

Notes while reading Reinforcement learning an introduction (Sutton/Barto)

Willem

January 2013

Contents

1	Introduction	1
2	Mutli-armed bandits	3
3	Finite Markov Decision Process	5
4	Dynamic Programming	7
4.1	Exercise 4.8	7
5	Monte Carlo Methods	9
5.1	Exercises	9
5.1.1	Exercise 5.1 page 94	9
5.1.2	Exercise 5.2 page 94	9
5.1.3	Exercise 5.4 page 99	9
5.1.4	Exercise 5.5 page 105	9
5.1.5	Exercise 5.6 page 108	10
5.1.6	Exercise 5.7 page 108	10
5.1.7	Exercise 5.8 page 108	10
5.1.8	Exercise 5.11 page 111	11
6	TD Prediction	13
6.1	Summary	13
6.1.1	TD prediction	13
6.1.2	TD Advantages	13
6.1.3	Optimality of TD(0)	14
6.1.4	SARSA	14
6.1.5	Q-Learning	14
6.1.6	Difference between SARSA and Q-Learning	14
6.1.7	Expected Sarsa	14
6.1.8	Double learning	15
6.2	Exercises	15

6.2.1	Exercise 6.1	15
6.2.2	Exercise 6.2	16
6.2.3	Exercise 6.3	16
6.2.4	Exercise 6.4	16
6.2.5	Exercise 6.5	16
6.2.6	Exercise 6.6	16
6.2.7	Exercise 6.7	17
6.2.8	Exercise 6.8	17
6.2.9	Exercise 6.11	17
6.2.10	Exercise 6.12	17
6.2.11	Exercise 6.13	17
6.2.12	Exercise 6.14	17
7	n-step bootstrapping	19
7.1	Summary	19
7.1.1	n-step TD prediction	19
7.1.2	n-step Sarsa	19
7.1.3	n-step Off-policy Learning	20
7.1.4	Per Decision Methods with Control Variates	20
7.1.5	Off-Policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm	20
7.1.6	*A Unifying Algorithm: n-step Q(Sigma)	21
7.2	Exercises	21
7.2.1	Exercise 7.1	21
7.2.2	Excercise 7.3 page 145	22
7.2.3	Excercise 7.4 page 148	22
8	Planning and learning with Tabular methods	23
8.1	Summary	23
8.1.1	Models and planning	23
8.1.2	Dyna: integrated planning an learning	23
8.1.3	When the model is wrong	23
8.1.4	Prioritized sweeping	24
8.1.5	Expected vs Sample updates	24
8.1.6	Trajectory Sampling	25
8.1.7	Real-time dynamic programming	26
8.1.8	Planning at decision time	26
8.1.9	Heuristic Search	26
8.1.10	Roll-out algorithms	26
8.1.11	Monte Carlo tree search	26
8.2	Exercises	27

8.2.1	Exercise 8.1	27
8.2.2	Exercise 8.2	27
8.2.3	Exercise 8.3 page 168	27
8.2.4	Exercise 8.4 page 168	27
8.2.5	Exercise 8.5 page 168	27
8.2.6	Exercise 8.6 page 174	28
8.2.7	Exercise 8.7 page 177	28
8.2.8	Exercise 8.8 page 177	28

Chapter 1

Introduction

Chapter 2

Mutli-armed bandits

Chapter 3

Finite Markov Decision Process

Chapter 4

Dynamic Programming

4.1 Exercise 4.8

The reward is only obtained when the capital is above 99. When the capital is at 50, there is a 50% chance you can win the game. So this obviously is the optimal policy. When you reach 51: it would be rather odd to bet the entire capital, as you don't need to risk it all to reach 100. Bigger downside, but same upside. So the best course of action is to bet with 1, see if you can grow this above 50. If you lose it, you still have a 50% chance to win by betting it all.

Chapter 5

Monte Carlo Methods

5.1 Exercises

5.1.1 Exercise 5.1 page 94

The last 2 rows in the rear means you either have 21, or 20, which means the odd's are very good you will win. (hence high value function)

The last row on the left means the dealer has an ace, so it's at an advantage to get a higher score.

The front row's are higher on the upper diagram, as there is a usable ace. Which means that if you get a bad hit that put's you over 21. It can count as 1.

5.1.2 Exercise 5.2 page 94

As this is Markov process eg. The cards drawn are not exhaustible. The odds of winning on the second time your in the same state is just as good as the first time.

5.1.3 Exercise 5.4 page 99

The "Append G to Returns (S_t, A_t) would be replaced by increasing a count and added it as running average to some table.

5.1.4 Exercise 5.5 page 105

question: Consider an MDP with a single Non-terminal state and a single action that transitions back to the nonterminal state with probability p and transitions to the terminal state with probability

$p-1$. Let the reward be 1 on all transitions, and let $\gamma = 1$. Suppose you observe one episode that lasts 10 steps, with a return of 10. What are the first-visit and every visit estimators of the value of the non-terminal state.

10 Steps means 9 towards the non-terminal, and one towards the terminal. The rewards are all-way's the same so the final cost=10.

If $\gamma = 1$ then $G = G + \gamma R_{k+1}$ in every iteration.

In case of all visit the complete horizon counts 10 times in the non-terminal state, as the 10th time we leave the non-terminal state for good and enter the terminal state. $(1+2+3+4+5+6+7+8+9+10)/10 = 55/10 = 5.5$ So the value is 5.

In case of the first-visit, we only count the first visit which has a reward of 1.

5.1.5 Exercise 5.6 page 108

question: What is the equation analogous to (5.6) for action values $Q(s, a)$ instead of state values $V(s)$, again given returns generated using b?

$Q(s, a)$ is similar to $V(s)$, it takes the $V(s)$ given a certain step was taken first.

$$Q(s, a) = \frac{\sum_{t \in J(s, a)} \rho_{t+1:T(t)-1} G_t}{\sum_{t \in J(s, a)} \rho_{t+1:T(t)-1}} \quad (5.1)$$

5.1.6 Exercise 5.7 page 108

question: In learning curves such as those shown in Figure 5.3 error generally decreases with training as indeed happened for the ordinary importance-sampling method. But for the weighted importance-sampling method error first increased and then decreased. Why do you think this happened

If there are but a few samples, the bias will be the dominating error. And it will increase as more and more samples are added. Until there are so many samples, it starts to disappear.

5.1.7 Exercise 5.8 page 108

question: The results with Example 5.5 and shown in Figure 5.4 used a first-visit MC method. Suppose that instead an every-visit MC method was used on the same problem. Would the variance

of the estimator still be infinite? Why or why not? A first Visit MC has less terms than a every Visit MC. All terms have a positive value, so it would also go to infinite.

5.1.8 Exercise 5.11 page 111

If the target policy is a greedy deterministic policy, and the loop is broken off if $\pi(S_t) \neq A_t$. Then $\pi(A_t|S_t) = 1$ by definition.

Chapter 6

TD Prediction

6.1 Summary

6.1.1 TD prediction

The basic formula for monte carlo prediction is $V(S_t) = V(S_t) + \alpha[G_t - V(S_t)]$. G_t is the final result, this means that the update only can happen at the end of the simulation. By replacing G_t with $R_{t+1} + \gamma V(S_{t+1})$ we get the TD method $V(S_t) := V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$.

The update of the TD method is called the **TD error** $\delta = G_t - R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$. An equivalent entity exists with monte carlo methods, and is called the monte-carlo error. The monte carlo error can be written as a sum of TD errors, illustrated by equation 6.1. (proof on page 121)

$$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k \quad (6.1)$$

6.1.2 TD Advantages

1. No model of the behavior is required
2. Naturally online/incremental algorithm (useful with long episodes)
3. Learns from experimental choices (monte carlo need to discard them)
4. In practice faster than monte carlo methods

6.1.3 Optimality of TD(0)

When using batch learning, as in only changing the value function everytime a whole batch is processes. TD(0) and Monte Carlo do not converge to the same solution. Monte Carlo methods finds the solution that minimized the error on the dataset. TD(0) finds the parameters that most like would cause a markov process to result in the dataset. This is called the certainty-equivalence estimate.

6.1.4 SARSA

SARSA stands for $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$. It uses an policy to generate A_t and A_{t+1} . Updates the Q value, applies A_{t+1} and then finds the next input A_{t+2} .

$$Q(S_t, A_t) := Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (6.2)$$

6.1.5 Q-Learning

Q-learning acts greedily in when predicting, but acts according to it's policy when finding an input to apply to the system. So in contrast to SARSA it won't reuse A_{t+1} it generated when predicting.

$$Q(S_t, A_t) := Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (6.3)$$

6.1.6 Difference between SARSA and Q-Learning

SARSA will act a bit more carefull, as it's prediction is not greedy. And it takes into account that the next action might not be the best one. Q-Learning will take the more risky route, as it uses the best(according to $Q(S, A)$) possible action in it's prediction.

6.1.7 Expected Sarsa

Expected SARSA uses the expected value of all possible actions A_{t+1} given the policy. Then it uses a greedy policy to act, just like with Q-learning. Expected Sarsa will work with $\alpha = 1$, which would not work very well with classical SARSA. This makes the short term behavior much better. But is more computational expensive.

$$Q(S_t, A_t) := Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \mathbb{E}[Q(A_{t+1}, S_{t+1})|S_{t+1}] - Q(S_t, A_t)] \quad (6.4)$$

6.1.8 Double learning

Equation 6.3 uses an argmax to estimate the value of Q . If one of these estimates is over-estimated, it will result in bad behavior(bias). Double learning reduces the odds of this happening by using two $Q(A, S)$ estimates. One to find the maximum action, and one to estimate it's value.(equation 6.5) It's less like that the overestimate will happen this way.

$$A = Q_2(\arg \max_a Q_1(S, a)) \quad (6.5)$$

It's good practice to swap Q_1 and Q_2 in equation 6.5 constantly. For example at random with odds 50/50.

6.2 Exercises

6.2.1 Exercise 6.1

$$V_{t+1}(s_t) = \alpha[R_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)] + V_t(s_t) \quad (6.6)$$

The difference between the value function at time t and $t+1$ is defined by equation 6.6.

The equality $G_t = R_{t+1} + \gamma G_{t+1}$ still holds. However the monte carlo error is slightly different in every iteration. $G_t - V_t(s_t)$ becomes $G_{t+1} - V_{t+1}(s_{t+1})$ in the next iteration. As the value function now changes at iteration t , with a difference of $d_t = \alpha[R_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)]$.

$$G_{t+1} - V_t(S_{t+1}) = G_{t+1} - V_{t+1}(S_{t+1}) - d_{t+1} \quad (6.7)$$

$$error = - \sum_{k=t+1}^{T-1} \gamma^{k-t} d_{k-1} \quad (6.8)$$

In conclusion the different factor is equation 6.8.

6.2.2 Exercise 6.2

If (as explained in the example of the hint) a part of the statespace is already well estimated. Then the TD prediction will be very good as you enter those states and if your path ends on one of those states. So you only have lesser predictions while in an unexplored part.

The Monte Carlo approach would still need to evaluate through the already well estimated part. Which is rather slow.

6.2.3 Exercise 6.3

The change on a value function is defined by: $\alpha[R_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)] = 0.1[0 + 0 - 0.5] = -0.05$ if $V_t(s_{t+1}) = 0$ so it ends on the left terminal state. And $\alpha = 0.1$ and $V_t(A) = 0.5$.

6.2.4 Exercise 6.4

The TD algo is over-fitting when $\alpha > 0.05$ we could try to make it a bit smaller. But at $\alpha = 0.05$ it seems to flatten out nicely, so I would not expect better results.

A similar story with the MC method, this time at $\alpha 0.02$ we get a nice flat tail. It's not as clear as with the TD method, but that's due the larger variance on the MC method.

So no, I would not expect any changes in results if more samples were ran with different values for α .

6.2.5 Exercise 6.5

Overfitting, the step is too large so TD cannot find the optimal values. But keeps over/under estimating every time it runs through an episode.

6.2.6 Exercise 6.6

You setup the bellman optimality equation, and the pick a method to solve it. As this is a rather simple example, you could just manually solve the equation.

$$\begin{aligned}
V(A) &= 0.5V(B) \\
V(B) &= 0.5V(A) + 0.5V(C) \\
V(C) &= 0.5V(B) + 0.5V(D) \\
V(D) &= 0.5V(C) + 0.5V(E) \\
V(E) &= 0.5V(D) + 0.5
\end{aligned} \tag{6.9}$$

This seems like the simplest way to do it, as it's small.

6.2.7 Exercise 6.7

The normal on-policy TD(0) update looks like $V(s_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$. I would expect that $\alpha = \frac{\rho}{\sum_t \rho^t}$ as it becomes a weighted average due too the importance sampling.

6.2.8 Exercise 6.8

todo, not hard, but a bit of bookkeeping to be done.

6.2.9 Exercise 6.11

In Q-learning the actions that are applied to the system are learning through a ϵ -greedy policy(behavior policy) are not used for the prediction(Q). This is by definition an off-policy control.

6.2.10 Exercise 6.12

It would be nearly the same, SARSA selects the next action before updating Q and Q-learning selects it after. So the update of Q might make a difference in some cases.

6.2.11 Exercise 6.13

todo

6.2.12 Exercise 6.14

todo

Chapter 7

n-step bootstrapping

7.1 Summary

7.1.1 n-step TD prediction

Monte Carlo updates the estimate $V(S)$ using the complete return (equation 7.1). TD(0) only watches one step ahead, a compromised is too take an n-step prediction window as illustrated by equation 7.2. The state learning algorithm then becomes equation 7.3.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-t-1} R_T \quad (7.1)$$

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \quad (7.2)$$

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha [G_{t:t+n} - V_{t+n-1}(S_t)] \quad (7.3)$$

7.1.2 n-step Sarsa

The previous Sarsa is often called Sarsa(0), the generalized version is called n-step Sarsa. The return value can be estimated by equation 7.4. The update rule for Q then becomes equation 7.5.

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} \dots \gamma^{n-1} R_{t+n-1} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \quad (7.4)$$

$$Q_{t+n}(S_t, A_t) := Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \quad (7.5)$$

The same logic can be applied to expected SARSA. The value function can be defined as equation 7.6. Using equation 7.6 the n-step expected sarsa

return value is defined by equation 7.7. The update is still equation 7.5 but using the new $G_{t:t+n}$ from equation 7.7.

$$V_t(S_t) = \sum_a P(a)Q(S_t, a) \quad (7.6)$$

$$G_{t:t+n} = R_t + \gamma R_{t+1} + \dots \gamma^{n-1} R_{t+n-1} + V_t(S_{t+n}, A_{t+n}) \quad (7.7)$$

7.1.3 n-step Off-policy Learning

$$Q_{t+n}(S_t, A_t) := Q_{t+n-1}(S_t, A_t) + \rho_{t:t+n} \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \quad (7.8)$$

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \rho_{t:t+n-1} \alpha [G_{t:t+n} - V_{t+n-1}(S_t)] \quad (7.9)$$

$$\rho_{t:h} = \prod_{k=t}^{\min(h, T-1)} \frac{\pi(A_k | S_k)}{b(A_k | S_k)} \quad (7.10)$$

7.1.4 Per Decision Methods with Control Variates

The previous sections are inefficient implementations of the n-step algorithm. The n-step return can be written recursively as $G_{t:h} = R_{t+1} + \gamma G_{t+1:h}$, with $G_{h:h} = V_{h-1}(S_h)$. The importance sampling weighting is still $\rho_t = \frac{\pi(S_t, A_t)}{b(S_t, A_t)}$. Using this recursive definition we can define the return as equation 7.11.

$$G_{t:h} = \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho)V_{h-1}(S_t) \quad (7.11)$$

The term $(1 - \rho)V_{h-1}(S_t)$ in equation 7.11 is called the control variate. It has an expected value of one because $\mathbb{E}[\rho] = 1$ so $\mathbb{E}[1 - \rho] = 0$.

The return state of a n-step off policy with control variate is defined by equation 7.12. The recursion ends with $G_{h:h} = Q_{h-1}(S_h, A_h)$

$$G_{t:h} = R_{t+1} + \gamma[\rho_{t+1}G_{t+1:h} + V_{h-1}(S_{t+1}) - \rho_{t+1}Q_{h-1}(S_{t+1}, A_{t+1})] \quad (7.12)$$

7.1.5 Off-Policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm

Instead of only using the value estimations of the actual path taken in n-steps. We can generalize the expected SARSA algorithm, from equation 7.13 to equation 7.14.

$$G_{t:t+1} = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q_t(S_{t+1}, a) \quad (7.13)$$

$$\begin{aligned} G_{t:t+n} &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n} \\ &= R_{t+1} + \gamma \pi(A_{t+1}|S_{t+1}) [G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})] + \gamma V_{h-1}(S_{t+1}) \end{aligned} \quad (7.14)$$

Adding the control variate to Equation 7.14 gives equation 7.15.

$$G_{t:t+n} = R_{t+1} + \gamma \pi(\sigma_{t+1} \rho_{t+1} + (1 - \sigma_{t+1}) \pi(A_{t+1}|S_{t+1})) [G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})] + \gamma V_{h-1}(S_{t+1}) \quad (7.15)$$

7.1.6 *A Unifying Algorithm: n-step Q(Sigma)

7.2 Exercises

7.2.1 Exercise 7.1

The Monte carlo error can be written as a sum of TD errors, with TD(0) this becomes:

$$\begin{aligned} G_t &= R_{t+1} + \gamma G_{t+1} \\ \delta_t &= R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \\ G_t - V(S) &= R_{t+1} + \gamma G_{t+1} - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k \end{aligned}$$

With an n step we get:

$$\begin{aligned} G_t &= R_{t+1} + \gamma G_{t+1} \\ \delta_t &= \sum_{k=1}^n \gamma^{k-1} R_{t+k} + \gamma^n V(S_{t+n}) - V(S_t) \end{aligned}$$

By putting them together we get:

$$\begin{aligned} &G_t - V(S_t) \\ &= R_{t+1} + \gamma G_{t+1} - V(S_t) \\ &= R_{t+1} + \gamma G_{t+1} - V(S_t) \\ &+ \sum_{k=2}^n \gamma^{k-1} R_{t+k} - \sum_{k=2}^n \gamma^{k-1} R_{t+k} \\ &+ \gamma^n V(S_{t+n}) - \gamma^n V(S_{t+n}) \\ &= \delta_t + \gamma (G_{t+1} - \gamma^{n-1} V(S_{t+n})) - \sum_{k=2}^n \gamma^{k-1} R_{t+k} \end{aligned}$$

I don't see how to continue from here.

7.2.2 Exercise 7.3 page 145

question: Why do you think a larger random walk task (19 states instead of 5) was in the examples of this chapter? Would a smaller walk have shifted the advantage to a different value of n ? How about the change in left-side outcome from 0 to -1 made in the larger walk? Do you think that made any difference in the best value of n ?

If a random walk with length of 5 was used the results would not be best around an n of 4 or 8. As you need a large enough episode to learn from a step of 4 or 8 samples.

When the walk is smaller, smaller n 's will give better results than current results.

If the left side is negative, the first time the walk will go into it. That negative result will propagate n -steps, instead of 1 with $TD(0)$.

7.2.3 Exercise 7.4 page 148

Similar to exercise 7.1, still TODO, first finish 7.1.

Chapter 8

Planning and learning with Tabular methods

8.1 Summary

8.1.1 Models and planning

Planning uses a model of the System, while learning uses experience.

There are 2 major types of planning

1. State space planning
2. Plan space planning (for example genetic programming)

8.1.2 Dyna: integrated planning and learning

A value function/policy leads to an action, this results in experience that can directly improve the value/policy via reinforcement learning. It can also be used to improve a model of the system. A planning algorithm can then improve the policy/value function via the model. This is illustrated in figure 8.1.

Dyna Q is an example of a Dyna algorithm. It uses Q-learning on the experience to improve the value function. Saves the state/action pair with the result. And then uses Q-learning on the model(table) to further improve the value function.

8.1.3 When the model is wrong

If the optimal path is suddenly significantly worse than before, it takes quite a while to adjust with Dyna. But even worse is when a better path sud-

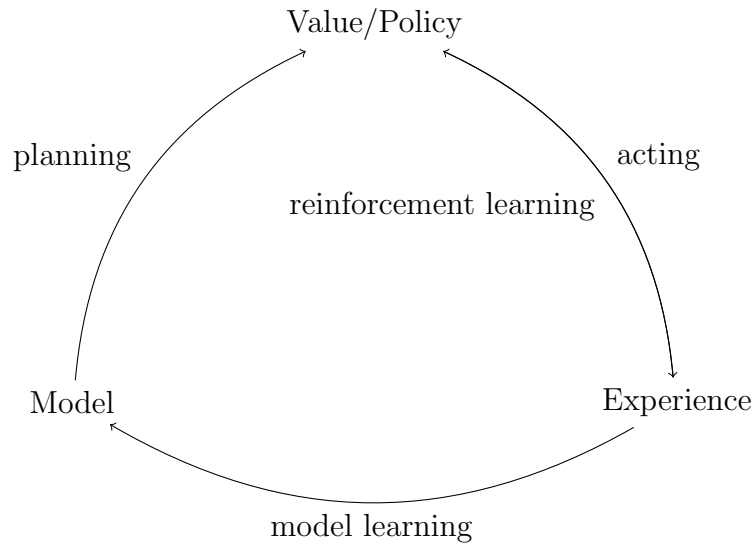


Figure 8.1: Dyna

denly becomes available, as Dyna has no reason to look for it. It might stay undiscovered.

DynaQ+ Tries to asses this weakness by adding a bonus reward on each state in the value function when planning. $r + k\sqrt{\tau}$, where r is the normal simulated reward, k is a small number (parameter), and τ is the number of step since the last visit.

8.1.4 Prioritized sweeping

Prioritized sweeping is a form of planning called **backward focusing**. When a state value changes, it identifies the predecessor states and the change in value function. It puts them on a queue, ordered according to their value function change. The state with the highest change then gets the same treatment.

The opposite approach would be **forward focusing** which updates all the successors of a state that has a value change.

8.1.5 Expected vs Sample updates

Value function updates have 3 binary dimensions:

1. state vs action value function
2. estimated using arbitrary policy or optimal policy

3. expected vs sample update

This sums up to 8 possible combinations illustrate in table 8.1.5.

value estimation	expected updates	sample updates
v_π	policy evaluation	TD(0)
v_*	value iteration	
$q_\pi(s, a)$	q-policy evaluation	sarsa
$q_*(s, a)$	q-value iteration	Q-learning

Table 8.1: table of one step updates

The **expected update** is equation 8.1 uses \hat{p} the estimated dynamics of the system to estimated the probability of getting a certain state with a certain result.

The **sample update** from equation 8.2 doesn't need a model as updates with the actual state/action and reward applied to the system. This makes the sample update a lot less computational expensive.

If the system only has 1 possible next state, then the expected and sample update are the same. However once there is more then 1 possible next state, there is a **sample bias** on the sample updates.

The error of the sample update drops with $\sqrt{\frac{b-1}{bt}}$, where b is the branching factor, the number of possible next states s' for which $\hat{p}(s'|s, a) > 0$. And t the number of sample updates. This formula suggests that **sample updates are good with system with large branching factors**.

$$Q(s, a) = \sum_{s', r} \hat{p}(s', r|s, a) [r + \gamma \max_{a'} Q(s', a')] \quad (8.1)$$

$$Q(s, a) = Q(s, a) + \alpha [R + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (8.2)$$

8.1.6 Trajectory Sampling

In dynamic programming there are 2 way's to distribute updates of the value function. Either **exhaustively** in 1 sweep, or **according to some distribution**.

On-policy learning **ϵ -greedy will converge very fast**, as it skips a large part of the irrelevant state space. It will however not converge to the actual value in the long run, but will have some sample bias.

8.1.7 Real-time dynamic programming

RTDP is an on-policy trajectory sampling value iteration algorithm.

RTDP can find optimal policies without infinitely visiting all states. It's very good with big and stochastic problems.

8.1.8 Planning at decision time

The planning phase of an DP algorithm can happen on the background, this is called **background planning**.

Decision time planning only does the planning when a new state change occurs. It's less accurate, but has very fast response times.

8.1.9 Heuristic Search

Heuristics have man made value functions, they look at the now, so no simulations.

8.1.10 Roll-out algorithms

Decision-time planning based on Monte Carlo Control. They estimated the action values from the current state by simulating trajectories from the current state. And when the action value is accurate enough it applies the action. And forgets all the past runs. The simulations itself can be run in parallel.

The name **rollout** comes from the game backgammon. Roll-out algorithms were studied on backgammon games, the algorithm would randomly select dice rolls and play out the game.

8.1.11 Monte Carlo tree search

A **tree policy** keeps track of value function via tree of state/action values. It executes **monte carlo** simulations at the leafs of the trees to get return values to update the tree.

It's a **roll-out** algorithm as it starts from 1 specific state, and forgets everything after an action was taken. **decision time planning**. (although some information could be reused for the next tree)

There are 4 major steps:

1. Selection: A tree policy based on action values on the tree selects a leaf node

2. Expansion: Potentially the tree could be expanded with a new leaf by the tree policy
3. Simulation: A roll-out policy simulates to the end of the episode
4. Backup: The return value of the roll-out episode is backup-ed into the action values of the tree.

8.2 Exercises

8.2.1 Exercise 8.1

A Dyna method can do much better than a multi-step method. If the path is longer than the horizon of the multi-step method. The Dyna method can propagate the end result over the entire path. While the multi-step method is limited to its horizon.

8.2.2 Exercise 8.2

Dyna+ will get good rewards on previously bad states if that state was not visited for a while. Pushing the algorithm to re-visit it from time to time. The parameter k controls how much exploration Dyna+ needs to do.

8.2.3 Exercise 8.3 page 168

Dyna and Dyna+ had bad rewards on the previously good path. So relatively speaking older not so good states are getting better. Dyna+ accelerates this effect by also adding the bonus reward.

8.2.4 Exercise 8.4 page 168

programming task TODO

8.2.5 Exercise 8.5 page 168

The algorithm should keep updating its value function, but it might be a good idea to use a moving average. The moving average will help with the stochastic environment, but will make the change environment worse. So a trade-off on the window size should be made.

8.2.6 Exercise 8.6 page 174

If some new states are much more likely than others, then sample updates will work a lot better. As the very likely transitions will be the most influential on the behavior of the system, and due to be likely will also be very present in the sample updates.

8.2.7 Exercise 8.7 page 177

We observe b goes up, the lines get smoother. When b is very low, one good sample update can significantly improve things. While if b is large, the optional actions are more spread out as the next state s' is more stochastic. So you can only improve it on average. (not sure about this)

8.2.8 Exercise 8.8 page 177

programming TODO