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

Willem

January 2013

Contents

Chapter 1

Introduction

Chapter 2

Mutli-armed bandits

Chapter 3

Finite Markov Decision Process

3.1 Summary

MDP's are a formal framework to model sequential decision making. It can handle immediate and delayed rewards, incorporating state.

3.1.1 Agent-Environment Interface

1. agent: Leaner and decision maker.
2. environment: The thing the agent interacts with.
3. $P(s', r|s, a)$: dynamics of the system (as probability distribution)

If the next state S_{t+1} only depends on the current state S_t and the input A_t, then the state is said to have the **Markov property** .

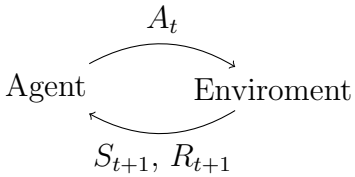

Figure 3.1: Agent Environment

3.1.2 Goals and rewards

The **reward hypothesis** say's that all goals/purposes can be expressed as maximizing reward.

Reward should only communicate **what** needs to be done, not **how**.

3.1.3 Returns and episodes

An **episodial task** has a finite number of steps, until it stops in the absorbing state. $G_t = R_{t+1} + R_{t+2} + ... + R_T$.

A **continuing task** never stops, so it keeps getting rewards. So an extra concept **discount factor**(γ) is needed to define the expected reward.

$$\begin{aligned} G_T &= R_{t+1} + \gamma R_{t+2} + ... \\ &= \sum_{k=0}^{\infty} \gamma R_{t+k+1} \\ 0 &\leq \gamma \leq 1 \end{aligned} \tag{3.1}$$

The discount factor is a geometric series, and equals to one.

$$\sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{3.2}$$

3.1.4 Policies and value function

The **state-value function** expressed how it is to be in a certain state under a certain policy(π). It has a recursive definition that is derived in equation 3.3 and known as the **bellman equation**.

$$\begin{aligned} V_\pi(s) &= \mathbb{E}\left[G_t | S_t = s\right] \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma\, \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \end{aligned} \tag{3.3}$$

The value of taking action a under state s is defined by the **action-value function** equation 3.4.

$$\begin{aligned} q_\pi(a,s) &= \mathbb{E}[G_t|S_t=s,A_t=a] \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t=s,A_t=a\right] \\ &= \sum_{r,s'} p(s',r|s,a)(r+\gamma v_\pi(s')) \end{aligned} \tag{3.4}$$

3.1.5 Optimal policies and optimal value function

Solving a **reinforcement learning** problem is finding a policy that gets a lot of reward. The best possible policy is called the **optimal policy**, the value-state function of this policy is the **optimal state-value function** $v_*(s) = \max_\pi v_\pi(s)$. And the action-value function is the **optimal value-state function** $q_*(s,a) = \max_\pi q_\pi(s,a)$

$$\begin{aligned} v_* &= \max_a q_\pi(s,a) \\ &= \max_a \mathbb{E}\left[G_t|S_t=a,A_t=a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1}+\gamma G_t|S_t=a,A_t=a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1}+\gamma V_*(S_{t+1})|S_t=a,A_t=a\right] \\ &= \max_a \sum_s p(s',r|a,s)[r+\gamma V_*(s')] \end{aligned} \tag{3.5}$$

The optimal state-value and action-value functions lead to **the bellman optimality equations** equation 3.5 and equation 3.6

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1}+\gamma \max_{a'} q(S_{t+1},a')|S_t=s,A_t=a\right] \\ &= p(s'r|s,a)[r+\gamma \max_{a'} q(s',a)] \end{aligned} \tag{3.6}$$

3.2 Exercises

3.2.1 Exercise 3.1

A Robot in a maze has a delayed reward, and needs to make a sequence of decisions. The position of the robot in the maze is the state, and the input is the decisions left/right/straight ahead. The reward is -1 until the absorbing state, which has a reward of zero.

A automatic poker player can be a mdp, the state is the current cards in the hand and the table.

Chapter 4

Dynamic Programming

4.1 Summary

Dynamic programming is a **collection of algorithms** that can be used to find the **optimal policy**. It assumes a perfect model of the system (MDP) and uses a lot of computational power.

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{4.1}$$

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{4.2}$$

4.1.1 Policy evaluation

The bellman equation from 3.3 can be converted into an iterative method called **iterative policy evaluation** to find the value function. It takes the expected value over all the same next states. All updates in dynamic programming are called **expected updates**, because they are based on expectation over all possible next states rather then the sample next states.

$$\begin{aligned} v(s)_{k+1} &= \mathbb{E}_\pi \left[R_{t+!} + \gamma v_k(S_{k+1})|S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v(s')\right] \end{aligned} \tag{4.3}$$

4.1.2 Policy improvement

$$\begin{aligned} q_\pi(a,s) &= \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_\pi(s')\big] \end{aligned} \tag{4.4}$$

Given a policy π and value function $v_\pi(s)$, one action a can be selected that maximizes equation 4.4 and all sequential actions follow the policy π. The **policy improvement theorem** say's that if a new policy π' satisfies equation 4.5, the the new policy will satisfy equation 4.6. And be as good or better then the original policy.(proof on page 78-79 of the book)

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{4.5}$$

$$v_\pi(s) \leq v_{\pi'}(s) \tag{4.6}$$

The new improved policy π' is formally written down in equation 4.7. The corresponding value function is formally written down in equation 4.8. Where we **end up with the bellman optimality equation**. Indicating that the policy can improve until it's the optimal policy.

$$\begin{aligned} \pi'(s) &= \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}|S_t = s, A_t = a)\right] \\ &= \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \end{aligned} \tag{4.7}$$

$$\begin{aligned} v_{\pi'}(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v'_\pi(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_{\pi'}(s')\right] \end{aligned} \tag{4.8}$$

4.1.3 Policy iteration

The iterative process of evaluating a policy, and then creating a new policy that is greedy towards the old one is called **policy iteration**.

4.1.4 Value iteration

Instead of evaluating the complete policy first, and then improving the policy. The policy can be improved after every state evaluation. Effective **turning**

the bellman optimality equation into the iterative update of equation 4.9.

$$v_{k+1} = \max_a = \sum_{s',r} p(s', r|s, a)\big[r + \gamma v_k(s')\big] \tag{4.9}$$

4.1.5 Generalized policy iteration

The iterative process of repeatedly evaluating a policy and using it to create an improved version of that policy, is referred to as **generalized policy iteration** or short GPI. Both policy iteration and value iteration are GPI, as do many stochastic methods.

4.2 Exercises

4.2.1 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 Summary

Monte Carlo methods require not model of the system, they work with samples. So they are based on **experience**.

5.1.1 Monte Carlo Prediction

Monte carlo prediction creates a trajectory/sample of the system. Starting from a certain state, following a certain policy v_π. And averaging the returns (equation 5.1)to the value function. The error on the standard deviation of the value function drops by $\frac{1}{\sqrt{n}}$ with n as the number of average returns.

$$G_t := \gamma G_{t+1} + R_{t+1} \tag{5.1}$$

5.1.2 Monte Carlo Estimation of action values

If no model is available $v_\pi(s)$ is not sufficient, as it's not clear what actions can be taken. We need to estimate the value function of the state/action pair $q(s, a)$.

Monte Carlo methods can suffer from the **problem of maintaining exploration**, as not all state/action combinations might be visited. One solution the this problem is using **exploring starts**, every state/action combination has an equal probability to be used as start state/action.

5.1.3 Monte Carlo Control with exploring starts

Just as with **dynamic programming** the principle of **generalized policy iteration** can be used. Monte Carlo exploring state is illustrated in Fig-

ure 5.1.3, it uses a random start pair to avoid the **exploring state problem**.

1. Take random S_0 and A_0
2. Generate an entire episode
3. Average the returns
4. Create an improved policy $\pi(s_t) = \arg\max_a +(s_t, a)$ and repeat

Figure 5.1: Monte Carlo Exploring starts

5.1.4 Monte Carlo without exploring starts

Online policy methods generally use **soft policies**. Soft policies have a no-zero probability for every action $p(a|s) > 0$.

On-policy method

ϵ-Greedy is a commonly used online policy, illustrated in equation 5.2. Generaly Policy iteration only requires that the policy moves towards the greedy policy. Which is still true, just a bit slower.

$$\begin{aligned} p_{non-greedy} &= \frac{\epsilon}{A(s)} \\ p_{greedy} &= 1 - \epsilon + \frac{\epsilon}{A(s)} \end{aligned} \tag{5.2}$$

Off-policy method

- $\pi(a|s)$: target policy
- $b(a|s)$: behavior policy
- Assumption of coverage: $b(a|s) > 0$
- G_t return after t
- $\tau(s)$ set of all time steps when s was visited.
- $T(t)$ first time of termination

Importance sampling is used in off-policy methods to translate the returns from the behavior policy to the target policy. Given a starting state S_t, the probability of a certain trajectory is defined by equation 5.3. The relation between the likelyhood of a trajectory using the behavior policy and the target policy is called the **importance-sampling ratio**. It's defined by equation 5.4. The value function of the target policy simulate under the behavior policy is equation 5.5.

$$\begin{aligned} &p(A_t, S_{t+1}, A_{t+1}...S_T|S_t, A_{t:T-1} \sim \pi) \\ &= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k) \end{aligned} \tag{5.3}$$

$$\begin{aligned} \rho_{t:T-1} &= \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} \\ &= \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)} \end{aligned} \tag{5.4}$$

$$\mathbb{E}[\rho_{t:T-1}G_t|s_t = s] = v_\pi(s) \tag{5.5}$$

There are 2 variants of importance sampling that can be used. Either **ordinary** (equation 5.6) or **weighted** (equation 5.7). The ordinary makes at first sight the most sense as it has no bias. It does however have an unbounded variance, which weight importance simple does not have. In practice weight variance tends to perform better.

$$V_{\text{ordinary}}(s) = \frac{\sum_{t\in\tau(s)} \rho_{t:T(t)-1}G_t}{|\tau(s)|} \tag{5.6}$$

$$V_{\text{weighted}}(s) = \frac{\sum_{t\in\tau(s)} \rho_{t:T(t)-1}G_t}{\sum_{t\in\tau(s)} \rho_{t:T(t)-1}} \tag{5.7}$$

ordinary importance sampling	weighted importance sampling
unbiased	biased but asymptotically converges to zero
unbound variance	bound variance

Importance sampling can use recursion to incrementally implement the value function.(equation 5.8) with $W_n = \rho_{t_n:T(t)-1}$

$$
\begin{aligned}
V_{k+1} &= V_n + \frac{W_n}{C_n}[G_n - V_n] \\
C_{n+1} &= C_n + W_{n+1} \\
W_{n+1} &= \frac{\pi(A_t|S_t)}{b(A_t|S_t)} W_n
\end{aligned} \tag{5.8}
$$

5.1.5 Off-Policy Monte Carlo Control

Algorithm 5.1 Off policy monte carlo control

$G \leftarrow 0$
$W \leftarrow 1$
for $t \leftarrow T - 1$ to 0 **do**
  $G \leftarrow \gamma G + R_{t+1}$
  $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
  $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
  $\pi(S_t) \leftarrow \arg\max_a(Q(S_t, a))$
  if $A_t \neq \pi(S_t)$ **then**
    break
  end if
  $W = \frac{1}{b(A_t|S_t)}$
end for

It's important to note that algorithm 5.1 only can learn from the tail of the trajectories. This can make the algorithm rather slow, if this is a problem, this can be addressed by using temporal difference learning.

5.1.6 Discount-aware Importance Sampling

Importance sampling calculates ρ using all the factors, even if γ is close to zero, when the returns don't really matter after a few timesteps. They do however still influence the importance factor, and so still increase the variance. Discount away importance keeps this in mind.

The return value can be written as a sum of flat partial returns(equation 5.9) as demonstrated by equation 5.10.

$$
\begin{aligned}
&\bar{G}_{t:h} = R_{t+1} + R_{t+2} + ... + R_h \\
&0 \leq t \leq h \leq T
\end{aligned} \tag{5.9}
$$

$$\begin{aligned}\bar{G}_{t:h} &= R_{t+1} + \gamma R_{t+2} ... \gamma^{T-t-1} R_h \\ &= (1-\gamma) R_{t+1} \\ &+ (1-\gamma)\gamma(R_{t+1} + R_{t+2}) \\ &+ (1-\gamma)\gamma^2(R_{t+1} + R_{t+2} + R_{t+3}) \\ &... \\ &\gamma^{T-t-1}(R_{t+1} + R_{t+2} + ... + R_T) \\ &= (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} + \gamma^{T-t-1} \bar{G}_{t:T}\end{aligned} \tag{5.10}$$

Using equation 5.10 we can define ordinary and weighted importance sampling:

$$V_{\text{ordinary}}(S) = \frac{\sum_{t \in \tau(s)} (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} \rho_{t:h-1} + \gamma^{T-t-1} \bar{G}_{t:T} \rho_{t:T(t)}}{|\tau(S)|} \tag{5.11}$$

$$V_{\text{ordinary}}(S) = \frac{\sum_{t \in \tau(s)} (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} \rho_{t:h-1} + \gamma^{T-t-1} \bar{G}_{t:T} \rho_{t:T(t)}}{\sum_{t \in \tau(s)} (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} + \gamma^{T-t-1} \bar{G}_{t:T}} \tag{5.12}$$

5.1.7 Per Decision Importance Sampling

The off-policy estimator can be written as a sum of rewards as demonstrated in equation 5.13. Each of the terms has a reward **and the same importance sampling term**. This term can be written out as demonstrated by equation 5.14. The **terms can be averaged out** (equation 5.15) to 1. Bringing the average of the sampling-ratio term's of equation 5.13 to a different value as demonstrated by equation 5.16. Finally the **per decision importance sampling value function** then becomes equation 5.17.

$$\begin{aligned}\rho_{t:T-1} G_t &= \rho_{t:T-1}(R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t-1} R_T) \\ &= \rho_{t:T-1} R_{t+1} + \gamma \rho_{t:T-1} R_{t+2} + ... + \gamma^{T-t-1} \rho_{t:T-1} R_T\end{aligned} \tag{5.13}$$

$$\rho_{t:T-1} R_{t+1} = \frac{\pi(A_t|S_t)\pi(A_{t+1}|S_{t+1})\pi(A_{t+2}|S_{t+2})}{b(A_t|S_t)b(A_{t+1}|S_{t+1})b(A_{t+2}|S_{t+2})} ... \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1} \tag{5.14}$$

$$\mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k))}\right] = \sum_a b(a|S_k)\frac{\pi(a|S_k)}{b(a|S_k)} = \sum_a \pi(a|S_k) = 1 \tag{5.15}$$

$$\begin{aligned}\mathbb{E}[\rho_{t:T-1}R_{t+1}] &= \mathbb{E}[\rho_{t:t}R_{t+1}] \\ \mathbb{E}[\rho_{t:T-1}R_{t+k}] &= \mathbb{E}[\rho_{t:t+k-1}R_{t+k}]\end{aligned} \tag{5.16}$$

$$\begin{aligned}&\mathbb{E}[\rho_{t:T-1}G_t] = \mathbb{E}[\widetilde{G_t}] \\ &\widetilde{G_t} = \rho_{t:t}R_{t+1} + \gamma\rho_{t:t+1}R_{t+2} + \gamma^2\rho_{t:t+2}R_{t+3} + ... \\ &+ \gamma^{T-t-1}\rho_{t:T-1}R_T \\ &V(S) = \frac{\sum_{t\in\tau(s)}\widetilde{G_t}}{|\tau(s)|}\end{aligned} \tag{5.17}$$

5.2 Exercises

5.2.1 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 usuable ace. Which means that if you get a bad hit that put's you over 21. It can count as 1.

5.2.2 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.

5.2.3 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.

5.2.4 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 != 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.

5.2.5 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.

$$Q(s,a) = \frac{\sum_{t \in J(s,a)} \rho_{t+1:T(t)-1} G_t}{\sum_{t \in J(s,a)} \rho_{t+1:T(t)-1}} \tag{5.18}$$

5.2.6 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.

5.2.7 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 then a every Visit MC. All terms have a positive value, so it would also go to infinite.

5.2.8 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 $R_{t+1}V(S_{t+1})$ 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** $\delta = G_t-R_{t+1}V(S_{t+1})$. 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)

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

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 then 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}.

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

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.

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

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 will 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.

$$A = Q_2(\arg\max_a Q1(S, a)) \quad (6.5)$$

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 $V_t(s_{+1}) = 0$ 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 optionality 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 them becomes equation 7.3.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1} R_T \tag{7.1}$$

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

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

7.1.2 n-step Sarsa

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

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

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

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.

$$V_t(S_t) = \sum_a P(a)Q(S_t, a) \tag{7.6}$$

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

7.1.3 n-step Off-policy Learning

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

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

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

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 $G_{h:h} = V_{n-1}(S_h)$. 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.

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

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 $G_{h:h} = Q_{h-1}(S_h, A_h)$

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

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.

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

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

Adding the control variate to Equation 7.14 gives equation 7.15.

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

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:

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

With an n step we get:
$G_t = R_{t+1} + \gamma G_{t+1}$
$\delta t = \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n V(S_{t+n}) - V(S_t)$

By putting them together we get:
$G_t - V(S_t)$
$= R_{t+1} + \gamma G_{t+1} - V(S_t)$
$= R_{t+1} + \gamma G_{t+1} - V(S_t)$
$+ \sum_{k=2}^{n} \gamma^{k-1} R_{t+k} - \sum_{k=2}^{n} \gamma^{k-1} R_{t+k}$
$+ \gamma^n V(S_{t+n}) - \gamma^n V(S_{t+n})$
$= \delta_t + \gamma(G_{t+1} - \gamma^{n-1} V(S_{t+n})) - \sum_{k=2}^{n} \gamma^{k-1} R_{t+k}$

I don't see how to continue from here.

7.2.2 Excercise 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 form 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 a 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 then 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 Excercise 7.4 page 148

Similar to excercise 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 an 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 plannings 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 algorihm. 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 then before, it takes quiet a while to adjust with Dyna. But even worse is when a better path sud-

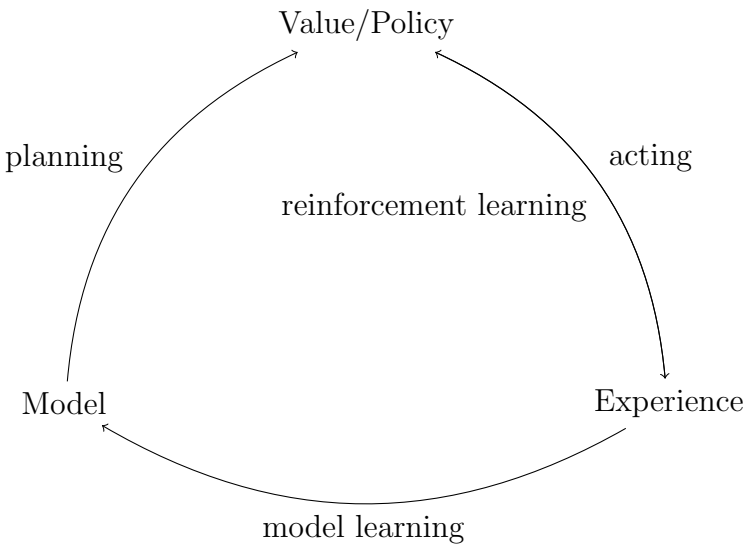

Figure 8.1: Dyna

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**.

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

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

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 then a multi-step method. If the path is longer then 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 it's 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 it's 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 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 then 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

progamming TODO

Chapter 9

On-Policy Prediction with Approximation

9.1 Summary

9.1.1 Function approximation

In the previous chapters tables were used to represent the value function. This means that every state is updated separately. If the number of states goes up, this becomes infeasible. The state function can be approximated by $\hat{v}(s, w)$, with $w \in \Re^d$, the dimension of w is much smaller then the number of states $d << |s|$. $\hat{v}$ generalizes the update of one state over many.

9.1.2 The prediction objective $\overline{VE}$

- $\mu(s)$: state distribution/on-policy distribution, indicates how important a certain state is.
- $h(s)$: probability that the episode began in state s
- $\eta(s)$: number of timesteps spend on average in state s

The value function updates don't update the state estimation directly but have to adjust the weights of a function. A common metric used is the **mean squared value error** illustrated in equation 9.1.

$$\overline{VE}(w) = \sum_{s \in S} \mu(s) \big[v_\pi(s) - \hat{v}(s, w) \big]^2 \tag{9.1}$$

When training a episodic task on-policy μ can be derived from η using equation 9.2. Solving the sytem of equations results in η, which can be used to find $\mu(s)$ $\mu = \frac{\mu(s)}{\sum_{s'} \mu(s')}$.

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_{a} \pi(a|\bar{s}) p(s|\bar{s}, a) \tag{9.2}$$

The value function $\overline{VE}$ might not always be the the best option, but it's a common used one. The updates of the approximation sometime lead to a local optimum, and not to a global one.

9.2 Stochastic gradient and semi-gradient methods

- $w = (w_1, w_2, ...w_d)^T$: weights
- $\hat{v}(s, w)$: approximation function
- w_t: update of w at timestep t

Often there will be no w that gets all the states right. So the w with the lowest error is picked by stochastic gradient descent (SGD). If the actual value function is present the update of the approximation is equation 9.3. However v_π is not known in practice, so take $v_\pi \approx U_t$ and the update rule then becomes equation 9.4.

$$\begin{aligned} w_{t+1} &= w_t - \frac{1}{2}\alpha \nabla \big[v_\pi(S_t) - \hat{v}(S_t, w_t)\big]^2 \\ &= w_t + \alpha \big[v_\pi(S_t) - \hat{v}(S_t, w_t)\big] \hat{\nabla}(S_t, w_t) \end{aligned} \tag{9.3}$$

$$w_{t+1} = w_t + \alpha \big[U_t - \hat{v}(S_t, w_t)\big] \hat{\nabla}(S_t, w_t) \tag{9.4}$$

Monte carlo methods approximate $U_t = G_t$, the update rule then becomes equation 9.5. We know that G_t is an unbiased estimate of v_π.($\mathbb{E}[U_t|S_t = s] = v_\pi(s)$). With **bootstrapping methods**(equation 9.6) this is not the case anymore, as U_t relies on the weights w_t it has a bias. They can however converge much quicker, but are as robust. We call these **semi-gradient methods**.

$$w_{t+1} = w_t + \alpha \big[U_t - \hat{v}(S_t, w_t)\big] \hat{\nabla}(S_t, w_t) \tag{9.5}$$

$$U_t = \sum_{a,s',r} \pi(a|S_t)p(s',r|S_t,a)[r + \gamma\hat{v}(s',w)] \tag{9.6}$$

So the TD(0) algorithm with approximation is a **semi-gradient method**, the update is illustrated in equation 9.7.

$$w_{t+1} = w_t + \alpha\big[R_{t+1} - \gamma\hat{v}(S_t,w_t)\big]\hat{\nabla}(S_t,w_t) \tag{9.7}$$

State aggregation is a simplified version of general function approximation. The states are grouped, and one component of w represents the whole group.

9.2.1 Linear methods

- $x(s)$: feature vector, representing basis functions
- $x_i \in S$: all features are just states
- $\hat{v}(s,w) = w^T x(s) = \sum_{i=1}^{d} w_i x_i$: linear value function
- $\nabla\hat{v}(s,w) = x(s)$: gradient value function

Monte carlo with linear approximation has the update rule of equation 9.8. It converges robustly to the global optimum.

$$w_{t+1} = w_t + \alpha\big[G_t - \hat{v}(S_t,w_t)\big]x(s_t) \tag{9.8}$$

TD(0) has the update rule of equation 9.9, which only converges to the local optimum.

$$\begin{aligned} w_{t+1} &= w_t + \alpha\big(R_{t+1} + \gamma w_t^T x_{t+1} - w_t^T x_t\big)x_t \\ &= w_t + \alpha\big[R_{t+1}x_t - x_t(x_t - \gamma x_{t+1})^T w_t\big] \end{aligned} \tag{9.9}$$

The expected TD(0) weight vector is derived from equation 9.9 in equation 9.10. The TD(0) steady state vector w_{TD} can be solved from this as a fixed point $w_{TD} = A^{-1}b$.

$$\begin{aligned} &\mathbb{E}[w_{t+1}|w_t] = w_t + \alpha(b - Aw_t) \\ &b = \mathbb{E}[R_{t+1}x_t] \in \Re^d \\ &A = \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T] \end{aligned} \tag{9.10}$$

The upper bound error of $\overline{VE}$ when using TD is defined by equation 9.11. The Monte Carlo error is $\min_w \overline{VE}$. When γ is near zero the error goes up drastically, but because it's a TD method, it has low variance.

$$\overline{VE}(W_{TD}) \leq \frac{1}{1-\gamma} \min_w \overline{VE}(w) \tag{9.11}$$

9.2.2 Feature construction for Linear Methods

A big limitation of the linear form is that no interactions between features can be modeled. Such as presence of a feature being good only in the absence of an other feature.

Polynomials

Polynomials are a simple way to model interpolation and regression. The calculation of the weights w is still a linear problem.

- high dimensional feature vector $x(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1^2 s_2, s_1 s_2^2, s_1^2, s_2^2)$

$$\begin{aligned} x_i(s) &= \prod_{j=1}^{k} s_j^{c_{i,j}} \\ c_{i,j} &= 0, 1, ..., n \\ (n+1)^k &\text{ features} \end{aligned} \tag{9.12}$$

Fourier basis

The Fourier series(linear combination) can be a basis for the function approximation. It works good with SARSA on smooth functions, it has some problems with discontinuations.

$$\begin{aligned} &x_i(s) = cos(\pi s^T c^i), s \in [0, 1] \\ &c^i = (c_1^i, ...c_k^i) \\ &c_j^i \in 0, ..., n \\ &(n+1)^k \text{ features} \end{aligned} \tag{9.13}$$

$$\alpha_i = \frac{\alpha}{\sqrt{(c_1^i)^2 + ... + (c_k^i)^2}} \tag{9.14}$$

Coarse Coding

The state space is covered by overlapping shapes(typically circles). A state is a point on in the state space. All the shapes that have this point inside their borders become 1, the other zero. The shape/width of the shapes determines the generalization. Small shapes give very little generalization, while big ones will give a lot of generalization.

Tile coding

Similar to coarse coding, but the shapes (such as rectangle) do not overlap. Only points in the same tile are generalized with each other. In order to get the same kind of generalization as in coarse coding, several layers of tiles can be made. This results in faster then coarse coding, but equally good results. The step size is more intuitive $\alpha = \frac{1}{n}$ would move 1 tile per iteration. Or if there are multiple layers $\alpha = \frac{w}{n}$, with w the offset between the layers.

Radial basis functions

The Radian basis functions (RBF) are the natural generalization of coarse coding to continuous valued features. It can be anything in $[0, 1]$, while the coarse coding is a Boolean.

- c_i: position of the RBF curve
- σ: width of the RBF curve

$$x_i(s) = \exp\left(\frac{||s - c_i||^2}{2\sigma_i^2}\right) \tag{9.15}$$

9.2.3 Selecting step-size parameters manually

The classical choice for step size is $\alpha = \frac{1}{t}$, this works great with MC methods, but not with TD methods.

In the tabular case, $\alpha = 1$ would eliminate the error after just one step. $\alpha = \frac{1}{\tau}$ eliminates the error after τ steps. This rule can be generalized to the linear approximation $\alpha = \mathbb{E}[x^T x]]$, with x a random feature vector. This does assume that the size of $x^T x$ is about constant.

9.2.4 Nonlinear function approximation: Artificial Neural network

The text in the book is rather vague, the only relevant take away is that TD-error's are used as well to update the value function if you use ANN.

9.2.5 Least-Squared TD

Earlier on in the chapter we used Least-Squares to do linear approximation. We ended up a fixed point $W_{TD} = A^{-1}b$ where $A = \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T]$ and $b = \mathbb{E}[R_{t+1}x_t]$. Instead of interating over the fixed point, W_{TD} can be found in 1 go. This is illustrated in equation 9.16, a more detailed algorithm can be found on page 230 of the book.

$$\begin{aligned} \hat{A}_t &= \sum_{k=0}^{t-1} x_k(x_k - \gamma x_{k+1})^T + \epsilon I \\ w_t &= \hat{A}_t^{-1} b_t \end{aligned} \tag{9.16}$$

If all the samples are used in the update, we don't get the forgetting behavior of old samples, as we had before. This can cause some issues if the target policy π is updated. There are some extra steps required to add forgetting in the algorithm. There is however no need to pick a step size, but you do have to pick the size of ϵ, so this is not really a win.

9.2.6 Memory-based function approximation

Instead of using the training examples to find the proper value of function parameters. The training data itself can be saved, and when the approximation is called upon with a certain state. It could return the answer of **the nearest neighbor**. Alternatively **the weighted average** of a set of neighbors could be returned, or a surface could be fit on the neighbors. The value can then be approximated using this surface, this is called **locally weighted regression**.

The 2 major advantages of this kind of approximation are:

1. No parameters.
2. The effect on the value function of adding a sample to the set is immediate.

Memory-based functions can however become slow in execution as the number of samples grows. This can be mitigated by using **k-d trees**, to speedup the look ups.

9.2.7 Kernel-based function approximation

Any linear parametric regression method that uses feature vectors can be recasted as kernel regression with $k(s', s) = x(s)^T x(s)$. (the kernel trick)

- $g(s)$: Target for the state s.
- D: Set of stored samples.

$$\hat{V}(s, D) = \sum_{s' \in D} k(s, s') g(s') \tag{9.17}$$

The text on page 232-233 is very vague on this method, might require some further investigation.

9.2.8 Looking deeper at on-policy learning: Interest and Emphasis

Not all states are as relevant to the RL problem. For example states that only occur after a series of very poor choices don't need to have an as accurately value function as those frequently visited by the greedy policy.

- I_t: interest, the degree to which we are interested in accurately valuing the state. An interest of 0 means there is no interest, and a value of 1 means it's of maximum interest
- M_t: emphasis, is multiplied with the learning rate
- $M_t = I_t + \gamma^n M_{t-n}$

$$w_{t+n} = w_{t+n-1} + \alpha M_t \big[G_{t:n+n} - \hat{v}(s_t, w_{t+n-1}) \big] \nabla \hat{v}(s_t, w_{t+n-1}) \tag{9.18}$$

9.3 Exercises

9.3.1 Exercise 9.1 page 209

Show that tabular methods such as presented in Part I of the book are a special case of linear function approximations. What would the feature vectors be?

The weights w would be the value that used to be in the table. The feature vectors x(s) would be a vector that is 1 on the state that is active so, if $S_t = 3$ then $x(3) = (0, 0, 0, 1, 0, 0)^T$.(one hot encoding)) This means that the value function is the weight on the index of that state $\hat{v}(s = 3, w) = w^T x(3) = w[3]$.

9.3.2 Exercise 9.2 page 211

Why does (9.17 in the book) define $(n+1)^k$ distinct features for dimension k? The highest degree of polynomial is n, so there are $n+1$ terms in a polygon that can be turned on or off. There are k dimensions, so to the power of k.

9.3.3 Exercise 9.3 page 211

What n and $c_{i,j}$ produce the feature vectors $x(s) = (1, s_1, s_2, s_1s_2, s_1^2, s_2^2, s_1^2s_2, s_1s_2^2, s_1^2, s_2^2)$

$1, s_1, s_2, s_1s_2, s_1^2, s_2^2, s_1^2s_2, s_1s_2^2, s_1^2, s_2^2$ has basis functions of a degree at most 2, and a dimension of 2 ($k = 2$). A max degree of 2 leads to $n = 2$.

- $c_{i,j}$: weight vector
- i: feature base index
- j: power of polynomial term

The basis functions exists at most out of 2 terms, one of each base index. The table below show the powers used on the terms to construct the base function.

i	basis function	$c_{i,0}$	$c_{i,1}$
0	1	0	1
1	s_1	1	0
2	s_2	0	1
3	s_1s_2	1	1
4	$s_1^2s_2$	2	1
5	$s_1s_2^2$	1	2
6	$s_1^2s_2^2$	2	2

9.3.4 Exercise 9.4 page 221

Suppose we believe that one of two state dimensions is more likely to have an effect on the value function than is the other, that generalization should be primarily across the dimension rather then along it. What kind of tilings could be used to take advantage of this prior knowledge

This is mentioned on page 220, a form of tiling that is more elongated along the less relevant dimension could work very well.

9.3.5 Exercise 9.5 page 223

- 7 dimensions
- 8 strip tiling's
- 7*8 = 56 dimension independent tiling's
- $\binom{7}{2}$=21 pairs of 2 dimensional
- total tiling's = 21*2 + 56 = 98

For any x, it will belong to exactly 8 strip tilings. And it will belong to 6 pairwise interactions so 12 tilings. To represent this you need at 20 non-zero entries equal to 1, therefore $\mathbb{E}[x^T x] = 20$. This leads to $\alpha = (\tau\,\mathbb{E}[x^T x])^{-1} = 20^{-1}$

9.3.6 Exercise 9.6 page 223

If $\tau = 1$ prove that (9.19) together with (9.7) results in the error being reduced to zero in one update

- equation 9.19 from book page 223: $\alpha = (\tau\,\mathbb{E}[x^T x])^{-1}$
- equation 9.7 from book page 202: $w_{t+1} = w_t + \alpha\big[U_t - \hat{v}(S_t, w_t)\big]\nabla\hat{v}(S_t, w_t)$

I am not sure how to continue here, it seems $\bar{x}$ must be 1, for this to work out.

$$
\begin{aligned}
w_{t+1} &= w_t + \alpha\big[U_t - \hat{v}(S_t, w_t)\big]\nabla\hat{v}(S_t, w_t) \\
&= w_t + \mathbb{E}[x^T x]^{-1} ex \\
&= w_t + (x^T x)^{-1} ex \\
&= w_t + e\bar{x} \\
e &= U_t - \hat{v}(S_t, w_t) \\
e &= w_{t+1} - w_t \\
\bar{x} &= \frac{x}{x^T x} \\
\nabla\hat{v}(S_t, w_t) &= x
\end{aligned}
\tag{9.19}
$$