Reinforcement Learning Notes

1. *k*-armed Bandits

1.1. The k-armed Bandit Problem

Setup

- K different actions to choose from at each time step
- p(reward|action) is a stationary (not changing over time) probability distribution
- Goal of the agent is to maximize the cumulative reward (over some time span)

What is difficult about solving this problem?

- Do not know what conditional distributions (over reward) are, which means that we need to
- Estimate the expected values of underlying distributions using observed data
- Given that our estimate is never perfect, an important question arises, that is, under what circumstances should we exploit / explore?

1.2. Action Value Methods

Perfect Information (Full Knowledge of Underlying Reward Distributions)

Notation:

- A_t: action (out of k actions) selected at time step t
- R_t: reward received at time step t (after an action has been taken in that time step) (a random variable)

Since we are first dealing with stationary reward distributions, values of actions do not depend on time step. Therefore, the t subscripts merely serve to emphasize that A and R happens in the same time step. In other words, A caused the agent to receive R.

Expected reward given an arbitrary action a:

$$q_\star(a) = \mathbb{E}[R_t | A_t = a]$$

Obviously, the action that should be chosen at each and every time step is $\text{argmax}_a q_\star(a)$.

Imperfect Information (Can Only Sample From Underlying Reward Distributions)

Since we do not know the true expected rewards (for all actions) but can only sample from their corresponding distributions, we need a way to estimate the expected rewards using the finite samples.

This is intuitively done using the **sample-average method**:

$$\begin{aligned} Q_t(a) &= \frac{\text{sum of rewards when a was take prior to t}}{\text{number of times a was taken prior to t}} \\ &= \frac{\sum_{i=1}^{t-1} R_i \mathbb{I}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{I}_{A_i=a}} \end{aligned}$$

Different algorithms use the sample-average rewards in different ways:

- Greedy method
  - Initialization: randomly choose between actions
  - Action selection: $A_t = \text{argmax}_a Q_t(a)$ (never explores)
  - Disadvantage: It is very likely that the true expected rewards of other non-greedy actions are higher, but this algorithm would never know because it never explore.
- ϵ-greedy method
  - Initialization: randomly choose between actions
  - Action selection:
    - Behave greedily most of the time: $A_t = \text{argmax}_a Q_t(a)$
    - Once in a while, with small probability ϵ, select randomly from all other action
  - Advantage: In the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that $Q_t(a)$ converges to $q_\star(a)$.
  - Disadvantage: Such asymptotic guarantees say little about its practical effectiveness.
- Optimistic initial values method
  - Initialization: set the action values to very high (and non-realistic) values, randomly choose between actions
  - Action selection:
    - $A_t = \text{argmax}_a Q_t(a)$, but explores much more than greedy method without optimistic initial values and here's why:
    - For example, let's say that $q_\star(a)$'s in a multi-armed bandit problem are selected from a normal distribution with mean 0 and variance 1. In this case, an initial estimate of +5 is very optimistic for all $q_\star(a)$'s.
    - Whichever action is initially selected, the reward is less than the starting estimate and the estimate is decreased; the learner therefore switches to other actions.
    - All actions are explored several times before the value estimates $q_\star(a)$ converge.
- Upper confidence bound (UCB) method
  - Initialization: randomly choose between actions
  - Action selection:
    - $A_t = \text{argmax}_a[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$
    - Motivation:
      - ϵ-greedy forces non-greedy actions to be tried, but indiscriminately.
      - It would be better to select among non-greedy actions
        1. according to their potential for being actually optimal ($Q_t(a)$)
        2. take into account uncertainties in their value estimates ($\sqrt{\frac{\ln t}{N_t(a)}}$)
  - Disadvantage: hard to generalize to other reinforcement learning tasks (we do not know why yet)

1.3. The 10-armed Testbed

The 10-armed testbed consists of 1000 tasks similar to the one shown in Fig. 1.

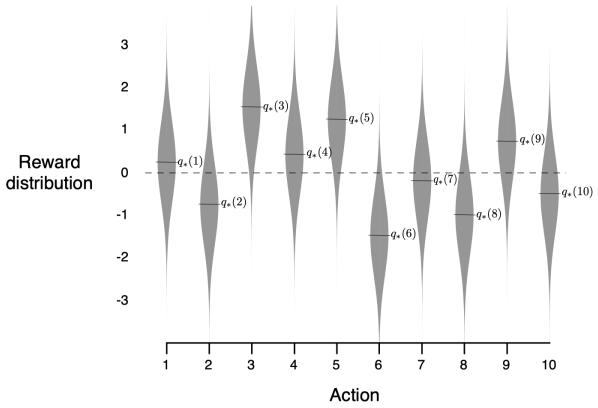


Figure 1: One task from the 10-armed testbed.

In Fig. 2, we can see that the highest average reward over first 1000 steps is achieved by UCB.

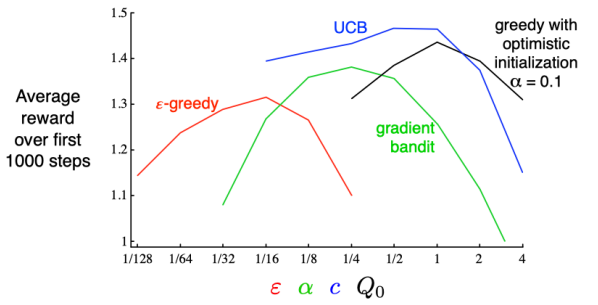


Figure 2: Performance of various algorithms on the 10-armed testbed.

1.4. Compute Sample-Averages Incrementally

If we focus on one action a_k, let R_i denote the reward received by the agent after the ith selection of this action and let Q_n denote the estimated reward:

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

Native implementation: Keep a list of R_i's and compute the sum every time.
Incremental implementation: One can show that Q_{n+1} can be conveniently expressed as a function of Q_n:

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$$

where R_n is a new reward received as a consequence of a_k.

1.5. Tracking Non-stationary Problems

In this case, it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is called **exponential-recency-weighted average** (still a way to estimate the value function).

The incremental update rule is modified to:

$$Q_{n+1} = Q_n + \alpha[R_n - Q_n]$$

where $\alpha \in (0, 1]$ and a common choice of α is 0.99.

To understand this update rule more intuitively, let's expand it. When expanded, this rule becomes:

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$$

We can see, as i decreases, the weight given to the ith reward decreases exponentially.

1.6. Gradient Bandit Algorithms

In previous sections we've estimated the value of each action and used those estimates to take semi-informed actions. Decision making based on value estimates can work quite well, but it is not the only selection method available. One alternative is learn a preference for each action denoted $H_t(a)$ and makes decisions based on the relative preference of one action over another. Here we will use the *soft-max distribution* to define preference as a distribution over all possible actions.

$$Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a) \tag{1}$$

In this case $\pi_t(a)$ is the probability of taking action a at time t. All preferences are initially set to be the same.

Now we will begin to introduce an algorithm for this setting based on stochastic gradient descent using the following pair of formulas:

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), \tag{2}$$

$$H_{t+1}(a) \doteq H_t(a) + \alpha(R_t - \bar{R}_t)\pi_t(a), \forall\, a \neq A_t \tag{3}$$

Where $\alpha > 0$ is the step size parameter and $\bar{R}_t$ is the average reward of all rewards up to time t. Note: if $\bar{R}_t$ is set to be a constant 0 the performance of the algorithm is reduced (e.g. any baseline works so long as it exists and isn't constant).

Now let's walk through how the gradient bandit algorithm approximates to gradient ascent.

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \tag{4}$$

Where the expected reward is:

$$\mathbb{E}[H_t(a)] = \sum_x \pi_t(x) q_*(x),$$

$$\begin{aligned}
\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} [\sum_x \pi_t(x) q_*(x)] \\
&= \sum_x q_*(x) \frac{\partial \pi_t(x)}{\partial H_t(a)} \\
&= \sum_x (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)}
\end{aligned}$$

We can add the B_t during the final step above because the total change in $H_t(a)$ must be zero since all probabilities must sum to one.

$$\sum_x \frac{\partial \pi_t(x)}{\partial H_t(a)} = 0$$

Now we multiply by $\pi_t(x)/\pi_t(x)$

$$\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} = \sum_x (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)} \pi_t(x)/\pi_t(x)$$

Now we have an expectation that will allow us to sum over all possible values of x of the random variable A_t

$$\begin{aligned} &= \mathbb{E}[(q_*(A_t) - B_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)] \\ &= \mathbb{E}[(R_t - \bar{R}_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)] \end{aligned}$$

In this case we let $B_t = \bar{R}_t$ and $R_t = q_*(A_t)$, which we can do since $\mathbb{E}[R_t|A_t] = q_*(A_t)$

Briefly let $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$ without proof netting us:

$$\begin{aligned} &= \mathbb{E}[(R_t - \bar{R}_t)\pi_t(A_t)(\mathbb{1}_{a=A_t} - \pi_t(a))/\pi_t(A_t)] \\ &= \mathbb{E}[(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a))] \end{aligned}$$

Substituting this back into (4) we get:

$$H_{t+1}(a) \doteq H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a)), \forall a$$

Which turns out to be equivalent to what was laid out in (2) and (3).

Now let's go back and show $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$ starting with the quotient rule:

$$\frac{\partial}{\partial x}[\frac{f(x)}{g(x)}] = \frac{\frac{\partial f(x)}{\partial x} g(x) - f(x) \frac{\partial g(x)}{\partial x}}{g(x)^2}$$

Thus:

$$
\begin{aligned}
\frac{\partial \pi_t(x)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \pi_t(x) \\
&= \frac{\partial}{\partial H_t(a)} [\frac{e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}}] \\
&= \frac{\frac{\partial e^{H_t(x)}}{\partial H_t(a)} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} \frac{\partial \sum_{y=1}^{k} e^{H_t(y)}}{\partial H_t(a)}}{(\sum_{y=1}^{k} e^{H_t(y)})^2} \\
&= \frac{\mathbb{1}_{\mathbb{a}=\mathbb{x}} e^{H_t(x)} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} e^{H_t(a)}}{(\sum_{y=1}^{k} e^{H_t(y)})^2} \\
&= \frac{\mathbb{1}_{\mathbb{a}=\mathbb{x}} e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}} - \frac{e^{H_t(x)} e^{H_t(a)}}{(\sum_{y=1}^{k} e^{H_t(y)})^2} \\
&= \mathbb{1}_{\mathbb{a}=\mathbb{x}} \pi_t(x) - \pi_t(x) \pi_t(a) \\
&= \pi_t(x)(\mathbb{1}_{\mathbb{a}=\mathbb{x}} - \pi_t(a))
\end{aligned}
$$

1.7. Associative Search (Contextual Bandit)

Up to this point the formulations of the bandit problem have been non-associative tasks, meaning there is no need to tie certain actions to certain situations. The algorithms we have seen thus far have sought to find a best action that is either stationary or very slowly changing, but if we want to expand our horizons to the general reinforcement learning task we need to be able to learn a policy (a mapping from a situation a set of actions).

If for example we saw a version of the bandit problem where the value of each action changed randomly and drastically with each step the algorithms we've previously seen would be worthless. But if we have some distinct and predictable clue that queues us into how the situation has changed (but not its action values) then we may be able to begin to learn a policy associating that clue with a best action.

1.8. Summary

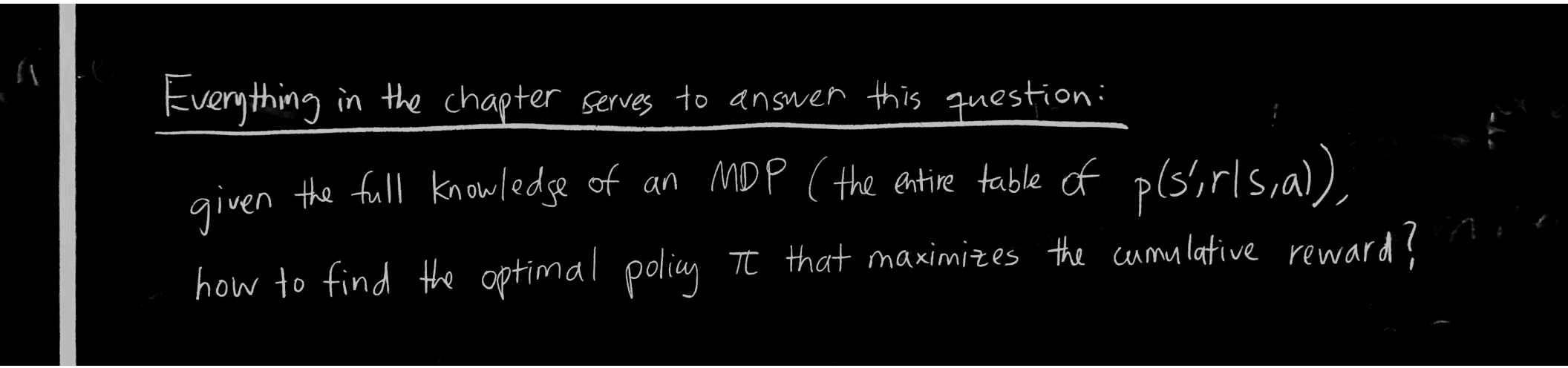
Despite the simplicity of these methods they can actually be used to approach very complex problems and are in some way cutting edge. In the form we've laid out so far, however, they lack the complexity needed to tackle the full reinforcement learning problem, but as we'll see later on they can be used in that context when proper considerations are taken.

Another method for approaching the k-armed bandit problem not mentioned is a

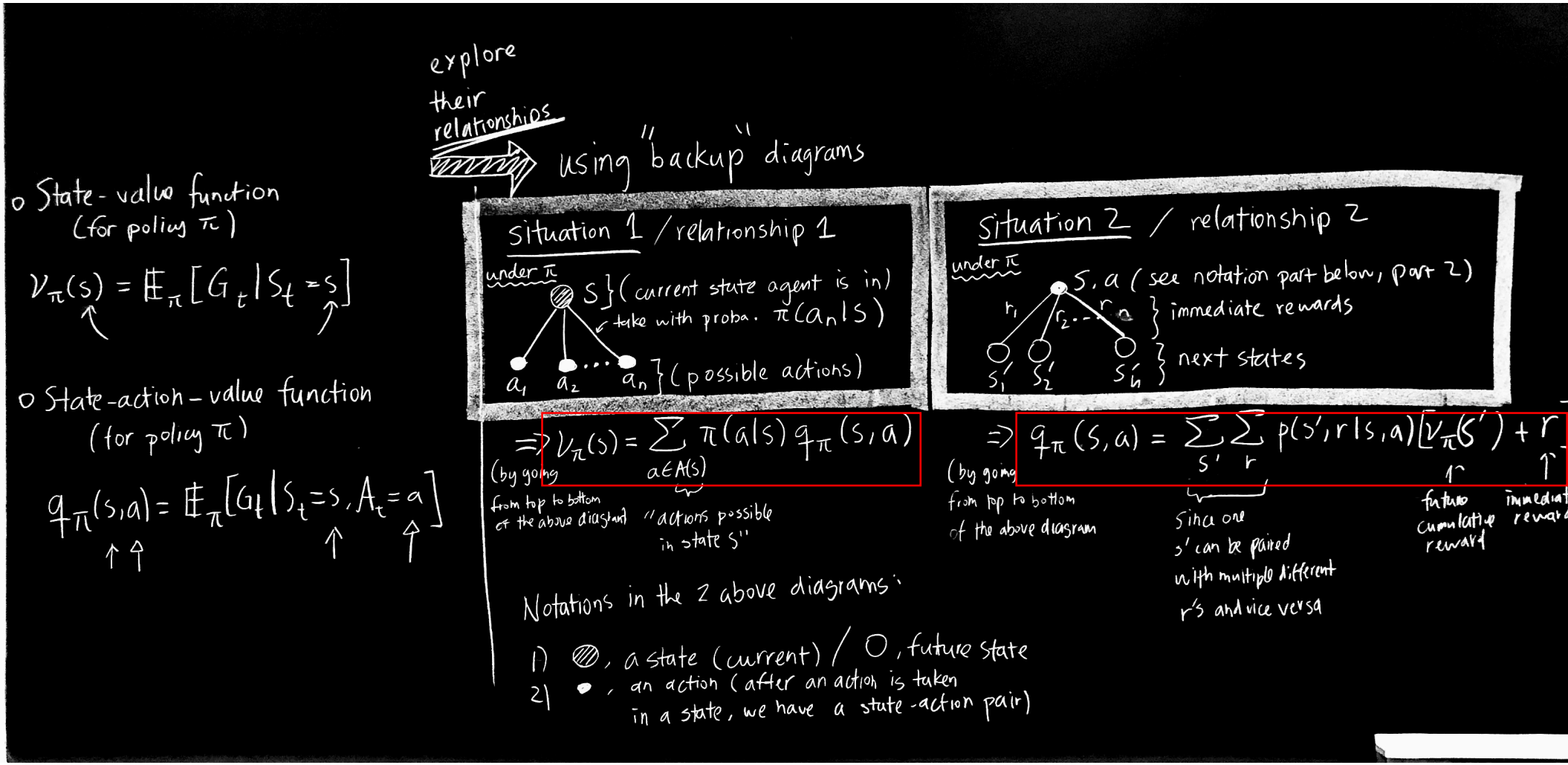
References

Reinforcement Learning: An Introduction, 2nd Edition, Richard S. Sutton, Andrew G. Barto, 2018.

Motivation of Dynamic Programming Chapter

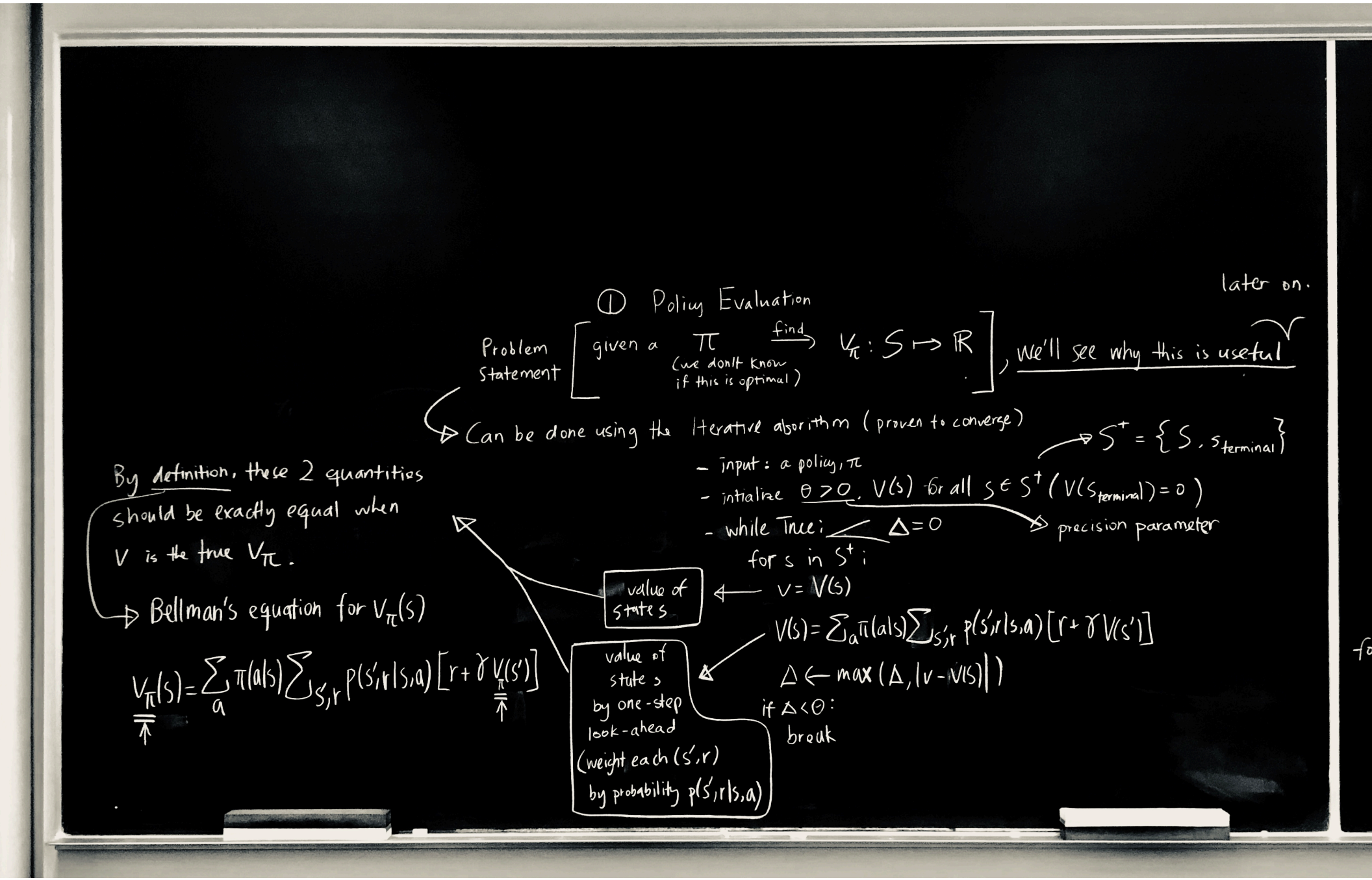


Understand the connection between state-value function v(s) and action-value function q(s, a)



We have found the true value of all states or all state-action pairs when the two equations in red boxes are true for all states / all state-action pairs. Both of these come from the definition of "value", the cumulative reward onwards. For example, the equation in the left red box describes that the cumulative reward from s onwards should equal the weighted sum of the values of possible actions (when the agent is in that state). This definition is justified by our daily experience as agents.

Policy Evaluation Algorithm For Full-MDP



Policy Improvement

Sutton 2018, Chapter 4, Dynamic Programming

Zhihan Yang

Policy improvement algorithm

The policy improvement algorithm happens after each iteration of policy iteration and can be summed up very concisely:

1. For each accessible state, sum up the immediate reward of arriving in that state and the value of that state (expected cumulative reward of being in that state and onwards).
2. Choose the action that take yourself to the accessible state with the highest sum. This is called **greedy action selection**.

However, the most important question is why this simple algorithm works; we'll answer this by proving the convergence of this algorithm (to the optimal value function and the optimal policy) when used in combination with policy evaluation (which we discussed in the previous session).

Proof

The second step of the algorithm updates the old policy such that now the new policy $\pi(a|s)$ gives the highest probability to the action a that maximizes $q_\pi(s,a)$ - the highest value of $\pi(a|s)$ and the highest value of $q_\pi(s,a)$ now occur for the same a. By understanding this alignment and inspecting equation 1 (expressing v_π in terms of q_π), it becomes clear why $V_{\text{new}}(s) \geq V_{\text{old}}(s)$.

(Recall that the value of a state v_π is related to the value of its accessible states $q_\pi(s,a)$ by the following relationship.)

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) \qquad (1)$$

There are two cases that allow $V_{\text{new}}(s) \geq V_{\text{old}}(s)$:

1. $V_{\text{new}}(s) = V_{\text{old}}(s)$
    - This is the Bellman's optimality equation, which indicates that both V are already optimal.
2. $V_{\text{new}}(s) > V_{\text{old}}(s)$
    - This is what happens otherwise.

Therefore, we see that V improves unless it is already optimal.

RL - W5 - Monte Carlo Methods I

5.2 Monte Carlo Estimation of Action Values

$$\pi(s) = \text{argmax}_a\, q(s,a) \tag{1}$$
$$= \text{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma v(s')] \tag{2}$$

With a model of the environment ($p(s',r|s,a)$), state values alone are sufficient to determine a policy; one simply looks ahead one step and chooses whichever action leads to **the best combination of reward and next state**.

For example, in the GridWorld example, given a specific box, $p(s',r|s,a) = 1$ for all nearby boxes when actions are chosen correspondingly (e.g. if you want to go to the box on the left and your action is to move to the left, then you will always end up in the box on the left). $p(s',r|s,a) = 0$ if otherwise. In this context, the action that maximizes the expression $\sum_{s',r} p(s',r|s,a)[r + \gamma v(s')]$ is the one that assigns probability 1 to the best combination of r and $v(s')$.

Without a model, however, state values alone are not sufficient, because computing the expression $\text{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma v(s')]$ requires knowledge of r's. Since the specific values of r's are part of the environment and we assume no knowledge of the environment, we do not have knowledge of the values of r's . Therefore, instead of just estimating the values of states, we directly estimate the values of state-action pairs and perform the argmax directly according to equation (1).

The only complication is that many state–action pairs may never be visited. If π is a deterministic policy, then in following π one will observe returns only for one of the actions from each state.

This is the general problem of maintaining exploration, there are two methods to help with the problem.

- Exploring starts. By specifying that the episodes start in a state–action pair, and that every pair has a nonzero probability of being selected as the start.
- Consider only policies that are stochastic with a nonzero probability of selecting all actions in each state.

5.3 Monte Carlo Control (with "exploring starts")

In policy iteration one maintains both an approximate policy and evaluation an approximate value function.

For the moment, let us assume that we do indeed observe an infinite number of episodes and that, in addition, the episodes are generated with exploring starts. Under these assumptions, the Monte Carlo methods will compute each q_{π_k} exactly, for arbitrary π_k.

For any action-value function q, the corresponding greedy policy is the one that, for each $s \in S$, deterministically chooses an action with maximal action-value:

$$\pi(s) \doteq \text{argmax}_a\, q(s,a) \tag{3}$$

Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}.

$$\pi_{k+1}(s) = \text{argmax}_a\, q_{\pi_k}(s,a) \tag{4}$$

In the previous dynamic programming chapter we have demonstrated that, by using this strategy of acting greedily w.r.t to the current value function, the new value function is strictly greater than the current value function when the current value function is not optimal and the new value function is equal to the current function only when the current value function is optimal.

Two unlikely assumptions

- Episodes have exploring starts
- Policy evaluation could be done with an infinite number of episodes (**the book tackles the second one first)**
    - first approach: hold firm to the idea of approximating q_{π_k} in each policy evaluation
        - allows a boundary of errors
    - second approach: give up trying to complete policy evaluation before returning to policy improvement.
        - don't evaluate the values of all state-action pair / estimate them for a fixed number of times, regardless of whether convergence occurs

This algorithm is still using first-visit of state-action pairs

Note: a G value is accumulated in each Returns(s, a) per outer loop

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
if not the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Temporal Difference Learning Part 1 Notes

Zhihan Yang, Oct 26, 2019

TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

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

A simple every-visit Monte Carlo method suitable for nonstationary environments is $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$

- Where:
  - $V(S_t)$ on the right is the new value estimate for S_t
  - $V(S_t)$ on the left is the old value estimate for S_t
  - α is the stepsize $(0, 1]$
  - $[\cdot]$ is called the error term; notice how this update rule moves the old value estimate in the direction of the error
  - G_t is the cumulative reward starting from S_t and onwards
- This is because $V(S_t) = \frac{1}{N(S_t)} \sum_{n=1}^{N(S_t)} G_t^n$ can be written as the update rule $V(S_t) \leftarrow V(S_t) + \frac{1}{N}(G_t^{N(S_t)} - V(S_t))$. If we consider nonstationary environments (values of states are changing) are we want to give more emphasis to recent G_t's, setting a stepsize α greater than $\frac{1}{N}$ is the way to go.
- Also called constant-alpha MC
- Wait until the end of the episode to determine the increment to $V(S_t)$; this is the definition of G_t

---

Let's think about the motivation behind this algorithm.

- This algorithm serves to estimate $v_\pi(s)$ by sampling G_t because
  - by definition, $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ (equation 1)
  - and we simply replace the expectation with an empirical mean; for a state s , $v_\pi(s) \approx \frac{1}{N(s)} \sum_{n=1}^{N(s)} G_n(s)$
- However, equation 1 can actually be framed in the following way (remember to add the discounting term)
  - $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[r_{t+1} + G_{t+1}|S_t = s]$
  - $= \mathbb{E}_\pi[r_{t+1}|S_t = t] + \mathbb{E}_\pi[G_{t+1}|S_t = s]$
  - $= \mathbb{E}_\pi[r_{t+1}|S_t = t] + \sum_{s'} p(s'|s)\mathbb{E}_\pi[G_{t+1}|S_t = s']$

- $= \mathbb{E}_\pi[r_{t+1}|S_t = t] + \sum_{s'} p(s'|s)v_\pi(s')$
- $= \mathbb{E}_\pi[r_{t+1}|S_t = t] + \mathbb{E}_\pi[v_\pi(S_{t+1})|S_t = s]$
- $= \mathbb{E}_\pi[r_{t+1} + v_\pi(S_{t+1})|S_t = t]$

- What if, for a state $S_t = t$, instead of sampling G_t's, we sample $[r_{t+1} + V(S_{t+1})]$'s, the immediate reward plus the value estimate of the next state?

The simplest TD method makes this update

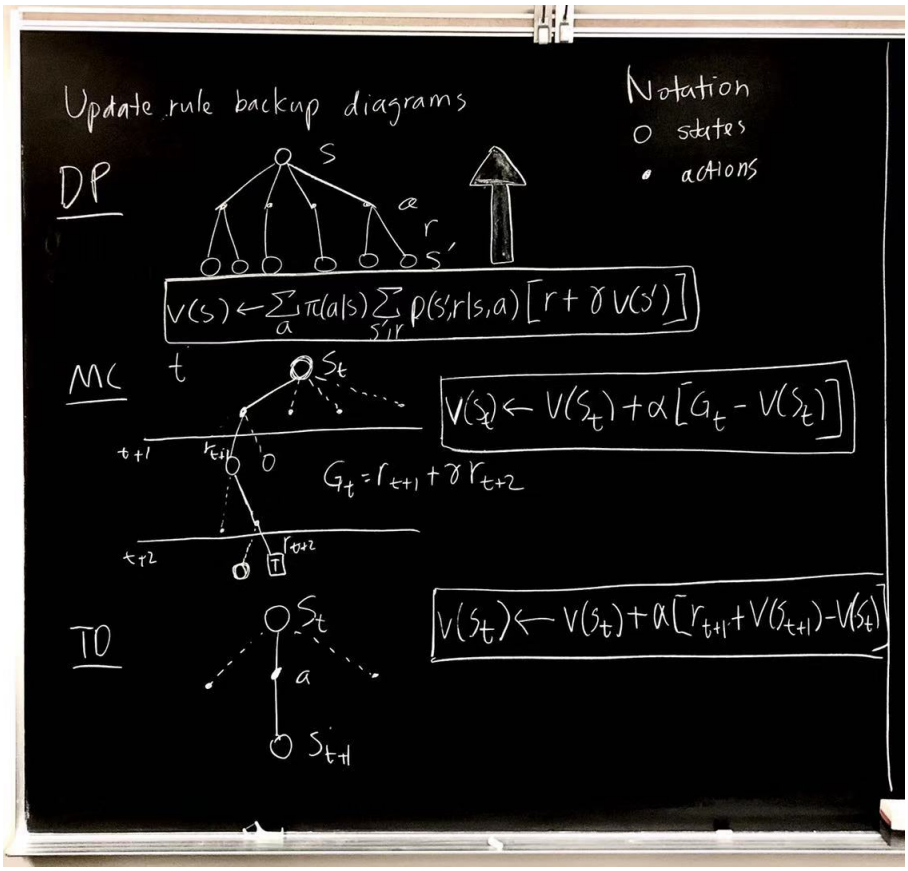
$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \qquad (1)$$

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

Notice how G_t is replaced by something else.

This method involves bootstrapping because the estimate $V(S_t)$ is updated by sampling a quantity that involves another estimate $V(S_{t+1})$. So a natural question is, why does this move $V(S_t)$ closer to $v_\pi(S_t)$? A simple answer is that each step of the update rule includes R_{t+1} as something from real dynamic of the underlying MDP (markov decision process).

Compare the backup diagrams of Dynamic Programming, Monte Carlo Method and Temporal Difference Method



An example of TD

Read highlighted parts off textbook

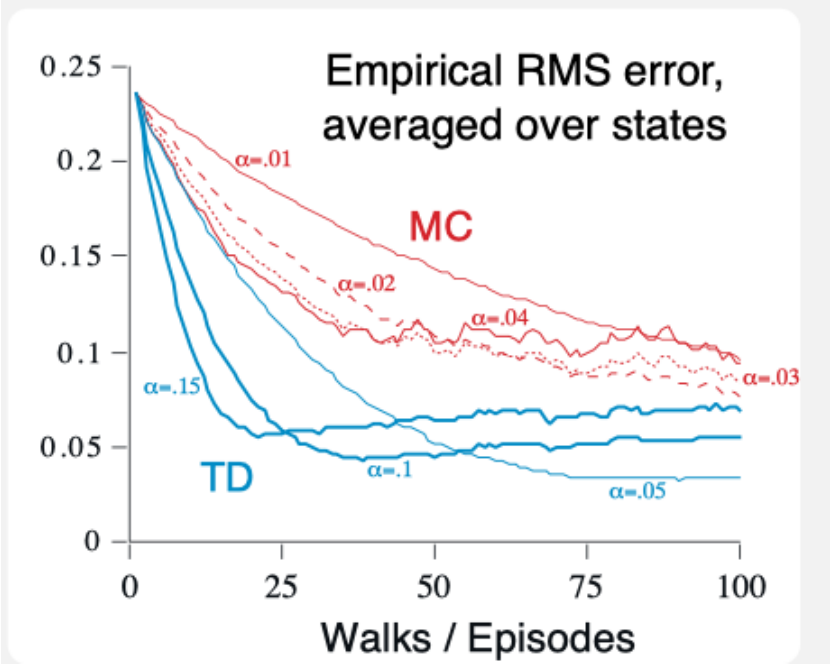
Advantages of TD Prediction Methods

- Essentially, what is the advantage from boostrapping, that is, learn a guess from a guess?

- Advantage over DP methods
  - they do not require a model of the environment, of its reward and next-state probability distributions.
- advantage of TD methods over Monte Carlo methods
  - naturally implemented in an online, fully incremental fashion
    - With Monte Carlo methods one must wait until the end of an episode, because only then is the return known, whereas with TD methods one need wait only one time step.
    - Some applications have very long episodes, so that delaying all learning until the end of the episode is too slow. Other applications are continuing tasks and have no episodes at all.
- Are TD methods sound? Can we guarantee convergence to the correct answer?
  - Yes.
  - For any fixed policy, TD(0) has been proved to converge to v pi , in the mean for a constant step-size parameter if it is su!ciently small, and with probability 1 if the step-size parameter decreases according to the usual stochastic approximation conditions (2.7), for example $1/n$.
- If both TD and Monte Carlo methods converge asymptotically to the correct predictions, then a natural next question is “Which gets there first?”
  - In fact, it is not even clear what is the most appropriate formal way to phrase this question!
  - In practice, however, TD methods have usually been found to converge faster than constant-alpha MC methods on stochastic tasks, as illustrated in the following example.

In this example we empirically compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process:

- Markov reward process, consider P(S', r|S), not including actions.
- All episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability.
- Recall that the value of a state is its expected reward into the future, until terminating states, which the agent gets stuck adn receives 0 reward onwards.
- The probability of reaching the rightmost terminating block is 1/6 from A; 2/6 from B, 3/6 from C, 4/6 from D, and 5/6 from E.
- Since reaching the rightmost terminating block gives a reward of 1, the corresponding true values are 1/6, ..., 5/6.
- After 100 iterations


Empirical RMS error,
averaged over states
0.25
0.2
0.15
0.1
0.05
0
α=.01
MC
α=.02
α=.04
α=.03
α=.15
TD
α=.1
α=.05
0
25
50
75
100
Walks / Episodes

Reinforcement Learning Notes

1. *k*-armed Bandits

1.1. The k-armed Bandit Problem

Setup

- K different actions to choose from at each time step
- p(reward|action) is a stationary (not changing over time) probability distribution
- Goal of the agent is to maximize the cumulative reward (over some time span)

What is difficult about solving this problem?

- Do not know what conditional distributions (over reward) are, which means that we need to
- Estimate the expected values of underlying distributions using observed data
- Given that our estimate is never perfect, an important question arises, that is, under what circumstances should we exploit / explore?

1.2. Action Value Methods

Perfect Information (Full Knowledge of Underlying Reward Distributions)

Notation:

- A_t: action (out of k actions) selected at time step t
- R_t: reward received at time step t (after an action has been taken in that time step) (a random variable)

Since we are first dealing with stationary reward distributions, values of actions do not depend on time step. Therefore, the t subscripts merely serve to emphasize that A and R happens in the same time step. In other words, A caused the agent to receive R.

Expected reward given an arbitrary action a:

$$q_\star(a) = \mathbb{E}[R_t | A_t = a]$$

Obviously, the action that should be chosen at each and every time step is $\text{argmax}_a q_\star(a)$.

Imperfect Information (Can Only Sample From Underlying Reward Distributions)

Since we do not know the true expected rewards (for all actions) but can only sample from their corresponding distributions, we need a way to estimate the expected rewards using the finite samples.

This is intuitively done using the **sample-average method**:

$$Q_t(a) = \frac{\text{sum of rewards when a was take prior to t}}{\text{number of times a was taken prior to t}} \tag{1}$$

$$= \frac{\sum_{i=1}^{t-1} R_i \mathbb{I}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{I}_{A_i=a}} \tag{2}$$

Different algorithms use the sample-average rewards in different ways:

- Greedy method
  - Initialization: randomly choose between actions
  - Action selection: $A_t = \text{argmax}_a Q_t(a)$ (never explores)
  - Disadvantage: It is very likely that the true expected rewards of other non-greedy actions are higher, but this algorithm would never know because it never explore.
- ϵ-greedy method
  - Initialization: randomly choose between actions
  - Action selection:
    - Behave greedily most of the time: $A_t = \text{argmax}_a Q_t(a)$
    - Once in a while, with small probability ϵ, select randomly from all other action
  - Advantage: In the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that $Q_t(a)$ converges to $q_\star(a)$.
  - Disadvantage: Such asymptotic guarantees say little about its practical effectiveness.
- Optimistic initial values method
  - Initialization: set the action values to very high (and non-realistic) values, randomly choose between actions
  - Action selection:
    - $A_t = \text{argmax}_a Q_t(a)$, but explores much more than greedy method without optimistic initial values and here's why:
    - For example, let's say that $q_\star(a)$'s in a multi-armed bandit problem are selected from a normal distribution with mean 0 and variance 1. In this case, an initial estimate of +5 is very optimistic for all $q_\star(a)$'s.
    - Whichever action is initially selected, the reward is less than the starting estimate and the estimate is decreased; the learner therefore switches to other actions.
    - All actions are explored several times before the value estimates $q_\star(a)$ converge.
- Upper confidence bound (UCB) method
  - Initialization: randomly choose between actions
  - Action selection:
    - $A_t = \text{argmax}_a[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$
    - Motivation:
      - ϵ-greedy forces non-greedy actions to be tried, but indiscriminately.
      - It would be better to select among non-greedy actions
        1. according to their potential for being actually optimal ($Q_t(a)$)
        2. take into account uncertainties in their value estimates ($\sqrt{\frac{\ln t}{N_t(a)}}$)
  - Disadvantage: hard to generalize to other reinforcement learning tasks (we do not know why yet)

1.3. The 10-armed Testbed

The 10-armed testbed consists of 1000 tasks similar to the one shown in Fig. 1.

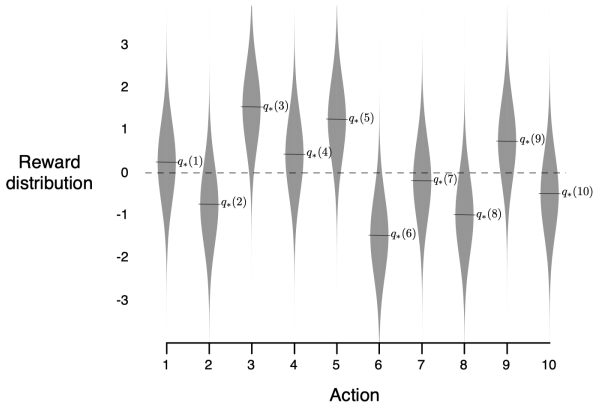


Figure 1: One task from the 10-armed testbed.

In Fig. 2, we can see that the highest average reward over first 1000 steps is achieved by UCB.

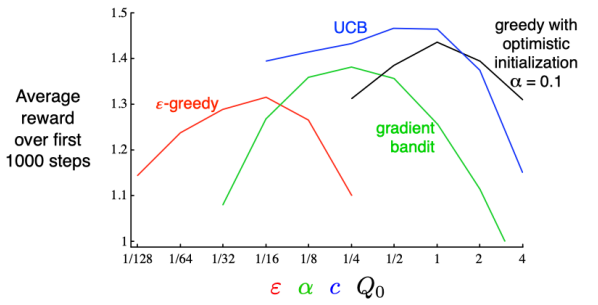


Figure 2: Performance of various algorithms on the 10-armed testbed.

1.4. Compute Sample-Averages Incrementally

If we focus on one action a_k, let R_i denote the reward received by the agent after the ith selection of this action and let Q_n denote the estimated reward:

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

Native implementation: Keep a list of R_i's and compute the sum every time.
Incremental implementation: One can show that Q_{n+1} can be conveniently expressed as a function of Q_n:

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n]$$

where R_n is a new reward received as a consequence of a_k.

1.5. Tracking Non-stationary Problems

In this case, it makes sense to give more weight to recent rewards than to long-past rewards. One of the most popular ways of doing this is called **exponential-recency-weighted average** (still a way to estimate the value function).

The incremental update rule is modified to:

$$Q_{n+1} = Q_n + \alpha[R_n - Q_n]$$

where $\alpha \in (0, 1]$ and a common choice of α is 0.99.

To understand this update rule more intuitively, let's expand it. When expanded, this rule becomes:

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i$$

We can see, as i decreases, the weight given to the ith reward decreases exponentially.

1.6. Gradient Bandit Algorithms

In previous sections we've estimated the value of each action and used those estimates to take semi-informed actions. Decision making based on value estimates can work quite well, but it is not the only selection method available. One alternative is learn a preference for each action denoted $H_t(a)$ and makes decisions based on the relative preference of one action over another. Here we will use the *soft-max distribution* to define preference as a distribution over all possible actions.

$$Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a) \tag{3}$$

In this case $\pi_t(a)$ is the probability of taking action a at time t. All preferences are initially set to be the same.

Now we will begin to introduce an algorithm for this setting based on stochastic gradient descent using the following pair of formulas:

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), \tag{4}$$

$$H_{t+1}(a) \doteq H_t(a) + \alpha(R_t - \bar{R}_t)\pi_t(a), \forall\, a \neq A_t \tag{5}$$

Where $\alpha > 0$ is the step size parameter and $\bar{R}_t$ is the average reward of all rewards up to time t. Note: if $\bar{R}_t$ is set to be a constant 0 the performance of the algorithm is reduced (e.g. any baseline works so long as it exists and isn't constant).

Now let's walk through how the gradient bandit algorithm approximates to gradient ascent.

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}, \tag{6}$$

Where the expected reward is:

$$\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x),$$

$$\begin{aligned}
\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)}[\sum_x \pi_t(x) q_*(x)] \\
&= \sum_x q_*(x) \frac{\partial \pi_t(x)}{\partial H_t(a)} \\
&= \sum_x (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)}
\end{aligned}$$

We can add the B_t during the final step above because the total change in $H_t(a)$ must be zero since all probabilities must sum to one.

$$\sum_x \frac{\partial \pi_t(x)}{\partial H_t(a)} = 0$$

Now we multiply by $\pi_t(x)/\pi_t(x)$

$$\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} = \sum_x (q_*(x) - B_t) \frac{\partial \pi_t(x)}{\partial H_t(a)} \pi_t(x)/\pi_t(x)$$

Now we have an expectation that will allow us to sum over all possible values of x of the random variable A_t

$$\begin{aligned} &= \mathbb{E}[(q_*(A_t) - B_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)] \\ &= \mathbb{E}[(R_t - \bar{R}_t) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)] \end{aligned}$$

In this case we let $B_t = \bar{R}_t$ and $R_t = q_*(A_t)$, which we can do since $\mathbb{E}[R_t|A_t] = q_*(A_t)$

Briefly let $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$ without proof netting us:

$$\begin{aligned} &= \mathbb{E}[(R_t - \bar{R}_t)\pi_t(A_t)(\mathbb{1}_{a=A_t} - \pi_t(a))/\pi_t(A_t)] \\ &= \mathbb{E}[(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a))] \end{aligned}$$

Substituting this back into (4) we get:

$$H_{t+1}(a) \doteq H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{1}_{a=A_t} - \pi_t(a)), \forall a$$

Which turns out to be equivalent to what was laid out in (2) and (3).

Now let's go back and show $\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)(\mathbb{1}_{a=x} - \pi_t(a))$ starting with the quotient rule:

$$\frac{\partial}{\partial x}[\frac{f(x)}{g(x)}] = \frac{\frac{\partial f(x)}{\partial x} g(x) - f(x) \frac{\partial g(x)}{\partial x}}{g(x)^2}$$

Thus:

$$
\begin{aligned}
\frac{\partial \pi_t(x)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \pi_t(x) \\
&= \frac{\partial}{\partial H_t(a)} [\frac{e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}}] \\
&= \frac{\frac{\partial e^{H_t(x)}}{\partial H_t(a)} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} \frac{\partial \sum_{y=1}^{k} e^{H_t(y)}}{\partial H_t(a)}}{(\sum_{y=1}^{k} e^{H_t(y)})^2} \\
&= \frac{\mathbb{1}_{\mathbb{a}=\mathbb{x}} e^{H_t(x)} \sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)} e^{H_t(a)}}{(\sum_{y=1}^{k} e^{H_t(y)})^2} \\
&= \frac{\mathbb{1}_{\mathbb{a}=\mathbb{x}} e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}} - \frac{e^{H_t(x)} e^{H_t(a)}}{(\sum_{y=1}^{k} e^{H_t(y)})^2} \\
&= \mathbb{1}_{\mathbb{a}=\mathbb{x}} \pi_t(x) - \pi_t(x) \pi_t(a) \\
&= \pi_t(x)(\mathbb{1}_{\mathbb{a}=\mathbb{x}} - \pi_t(a))
\end{aligned}
$$

1.7. Associative Search (Contextual Bandit)

Up to this point the formulations of the bandit problem have been non-associative tasks, meaning there is no need to tie certain actions to certain situations. The algorithms we have seen thus far have sought to find a best action that is either stationary or very slowly changing, but if we want to expand our horizons to the general reinforcement learning task we need to be able to learn a policy (a mapping from a situation a set of actions).

If for example we saw a version of the bandit problem where the value of each action changed randomly and drastically with each step the algorithms we've previously seen would be worthless. But if we have some distinct and predictable clue that queues us into how the situation has changed (but not its action values) then we may be able to begin to learn a policy associating that clue with a best action.

1.8. Summary

Despite the simplicity of these methods they can actually be used to approach very complex problems and are in some way cutting edge. In the form we've laid out so far, however, they lack the complexity needed to tackle the full reinforcement learning problem, but as we'll see later on they can be used in that context when proper considerations are taken.

Given that in our formulations of the bandit problem we are trying to discover the expected values of a set of actions its easy to formulate a Bayesian approach to the problem. Using Bayesian methods to solve the problem can be computationally complex, but allow us to solve the problem very precisely. By setting prior distributions for each action and then iterative computing the posterior probability that a given action is optimal. It is even possible to find the optimal approach by computing the possible outcome over all values, although in practice doing so is much too complex. Approximation may help reduce that problem and will be touched on later, but is an ongoing research topic at the moment.

2. Finite Markov Decision Processes

In this section we're going to go over the basics of Markov decision processes, or MPDs. When solving MPDs we estimate the value $q_*(s, a)$ of each action a in each state s. Alternatively, we can estimate the value $v_*(s)$ of each state given the optimal action is chosen.

2.1. The Agent-Environment Interface

At each step the agent sees a state $S_t \in S$ and chooses an action $A_t \in A(s)$. The agent then receives a reward R_{t+1} during the next step and enters state S_{t+1}.

For random variables $s' \in S, r \in R$ there is a chance of those values occurring at time s given the previous state and action:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{7}$$

It's important to note that MDPs are *Markov* processes, because the probabilities for S_t and R_t are only dependent on the immediately preceding S_{t-1} and A_{t-1}. The above equation can be slightly augmented to produce the equations below.

2.2. Goals and Rewards

At each step the agent receives a reward $R_t \in \mathbb{R}$. The goal of the agent is to maximize the total amount of reward it receives over time. That is to say, that the agent does not seek to maximize its next reward, but the total reward it will receive over the long run.

Given this simple definition of an agents reward seeking behavior, our reward choice can be quite flexible. For example, an agent learning to play a game can be given +1 for a win, 0 for a tie, and -1 for a loss, or a maze solving agent can be given a reward of -1 for each second it doesn't find the solution. In both cases the agent learns to solve the problem by maximizing the reward, but how we choose that reward determines *what* we want the agent to do, but not *how* we want it to do it.

2.3. Returns and Episodes

As previously, mentioned we want to maximize the cumulative reward, but more generally we want to maximize the expected return. Return in this case is defined as a specific function of the reward sequence $G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$ where T is the final time step. Tasks where we have a final step are called *episodic* tasks.

Sometimes we're able to break up a task into a series of *episodes*, but in many cases we can't. In those cases, when we have a continual process and are working on a *continuing task*, our previous return formula is no long valid given that $T = \infty$. To combat that issue we can use *discounting*, where we work with a sum of discounted rewards to maximize the *discounted return* given a chosen A_t:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = \tag{8}$$

where $0 \leq \gamma \leq 1$ is the discount rate.

In selecting a discount rate we choose how much less we value a future reward than an immediate reward. If γ is 0 or close to 0 than we focus near exclusively on the immediate reward, while a γ near 1 will take into account further out scenarios.

As is the case in many reinforcement learning algorithms we can define G_t in terms of successive steps:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{9}$$

We should also mention that when $\gamma < 1$ and the reward is constant

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{10}$$

2.4. Unified Notation for Episodic and Continuing Tasks

To simplify the notation used in the section above, we define the expected return as:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{11}$$

with the possibility that $T = \infty$ or $\gamma = 1$ but not both.

2.5. Golf Example (Optimal Value Functions)

Imagine we can use either a putter or driver to complete a section of a golf course. At some points the driver is advantageous and at others the putter is better. We want our agent to learn how to optimally complete the hole through a combination of puts and drives.

We'll approach this problem through the Bellman optimality equation, since the value of a state under an optimal policy must equal the return of the best action.

$$\begin{aligned} v_*(s) &= max_{a \in A(s)} q_{\pi *}(s,a) \\ &= max_a \mathbb{E}_{\pi *}[G_t | S_t = s, A_t = a] \\ &= max_a \mathbb{E}_{\pi *}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= max_a \mathbb{E}_{\pi *}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{12}$$

Then

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma max_{a'} q_*(s', a')] \end{aligned} \tag{13}$$

With these two sets of equations we can solve the question of which action is optimal relatively easily. This can be done by either finding a v_* greedily since the optimal policy will give an optimal value. Similarly with q_* the agent can simply solve for the best action at every state in order to optimize q_*

3. Dynamic Programming

In this section we'll discuss how dynamic programming can be used to solve the equations laid out in the previous section.

3.1. Policy Evaluation

Given that equation 14 satisfies the Bellman optimality principle (which it should considering that unless something has been messed up we should achieve convergence as $k \to \infty$) we should be able to solve for v_π using *iterative policy evaluation*. That is to say, that we can repeatedly generate successive approximations until we believe convergence has been achieved. Each update is considered an *expected*

update due to our evaluation of the expectation of the function over all possible next states.

In the algorithm below we can see generally speaking how iterative policy evaluation is handled, but we still haven't seen precisely how $V(s)$ is calculated with each passing loop. In the next section we'll approach the computation behind $V(S)$.

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Figure 3

3.2. Policy Improvement

Using the algorithm from the previous section we can find some deterministic policy $\pi(s)$ using v_π. That policy could be very good, but what if we want to choose an action that deviates from $\pi(s)$. To do that we need to find a way to update and improve or policy.

3.3. Policy Iteration

Each policy is updated through a sequence of evaluation and improvement operations, which are guaranteed to result in a new policy that is strictly better than the previous one. Combined with the nature of MDPs (which have a finite number of policies) we can converge upon an optimal policy within a finite number of steps using solely evaluation and improvement. We call this process policy iteration.

References

Reinforcement Learning: An Introduction, 2nd Edition, Richard S. Sutton, Andrew G. Barto, 2018.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

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

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \,|\, s, \pi(s)) \big[r + \gamma V(s') \big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \,|\, s, a) \big[r + \gamma V(s') \big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow false$
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 4

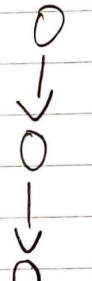
Input π_i to be eval

Initialize:
$V(s) \in \mathbb{R}$ arbitrarily for all $s \in S$
Returns$(s) \leftarrow$ empty list $\forall s \in S$

Loop forever (for each episode)
gen an ep following π: $S_0, A_0, R_1, S_1, A_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each state in the ep, $t = T-1, T-2, \ldots, 0$
$G \leftarrow \gamma G + R_{t+1}$
Unless S_t in $S_0, S_1, \ldots S_{t-1}$:
Append G to Returns(S_t)
$V(S_t) \leftarrow$ average(Returns(S_t))

Simulate many games using a policy with an MC approach then average the returns for each state

No need to pre-comp probs



policy improvement - use greedy

$$q_{\pi_k}(s, \pi_{k+1}(s)) = q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s,a))$$

$$= \max_a q_{\pi_k}(s,a)$$

$$\geq q_{\pi_k}(s, \pi_k(s))$$

$$\geq V_{\pi_k}(s)$$

$$\pi_0 \to q_{\pi_0} \to \pi_1 \to q_{\pi_1} \to \quad \to \pi_* \to q_*$$

Init

$\pi(s) \in A(s)$ arbitrarily $\forall s \in S$

$Q(s,a) \in \mathbb{R}$ $\forall s \in S, a \in A(s)$

Returns $(s,a) \leftarrow$ empty list

Loop Forever (for each ep)

Choose $S_0 \in S$, $A_0 \in A(S_0)$ randomly $(p(s_0, A_0) > 0 \ \forall A_0, S_0)$

generate an ep π: $S_0 A_0 R_1 \cdots S_{T-1} A_{T-1} R_T$

$G \leftarrow 0$

Loop for each step $t = T-1, \ldots$

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

Unless the pair S_t, A_t in our ep

Append G to Returns (S_t, A_t)

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

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

On policy first visit

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$

$\forall a \in A(S_t)$

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|A(S_t)| \\ \varepsilon/|A(S_t)| \end{cases}$$

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

$$= \frac{\varepsilon}{|A(s)|} \sum q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a)$$

$$\geq \frac{\varepsilon}{|A(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum \frac{\pi(a|s) - \frac{\varepsilon}{|A(s)|}}{1-\varepsilon} q_\pi(s,a)$$

$$= \frac{\varepsilon}{|A(s)|} \sum q_\pi(s,a) - \frac{\varepsilon}{|A(s)|} \sum q_\pi(s,a) + \sum \pi(a|s) q_\pi(s,a)$$

$$= v_\pi(s)$$

Off policy MC

Need to behave non-optimally to learn but want to act optimally

Two policies
- Target
- behavior

High var low convergence more power

b (behavior), π (target)

$\pi(a|s) > 0 \Rightarrow b(a|s) > 0$

$Pr\{A_t, S_{t+1}, A_{t+1} \dots S_T | S_t, A_{t:T-1} \sim \pi\}$

$= \pi(A_t|S_t) p(S_{t+1}|S_t, A_t) \pi(A_{t+1}|S_{t+1}) \dots p(S_T|S_{T-1}, A_{T-1})$

$= \prod_{k=t}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k)$

Importance sampling ratio - ~~estimating expected values~~ of one distro from another

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k) p(S_{k+1}|S_k, A_k)}{b(A_k|S_k) p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

$E[G_t | S_t = s] = v_b(s)$ - won't give us π

$E[\rho_{t:T-1} G_t | S_t = s] = v_\pi(s)$

$$V(\tau) = \frac{\sum_{t \in \tau(\tau)} P_{t:T(t)-1} \, G_t}{|\tau(\tau)|}$$

τ given here

weighted version

$$V(\tau) = \frac{\sum_{t \in \tau(\tau)} \rho_{t:T(t)-1} \, G_t}{\sum_{t \in \tau(\tau)} \rho_{t:T(t)-1}}$$

Want

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum W_k}$$

$$V_{n+1} = V_n + \frac{W_n}{C_n}\left[G_n - V_n\right] \qquad (C_n \text{ is sum of weights})$$

$$C_{n+1} = C_n + W_{n+1}$$

MCTS

1.) Selection: Follow the tree policy to a leaf

2.) Expansion: expand the tree by adding a new leaf node to the tree

3.) Simulation: From the selected node or new child run a MC episode with the rollout policy

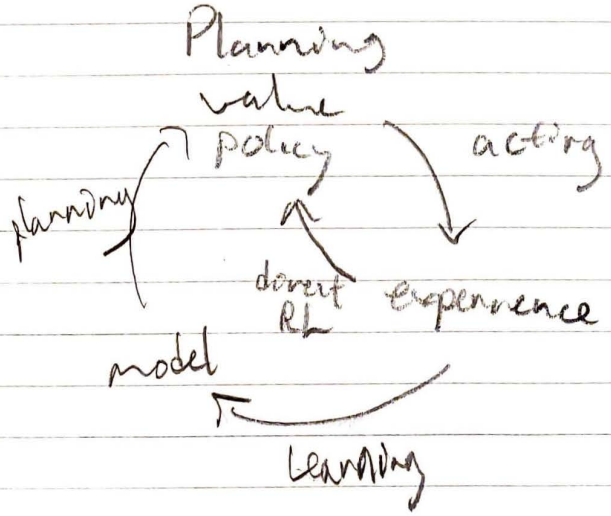
4.) Backup: Return the generated episode and its return

Rollout Algos

Estimate action values by averaging the rewards of many trajectories that start with a given action.

Unlike MC we don't estimate q_* or q_π

Instead they only get values for only the actions associated with the current state.



Tab Dyna Q

Init $Q(s,a)$, $Model(s,a)$ $\forall s \in S, a \in A(s)$

Loop:

$S \leftarrow$ current state

$A \leftarrow \varepsilon$-greedy (S,Q)

Take action A: observe resultant reward R, S'

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

$Model(S,A) \leftarrow R, S'$ (assumes deterministic env)

Loop n times:

$S \leftarrow$ random previous state

$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow Model(S,A)$

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