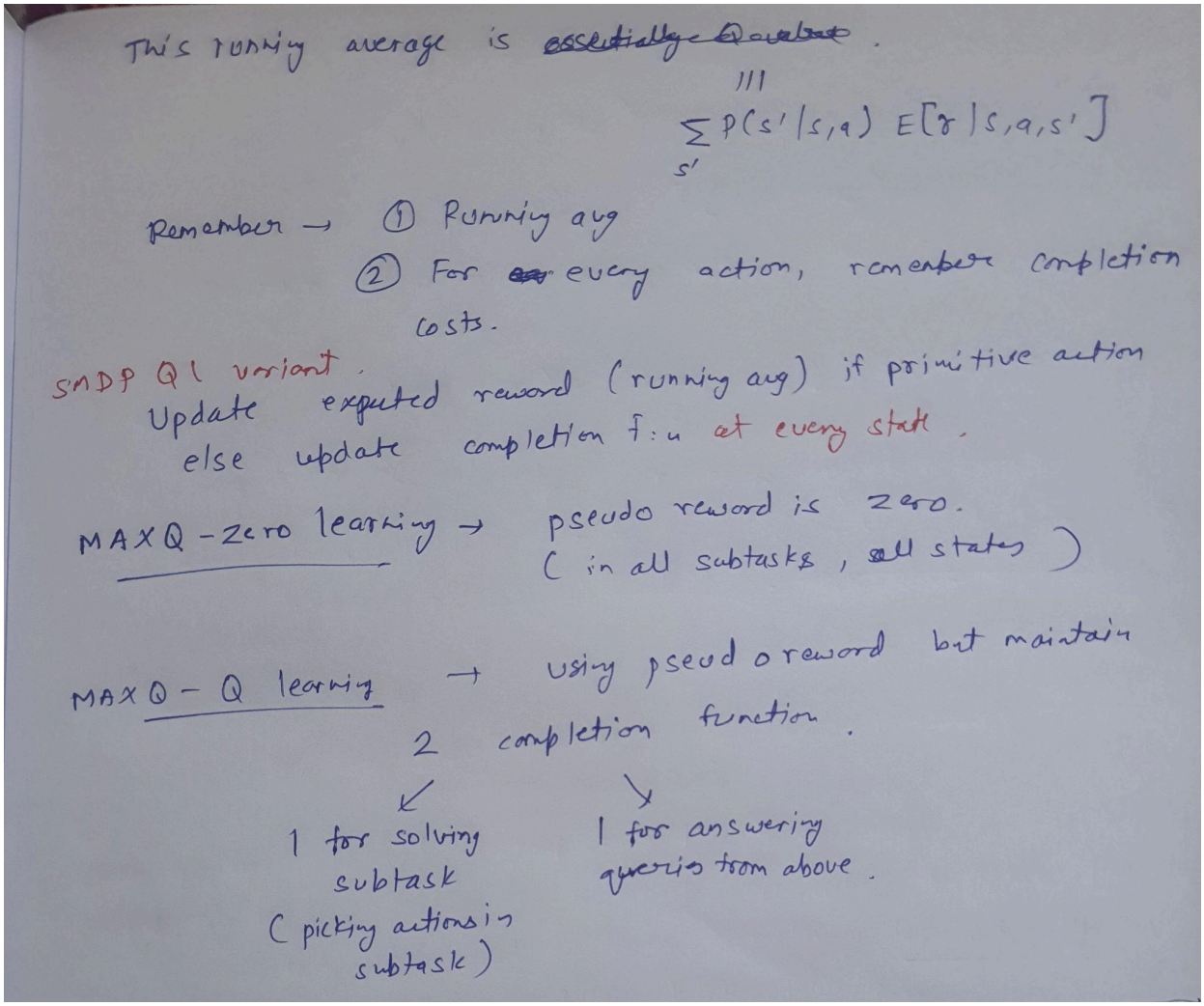
bookkeeping.

SMDP QL variation : at every subtask update Q-value for subtask as well as completion f:n.

go down using V recursively till primitive action but

$\sum_{s'} p(s' \mid s, a)\, E[r \mid s, a, s']$ is not known

Maintain running average & return it when hitting primitive action. Just need to store running avg



Different value function in a later paper which said Dietrich one uses projected value function so cannot learn hierarchically optimal policy

Option Discovery

Discovery becomes harder with increased complexity of hierarchical architecture
Good option: reusable, cuts down on exploration (aids in learning faster), transfer learning
Above are hard ways to evaluate goodness of options
Use surrogate measures: bottlenecks (access states: allow access to some other part of MDP)
Why bottlenecks: cut down on exploration, reusable, transfer learning
How to find bottlenecks: depending on information about the problem
If given transition function, segment MDP and find graph cuts (components of graphs very weakly connected)
Those states where weak connections happen will be **bottleneck states**

1. Graph partitioning ideas if transition function is available (at least a good model of MDP)
2. If transition function is not available:
    a. walk around in MDP, collect data and construct graph and then partition it

b. **Diverse density:** Assume that generated many trajectories while solving the problem in MDP (successful trajectories) and many trajectories you have aborted (unsuccessful trajectories: did not reach the goal)
   Look at states that appear lot on successful trajectories but don't appear a lot on unsuccessful trajectories

3. **Betweenness**: Node has high betweenness if lots of shortest paths pass through it
Strong correlation between betweenness and bottlenecks
Can define bottlenecks through betweenness: **high betweenness implies bottlenecks**
Other centrality measures like degree centrality need not necessarily yield bottlenecks

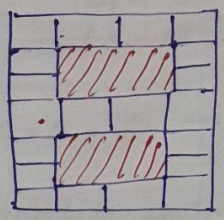
Bottlenecks states give us Beta ie terminal states
But appropriate start states and policy to follow in options need to be determined
In diverse density can use experience replay to learn Q function and hence policy
Random options (with length sampled from a distribution) in small problems cuts down exploration time significantly
Small world options make best use of given (fixed amount) data
Should help in transfer learning

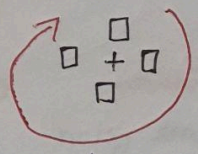
Week 12

POMDP Introduction

Till now assumed that every timestep, agent has access to complete state information
Gave up on some information to learn compact policy in navigation problem but that loss was not much as the features ignored were not relevant
Still had complete information just chose to ignore it
Sometimes, complete information is not even available
Eg. inside cubicle know that we are in a cubicle but not know which cubicle we are in
Perceptual aliasing: refining info through multiple sources
Partial observability

POMDP





can observe 4 cells.

If all clear then 0, 0, 0, 0

No 0, 0, 0, 0 here.

Possible states

0 0 0 1	1
1 0 1 0	6
0 1 0 1	8
1 0 0 1	1
1 1 0 0	1
0 1 1 0	1
0 0 1 1	1
0 1 0 0	1

Input is 4 digits (binary)

If observe 0101 then can't say which of these 8 cell we are in.

Initially 1/8 prob.

telling going North & seeing 0101 ~~prob~~

reduced to 4 possibilites Now 1/4 prob

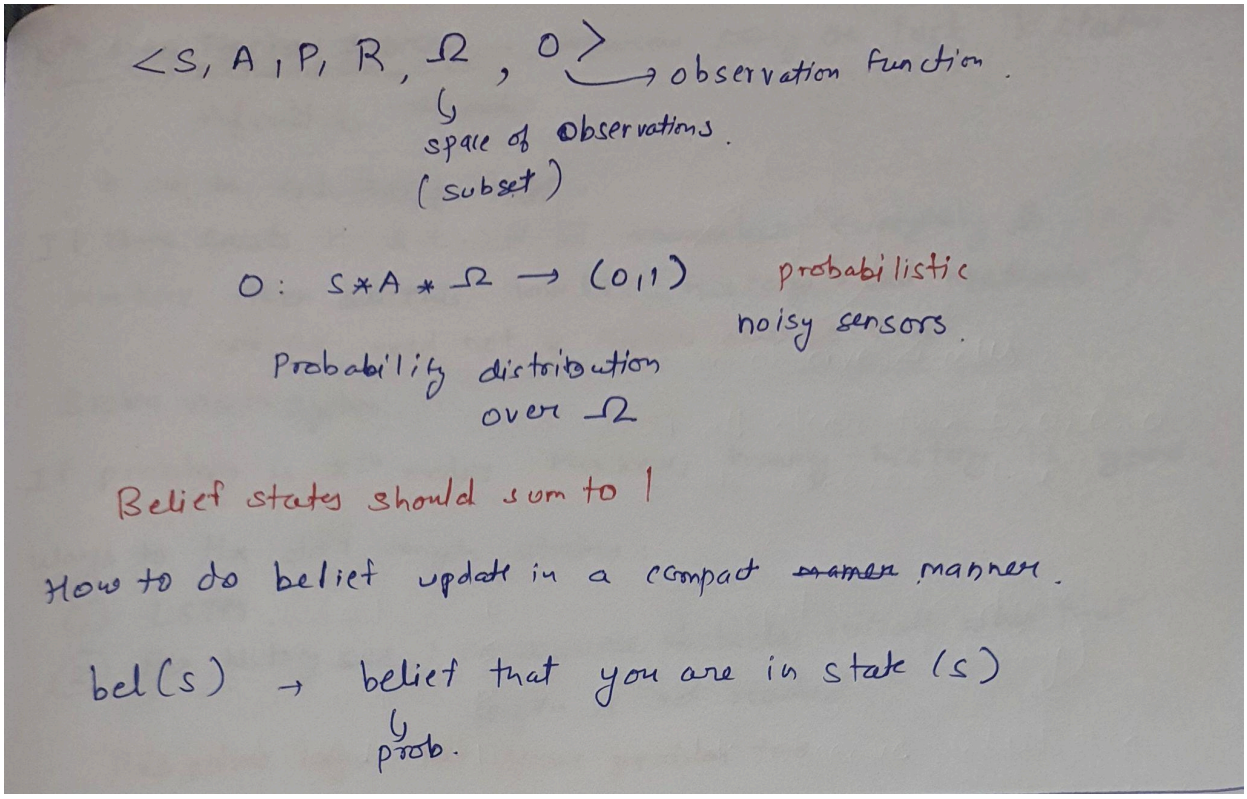
go north again & see 1001 (top left one)

reduced to 1 Now 1 prob

History helps in reducing uncertainity about where we are now.

Sequence can be uniquely identifiable even tho individual observations might not be.

One way of tackling partial observability is to use some amount of history.



With more observations, belief of being in a state keeps getting more and more focused
Belief state
Assuming MDP with underlying states but not able to observe states of the MDP. Can make only those observations which are some function of the states. Based on this, we form a belief about where I am in the state space.
Standard model for partially observable problems is POMDP (not necessarily needed in all partially observable problems tho)
POMDP = MDP + additional component to handle partial observability

Solving POMDP

Different ways of solving partially observable problems
All methods apart from history based methods assume that you know the POMDP

1. **History based methods**: use history directly (other methods also use but not directly) This does not use belief

History based methods

<0101>

<0101, 0101>

<0101, 0101, 1001>

state involves history

remember whole history,

history can involve action too

<s, a, s, a, ...> here north & south help in prev example.

Issues → ① large state space

② in parameterized space, states have different lengths.

③ wastage of computation.

sequence models for modeling history LSTM etc.
HMM etc.

* Don't want current state to have more info than by remembering history. Both ways. neither less nor more.

k^{th} order Markov system → depends only on last k states.

default is 1st order.

k can be arbitrarily large.

If there exists k s.t. if I remember everything, it is Markov then all this works. (History based methods)

need not be Markov always true.

Broken vision system.

If problem is k^{th} order Markov, having history is good.

Ways to fix diff length states:

1. LSTM.
2. Fix history size: can assume obstacles initially when that length is not reached.

This solves large state space problem too

Still wastage of computation: sometimes some steps of history are not needed. (localizing perfectly).

Some places need history of length 2, some places 3 etc.

Need not be uniformly k^{th} order Markov.

Once decided to be k^{th} order Markov, stuck with keeping k size window of history. relearning even two those states not required ⇒ wastage

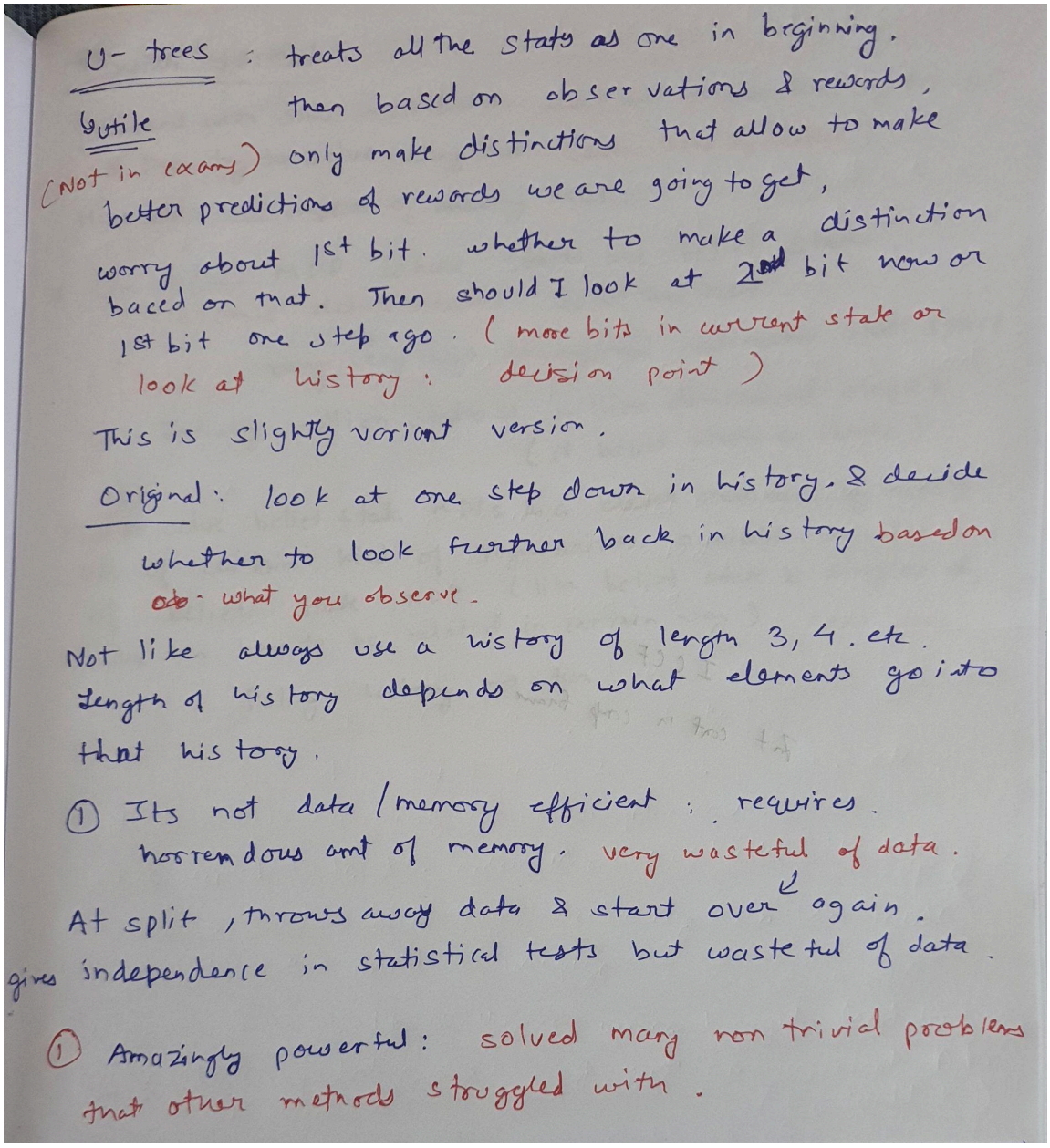
Determining k is itself challenging: wrong k might lead to wasteful computation (too large k) or missing out on important info (too small k)

Adv of history based methods ⇒ 1. very easy to use.

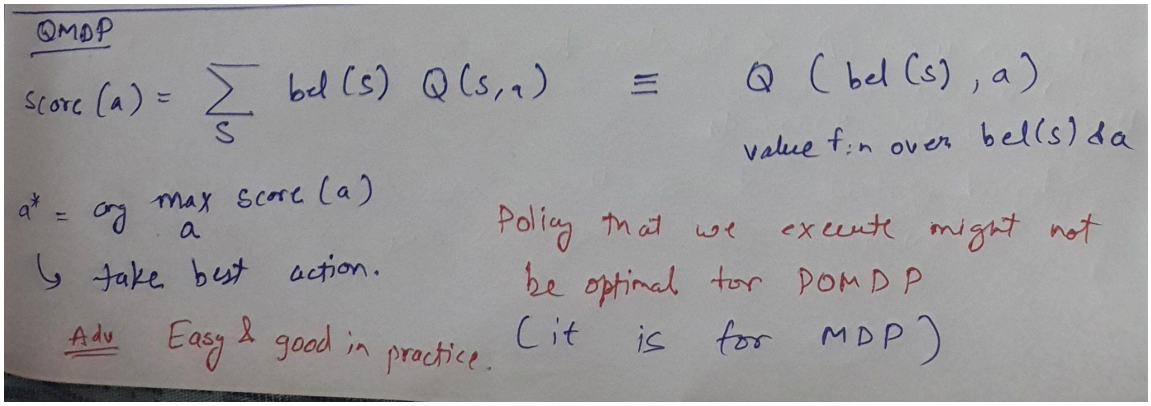
No belief state update & all

2. If MDP happens to have small k, history based methods are quite good at solving problems.

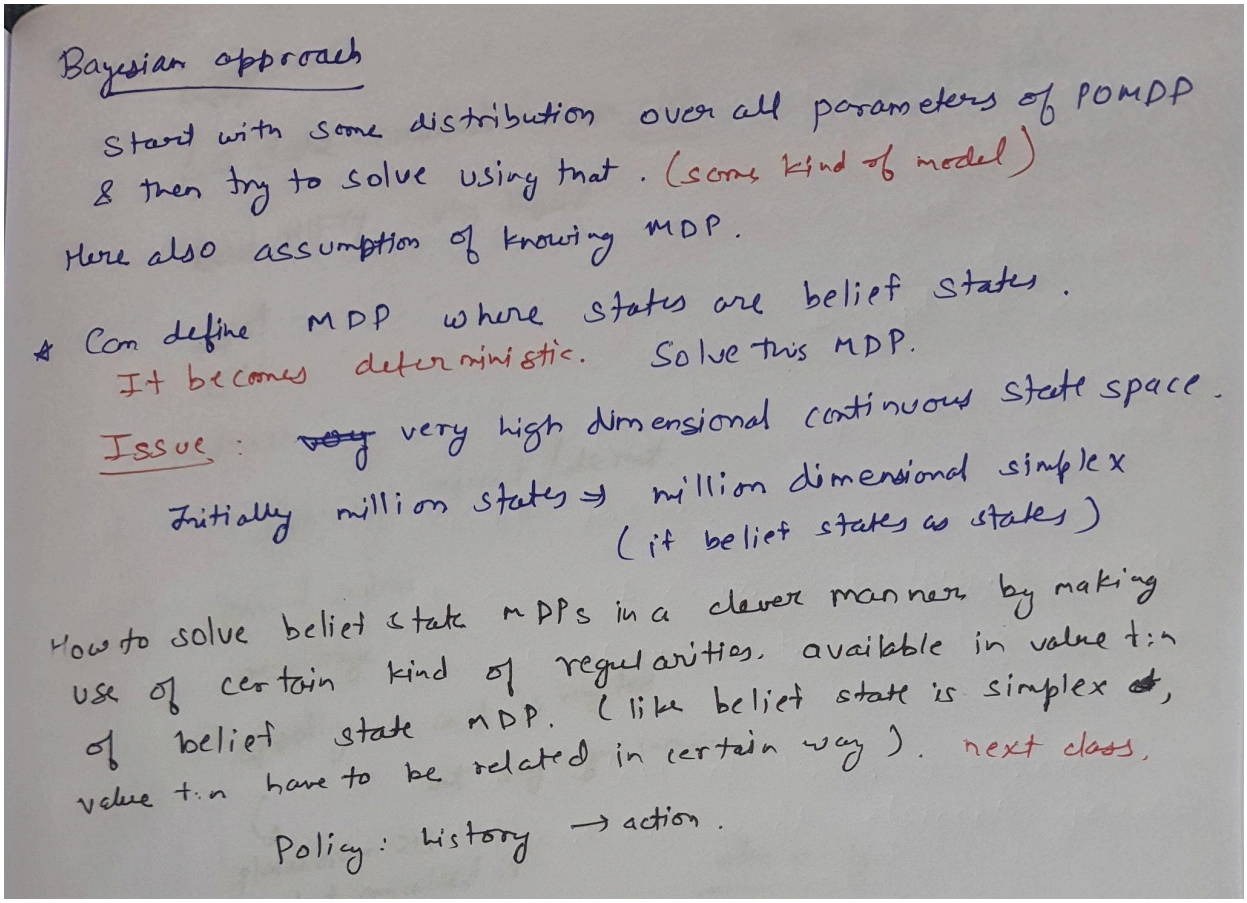
U trees are a type of history based method



2. Q-MDP type approach: assume that MDP (S,A, P, R) is available. Go ahead and solve it (using VI and PI) to obtain a policy. Solved using original states (MDP component of POMDP). Execution of policy is a problem.



Bayesian approach:



3. Directly solving POMDP: 2 classes of algorithms
    a. Assumes the model
    b. Bayesian approach: assume distribution and update it
    c. Non Bayesian approach when model is not known: Frequentist approach
4. Predictive state representations (PSR)
   We need history to make better predictions about the future (where I will end up). So, instead of looking at history, focus on how well we can predict the future.

We make certain assumptions about the dynamics of the world and predict where we might end up in future in a probabilistic fashion. Eg. prob (north, clear, north, clear, north, wall)
PSR consist of a set of **tests** and the probabilities that the tests are true
Don't assume that there is an underlying MDP like the POMDP does.
If build PSR on top of POMDP, they are as powerful as direct POMDP solving methods
PSR can go way beyond regular POMDP settings as they don't have any notion of underlying state space.
Hard to learn PSR
History based methods are easy
Can learn linear PSR from which we can recover probability of any test using linear combination
Linear PSR will be large (same as number of states)
Finding non linear PSR is incredibly hard