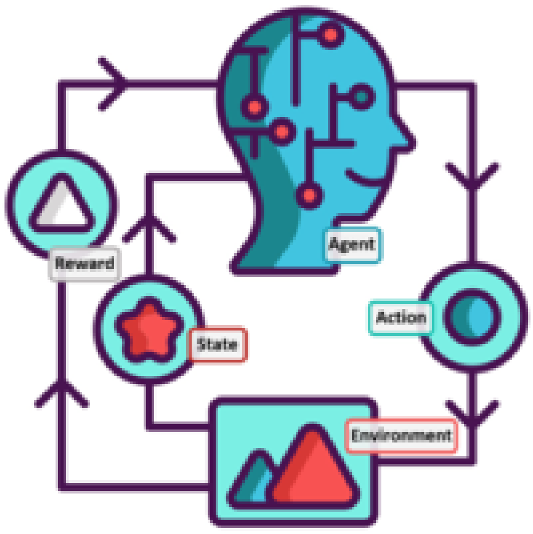
SOS Midterm Report

CS22: Reinforcement Learning

Abhilasha Sharma Suman

23B1011



Contents

1 Introduction to Reinforcement Learning

Learnt the core idea behind reinforcement learning basic definitions like agent, policy, reward, environment, history, state, et cetera.

→ The **agent** is the "learner" or the part of our enterprise that interacts with the environment and takes the decisions based on these interactions and changes it's behaviour depending on the feedback of the environment.

→ The **environment** is the simulation or the model of the world which the agent interacts with. It gives the agent information about the state it is in and provides it with a reward based on the action taken by the agent.

→ **Action** is the output (the decision) that the agent takes based on the input (information) given to it by the environment. The set of all possible actions is called the Action space.

→ **History** (H_t) is the sequence of all observations, actions, and rewards that an agent experiences over time in its environment.

→ The **reward** is feedback the environment provides to the agent based on how good of an action the agent took. This is the "reinforcement" based on which the agent alters it's policy by strengthening a behaviour that gets more favoured feedback from the environment and tends to not repeat the one that doesn't.

→ The **model** is the agent's replica or simulation of the environment which tries to predict the feedback of the environment in the action it will take (basically estimate the value of the actions in the action space).

→ The **return** is the total accumulated reward that an agent aims to maximize over time. It represents the sum of rewards received over a sequence of time steps in case of episodic tasks and as discounted sum of successive rewards in case of continuing tasks. The only difference is for episodic tasks T is finite and γ is 1 and for continuous tasks T is infinite and $\gamma \epsilon [0, 1)$

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{\text{t+k+1}}$$

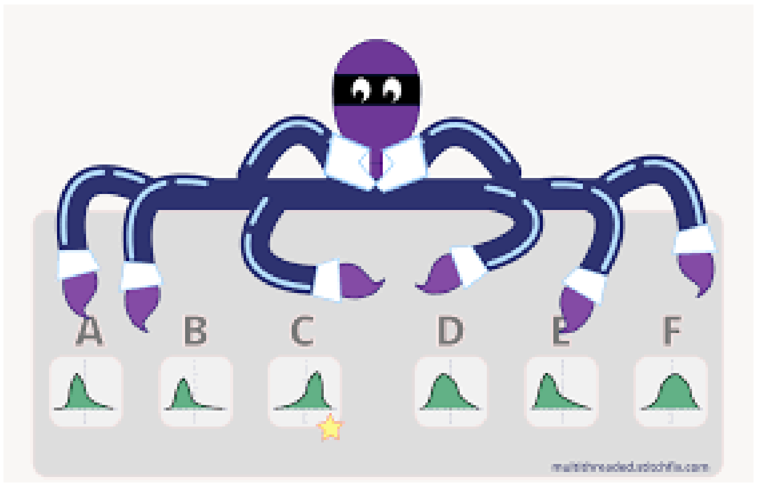
→ **Policy** of an agent defines it's behaviour. A policy $\pi(a, s) = \mathbb{P}[A_t = a | S_t = s]$ thus gives the probability distribution over actions given the present state of the process.

→ **Value function** is a function to approximate the goodness of being in a particular state (State Value function) or taking a particular decision (Action Value function, $q_\pi(s, a)$).

2 Multi Armed Bandits Problem

2.1 Key Ideas

Problem Statement: In a casino, you have in front of you n slot machines. Each of them has a certain possibility of winning but you are not aware of this probability distribution. How do you play to maximise your reward in a specific number of trials- do you keep playing a single machine or do you play a combination of different machines?



→ The key idea behind this problem is to introduce the balance between exploration (trying out unknown actions to find any hopefully find better prospects and maximise your reward) and exploitation (using your knowledge to keep getting a good reward). Too much exploitation might lead to losing out on possibly more rewarding actions and too much exploration might lead to never losing rewards due to trying out too many different possibilities. There are a few approaches to this problem, the ones that I went through are described below.

→ ϵ-Greedy method: This method aims to strike a balance between exploration by following a exploitative strategy with $1-\epsilon$ probability and exploring rest of the trials. This way, with the optimal ϵ, the agent can exploit the known facts about the environment and sometimes can explore for better prospects and then exploit that to make up for the potential reward lost during exploration which didn't lead the agent to a better (more rewarding) action.
To see the work done on ϵ-greedy method, click here.

→ Upper Confidence Bound (UCB method): This method tries to balance the exploration-exploitation conundrum by leveraging the uncertainty in the accuracy of the action-value estimates when using a sampled set of rewards to satiate the need for exploration.
Given $Q_t(a)$ represents the current estimate for action a at time t and $N_t(a)$ is the number of times action a has been taken till time t,

$$A_t = argmax_t(Q_t(a) + c\sqrt{\frac{ln(t)}{N_t(a)}}$$

To see the work done on UCB method, check this out!

3 Markov Decision Processes

3.1 Markov Property

.The point of any RL problem is to ultimately find the best possible policy and get the maximum possible reward. Although it seems to be limiting to just limit the agent to try to maximise the reward it has proven to be very adaptable and widely applicable.

The future of a Markov process is dependent solely on present state and not the entire history that has occurred before it (this condition is the **Markov property**). Thus a Markov process can be represented as a tuple $< S, P_{\text{ss'}} >$ where S is a finite set of states and $P_{\text{ss'}}$ is a state transition probability matrix (gives the possibility of going to a state s' from present state s). In a usual case the state dynamics that is $p(s', r|s, a)$ would have depended on the entire sequence of states,actions and rewards but in case of a Markov signal it depends on the present state and action only. Even for a non-Markov signal we can think of such a state as an approximation of a Markov state.

Due to the state being the sole basis for future rewards and selecting actions we need to select the definition of state accordingly. Using the dynamic described above we can compute anything we wish to know about the environment, such as

the expected reward,

$$r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r\epsilon\mathbb{R}} \sum_{s'\epsilon\mathbb{S}} p(s', r|s, a)$$

the state transition probability,

$$r(s,a) = Pr\{S_{t+1} = s'|S_t = s,\ A_t = a\} = \sum_{r\epsilon\mathbb{R}} p(s', r|s, a)$$

here $\mathbb{R}$ is the set of all possible rewards.

3.2 State and Policy value evaluation

Most RL problems involve computation of value function of states or state-action pair to estimate how good it is for an agent to be in a state or to choose to perform a particular action.
We defined policy to be a mapping from each state s and action a to the probability of the agent choosing action a when in state s. For MDPs, we define value of a state under a policy as follows,

$$v_\pi(s) = \mathbb{E}_\pi\{G_t \mid S_t = s\} = \mathbb{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k R_{\text{t+k+1}} \mid S_t = s\}$$

Similarly we define the value of taking action a when in state s under policy π as $q_\pi(s,a)$ or the action value function for policy π as follows

$$q_\pi(s,a) = \mathbb{E}_\pi\{G_t \mid S_t = s,\ A_t = a\} = \mathbb{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k R_{\text{t+k+1}} \mid S_t = s,\ A_t = a\}$$

3.3 Bellman Equations

→ Bellman's Expectation Equation

Bellman's expectation equation, often referred to as the Bellman equation for Markov Decision Processes (MDPs), is a fundamental concept in reinforcement learning and dynamic programming. It provides a way to calculate the expected value of being in a certain state and taking a certain action, incorporating both immediate rewards and the expected future rewards.

A fundamental property of value functions that we use throughout RL and DP is that they satisfy particular recursive relations. For any policy π and any state s, the value $v_\pi(s)$ can be dissociated into the immediate reward of following said policy and the discounted value of the successor state in the following manner: $\mathrm{v}_\pi(s) \;=\; \mathbb{E}[G_t|S_t = s]$

$$\begin{aligned}
=> v_\pi(s) &= \mathbb{E}_\pi\{\textstyle\sum_{k=0}^{\infty}\gamma^k R_{\mathrm{t+k+1}} \mid S_t = s\}\\
=> v_\pi(s) &= \mathbb{E}_\pi\{R_{t+1} + \textstyle\sum_{k=0}^{\infty}\gamma^k R_{\mathrm{t+k+2}} \mid S_t = s'\}\\
=> v_\pi(s) &= \mathbb{E}_\pi\{R_{t+1} + \gamma v_\pi(s')\}\\
=> v_\pi(s) &= \textstyle\sum_a \pi(a|s)\ \sum_s' \sum_r p(s', r|s, a)[r + \gamma\mathbb{E}_\pi\{\sum_{k=0}^{\infty}\gamma^k R_{\mathrm{t+k+2}} \mid S_t = s'\}]\\
=> v_\pi(s) &= \textstyle\sum_a \pi(a|s)\ \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]
\end{aligned}$$

The final equation is the Bellman's Expectation equation for v_π. It gives us a relation between the value of the present state and that of it's successor states. The value function v_π is the unique solution to its Bellman equation. Similar to the state value function, $Q_\pi(s, a)$ can be dissociated into immediate reward and discounted successive rewards. So state-action value function's Bellman equation is

$$q_\pi(s, a) = R_s^a + \textstyle\sum_{s'\epsilon\mathbb{S}}\ P^a_{\mathrm{ss'}} \sum_{a\epsilon\mathbb{A}} \pi(a'|s')q(s', a')$$

here $\mathbb{A}$ is the action space.

These can also be represented in matrix form. for state value function,

$$v = R + \gamma P v$$
$$R = (I - \gamma P)v$$
$$v = (I - \gamma P)^{-1}\ R$$

Here, v is the matrix containing the state values of all states under policy π, γ is the discount factor, P is the state transition dynamics/ probability matrix.

The time complexity for the standard matrix inversion algorithm is $O(n^3)$ which makes direct solution possible only for small MDPs. For larger MDPs we use algorithms such as Dynamic Programming, Monte Carlo Evaluation and Temporal Difference Learning.

→ Bellman's Optimality Equation

Optimal state value function $v_*(s) = \max_\pi\ v_\pi(s)$
Optimal state-action value function $q_*(s, a) = \max_\pi\ q_\pi(s, a)$
An MDP is solved when the optimal state value function is known which basically means that the best policy is found. Bellman's Optimality equation helps us find this by recursively relating the optimal value functions.

$$v_*(s) = \max_a q_*(s, a)$$
$$q_*(s, a) = \mathbb{R}_s^a + \gamma \textstyle\sum_{s'\epsilon\mathbb{S}} P^a_{ss'} v_\pi(s')$$

here $\mathbb{S}$ is the set of all possible states.

The optimality equation is a non linear equation which means that we won't have a closed form solution in general cases. The possible iterative solution methods to solving Bellman's optimality equation are value iteration, policy iteration, Q-Learning and SARSA (won't be covering this under SoS).

4 Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process

(MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. In this discussion, we will assume that the environment is a **finite MDP** only.

4.1 Policy Evaluation

First we consider how to compute state value function $v_\pi$for an arbitrary policy π. We now know that

$$v_\pi(s) \;=\; \textstyle\sum_a \pi(a|s)\; \sum_{s',r} p(s',r|s,a)[r+\gamma v_\pi(s')]$$

If the dynamics of the environment are known to us then this equation is a system of $|\mathbb{S}|$ simultaneous linear equations in $|\mathbb{S}|$ unknowns. or our purposes, iterative solution methods are most suitable. Consider a sequence of approximate value functions v_0, v_1, v_2, each mapping $|\mathbb{S}^+|$ to R. The initial approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π as an update rule:

$$v_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s]$$
$$v_{k+1} = \textstyle\sum_a \pi(a,s) \sum_{s',r} p(s',r|s,a)[r+\gamma v_k(s')]$$

Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation assures us so. This sequence of v_k can be shown to converge to v_π as $k \to \infty$ under the same conditions which guarantee the existence of v_π. This is the **iterative policy evaluation algorithm**. To produce these successive approximations, iterative policy evaluation applies the same operation to each state s: it replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated. We call this kind of operation a full backup. All backups done in DP are full backups due to all of them being based in all possible successor states rather than a sample successor.

4.2 Policy Improvement

The point of calculating value function for a policy is to find better policies. Suppose we have determined the value for an arbitrary deterministic policy. For some state, we would like to know whether or not we should change the policy to deterministically choose an action that wouldn't be chosen under the current policy. We know how good it is to follow the current policy from this state but how would it be to change to a new policy? The value of this way of behaving is

$$q_\pi(s,a) = \textstyle\sum_{s',r}[r+\gamma v_\pi(s')]$$

Maybe it is better to follow policy π' just for this state and then continue to stick to policy π? The key criterion to check is whether this is greater or lesser than $v_\pi(s)$. If it is greater that is, if it is better to select a once in s and thereafter follow than it would be to follow all the time then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall.
This is true under a special case of a general result, the policy improvement theorem. If the

state-action value of a alternate policy π' is greater than or equal to the state value under policy π for all possible states then the said alternate policy is either as good as or better than the latter. That is, π' must garner higher returns than the policy π. In case of a strict inequality in the first case there must be a strict inequality in the second case as well. It is very natural to extend this thought process and check if changes are required for all states and all possible actions that is to select the best action according to the state-action value function at each state. In other words, we try to see if we need to consider a new *greedy policy* π' given by

$$\pi' = argmax_a \; q_\pi(s, a)$$
$$\pi' = argmax_a \; \sum_{s',r}[r + \gamma v_\pi(s')]$$

But this is the same as Bellman's optimality equation which means that v'_π must be v_*. Thus we see that policy improvement gives us a strictly better policy given that the original policy wasn't already the optimal policy.

We have discussed deterministic policies, now coming to stochastic policies. (The references that I used didn't go into detail on this but said ..) All the ideas applicable to deterministic policy can be extended to stochastic policies as well. The policy improvement theorem carries as stated for stochastic case, under the natural definition,

$$q_\pi(s, \pi'(s)) = \sum_a \pi'(a|s) q_\pi(s, a)$$

Additionally, if there are ties in policy improvement steps such as having multiple actions which achieve the maximum value at a particular state, in stochastic case we can make the choice of assigning probabilities of choosing amongst those actions rather than choosing just one of these actions. Any way to implement this would be appropriate given that the probability of choosing any sub-maximal action is zero.

4.3 Policy Iteration

After we are done using an original sub-optimal policy to improve to a better policy and then yet better policy until we cannot have a better policy than the one we have, we will have found the optimal policy v_*. This way we also obtain a sequence of monotonically improving policies and value functions. Since a finite MDP can have a finite number of policies only, this process will converge to an optimal policy in a finite number of iterations. This is **policy iteration**.
The work I did on policy iteration was going through the problem of Jack's car rentals. Click here to see.

4.3.1 Jack's Car Rentals

Problem Statement (Taken from Sutton & Barto):
Jack manages two locations for a nationwide car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited \$10 by the national company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between

the two locations overnight, at a cost of \$2 per car moved. We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is n is $\frac{\lambda^n}{n!}\exp(-\lambda)$, where is the expected number. Suppose is 3 and 4 for rental requests at the first and second locations and 3 and 2 for returns. To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the nationwide company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be = 0.9 and formulate this as a continuing finite MDP, here the time steps are days, the state is the number of cars at each location at the end of the day, and the actions are the net numbers of cars moved between the two locations overnight.

4.4 Value Iteration

A disadvantage of policy iteration is that it involves policy evaluation which itself is a computation that goes over the state set multiple times. If it is done iteratively then the converge exactly to v_π occurs only in the limit. But it so happens that we can truncate this process through multiple methods, of which one is value iteration which is a special case wherein we stop policy evaluation just after going over the state set once. It is simple backup operation which combines the policy improvement and the truncated evaluation process.

$$v_{k+1}(s) = \max_a \ \mathbb{E}[R_{t+1} + \gamma v_k(S_t + 1) \mid S_t = s, A_t = a]$$
$$v_{k+1}(s) = \max_a \ \sum_{s',r} p(s', r|s, a[r + \gamma v_k(s')]$$

That is value iteration simply turned the Bellman optimality equation into an update rule! Value iteration backup function exactly the way policy evaluation backup does.

Value iteration should theoretically take an infinite amount of iterations to converge to the optimal policy but in practice we terminate the algorithm when the change in value function drops below a certain threshold. Effectively going over a state set one time through value iteration is equivalent to a sweep by policy iteration and evaluation each.

4.5 Asynchronous Dynamic Programming

A significant drawback of DP methods is that they involve operations over the whole state set of the MDP. In case of a very large state set, even a single iteration of these algorithms might be computationally heavy.
Asynchronous DP methods are in-place DP iterative DP algorithms that are not organized in terms of systematic sweeps of the state set meaning that they back up state values in a random order depending on whichever state is available. This implies that we might be backing up some state value several times before we get to the ones that we haven't gone through yet. For accurate convergence we need to back up all state values. Asynchronous DP algorithms are very flexible in the selection of states on which backup operations need to be applied.
Naturally, avoiding sweeps doesn't necessarily mean that we can get away with less computation. it just implies that an algorithm doesn't get stuck in a hopelessly long sweep before it makes progress. We leverage this flexibility by selecting the states to which back ups will be applied. We can try to order these to let the information flow from

state to state and maybe skip states that are irrelevant to optimal behaviour. Some states also might not need to have their values backed up as often as other.

Asynchronous algorithms also make it easier to integrate computation with real time interaction. To solve a given MDP, we can run an iterative MDP as the agent experiences MDP real-time.

4.6 Generalized Policy Iteration

In policy iteration, we work with two simultaneous, interacting processes- making the value function consistent with the current policy (policy evaluation) and the other, making the policy greedy wrt current value function (policy iteration). In policy iteration these two unnecessarily alternate. In asynchronous DP methods, the evaluation and improvement processes are interleaved at an even deeper level.

Generalized policy iteration (GPI) is used to refer to the concept of letting these two processes interact irrespective of the finer details of their working. Since the value function stabilises only when it is consistent with the current policy and the policy stabilises only when it is greedy with respect to current value function, both processes stabilise only when a policy is found to be greedy with respect to it's own value function. Almost all RL processes have identifiable policies and value function with policies being improved wrt the value function which qualifies them as GPIs as well. The evaluation and iteration processes are both cooperating with and competing against each other at the same time in the sense that they both pull in opposite directions.

4.7 Efficiency of DP Algorithms

Even though they aren't the best way to go about larger MDPs but they are more efficient than many other methods to do so. If we don't harp on some fine technical details, then the worst case time complexity of DP algorithms to find the optimal policy is a polynomial in the number of states (n) and actions (m) even when the number of deterministic policies in m^n. This makes DP exponentially faster than d=any direct search in policy space.
Linear programming methods can also be used to solve MDPs, and in some cases they are more efficient than DP methods but they become very impractical at a much smaller number of states than DP methods do.
On larger state spaces, asynchronous DP methods are oft preferred. To complete even one sweep of a synchronous method requires memory and computation for every state which makes them impractical for some problems yet the problem is still solvable because relatively fewer states occur along optimal solution trajectory. Asynchronous methods and GPI variants are applicable and viable for such cases too and are much faster than synchronous methods can.

To check the work done on Gambler's problem: Click here

5 Monte Carlo Method

In this discussion we will not be assuming complete knowledge of environment. Monte Carlo methods require only experience- a few sample states, actions and rewards from actual or simulated interactions with an environment

Monte Carlo methods are ways of solving RL problems based on averaging sample returns. To ensure the availability of well defined returns, we define these methods solely for episodic tasks implying that we assume all experiences of the agent are divided into discrete episodes and that all these episodes will terminate irrespective of the agent's choice of actions.

In it's way of sampling and averaging returns for each state-action pair it is quite similar to the bandit methods we explored under MAB but the most significant distinction is that there are more states, each of which acts like a different bandit problem and all these are interrelated. This means that the return of an action in a state depends on the action taken in later states of the same episode implying that the action selections are undergoing learning making this a dynamic problem from the reference of a former state. To handle this, we extend the idea of GPI.

5.1 Monte Carlo Predictions

We start by considering Monte Carlo methods for learning the state value function for a given policy. The value of a state is the expected cumulative discounted future rewards that is the expected return from that state. The underlying idea behind Monte Carlo methods is to simply average the returns observed after experiencing the state and with more observations we can expect that the average return will converge to the actual value of the state.
Let's consider a case where we wish to estimate $v_\pi(s)$, given a set of episodes obtained by following policy π and passing through state s which we might be visiting multiple times. We could either start estimating $v_\pi(s)$ by returning the average of returns following the first visit to s (*first visit MC method*) or we could first go through all the visits and finally return the average as $v_\pi(s)$ (*every-visit MC method*). Both these methods converge to $v_\pi(s)$ when the number of visits to s approach infinity.
We also extend the idea of back up diagrams to Monte Carlo algorithm. For the estimation of v_π the root is the state node and the entire flow of transition along a particular episodes lies beneath this root. Under DP we would have included just single step transitions but Monte Carlo diagrams go on until the end of an episode. The estimates for all states are independent implying that MC methods do not bootstrap unlike DP methods.
The computational expense of MC methods is independent of the number of states which makes them particularly lucrative when the value of only a state or a subset of states is required.

5.2 Monte Carlo Estimation of Action Values

Since we don't have a model simulating the behaviour of the environment, it is useful to estimate state-action values rather than just state values which in this case are insufficient to determine/suggest the policy. The way for this pretty much like how we estimated state values using MC methods except now we will be talking about visits to state-action pairs instead of just states. These methods converge quadratically(meaning that the error decreases proportional to the square of the error in the previous iteration).

However we face a complication here- some state action pairs might never be visited. In case we follow a deterministic policy we will only see a single action being chosen over and over again in state s. This is a problem because we never experience other possibly better action state pairs for which it is imperative to know the value of all actions from a state rather than just the one we currently favour. This brings us back to the problem of maintaining exploration as we saw back in the problem of Multi Armed Bandits. For policy evaluation we must assure continued exploration. One way to go about this is to specify that episodes start in a pair of state-action pair and each such pair has a non zero probability of being chosen as the start. This guarantees that each said pair will be visited an infinite number of times and this is called the assumption of *exploring starts.*

5.3 Monte Carlo Control

To approximate optimal policies we go proceed with the same DP approach of general policy iteration wherein we will maintain both an approximate policy and an approximate value function the former of which will have to be continually improved wrt the current value function and the latter will be altered to approximate the current policy's value function. As discussed earlier, both these processes are always at juxtaposition but eventually approach optimality.
Assuming that we experience an infinite number of episodes which are generated with exploring start, the MC methods will compute each q_{π_k} exactly, for arbitrary policy π_k.

Policy improvement is done my making the policy greedy with respect to the current value function. In our scenario, we have action values which frees us of the need for a model to construct the greedy policy which we define as follows

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

We can now carry forward by constructing each π_{k+1} as the greedy policy wrt action value function under current policy.

$$\begin{aligned} q_\pi(s, \pi_{k+1}) &= \; q_{\pi_k}(s, \arg\max_a \; q_{\pi_k}(s,a)) \\ q_\pi(s, \pi_{k+1}) &= \; \max_a \; q_{\pi_k}(s,a) \\ q_\pi(s, \pi_{k+1}) &\geq \; q_{\pi_k}(s, \pi_k(s)) \\ q_\pi(s, \pi_{k+1}) &= \; v_{\pi_k}(s) \end{aligned}$$

This makes sure that whatever policy we end up getting, it either better or equally good (in which case they are both optimal policies).

For Monte Carlo policy evaluation it is natural to alternate between evaluation and improvement on an episode-by-episode basis. After each episode, the observed returns are used for policy evaluation, and then the policy is improved at all the states visited in the episode. This is the Monte Carlo ES algorithm ie *Monte Carlo with Exploring Starts.*
In this method, the returns for each pair of state and action is sampled and averaged irrespective of the policy under which they were experienced. It can be seen that this way we can't possible converge to any sub-optimal policy because then the value function would approximate the value function of that policy which would cause the policy to change. Stability is achieved only when both value function and policy are optimal. (However convergence is yet to be formally proved and it continues to remain one of the fundamental open theoretical questions in the field).

5.4 Monte Carlo Control without Exploring Starts

The only general method to ensure that all actions are chosen infinitely often is to make sure that the agent keeps choosing them. Two ways to do this are:

* **On-policy methods** which work by evaluating and improving upon the policy that was used to generate the experiences of the agent, and,
* **Off-policy methods** which work by doing the same for a policy other than that used to generate data.

Here we will try to see how we can do away with the unrealistic assumption of exploring starts.

On policy control methods the policy if generally soft meaning that at first we start with a stochastic policy which gradually shifts closer and closer to a deterministic one. Similar to the ϵ-greedy method, we select a random action with possibility ϵ and rest of the time we go with the action with the maximal value. Amongst ϵ soft policies, ϵ greedy are in a way the ones closest to greedy.
At it's core, MC control still works on the concept of GPI. Without exploring starts however, we cannot improve upon a the current policy by making it greedy with respect to the present value function because that way we would be curtailing possible further exploration. But, GPI only mandates change of the current policy towards the greedy policy not a complete change of it the former into the latter, which allows us to move it only to an ϵ greedy policy ,π', which will definitely be better than or as good as any ϵ soft policy,π.

$$q_\pi(s, \pi'(s)) = \ \textstyle\sum_a \ \pi'(a|s) q_\pi(s,a)$$
$$q_\pi(s, \pi'(s)) = \ \tfrac{\epsilon}{|\mathbb{A}(s)|} \ \textstyle\sum_a q_p i(s,a) \ + \ (1-\epsilon)\max_a q_\pi(s,a)$$
$$q_\pi(s, \pi'(s)) \geq \tfrac{\epsilon}{|\mathbb{A}(s)|} \ \textstyle\sum_a q_p i(s,a) \ + \ (1-\epsilon)\max_a \frac{\pi(a|s) - \frac{\epsilon}{|\mathbb{A}(s)|}}{1-\epsilon} q_\pi(s,a)$$
$$q_\pi(s, \pi'(s)) = \ \tfrac{\epsilon}{|\mathbb{A}(s)|} \ \textstyle\sum_a q_p i(s,a) \ - \epsilon \max_a q_\pi(s,a) \ + \ \textstyle\sum_a \pi(a|s) q_\pi(s,a)$$
$$q_\pi(s, \pi'(s)) = \ v_\pi(s)$$

Thus we have shown that iteration and evaluation will still work for epsilon soft policies. Using the natural notion of greedy policy for ϵ-soft policies, one is assured of improvement on every step, except when the best policy has been found among the ϵ-soft policies. This analysis is independent of how the action-value functions are determined at each stage, but it does assume that they are computed exactly. Thus we achieve only the best policy among the epsilon soft policies and we have also done away with the assumption of exploring starts.

6 Temporal Difference Learning

TDL is an amalgamation of Monte Carlo ideas and Dynamic Programming idea. Like MC methods, it can learn from the agent's real time experience of the environment and at the same time, like DP methods, they update estimates based on other predictions without waiting for a conclusive outcome.

6.1 TD Prediction

Both TD and MC methods use experience to solve the estimation issue. With some experience of some policy π, both methods keep updating their estimates of the policy's value function for non terminal states occurring in this experience. Monte Carlo methods suitable for dynamic environments is

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

Here α is a constant step size parameter which earns this method the name of *constant-α* MC method. Contrary to MC methods which wait until the end of the episode to compute the change in $V(S_t)$, TD methods do the same task at the end of every time step. The simplest TD method, TD(0), is

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

Effectively, the target for the MC methods is G_t whereas that for TD method is

$$R_{t+1} + \gamma V(S_t + 1).$$

Due to TD methods' behaviour of basing it's updates on existing predictions, it is a bootstrapping method like DP. From our discussion of MDPs, it is known that

$$v_\pi(s) \;=\; \mathbb{E}_\pi[G_t|S_t = s]$$
$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

In a way, MC methods use an approximation of the former as the target whereas DP methods use that of the latter for the same purpose. The Monte Carlo target is an estimate because the expected value is unknown; it uses a sampled return in the place of the actual expected return. The DP target is an estimate not due to this, but because $v_\pi(S_{t+1})$ is not known and the current estimate, $V(S_{t+1})$, is used instead. TD methods' target is an estimate for both of these reasons, thus, it combines the sampling of MC methods and the bootstrapping of DP.

It is here that the advantage of TD methods lies. The obvious advantage of TD methods over DP is that they do not require a model of the environment and that of its reward distribution and successor state dynamics. The advantage over MC methods is that TD methods are implemented naturally in a fully incremental and on-line way whereas MC methods need to stall until the end of episodes to find out the returns unlike TD methods which need only wait for a time step. Additionally, MC methods must discount or disregard episodes where random experimental actions were taken which can slow down the learning of the agent. TD methods are much less susceptible to this flaw because they learn from every transition irrespective of the action that was chosen by the agent. Also, in practice TD methods have been found to converge to v_π faster than MC methods but a formal proof for this continues to be an open problem.

6.2 Optimality of TD(0)

In case of a finite experience span, a common approach incorporation incremental learning methods is to present the experience over and over again until we converge to an answer. Given an approximate value function, V , the increments are computed for every time step t at which a non-terminal state is visited, but the value function is changed only once, by the sum of the increments. Then we use the available experience along with the updated value function to update the value function again until we the function converges. This is called *batch updating* because an update is made only after processing a complete batch of training data. Under this method, TD(0) converges deterministically to an answer irrespective of the step size parameter, specified earlier, as long as it is small enough. The constant α method also converges deterministically under these very conditions but to a different answer.

Batch Monte Carlo methods always finds estimates such that mean square error is minimized whereas batch TD(0) always finds estimates which would be exactly correct for for the maximum likelihood model of the Markov process. Generally, the maximum likelihood estimate of a parameter is the parameter that is most likely to generate the data. The maximum-likelihood estimate of a parameter is the parameter value whose probability of generating the data is the greatest. This means we could estimate the value function that would be the exact value function if the model precisely replicated the underlying process behind the Markov processes. This is called the certainty-equivalence estimate. Generally batch TD(0) converges to the certainty equivalence estimate.

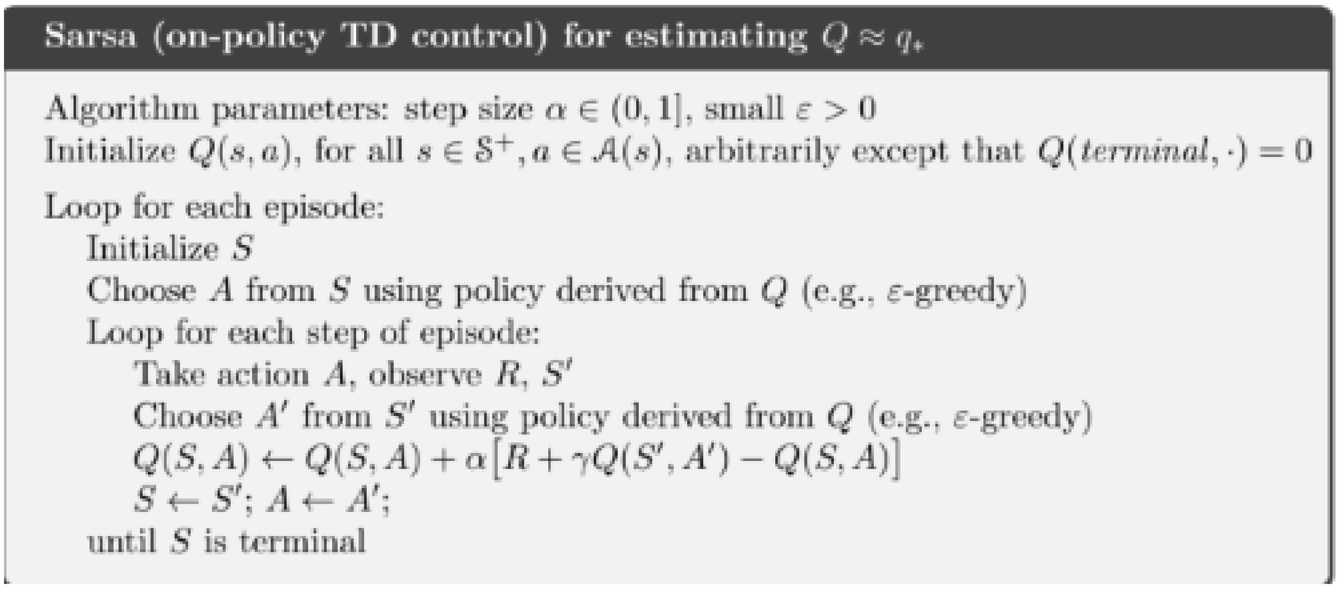
6.3 SARSA: On-policy Control

Turning to the use of TD prediction methods for the control, we follow the pattern of generalized policy iteration as we have been doing before only now, we use TD methods for the evaluation or prediction. Again we need to strike a balance between exploration and exploitation and the approaches possible are on-policy and off-policy control.

The theorems assuring the convergence of TD(0) also apply to the corresponding algorithm for action values:

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

This update is done after each transition from a non terminal state S_t. If the successor state happens to be terminal state, the $Q(S_t, A_t)$ is defined as zero. This uses every element of quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ [Thus the name SARSA] that makes up a transition from one state-action pair to the next.



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

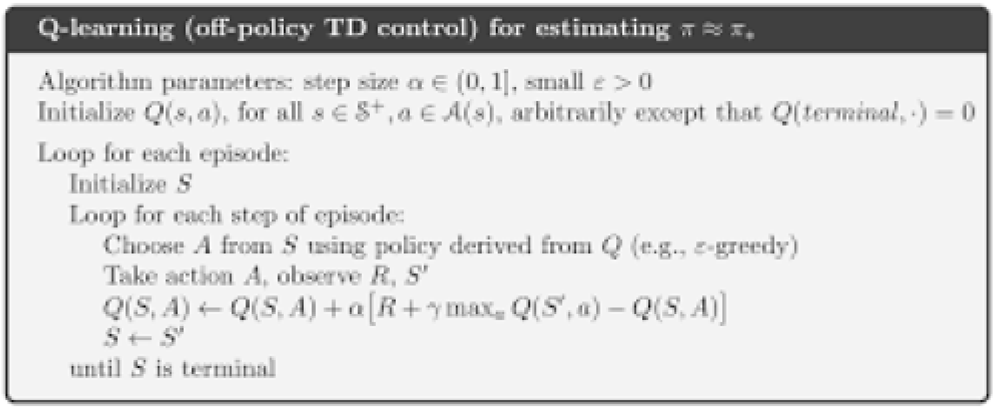
```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Loop for each step of episode:
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

6.4 Q-Learning: Off Policy TD Control

Q-learning iteratively updates Q-values based on the agent's experiences, using the Bellman equation as an update rule. The agent takes an action, observes the reward and the next state, and updates the Q-value for the state-action pair. Being off-policy, it learns the value of the optimal policy independently of the agent's actions, which can be exploratory. This makes Q-learning a powerful and flexible algorithm for solving a wide range of reinforcement learning problems, including those with large and complex state spaces making it an easy choice to use in cases such as when we are training the agent to play atari games which have a complex environment.

Q-Learning is defined by

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



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until $S$ is terminal
```

6.5 Maximisation Bias, Double Learning

All control algorithms discussed until now involve maximisation of their target policies, Considering a single state s, where there are as many actions a whose true values $q(s, a)$ are all zero but their estimated values $Q(s, a)$ are not known precisely and thus are distributed amongst positive and negative numbers. The maximum of the true values is obviously zero but the maximum of all the estimates is positive resulting in a positive bias; this is the *maximization bias.*

If we were to divide the plays in two sets and used them to learn two independent estimates, calling them Q1(a) and Q2(a), both estimates of the true value q(a). We could then use either estimate, say Q1, to determine the maximizing action $A' = argmax_a\ Q1(a)$ and the other, Q2, to provide the estimate of its value, $Q2(A') = Q2(argmax_a\ Q1(a))$. This estimate will then be unbiased in the sense that $E[Q2(A')] = q(A')$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q1(argmax_a\ Q2(a))$. This is the idea of *double learning.*

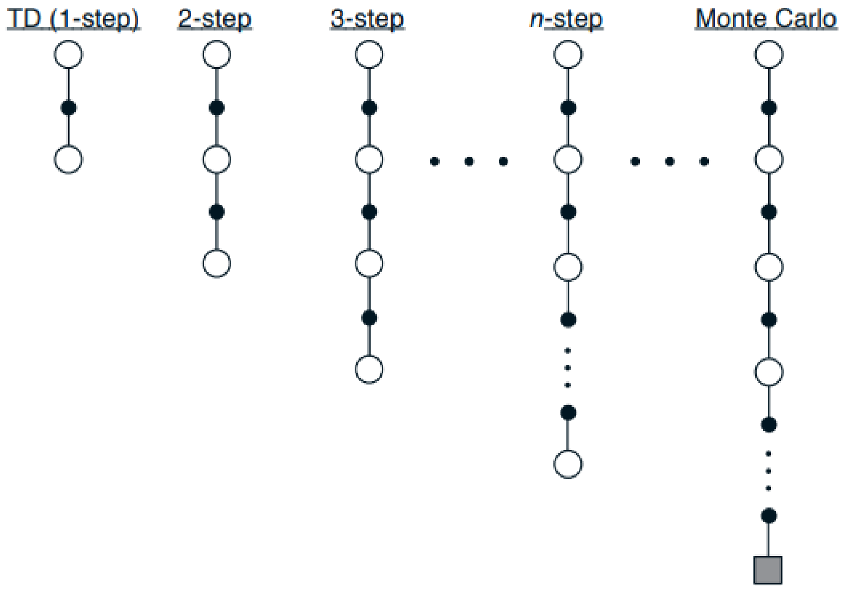
7 Eligibility Traces

From a mechanistic view of the matter, eligibility traces are temporary records of the occurrence of an event (like visiting a state or taking an action) which mark the said events eligible for

undergoing learning changes. From a more theoretical vies, they are a bridge between TD and Monte Carlo methods.
In the very popular TD(λ) method, the λ refers to employment of eligibility traces. In almost any TD method (such as SARSA or Q learning), a more general and more efficient learning method can be obtained by using eligibility traces.

7.1 n-Step Prediction

The main 'gap' between MC methods and TD methods is that for the estimation of policy value function from sample episodes, MC methods perform a backup of each state based on the entire sequence of observed rewards from the state until the end of the episode. Whereas, TD methods' backup is based on just the next reward using the value of the successor state as the proxy for the other rewards. A suitable intermediate to these would be to perform backup on an intermediate number of rewards. n-step backups for v_π are still TD methods because they too alter a prior estimate based on how much it deviates from a later estimate. Only now, the later estimate is n steps away instead of one. These are called n step TD methods because the temporal difference extends over n steps.
Given below is an illustration of the backups of various methods



The target for any arbitrary n step update is the n step return:

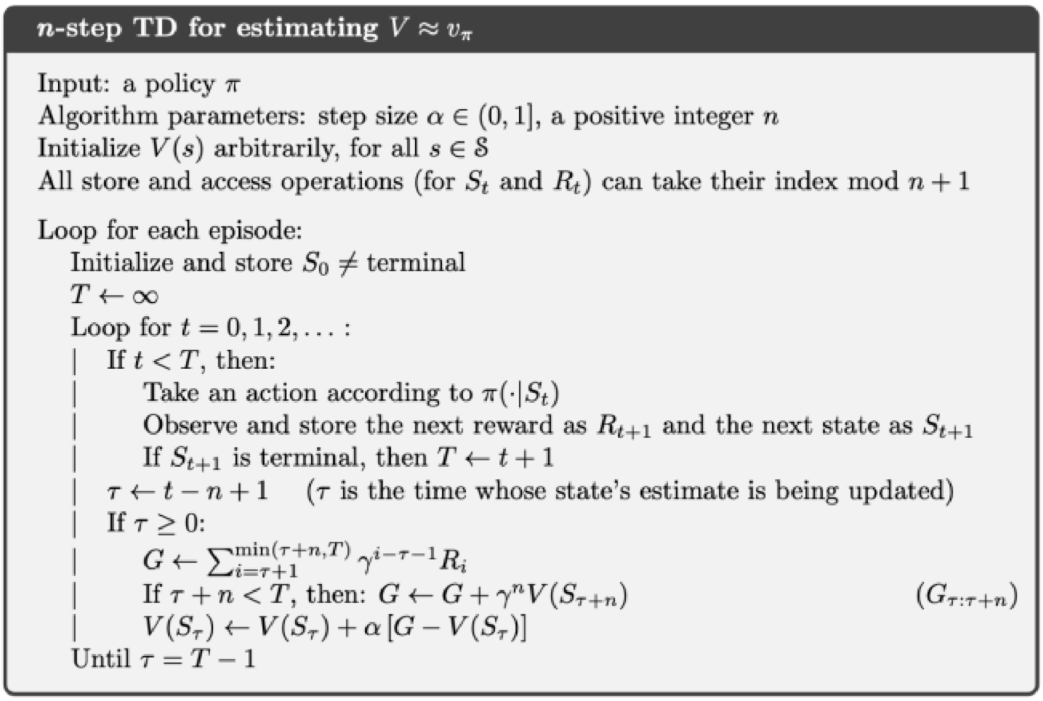
$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$$

If $t + n \geq T$ (that is if the n-step return extends beyond the termination of the episode), then all the missing terms are taken to be zero and the n step return is defined to be equal to the ordinary full return. Thus, the natural state value learning algorithm when using n step returns is:

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

The worst error of the expected n step return is guaranteed to not exceed γ^n times the worst error under V_{t+n-1}

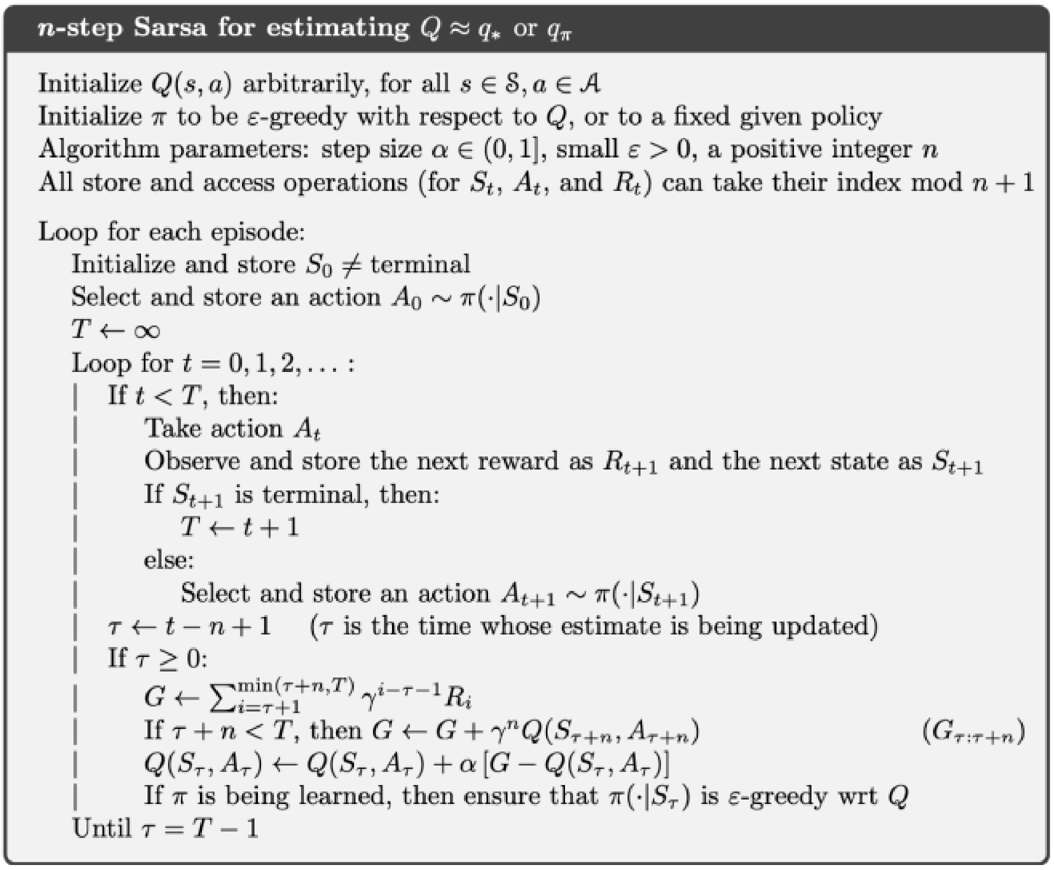
$$\max_s |E_\pi[G_{t:t+n}|S_t = s] - v_\pi(s)| \leq \gamma^n \ \max_s |V_{t+n-1}(s) - v_\pi(s)|$$



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

```
Input: a policy $\pi$
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer $n$
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for $S_t$ and $R_t$) can take their index mod $n + 1$

Loop for each episode:
   Initialize and store $S_0 \neq$ terminal
   $T \leftarrow \infty$
   Loop for $t = 0, 1, 2, \ldots$ :
   |  If $t < T$, then:
   |     Take an action according to $\pi(\cdot|S_t)$
   |     Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
   |     If $S_{t+1}$ is terminal, then $T \leftarrow t + 1$
   |  $\tau \leftarrow t - n + 1$   ($\tau$ is the time whose state's estimate is being updated)
   |  If $\tau \geq 0$:
   |     $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
   |     If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$        $(G_{\tau:\tau+n})$
   |     $V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$
   Until $\tau = T - 1$
```

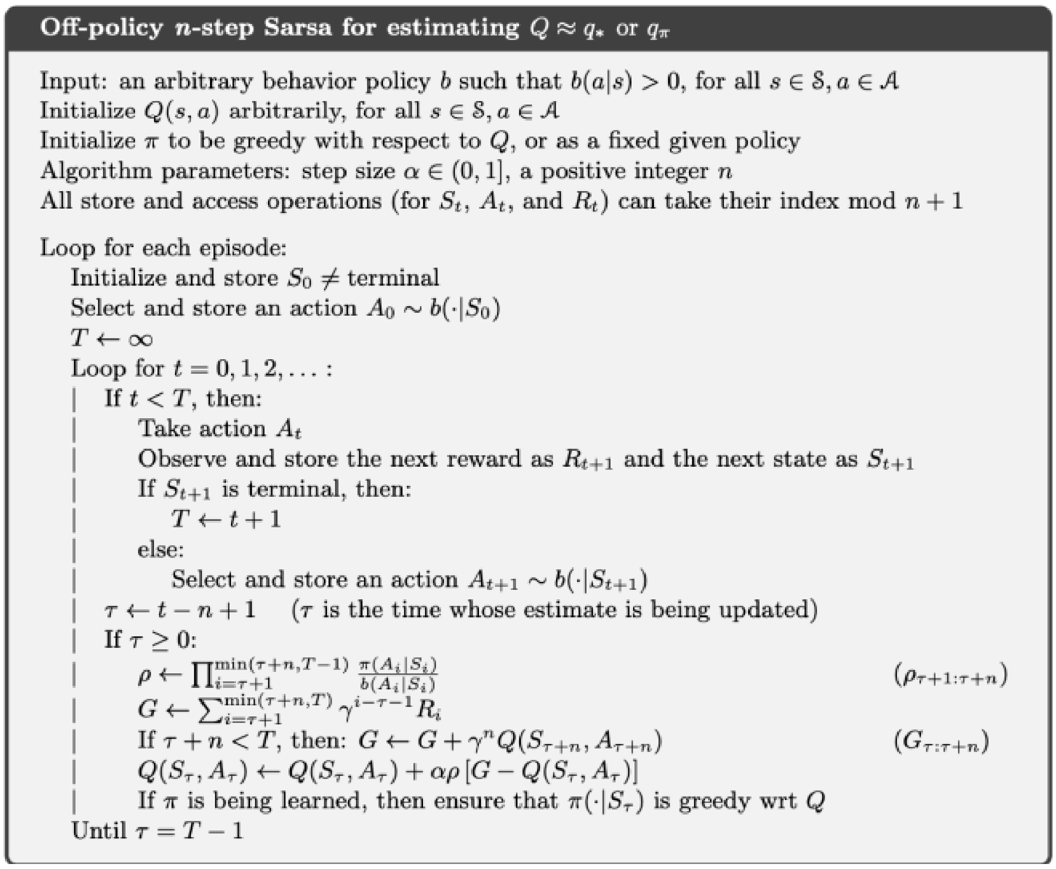
Similarly we have n-step SARSA method



n-step Sarsa for estimating $Q \approx q_*$ or q_π

```
Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be $\varepsilon$-greedy with respect to $Q$, or to a fixed given policy
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, and $R_t$) can take their index mod $n + 1$

Loop for each episode:
   Initialize and store $S_0 \neq$ terminal
   Select and store an action $A_0 \sim \pi(\cdot|S_0)$
   $T \leftarrow \infty$
   Loop for $t = 0, 1, 2, \ldots$ :
   |  If $t < T$, then:
   |     Take action $A_t$
   |     Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
   |     If $S_{t+1}$ is terminal, then:
   |        $T \leftarrow t + 1$
   |     else:
   |        Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
   |  $\tau \leftarrow t - n + 1$   ($\tau$ is the time whose estimate is being updated)
   |  If $\tau \geq 0$:
   |     $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
   |     If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$        $(G_{\tau:\tau+n})$
   |     $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$
   |     If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is $\varepsilon$-greedy wrt $Q$
   Until $\tau = T - 1$
```

and off policy n step methods.



Off-policy n-step Sarsa for estimating $Q \approx q_*$ or q_π

```
Input: an arbitrary behavior policy $b$ such that $b(a|s) > 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\pi$ to be greedy with respect to $Q$, or as a fixed given policy
Algorithm parameters: step size $\alpha \in (0,1]$, a positive integer $n$
All store and access operations (for $S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    Select and store an action $A_0 \sim b(\cdot|S_0)$
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \dots$ :
    |   If $t < T$, then:
    |       Take action $A_t$
    |       Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    |       If $S_{t+1}$ is terminal, then:
    |           $T \leftarrow t+1$
    |       else:
    |           Select and store an action $A_{t+1} \sim b(\cdot|S_{t+1})$
    |   $\tau \leftarrow t - n + 1$   ($\tau$ is the time whose estimate is being updated)
    |   If $\tau \geq 0$:
    |       $\rho \leftarrow \prod_{i=\tau+1}^{\min(\tau+n, T-1)} \frac{\pi(A_i|S_i)}{b(A_i|S_i)}$      $(\rho_{\tau+1:\tau+n})$
    |       $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
    |       If $\tau + n < T$, then: $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$      $(G_{\tau:\tau+n})$
    |       $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\rho\,[G - Q(S_\tau, A_\tau)]$
    |       If $\pi$ is being learned, then ensure that $\pi(\cdot|S_\tau)$ is greedy wrt $Q$
    Until $\tau = T - 1$
```

8 Planning and Learning with Tabular Methods

We usually think of dynamic programming and heuristic search as planning methods and Monte Carlo methods and Temporal Difference methods as learning methods. Although there are differences between these two types of methods but there also exist a lot of similarities. Particularly at the core of both types of methods, there is computation of value functions. And, all methods are on looking ahead to update an estimate of the value function.

8.1 Models and Planning

A model of the environment predicts how the environment will respond to the current state-action pair. The model might be stochastic, distribution based or sample models. Distribution models are stronger than sample models in that they can always be used to produce samples but it in surprisingly many cases it is more convenient to obtain a sample model compared to a distribution model.

$$\begin{tikzcd}\text{model} \arrow[r, "\text{planning}"] & \text{policy}\end{tikzcd}$$

Given a starting state and policy, a sample model could produce an entire episode whereas distribution model could produce all possible episodes. Thus the model is used to simulate the environment and what it produces is called a simulated experience.

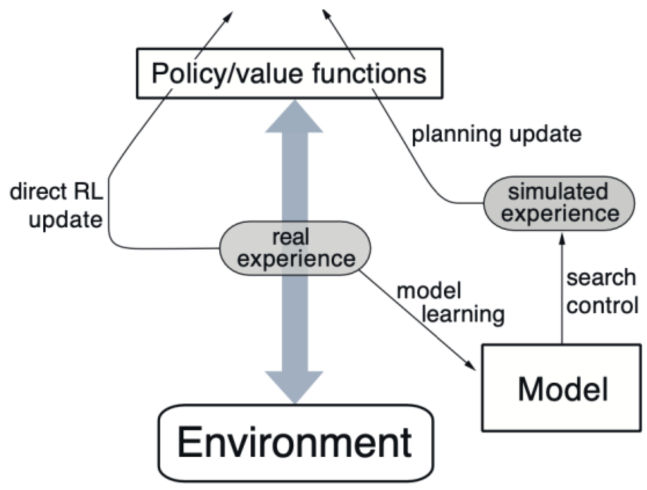


Figure 1: Caption

Planning is basically the computation process which takes a model as the input and improves or creates a policy to interact with the modeled environment.

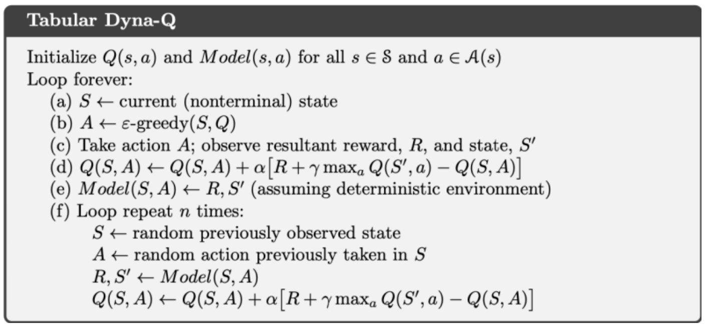
Without AI there exist two distinct approaches state space planning and plan space planning. In *state-space planning*, planning is approached with the view of a search through the state space for an optimal policy whereas in the latter planning viewed as a search throughout the plan space. Operators transform one plan to the other and value functions which if they exist, are defined over the space of plans. Plan space includes evolutionary methods and partial order planning which is a popular kind of planning in AI in which the steps' ordering isn't completely defined at all stages of planning. Plan space methods are difficult to execute efficiently to the stochastic optimal control problems that are in focus RL.

$$\begin{tikzcd}
\text{model} \arrow[r] & \text{simulated experience} \arrow[r, "\text{backups}"] & \text{values} \arrow[r] & \text{policy}
\end{tikzcd}$$

The two basic ideas behind state space planning methods is that (1) they involve value computation as an intermediate step toward policy improvement and, (2) the compute their values by backup operations applied to simulated experience. The above diagram shows this common structure, and we can see that dynamic programming methods clearly fit this structure- they make sweeps of the entire state space, generate the probability distribution for transitions from each of them. Then these distributions are use to update the state's estimated value. Examples of state space planning algorithms include Breadth First Search, Depth First Search, Iterative deepening search,etc.

9 Dyna

Within a planning agent, there are at least two roles for real experience: it can be used to improve the model (it can more closely replicate the real environment) and it can be used to directly improve the value function and policy. The former is called model-learning, and the latter we call direct reinforcement learning (direct RL). Indirect methods often make greater use of limited experience and thus are able to get to a better policy with fewer interactions



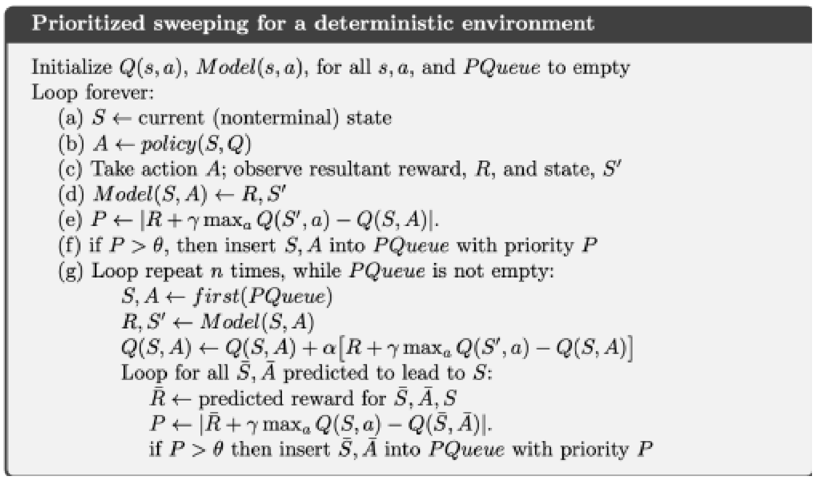
```
Tabular Dyna-Q
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow \varepsilon$-greedy$(S, Q)$
  (c) Take action $A$; observe resultant reward, $R$, and state, $S'$
  (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
  (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
  (f) Loop repeat $n$ times:
        $S \leftarrow$ random previously observed state
        $A \leftarrow$ random action previously taken in $S$
        $R, S' \leftarrow Model(S,A)$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
```

with the environment. In contrast, direct learning methods are way more simple and aren't affected by the biases in the model's design. **Dyna-Q** consists of all planning, acting, model learning and direct learning activities occurring over and over again as shown in the diagram above.
The pseudocode for Dyna-Q tabular method is given below. There are variants of Dyna Q:- Dyna Q+ which updates rewards with time, if a state was visited long ago, the reward to visit said state will eb increased to encourage the agent to come back and explore the state again (basically a sort of an exploration bonus), Dyna-Q AC method uses actor critic methods instead of Q learning methods (which are TDL methods that represent the policy function which does not depend on the value function).

9.1 Prioritized Sweeping

Suppose that the initial values are correct given the erstwhile model, if the agent discovers a change in the environment and changes it's pre-existing value estimate for a state. Typically, this will imply that the values of many other states should also be changed, but the only useful one-step updates are those of actions that lead directly into the one state whose value has been changed. If the values of these actions are updated, then the values of the predecessor states may change in turn. If so, then actions leading into them need to be updated, and then their predecessor states may have changed. In this way one can work backward from arbitrary states that have changed in value, either performing useful updates or terminating the propagation. This general idea might be termed backward focusing of planning computations.

The preceding pairs of those that have been changed more will be more likely to change themselves too. In a stochastic environment, variations in magnitude of change and the abruptness(?) with which pairs need to be updated is influenced by the variation of state transition probability distribution.



Prioritized sweeping for a deterministic environment

```
Initialize $Q(s,a)$, $Model(s,a)$, for all $s, a$, and $PQueue$ to empty
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow policy(S, Q)$
  (c) Take action $A$; observe resultant reward, $R$, and state, $S'$
  (d) $Model(S, A) \leftarrow R, S'$
  (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$.
  (f) if $P > \theta$, then insert $S, A$ into $PQueue$ with priority $P$
  (g) Loop repeat $n$ times, while $PQueue$ is not empty:
        $S, A \leftarrow first(PQueue)$
        $R, S' \leftarrow Model(S, A)$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        Loop for all $\bar{S}, \bar{A}$ predicted to lead to $S$:
          $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
          $P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|$.
          if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority $P$
```

9.2 Trajectory Sampling

Trajectory sampling is a method for generating experience in reinforcement learning. In an episodic task, an agent begins at a specific starting point and continues until it reaches a final goal. In a continuous task, the agent simply keeps going without a defined end. For both types, the agent's actions are determined by its current policy, and the outcomes of these actions (state transitions and rewards) are provided by the environment. In essence, this approach involves creating step-by-step sequences of actions and their consequences, and using these sequences to update the agent's knowledge.

9.3 Heuristic Search

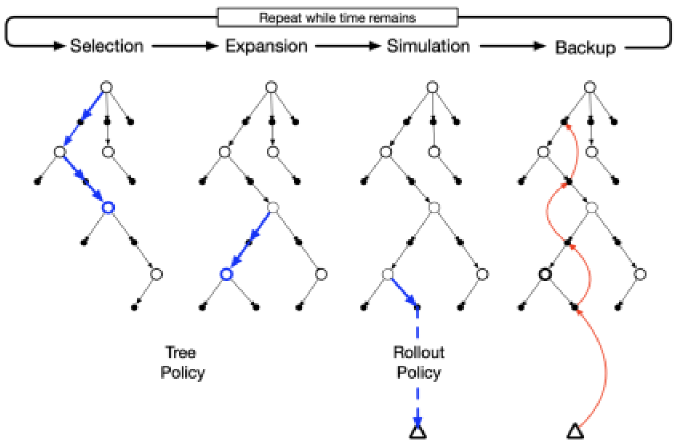
Heuristic search is basically when the agent uses prior knowledge of the environment meaning that the agent goes through the entire tree of possibilities is considered. Once the backed up values of all nodes are considered, the optimal course amongst these is chosen and rest of the values are discarded.

9.4 Rollout Algorithms

Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to model experience trajectories that all begin at the current environment state. They estimate action values for a given policy by averaging the returns of many trajectories that start with each possible action and then follow the given policy. When the action-value estimates are considered to be accurate enough, the action having the highest estimated value is executed, after which the process is carried out freshly over again from the resulting successor state.

9.5 Monte Carlo tree Search

Monte Carlo Tree Search (MCTS) is a recent and strikingly successful example of decision-time planning. At its base, MCTS is a rollout algorithm as described above, but enhanced by the addition of a means for accumulating value estimates obtained from the Monte Carlo



simulations in order to successively direct simulations toward more highly-rewarding trajectories.

1. Selection: Starting at the root node, a tree policy based on the action values attached to the edges of the tree traverses the tree to select a leaf node.
2. Expansion: On some iterations, the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.
3. Simulation: From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.
4. . Backup: The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS.

10 Applications

The applications of RL are abound, it finds it's uses in self driving cars, mobile robots, business investment models, job shop scheduling.

In gaming, RL has achieved remarkable success in mastering complex games, from Go and chess to real-time strategy games. AlphaGo, developed by DeepMind, famously defeated world champions, showcasing the potential of RL in solving challenging problems.

In healthcare, it has the potential to revolutionize healthcare by optimizing treatment plans, drug discovery, and patient care. For example, it can be used to personalize treatment regimens based on individual patient data, leading to improved outcomes.

Other notable applications include use in education services or in natural language processes. Obviously the field is just entering its golden era and going towards revolutionizing this field.

11 References

1. Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto, MIT Press

2. RL Course Lectures at UCL by Professor David Silver
3. CS 747: Foundations of Intelligent and Learning Agents by Professor Shivaram Kalyanakrishnan
4. CS 234 by Professor Emma Brunskill at Stanford University
5. Algorithms for Reinforcement Learning by Csaba Szepesvari, Morgan and Claypool Publishers