RL Book Notes

bjb3az

August 2025

Chapter 3: Finite MDPs

Policies and Value Functions: Formal definitions:

1. Policy - mapping from states to probabilities / action distribution $\pi(a|s)$.
2. State-Value function - denoted $v_\pi(s)$, is the expected return (cumulative reward), with respect to π, starting at t and following the π after: $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$
3. Action-value function - the function which takes in a current state and action and denotes the expected return with respect to π: $q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a]$

Exercise 3.11: Expectation of R_{t+1} with respect to π is: $\mathbb{E}[R_{t+1}] = \sum_a \pi(a|s) r(s,a) = \sum_a \pi(a|s) \sum_r r \sum_{s'} p(s', r|, s, a)$
Exercise 3.12: $v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a)$
Exercise 3.13: $q_\pi(s,a) = \sum_{s'} p(s', r|s,a)[r + \gamma v_\pi(s')]$

Value functions, v_π, q_π can be estimated from experience - by following a policy and maintaining the average of the returns from a state, the average will converge to the true $v_\pi(s)$, similarly for the action-value. These are called *Monte-Carlo* methods. Instead of tracking all values, we can you use parameterized functions with less parameters than states. A fundamental property of value functions in RL and DP is that they satisfy recursive relationships, like the **Bellman equation**:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{1}$$
$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s,a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] \tag{2}$$
$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s,a)[r + \gamma v_\pi(s')] \tag{3}$$
$$= \sum_{s',a,r} \pi(a|s) p(s', r|s,a)[r + \gamma v_\pi(s')] \tag{4}$$
$$\tag{5}$$

Note that the final expression can be read as an expected value over the actions, states and rewards, the three defining sets in a MDP. For each possible triple s, a, r, we compute the probability of this triplet under the policy, given by $\pi(a|s)p(s', r|s,a) = p(s', r, a|s)$. We sum over these probabilities to get 1, and we give s by $v_\pi(s)$. We interpret the Bellman equation: starting from a state s, we can take some $a \in \mathcal{A}(s)$, which will give a corresponding r, s' depending on p. We then take a weighted average over all these possibilities weighted by the reward and discounted value of the next state. It is important to make the distinction - value functions are the expectations of return with respect to the policy.

The value function is the unique solution to the Bellman equation, i.e there is only one $v_\pi(s)$.

Exercise 3.14: $((2.3 + 0.4 - 0.4 + 0.7) * \gamma) * 0.25 = 0.675$

Exercise 3.15: Adding a constant c to all rewards means that the returns are now: $G'_t = (R_{t+1} + c) + \gamma(R_{t+2} + c) + .. = G_t + \sum_{k=0}^{\infty} \gamma^k c = G_t + \frac{c}{1-\gamma}, v_c = \frac{c}{1-\gamma}$

Exercise 3.16: Yes, it would have an effect since now there is no discounting - this means that this could cause unexpected infinite behaviors, where a model could find a loop that has infinite reward when it's expected to be 0.

Exercise 3.17: Bellman equations for action-value functions:

$$q_\pi(s,a) = \mathbb{E}_\pi[R_t + \gamma G_{t+1}|S_t = s, A_t = a] = \sum_a \pi(a|s) \sum_{s,r'} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{6}$$

$$= \sum_{s',r} p(s',r|s,a)[r + \gamma[\sum_{a'} q_\pi(s',a')\pi(a'|s')]] \tag{7}$$

Interpretation of this equation - given a starting (s,a) pair, we can look at the corresponding reward and next state s' that come from taking this pair. In order to define a recursive relation, we need to take a next action a' per s', that is averaged over the policy to give the state-action setup.

Exercise 3.18: $v_\pi(s) = \mathbb{E}_\pi[q_\pi(s,a)|S_t = s] = \sum_a q_\pi(s,a)\pi(a|s)$

Exercise 3.19: $q_\pi(s,a) = \mathbb{E}_p[R_{t+1} + v_\pi(S_{t+1})|S_t = s, A_t = a]$. Second equation with explicit sum: $q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$.

Optimal Policies and Value Functions

: For Finite MDPs, we can utilize the value functions to define a partial ordering over the policies. A policy π is better than or equal to π' if its expected return is greater than π' for all states: $\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \forall s \in \mathcal{S}$. There is always at least one policy that is better than OR equal to all other policies - this is the optimal policy, π_*. There may be more than one, but they all share the same optimal state-value function $v_*(s) = \max_\pi v_\pi(s)$. We can also find the optimal action-value function in terms of $v_*(s)$: $q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]$. An important note is that the optimal $q_*(s,a)$ is the expected return from taking just a selected a from s - after that it follows the optimal policy.

Bellman optimality equation: We know that v_*, by virtue of being a value function, must satisfy the Bellman update equation, but there is a special optimal form - the intuition comes from the fact that the value of a state under the optimal policy must equal the expected return from the best action from that state:

$$v_*(s) = \max_{a \in A(s)} q_{\pi_*}(s,a) \tag{8}$$

$$= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \tag{9}$$

$$= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{10}$$

$$= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \tag{11}$$

Here, q_{π_*} represents the action-value function under the optimal policy π_*. The last two lines are forms of the Bellman optimality equation for v_*. To summarize, we choose the action that maximizes our expected return. Taking any given action a will present a distribution over the next states s' and their corresponding rewards r by $p(s',r|s,a)$ - use this to get the expected return. Notice that now we don't use the probability $\pi(a|s)$ because we're taking the maximum over action-value functions now.

For the $q_*(s,a)$, we are now already in a s,a pair:

$$q_*(s,a) = \mathbb{E}_\pi[G_t|S_t = a, A_t = a] \tag{12}$$

$$= \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \tag{13}$$

$$= \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a' \in A(s')} q_*(s',a')] \tag{14}$$

For finite MDPs, the Bellman optimality equations, similar to the update equation have unique solution. It is a system of equations, one for each state, and if the environment dynamics are known, there are solvers for these systems. Once we have v_*, we can find the optimal policy - for each state s, there will be one or more actions that equal $v_*(s)$, and a policy assigning nonzero probability to only those actions is optimal. This is a one-step search, but because $v_*(s)$ does an expected **return**, is actually long-term optimal. Having $q_*(s,a)$ is even easier - at the extra cost of enumerating over both (s,a) pairs instead of just states, given any state s it can just find the action that maximizes $q_*(s,a)$ - these are examples of value-guided policy search, using the cached results of one-step ahead searches. We now don't even need to worry about looking at $\gamma v_*(s')$, we can just use s,a again.

Solving the Bellman optimality equations gives one path to finding the optimal policy, but this is challenging because there are three requirements which are rarely all true: 1) The environment dynamics are known, 2) There are computational resources sufficient to complete the calculation and the system solving, 3) The environment follows the Markov property. Many RL approaches are just approximately solving the Bellman optimality equation - one approach is to perform a more shallow search, where we approximate $v_*(s)$ at the leaf nodes and perform a backup.

Exercise 3.20: The optimal state-value function for golf: If you are outside of the green, then it copies the action-value function of using the driver outside of the green, then just use the state-value function for putting inside the green.

Exercise 3.21: Kind of a long description: but after the first putt, you should immediately use a driver. So it is basically the contours of the driver, but shifted slightly and a few more lines.

Exercise 3.22: $\gamma = 0$ the left policy is optimal. $\gamma = 0.9$:

$$v_\pi(s_0) = a + b * \gamma + a\gamma^2 + b\gamma^3 + .. = a\sum_{k=1}^{\infty}\gamma^{2k} + b\sum_{k=1}^{\infty}\gamma^{2k+1} \tag{15}$$

$$= \frac{a}{1-\gamma^2} + \frac{b\gamma}{1-\gamma^2} = ad + b\gamma d \tag{16}$$

$$\gamma = 0.5 : d = 4/3, \pi_{left} : a = 1, b = 0, v_\pi(s) = 4/3, \pi_{right} : a = 0, b = 2 : v_\pi(s) = 4/3 \tag{17}$$

$$\gamma = 0.9 : d = 5.263. v_{left}(s) = 5.263, v_{right}(s) = 2 * 0.9 * 5.263 \tag{18}$$

So for the 0.5 case, either works. For the 0.9 choose right. For the 0 choose left.

Exercise 3.24: Since all four directions are present, we know all corresponding actions are optimal in the 24.4 square, and using the definition of the return, let's represent it symbolically:

$$G_t = 10 + \gamma 0 + \gamma^2 0 + \gamma^3 0 + \gamma^4 0 + \gamma^5 10 + .. = \sum_{k=0}^{\infty} 10\gamma^{5k} = \frac{10}{1-\gamma^5} = \frac{10}{1-0.9^5} = 24.419 \tag{19}$$

Since all the actions are optimal, we know they all give the same value of 24.4

Exercise 3.25: $v_* = \max_a q_*(s, a)$

Exercise 3.26:

$$q_*(s,a) = \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \tag{20}$$

$$= \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{21}$$

Exercise 3.27: $\pi_*(a|s) = \begin{cases} 1 & a = argmax_a q_*(s|a) \\ 0 & else \end{cases}$

Exercise 3.28: Given a state, and the corresponding optimal state-value function $v_*(s)$, we need to take an argmax over the actions, which give the corresponding (s,r) next pairs:

$$\pi_*(a|s) = \begin{cases} 1 & a = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \\ 0 & else \end{cases} \tag{22}$$

Exercise 3.29: Definitions:

$$p(s'|s,a) = \sum_{r \in R} p(s', r|s, a), r(s, a) = \sum_{s',r} r * p(s', r|s, a) \tag{23}$$

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + S_{t+1}|S_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{24}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) * r + \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\gamma v_\pi(s') \tag{25}$$

$$= \sum_a \pi(a|s) r(s, a) + \gamma \sum_a \pi(a|s) \sum_{s'} p(s'|s, a) v_\pi(s') \tag{26}$$

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')] \tag{27}$$

$$= \sum_{s',r} p(s', r|s, a) r + \gamma \sum_{s',r,a'} \pi(a'|s') p(s', r|s, a) q_\pi(s', a') \tag{28}$$

$$= r(s, a) + \gamma \sum_{s',a'} \pi(a'|s') p(s'|s, a) q_\pi(s', a') \tag{29}$$

Now for the optimal:

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{30}$$

$$= r(s, a) + \gamma \sum_{s',r} p(s', r|s, a) \max_{a'} q_*(s', a') \tag{31}$$

$$= r(s, a) + \gamma \sum_{s'} p(s'|s, a) \max_{a'} q_*(s', a') \tag{32}$$

$$v_*(s) = \max_a q_*(s, a) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{33}$$

$$= \max_a \{r(s, a) + \gamma \sum_{s',r} p(s', r|s, a) v_*(s')\} \tag{34}$$

$$= \max_a \{r(s, a) + \gamma \sum_{s'} p(s'|s, a) v_*(s')\} \tag{35}$$

By reformulating in terms of state-transition probabilities and expected state-action rewards, the Bellman optimality equations become much more compact and interpretable. The action-value is just the expected state-action reward + the discounted sum of the maximum next-state action-value, weighted by transition probabilities.

Lastly, we have already seen that the computational and memory constraints on MDPs prevent exact solving of finite MDP problems for most cases. Thus, we require more compact, parameterized approximations of the optimal state/action-value functions, but there are *tabular* cases where we can enumerate all the possibilities as well. We can also get away with making bad decisions in infrequent states in the online nature of RL, which allows us to settle for approximations, as long as our model is performing well in the frequent states.

To summarize, the seminal equation of this section are the Bellman optimality equations, which can be solved for the optimal value functions. From there, we can use a greedy rule to find the optimal policy, since any policy that is greedy with respect to the optimal value functions is an optimal policy. The optimal value function is also unique, but there are *classes* of optimal policies that satisfy the equations.

Chapter 4: Dynamic Programming

These are a collection of algos used to compute optimal policies given a perfect model of the environment as a MDP. In the case that our environment is not a finite MDP, or we have continuous action and state spaces, we can quantize the continuous spaces and then apply finite-state DP methods. The key idea behind DP is to use the value functions to organize and inform the search for good policies. This is done by turning Bellman equations into assignments / update rules for improving value function approximations.

Policy Evaluation / Prediction: This strategy is also referred to the prediction problem, where we are trying to compute $v_\pi(s)$ for policy π. When the environment's dynamics are completely known, it is simple to use an iterative method. Using the Bellman equation, we can create a sequence of approximate value functions, where v_0 can be arbitrarily chosen:

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \tag{36}$$
$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{37}$$

Because v_π is a unique solution to the update equation, $v_k = v_\pi$ is a fixed point to the update rule, and the sequence is shown to converge to v_π under the same regularity conditions that gaurantee the uniqueness. This is *iterative policy evaluation.* The type of equation update is called an expected update, since it is done based on an expectation over all possible next states rather than a sample.

There are two possible computer implementations of state-based DP - one where you use two arrays and completely update the new array, or the one-array, in-place approach: the convergence rates of policy iteration depend on this order updating in order to keep the data constantly fresh.

Exercise 4.1: $q_\pi(11, down) = -1, q_\pi(7, down) = -2$

Exercise 4.2: $v_\pi(15) = 1/4(-1 + 0) * 4 = -1$. $v_\pi($

Exercise 4.3:

$$4.3 : \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \tag{38}$$
$$4.4 : q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')] \tag{39}$$
$$4.5 : q_{k+1}(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_k(s', a')] \tag{40}$$

Policy Improvement: Another axes of improving policies is by comparing the $q_\pi(s, a)$ and $v_\pi(s)$ of different policies - we want to see if we should change our deterministic action to something different from the current policy. The *policy improvement theorem* says that for π, π', any pair of deterministic policies such that $\forall s \in S$:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{41}$$

So for each state s, we take the π' action then follow π afterwards. Then the policy π' is at least as good as π: $v_{\pi'}(s) \geq v_\pi(s)$ when the above equation is true. By writing out the Bellman update, we can see this concretely:

$$v_\pi(s) \leq q_\pi(s, \pi'(s)) \tag{42}$$
$$= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | A_t = \pi'(s), S_t = s] \tag{43}$$
$$\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \tag{44}$$
$$= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_\pi[R_{t+2} + \gamma v_\pi(S_{t+2}) | A_{t+1} = \pi'(S_{t+1}), S_{t+1}] | S_t = s] \tag{45}$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \tag{46}$$
$$... \leq v_{\pi'}(s) \tag{47}$$

What we're doing is just constantly reapplying the policy improvement inequality and the action-value in terms of the state-value. We can keep removing the expectations with respect to π inside, since using the inequality allows us to take the better actions with π' for every state. So this allows to evaluate a change in the policy at a given state.

If we now consider changes in *all* states, so our new policy is $\pi'(s) = argmax_a q_\pi(s, a)$. This greedy policy chooses the action which maximizes the value function after one-step lookahead. Since we just showed through the policy improvement theorem that choosing the policy that maximizes q and tries to outperform v always gives at least better expected returns, *policy improvement* is the process of finding new policies that are greedy with respect to the value function or the original policy. A nice free lunch of this method is that when $v_\pi = v_{\pi'}$ it follows that the relation between the two policies is actually the Bellman optimality equation.

Policy Iteration: We can now combine the two - policy evaluation and improvement, to create a sequence of monotonically improving value and policy functions. We first evaluate a policy π_0, using the Bellman update to get a value function v_{π_0}, which we then use for policy improvement by maximizing $q(s, a)$, which gives us a new policy. Another bonus is that policy evaluation, which is itself iterative, can use the previous policy function as a starting point, increasing convergence further.

Exercise 4.4: I think all you need to do is just keep track of the expected return of both policies as well? Then stop when either policy is stable or the expected returns are equal.

Exercise 4.5: Changes, in order of the pseudocode:

1. Initialize $Q(s,a) \in \mathbb{R}, Q(terminal, a) = 0 \forall a \in \mathcal{A}$
2. Change policy evaluation so that it now uses the Bellman update function for action-values:
$$Q(s,a) = \sum_{s',r} p(s',r|s,a)[r + \sum_{a'} \pi(a'|s')q(s',a')] \tag{48}$$
3. Policy Improvement: Instead of writing out the long form that is required since state-value functions were used in the book, we just replace with $\pi(s) = argmax_a q(s,a)$

Exercise 4.6: This means that each action has at least some probability given by $\epsilon/|A(s)|$, so the policy is not deterministic. For step 3: when we design a new policy, ensure all other actions besides the maximal one have the epsilon probability, while the maximal action has probability: $1 - (|A| - 1) * \frac{\epsilon}{|A|}$. In step 2, we should use probabilities over all actions now, and in step 1 we initialize with a equiprobable policy.

Value Iteration: Policy iteration is powerful but requires running nested iterative loops, with policy evaluation also being an iterative procedure. *Value iteration* is when we always stop the Bellman update in policy evaluation after the first round of updates. If we now combine both equations, we get the new update operation:

$$v_{k+1}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s, A_t = a] \tag{49}$$

So the expression inside the max is one-update, while the max is taken for the policy improvement steps. This update equation can also be viewed as the Bellman optimality equation turned into an update equation as well. Now, instead of performing sweeps over both the policy and value functions, we perform value iteration until convergence, and then our policy is deterministic and acts greedily with respect to the value function.

In general, value iteration is interpolating multiple policy evaluation sweeps between each policy improvement sweep, and has faster convergence. The only difference between iterative value iteration and iterative policy evaluation is the max operator inside value iteration, which is added to some sweeps of policy evaluation. All three of these algorithms converge to optimal policies for **discounted finite MDPs**.

Exercise 4.8: It takes this curious form because of the nature of the problem - when $p_h = 0.4$, the coin does not favor the gambler, so the gambler should try to reach 100 in the least amount of flips possible. This is why it takes such a large bet at the 50 mark, but at 51 it takes a bet size of 1, because it is ok with losing getting back down to 50. This is why at around 62.5, it is ok with staking the distance to get to 75, because if it loses it will get down to 50, otherwise it gets to 75.

Exercise 4.10:

$$q_{k+1}(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_k(s',a')] \tag{50}$$

Asychronous DP: This verison addresses a major drawback of most DRP methods - they involve operations over the entire set of the state space, which can be prohibitively expensive. Async DP is indiscriminate about the order in which it updates states - as long as an async algo continues to update the values of all states, it will continue to converge. This new version is not meant to decrease computation - instead it doesn't force us into iterative rounds of updates where we need to perform sweeps. We can be selective about what states we choose to update (perhaps ones that occur more frequently) in order to improve convergence. This new format also allows to try online methods of solving, where we run iterative DP algorithms, that process the current state an agent is experiencing, so it naturally updates on states that occur more frequently.

We could also achieve even more fine-grained levels of intermixing value iteration and policy evaluation updates

Generalized Policy Iteration: The core idea of policy iteration is the two simultaneous, interacting processes of policy evaluation and policy improvement. As long as both continue to see all the states and update them, we can gaurantee convergence. GPI is the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and mixing. Most RL algorithms can be described with this framework, where convergence is known when both the policy and the value functions stabilize.

GPI is described as the process where the value function aligns with the policy's value function, and the policy continues to improve with respect to the current value function. When both stabilize, that means that policy is greedy with respect to the value function that is aligned with it. GPI doesn't try to achieve both v_*, π_* as an optimization goal at any given step, instead by separately optimizing for each, it relies on the tight interaction between the two to reach the optimized goal.

One last special property of DP methods is that they are **boostrapping** methods, since they formulate new estimates based on estimates of successor states.

5: Monte Carlo Methods

In this chapter we consider learning methods that do not assume complete knowledge of the environment, and only require experience, through sample sequences of state actions and rewards from interaction. A model may be required, but only to generate S, A, R pairs, and not the associated transition probabilities. MC methods are methods of solving RL problems by averaging sample returns. They are usually formulated in the episodic setting, where we perform updates on value and policy functions after episodes are completed.

They are very similar to the bandit methods, which sampled and averaged returns in order to formulate estimates of which actions were best - the main difference is now the presence of multiple states (associateive-search/contextual bandits), and the fact that states depend on each other across timesteps. This is also an extremely nonstationary problem, since action and state updates are occurring at the same time as experience, so many extensions of DP-ideas are used here.

5.1: Monte Carlo Prediction: There are two ways of estimating / learning the state-value function by averaging returns, called *first-visit* and *every-visit*, which are literally just their names. First-visit is more common, but it just prescribes to use the first visit of a state's return as the one to average on. The algorithm is then for each generated episode, we loop from the last state, collecting return through $G = R_{t+1} + \gamma G$, and check if this state was visited in any of the previous ones, if it hasn't, add to tracker and average.

Exercise 5.1: It jumps up for the last two rows because when the user has 20,21 and chooses to always stick, regardless of what the dealer has 20,21 are the highest possible numbers, so the user always has a good chance of winning or drawing. It drops off when the Dealer has an Ace because the dealer has more 'opportunities' to get a 20/21, since the Ace is the highest number, or the lowest. The top ones have better frontmost values because when the ace is usable, the player essentially has "two chances" to beat the dealer, one with the usable ace and then another when it becomes unusable.

Exercise 5.2: If every-visit MC was used instead of first-visit MC, the results wouldn't be very different - this is because the states can only increase, so once a state is seen you can't reverse back to it.

An important distinction between MC and DP is that MC basically does depth and DP does breadth - furthermore the Monte Carlo methods do not bootstrap, since they simulate an entire trajectory from the environment. MC methods are also advantageous in environments like blackjack, where trying to precisely determine transition probabilities and +1 reward probabilities from a dealer's hand are much more intricate. MC methods also don't require sweeps or coverage over all the states - if we are only interested in a particular subset of states, we just need to generate episodes starting from those states and collect averages.

MC Action-Value Estimation: Without a model, state values are insufficient to determining the optimal policy, since if we have $v_*(s)$, we still would need to perform lookaheads using $p(s', r|s, a)$ and the combination of reward and next state. MC methods do the same episode rollouts, but will now check when the visits to a state-action pair in order to estimate the return, with the same first/every-visit method. This is shown to converge quadratically to the true expected values, which means the distance to the true value decreases by around a square factor for each iteration of action-value MC estimation. The only issue is that many (s, a) pairs aren't visited, due to the rollout nature. One way to handle this is to sample a state-action pair to start at for each episode, and ensure every combination has a nonzero probability of being picked, this is called *exploring starts.*

Exercise 5.3: The backup diagram for the Monte Carlo estimation of q_π is a similar to the v_π one - we only go along the state / action paths that the policy sees in a given episode.

Monte Carlo Control:

Monte Carlo estimation can also be used in control, i.e approximating optimal policies. A MC version of policy iteration, where we alternate between stages of policy evaluation and improvement, would look like this:

In the policy evaluation stage, we proceed as before, except the update equation is now substituted by averages of rollout estimates. If we assume infinite episodes and exploring starts, MC estimates will compute q_{π_k} exactly. The policy improvement stage is similar too - by computing q we just need to find the policy that aspects greedy with respect to q, and the policy improvement theorem ensures that each π_{k+1} is uniformly better than π_k. In order to get around the infinite state convergence requirement, we can use the same ideas from GPI, where we don't need to run policy evaluation for infinite episodes, only until some approximation bound is gauranteed.

For Monte Carlo Policy Iteration, we can alternate between evaluation and improvement per episode. Then after each episode, we use the observed returns for policy evaluation, and then we perform the policy improvement maximization using the new value function on all the states visited in the current episode.

Exercise 5.4: If we just maintain two variables, the mean M and count C, then the update equation would look: $M_{t+1} = (M_t C_t + R)/(C_t + 1), C_{t+1} = C_t + 1$

Without Exploring Starts: To ensure that all actions are selected infinitely often for convergence, we can either use on/off-policy methods. The MC-ES method is an example of an on-policy method, since it directly acts on rollouts. In the on-policy method, we are going to take advantage of the ϵ-greedy policy, which is greedy but with probability ϵ will take an action at random. On-policy MC control is still based off GPI - policy evaluation is still the same. However, during policy improvement we can't make the policy greedy with respect to the current value function because that leads it to be less exploratory. GPI only requires that we move towards a greedy policy every iteration, so we will just move towards an ϵ-greedy policy per step.

By the policy improvement theorem, an ϵ-greedy policy is always better than or equal to any other ϵ-soft policy:

$$WTS : q_\pi(s, \pi'(s)) \geq v_\pi(s) \forall s \tag{51}$$

$$q_\pi(s, \pi'(s)) = \sum_a \pi'(a|s) q_\pi(s, a) \tag{52}$$

$$= \frac{\epsilon}{|A(s)|} \sum_a q(s, a) + (1 - \epsilon) \max_a q_\pi(s, a) \tag{53}$$

$$\geq \frac{\epsilon}{|A(s)|} \sum_a q(s, a) + (1 - \epsilon) \sum_a \frac{\pi(a|s) - \frac{\epsilon}{|A(s)|}}{1 - \epsilon} q(s, a) \tag{54}$$

$$= \sum_a \pi(a|s) q(s, a) = v_\pi(s) \tag{55}$$

The inequality between the max and the sum operator is valid because the sum is a weighted sum where the weights add up to 1, so it must be less than the maximum value. Thus the policy improvement theorem holds - furthermore, we know that we are optimal. The book also proves that equality only holds when the policy is optimal - this is done by having two environments and comparing them. We have a general environment and then a ϵ-soft environment, where with prob $1 - \epsilon$ it acts normally, but otherwise resamples an action with equal probability. We are essentially moving the ϵ-soft behavior into the environment, and then we show that the Bellman optimality equations for an optimal ϵ-soft policy and a policy in this ϵ-soft environment are equivalent.

By proving this, we are trading away exploring starts (which may be prohibitively expensive) and swapping in ϵ-soft policies, which still give us the same optimality guarantees.

Off-policy Prediction with Importance Sampling: The previous approach is a compromise between exploration-exploitation - it learns the optimal action-values for a near optimal policy that explores. Instead, we can try using two policies - one that generates data and is exploratory, while the other learns, called the *target policy.* We can begin with the easiest case where both the target and behavior policies are fixed and given. Target = π, behavior = b. In order to use episodes from b to estimate π, there is a *coverage* condition: $\pi(a|s) > 0 \implies b(a|s) > 0$. From coverage, we know that b is stochastic in states where $b(s) \neq \pi(s)$, but π can still be a deterministic policy, which is nice for control.

All importance sampling methods utilize *importance sampling*, so returns are weighted by the *importance-sampling ratio*, which is given by:

$$p_{1:T-1} = \prod_{k=1}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{56}$$

The MDP transition probabilities appear in both parts of the fraction and are canceled out, and this equation is independent of whether the MDP is known. Given a return G_t from the behavior policy, we know that $v_\pi(s) = \mathbb{E}[p_{1,T-1}G_t|S_t = s]$. To now estimate $v_\pi(s)$, we can either use *ordinary or weighted* importance sampling:

$$V(s) = \frac{\sum_t p_{t:T-1}G_t}{|T(s)|}, V(s) = \frac{\sum_t p_{t:T-1}G_t}{\sum_t p_{t:T-1}} \tag{57}$$

The difference between these two methods is that ordinary IS is unbiased but has high variance, since the importance ratios are unbounded, while weighted IS has lower variance, since the highest largest weight is 1 for any given return, but is biased towards $v_b(s)$. The variance of the weighted IS estimator converges to 0, and has very low variance in practice and is preferred. So the idea is that we use the behavior policy to generate a bunch of episodes, and then use some importance-sampling to get it back into the target policy's area. Another downside of ordinary importance sampling is that the convergence is unsatisfactory if the estimates have infinite variance, which commonly occurs when the scaled returns have infinite variance, common in off-policy learning when trajectories can loop for an infinite number of steps.

Exercise 5.5: The episode is terminal after 10 steps, which means that we stayed in the nonterminal state, so the return is 10. For every visit estimators the average is $(1 + ...10)/10 = 5.5$.

Exercise 5.6:

$$G_t = \mathbb{E}_\pi[R_t + \gamma G_{t+1}] = \mathbb{E}_\pi[R_t + \gamma v_\pi(S)] \tag{58}$$

$$Q(s,a) = \frac{\sum_{t \in \mathcal{T}(s)} p_{t+1:T(t)-1}G_{t+1}}{\sum_{t \in \mathcal{T}(s)} p_{t:T(t)-1}} \tag{59}$$

Since we already took an action, we are curious about the return starting from the next state after the current one.

Exercise 5.7: Error first increased because the WIS method is naturally biased, so in the first few episodes it was still biasing towards the behavior policy, which was going away from the target policy.

Exercise 5.8: Yes, because the sum shown across episodes of $\mathbb{E}[X^2]$ is essentially the same - we're still adding across a bunch of lengths in the every-visit method.

Incremental Implementation: We can use the ideas from Chapter 2 again here, since MC methods just keep track of running averages of returns, similar to the bandit problem. For ordinary importance sampling, nothing changes from Chapter 2, since we are just performing a strict average over a set size. For WIS, we're performing a weighted average of the returns and the denominator also changes, so we need a different algorithm. If we are given a sequence of returns $G_1, ..G_{n-1}$, and each with a random weight $W_i = p_{t_i:T(t_i)-1}$, and a new return G_n comes in, then

$$V_n = \frac{\sum_{i=1}^{n-1} W_i G_i}{\sum_i W_i}, C_n = \sum_i^n W_i \tag{60}$$

$$V_{n+1} = \frac{\sum_i^n W_i G_i}{\sum_i^n W_i} = \frac{V_n C_{n-1} + W_n G_n}{C_n} = V_n \frac{C_{n-1}}{C_n} + \frac{W_n G_n}{C_n} \tag{61}$$

$$= V_n \frac{C_n - W_n}{C_n} + \frac{W_n G_n}{C_n} = V_n + \frac{W_n}{C_n}[G_n - V_n] \tag{62}$$

Notice this is similar to the incremental update equation in chapter 2, where we're maintaining an average and then adjusting it based on the distance for the new value, but now we weight by W_n/C_n, because the update needs to scale by how likely it is compared to the cumulative sum.

Exercise 5.9: The only part that needs modifying is the average, which we can rewrite:

$$V_n = \frac{\sum_i^{n-1} W_i G_i}{n-1} \tag{63}$$

$$V_{n+1} = \frac{\sum_i^n W_i G_i}{n} = \frac{V_n(n-1) + W_i G_i}{n} \tag{64}$$

$$= V_n + \frac{1}{n}[W_i G_i - V_n] \tag{65}$$

Exercise 5.10: Did above.

Off-policy Monte Carlo Control

: In the previous subchapters we looked at off-policy MC evaluation, which described how to estimate $Q(s,a), V(s)$ based on rollouts. As long as we maintain an ϵ-soft policy for b, we know that this will cover an infinite number of states. An example of Off-policy MC control with WIS and a deterministic greedy policy:

1. Initialize arbitrary $Q(s,a), C(s,a) = 0, \pi(s)$ is greedy with respect to $Q(s,a)$.
2. Start collecting rollouts per episode - initialize $G = 0, W = 1$. Then for each step of the episode, collect return and perform weighted incremental update on $Q(S_t, A_t)$. Update the policy so that $\pi_t = \arg\max_a Q(S_t, a)$
3. To ensure coverage, check if $A_t = \pi(S_t)$. If this is not true, go to the next episode, since we want to only collect estimates on rollouts that align with the greedy nature of the policy.

Exercise 5.11: This is because the policy is deterministic, greedy with respect to the action-value function.

Exercise 5.12:

Discounting-aware Importance Sampling: This introduces a natural extension of WIS - to allow importance sampling ratios to be discounting aware, to mitigate variance explosion. This is intuitive since returns are already naturally discounted - we can this in an example, where we have episodes of 100 steps with $\gamma = 0$. Then $G_0 = R_1$, but this gets multiplied by a product of 100 factors. The return is then scaled by a 100 factors when it is only necessary to scale by the first one, since after the first step the return is already determined. The later factors are independent of the return but have EV = 1, which causes the variance to explode.

To induce discounting-aware importance sampling, we reinterpret γ as a probability of termination, or the degree of partial termination. At any given timestep, k, we can expect the probability of terminating to be $(1-\gamma)\gamma^{k-1}$. The $(1-\gamma)$ means probability of partly terminating. We can define flat partial returns, which summed up will equal the conventional full return:

$$\bar{G}_{t:h} = R_{t+1} + ..R_h \tag{66}$$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + ... \tag{67}$$

$$= (1-\gamma)R_{t+1} + \gamma(1-\gamma)R_{t+2} + .. \tag{68}$$

Chapter 5: Summary

Monte Carlo methods are different from the DP methods we saw in Chapter 4 - they operate purely based on experience and episode rollouts and don't need a model of the environment. We can instead take rollouts from every single state / state-action pair in order to perform policy evaluation, and then perform some type of greedy matching to get the policy. Finding $Q(s,a)$ is convenient because then we don't need to even worry about transition probabilities. There are still some problems - MC deals with the exploration / exploitation dilemma, since the policy will naturally follow optimal paths, but it still needs to have some room to explore in order to gaurantee convergence to the optimal policy.

On-policy and off-policy are now two different forms of control methods. On-policy is when the target and behavior policies are equal to each other, so that the target policy learns from the experience it generates. Off-policy is when the two policies are different, and the target policy learns its value functions and policies from a behavior policy that is different from itself - this requires some corrections like importance sampling. Ordinary importance sampling is higher variance, and can even lead to infinite variance, while weighted importance sampling is a special version where the denominator is also a weighted sum - it is biased but has much lower variance in general and nice convergence properties.

Chapter 6: Temporal-Difference Learning

TD Learning takes the 'best of both worlds' from both MC and DP - it uses the MC idea of learning from experience, so it doesn't require a transition model, and uses the idea of bootstrapping from DP - thus we don't need to collect rewards from the terminal state.

TD Prediction: Like always, we start with the problem of correctly evaluating a value function given a policy π. A simple MC every-visit method for nonstationary environments is updating a moving average: $V(S_t) = V(S_t) + \alpha[G_t - V(S_t)]$. However in TD-learning we can now bootstrap: $V(S_t) = V(S_t)+\alpha[R_{t+1}+\gamma V(S_{t+1})-V(S_t)]$. This is also known as TD(0), or one-step TD. MC and TD methods are called *sample updates* since they involve looking ahead to sample successor states (or action pairs), using the values of the sucessor states and rewards to compute a backed-up value, and then updating the value of the original state.

A value that comes up a lot in RL is the *TD Error*: $R_{t+1}+\gamma V(S_{t+1})-V(S_t)$, and it is not available until time step $t+1$. If the array V of estimates does not change during the episode, then the MC error $G_t - V(S_t)$ can be written as a sum of TD errors, as seen using this recursive relation:

$$G_t - V(S_t) = R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \tag{69}$$
$$= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \tag{70}$$
$$= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V(S_{t+2})... = \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k \tag{71}$$

We use the camel trick, and then we identify the recursive relation - the last MC error at the terminal state cancels out.

Exercise 6.1: If the value array changes per timestep now, at V_t time step t when we update, then at time T the MC error is still given by:

$$G_t - V_t(S_t) = R_{t+1} + \gamma G_{t+1} - V_t(S_t) + \gamma V_t(S_{t+1}) - \gamma V_t(S_{t+1}) \tag{72}$$
$$= \delta_t + \gamma(G_{t+1} - V_t(S_{t+1}) - V_{t+1}(S_{t+1})) + \gamma V_{t+1}(S_{t+1}) \tag{73}$$
$$= \delta_t + \gamma(G_{t+1} - V_{t+1}(S_{t+1})) + \gamma V_{t+1}(S_{t+1}) - \gamma V_t(S_{t+1}) \tag{74}$$
$$= \delta_t + \gamma(\delta_{t+1} + \gamma(G_{t+2} - V_{t+2}(S_{t+2})) + \gamma V_{t+2}(S_{t+2}) - \gamma V_{t+1}(S_{t+2})) + \gamma V_{t+1}(S_{t+1}) - \gamma V_t(S_{t+1}) \tag{75}$$
$$= \sum_{k=t}^{T-1} \delta_k \gamma^{k-t} + \sum_{k=t}^{T-1} \gamma^{k-t} V_k(S_k) - \sum_{k=t}^{T-2} \gamma^{k-t} V_k(S_{k+1}) \tag{76}$$

Exercise 6.2: For the example provided in the exercise, TD methods would be better since we are still entering the highway at the same place - because we have already learned most of the highway segment, MC methods are wasteful since we need to wait until we go through all of the highway in order to update our beliefs. Instead, with a TD method, since we already have a good model of the returns starting from highway entry, we can learn predictions for the new building much more quickly. We can move a similar example to the current situation - say the highway we always use is closed for construction so we need to take another highway. MC methods would force us to get home first, so that we could use the actual outcome to update our predictions on the times for this new highway, while the TD method allows us to immediately update predictions once we get on the secondary road again. Another downside of MC is that rollouts will be noisy at the beginning.

Convergence properties of TD-methods: For any fixed policy π, TD(0) converges to v_π, in the mean for a constant step-size parameter (as long as it is sufficiently small), or if it follows a decreasing schedule under the Robbins-Monro conditions.

Exercise 6.3: If $V(A) = V(A) + \alpha[V(S') - V(A)]$, and we know only $V(A)$ decreased by a little, the first episode must have been C,B,A, so that $V(A)$ decreases by a factor of $\alpha * V(A)$, which looks like it is around $0.05/0.5 = 0.1$, which matches the step-size parameter given. The other states estimates don't change since their value functions are updated before $V(A)$ is updated.

Exercise 6.4: No - it looks like there was a sufficiently wide range of parameters used for MC, and it still underperformed compared to TD. MC naturally requires lower step-sizes because it is a higher variance method, but even then it couldn't perform as well as TD on a small step size of $\alpha = 0.5$. For TD(0), I think choosing a smaller, fixed value like $\alpha = 0.01$ could possibly make the performance better, while for MC it is hard to say, since the empirical RMS are noisy and overlapping across the different methods.

Exercise 6.5: Two reasons - firstly the α is too high, so this will cause errors in the updates not to converge to 0 but be more pronounced, causing the wobbly graph appearance. Second reason is because of the value initialization - when all the values are initialized to the same one, this can happen frequently: $V(D) = V(D) + \alpha[V(E) - V(D)]$. States with higher values than the initial will continue to push up, while states with lower values will continue to push down, leading to no convergence.

Exercises 6.6: One way is to perform iterations of dynamic programming using prediction - keep simulating episodes, and performing the Bellman update equation since we know the transition model probabilities.

Another way is to use Markov processes: If we say that $P_E(L)$ is the probability we end up in the left state from E, then

$$\begin{aligned} P_E(R) &= 1 - P_E(L) && (77)\\ &= 1 - P_E(D) * P_D(L) && (78)\\ &= 1 - P_E(D)[P_C(L) * P_D(C) + P_D(E) * P_E(L)] && (79)\\ &= 1 - 0.5[0.5 * 0.5 + 0.5 * P_E(L)] && (80)\\ \implies P_E(L) = 0.5[0.25 + 0.5 * P_E(L)] &\implies P_E(L) = 0.125 + 0.25 P_E(L) && (81)\\ &\implies 3/4 P_E(L) = 1/8, P_E(L) = 1/6, P_E(R) = 5/6 && (82) \end{aligned}$$

The second method was used since it is much simpler and faster than using dynamic programming.

Optimality of TD(0): When there is only a finite amount of experience, a common approach is to keep training on the same data until the method converges. Given an approximate value function V, we will keep computing the incremental changes given by the step-changes, whether it is TD(0) or constant α-MC are computed, but an update is only done once, by summing all the increments - this is **batch updating**. Under this method, TD(0) and MC actually converge deterministically to different values.

(After reading the examples), it seems like we can draw parallels from MC vs. TD to MLE vs. MAP in machine learning. MC evaluation methods directly draw conclusions about the values of states based on the training data, while TD(0) tries to construct a distribution over the training data, by generating a Markov process, and then computing state-values from that. The TD(0) method is better at predicting future data. In the book, they say that batch TD(0) converges to the *certainty-equivalence estimate*. This occurs when we assume the estimate of the value function would be exactly correct if the model were exactly correct, or we assume that the estimate of the underlying process was known with certainty, which is what TD(0) does by constructing a Markov process.

Exercise 6.7: For an off-policy version, we need to augment this update function:

$$V(S) = V(S) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{83}$$

To write a form of off-policy TD control, let's follow the structure of the Tabular TD(0) algorithm given. After appropriate initialization, we will first generate an episode with behavior policy b, and then loop for each step of the episode. For each state we will compute $\pi(a|s)$, and then keep a running $p_{1:t}$ importance sampling ratio. Let's say we are doing WIS, then the steps are:

$$\begin{aligned} G_t &= R_{t+1} + V(S_{t+1}) && (84)\\ C(S_t, A_t) &= C(S_t, A_t) + W && (85)\\ V(S_t) &= V(S_t) + \frac{W}{C(S_t, A_t)}\delta_t, \delta_t = R + \gamma V(S') - V(S) && (86)\\ W &= \frac{\pi(A_t|S_t)}{b(A_t|S_t)} W && (87) \end{aligned}$$

Recall that this was the incremental update equation derived in Exercise 5.10.

Sarsa: On-policy TD Control

The TD(0) update equations for state-value functions also have equivalent analogs to action-value functions, given by:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{88}$$

The method is called SARSA because at each step we use the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. The convergence properties of SARSA are pretty much the same, given that the step-size follows the Robbins-Monro conditions, and all state-action pairs are visited an infinite number of times. In order to ensure this, we should use ϵ-soft policies, so that the policy will converge to the ϵ-greedy policy. We could also anneal the epsilon effect by setting $\epsilon = 1/t$. In the implementation, the only addition from

TD(0) is that we now take action A from S, observe R, S', and then take action A' according to the policy derived from Q.

Exercise 6.8:

$$G_t - Q(S_t, A_t) = R_{t+1} + \gamma G_{t+1} - Q(S_t, A_t) + \gamma Q(S_{t+1}, A_{t+1}) - \gamma Q(S_{t+1}, A_{t+1}) \quad (89)$$

$$= \delta_t + \gamma(G_{t+1} - Q(S_{t+1}, A_{t+1})) \quad (90)$$

$$= \delta_t + \gamma(\delta_{t+1} + \gamma(G_{t+2} - Q(S_{t+2}, A_{t+2})) = \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - Q(S_{t+2}, A_{t+2})) \quad (91)$$

$$= \delta_t + \gamma\delta_{t+1} + ...\gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0 - Q(S_T, A_T)) \quad (92)$$

Q-learning : Off-policy TD Control

The off-policy version of TD control is where the learned action-value function Q directly approximates q_*, so now we use a max operation to fit the bellman equation update:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (93)$$

Now, the important difference is that we don't use the next action A' that we used in SARSA - hence this could also be called the SARS algorithm, since we are now treating $Q = q_*$ for the next state-value. Removing the dependence on the policy for the next state action simplifies the analysis and convergence proofs - under some minimal assumptions (all state coverage) and the usual Robbins Monro stohastic approximation conditions on the step-size parameters, Q converges to q_* with probability 1. The only difference in the Q-learning algo and SARSA is we don't take A' from S' using policy derived from Q, we just take the direct max, so no ϵ-greedy effect.

In the cliff example in the book, we see that Q-learning tends to take the 'riskier' path, since it doesn't have the additional knowledge of A' and instead takes maximizations of the value functions. This allows it to fall into the cliff sometimes due to the epsilon-soft nature of the policy, so it has worst online performance.

Exercise 6.11: Although the first action at a update step in Q-learning is taken with respect to the policy derived from Q, the actual Q-learning update equation does not depend on the policy. SARSA is on-policy because the value $Q(S', A')$ is used directly in the update equation and follows from the policy. The important distinction is this update: $A' \to A$, which forces the episode to follow the policy, while Q-learning always refreshes its action choice.

Exericse 6.12: They will mostly follow the same behavior, since $\max_a Q(S', a) = Q(S', A')$ since by definition now A' in SARSA is the argmax over the $Q(S', a)$. However there is one important distinction - on the next loop of a step of an episode, SARSA copies over the action using the 'stale' Q-function: $A' \to A$, while Q-learning will choose A from S using the freshly updated Q, which is probably why Q-learning also has nicer convergence properties, since it is using the most up-to-date Q-table.

Expected SARSA:

This is a slight modification to the Q-learning - instead of a maximization, we take an expectation over the next state-action pairs:

$$Q(S_t, A_t) += \alpha[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (94)$$

It moves deterministically in the same direction that SARSA moves in expectation, so it is called Expected SARSA. Although it requires more computation, it eliminates the variances due to the random selection of A_{t+1}, since we now enumerate over all A_{t+1}. Expected SARSA can be either on or off-policy - in general it is off-policy. If we suppose π is the greedy policy, while the behavior policy is exploratory, then Expected SARSA becomes Q-learning, since the policy expectation becomes a max again and the behavior policy becomes epsilon-greedy.

Maximization Bias and Double Learning

: Maximization in the construction of the target policies creates an issue - Q-learning does this when constructing a greedy policy relative to Q. This gives an issue because estimates of action-values are noisy at the beginning, and taking a maximum over the values will implicitly lead to significant positive biases. A good example is a MDP where transitions from a state B to a terminal state follow the reward distribution $\mathcal{N}(-0.1, 1)$. So the expected return from B is negative, but control methods might favor this path because of the occassional high variance, positive rewards.

A solution to this is to use Double Learning, since one way of explaining the maximization bias is that the same samples are being used to determine a maximizing action as well as learn a value estimate. If we instead divided our experiences into two sets and learned two independent value functions, $Q_1(a)$ and $Q_2(a)$, we could use one to determine the maximizing action, and one to determine the value: $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$. The vice versa holds as well. Double learning only doubles the memory requirements but not the computational requirements - only one estimate function is updated per step. The full double Q-learning update equation for Q_1 is given by:

$$Q_1(S_t, A_t) += \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a) - Q_1(S_t, A_t)] \tag{95}$$

Also, we can perform aggregations of the value function estimates in order to determine the behavior policy, like an epsilon greedy policy based on the averages of the two AV estimates. This can also naturally extend to ensembles of estimates as well.

Exercise 6.13:

$$Q_1(s, a) += \alpha(r + \gamma \sum_a \pi(a'|s') Q_2(s', a') - Q_1(s, a)), \tag{96}$$

$$\pi(a'|s') = \begin{cases} \frac{\epsilon}{|A(s)|}, a \neq \mathit{optimal} \\ 1 - \epsilon + \frac{\epsilon}{|A(s)|}, a = \mathit{optimal} \end{cases} \tag{97}$$

$$Q_2(s, a) = Q_2(s, a) + \alpha(r + \gamma \sum_{a'} \pi(a'|s') Q_1(s', a') - Q_2(s, a)) \tag{98}$$

Some other special cases of TD(0): In the case where we have knowledge of an initial part of the environment's dynamics but not, the full dynamics, we can use functions called *afterstate value functions*. These evaluate the state functions (or board positions in Tic-Tac-Toe) after the agent has made its move. This is useful in games since we know the immediate effects / transition dynamics of moves, but not how the opponent will reply, so it's useful to set up our grounding value function after our move.

Exercise 6.14: Jack's car rental is reformulated as an afterstates problem by: changing the state function to take in the states of the two locations *after* Jack moves the cars in between. This will speed up convergence since this reduces the state-action space, since there are many different actions Jack can take on a given night to transfer cars to get to a fixed set of combinations of cars at each location.

The version of TD presented in this chapter are *one-step, tabular, model-free* methods. There are extensions in each of these vectors that the later chapters cover.

N-step Bootstrapping

This chapter unifies MC and TD(0). N-step methods actually span a spectrum where MC methods are on one extreme and TD methods are on the other. Because we can now explore over multiple steps into the future, N-step algorithms are naturally better than TD(0).

n-step TD prediction:

N-step is simply seen as a generalization of the TD(0) algorithm we saw earlier - now the estimate for the n-step return is written as:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + .. + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), G_{t:t+n} = G_t, t + n \geq T \tag{99}$$

The n-step returns involve future rewards and states that aren't available during the one-step transitions - so at the beginning of each episode, we need to wait $n-1$ steps before we can start performing updates. To counteract this, we perform $n-1$ updates posthumously at the end of an episode. For a given timestep t, the first time the n-step returns are available is timestep $t + n$, so the update algo is then:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)], \tag{100}$$

$$V_{t+n}(S) = V_{t+n-1}(S), \forall S \neq S_t \tag{101}$$

Exercise 7.1:

$$G_{t:t+n} - V(S_t) = R_{t+1} + \gamma G_{t+1:t+n} + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) - V(S_t) \tag{102}$$
$$= \delta_t + \gamma(G_{t+1:t+n} - V(S_{t+1})) \tag{103}$$
$$= ... \sum_{k=t}^{t+n} \gamma^{k-t} \delta_k \tag{104}$$

Terminal states cancel out to 0 like in the chapter 6 example.

Exercise 7.2: Intuitively, it seems that the algorithm would perform worse on average than one where the value estimates do change step by step, since we could run into states inside $G_{t:t+n}$ that overestimate the value of state using stale functions and then bias towards that state. I'm going to test it on the random walk example given by the book.

Error reduction property: The n-step return uses the function V_{t+n-1} to correct for missing rewards beyond R_{t+n}. (The book doesn't provide a proof for this), but an important property of n-step returns is that the worst error of the expected n-step return is guarantee to be at least better than the discounted worst error under V_{t+n-1}:

$$\max_s |\mathbb{E}[G_{t:t+n}|S_t = s] - v_\pi(s)| = \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)| \tag{105}$$

This property allows us to confirm that all n-step TD methods converge to the correct predictions under appropriate technical conditions, probably through some convergence proofs.

Exercise 7.3: A larger random walk task was used in this chapter in order to see the effects of n-step TD more clearly - having more timesteps allows more updates to different states. A smaller walk allows episodes to terminate earlier, shifting higher advantage to smaller vlaues of n - it's no surprise that for a 19-step walk n-values that hover around the expected average length of a walk are the most advantageous, since they get to experience more states and provide better value updates. Making the change in the left side could affect the best value of n - a larger reward difference between the two sides would bias towards smaller values of n because you don't need as many timesteps to understand which side is better.

N-step SARSA

We can now combine n-step methods, with both evaluation (using the extended version to estimate return) and control. The first one is just creating a n-step version of SARSA. Recall that SARSA is on-policy TD(0) - this version is now called SARSA(0) and the new version is SARSA(n). The modifications are minimal - we are now incorporating action estimates everywhere so the equation becomes:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ...\gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \tag{106}$$
$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{107}$$

N-step SARSA follows the same relations with TD and MC methods - when N approaches infinity, it becomes a Monte Carlo method and when N = 0, then it is simply TD(0). Like always, to ensure convergence we use an ϵ-greedy policy so each state-action pair is visited an infinite number of times.

Exercise 7.4:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ...\gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}) \tag{108}$$
$$= R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - \gamma Q_t(S_{t+1}, A_{t+1}) + Q_{t-1}(S_t, A_t) - Q_{t-1}(S_t, A_t) + ... \tag{109}$$
$$\delta'_k = R_{k+1} + \gamma Q_k(S_{k+1}, A_{k+1}) - Q_{k-1}(S_k, A_k) \tag{110}$$
$$= Q_{t-1}(S_t, A_t) + \delta'_t - \gamma Q_t(S_{t+1}, A_{t+1}) + \gamma R_{t+2} + ... \tag{111}$$
$$= Q_{t-1}(S_t, A_t) + \delta'_t + \gamma(R_{t+2} - Q_t(S_{t+1}, A_{t+1}) + ... \tag{112}$$
$$= Q_{t-1}(S_t, A_t) + \delta'_t + \gamma(Q_t(S_{t+1}, A_{t+1}) - Q_t(S_{t+1}, A_{t+1}) + \delta'_{t+1} + \tag{113}$$
$$\gamma(R_{t+3} - Q_{t+1}(S_{t+2}, A_{t+2}) \tag{114}$$

The pattern will continue, where each $Q_k(S_{k+1}, A_{k+1})$ cancels out from the recursion relation, and we get the resulting sum that is equal to the novel TD error in Equation (7.6). The terminal condition just needs to be appropriately handled, so that the last Q value function becomes 0, or the reward goes to 0.

The n-step expected version of SARSA is nearly identical, the only difference is the return equation is rewritten using the *expected approximate value* of state s:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ...\gamma^{n-1} R_{t+n} + \gamma^n \bar{V}(S_{t+n}) \tag{115}$$
$$\bar{V}_t(s) = \sum_a \pi(a|s) Q_t(s,a) \tag{116}$$

We perform everything else the same - collect trajectories, save both next state and next action to stay on policy, use the $t-n+1$ to get the current update timestep.

N-step Off-policy

N-step off-policy utilizes the importance sampling ratio again, so when we are currently at timestep $t+n$, and updating the value functions for time step t, the equation will now:

$$V_{t+n}(S_t, A_t) = V_{t+n-1}(S_t, A_t) + \alpha \rho_{t:t+n-1}[G_{t:t+n} - V_{t+n-1}(S_t, A_t)] \tag{117}$$
$$\rho_{t:t+n-1} = \prod_{k=t}^{\min(T-1, t+n-1)} \frac{\pi(a_k|s_k)}{b(a_k|s_k)} \tag{118}$$

Get used to book notation - the order is $R_t \to S_t \to A_t$. Because off-policy methods subsume on-policy, using the importance sampling ratio we can generalize the n-step SARSA update to off-policy:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n+1}[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{119}$$

The slight modification is since we are using state-action pairs now, we shift the ρ by one-step, because we are updating a **state-action** pair and don't care about how the current action it is. Another nuance: Expected n-step SARSA would now use $\rho_{t+1:t+n}$, omitting the last step and replacing the $Q_{t+n-1}(S_{t+n}, A_{t+n})$ with an approximate value function sweep.

Per-decision methods with Control Variates: The motivation behind control variates to is allow efficient learning of the off-policy learning previously presented. The issue with previous off-policy methods is that when $\rho = 0$, all of the experience is ignored, and $G = 0$, causing high variance. If we have the vanilla n-step return equation, we can augment it using a control variate:

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}, t < h < T, h = t+n \tag{120}$$
$$G_{t:h} = \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1-\rho_t)(V_{h-1}(S_t)) \tag{121}$$

Now, when $\pi(a_t|s_t) = 0 \implies \rho_t = 0$, the control variate $1-\rho_t$ path will be taken, and we keep the estimate the same, no update, which is equivalent to ignoring the sample. This is a strict generalization of the original n-step return, so the two are identical in the on-policy case.

Exercise 7.5: Set up is similar, we initialize a V, as well as containers for states and then we also set up a ϵ-greedy policy π with respect to V as well as an exploratory behavior policy b. Then:

1. Start up episode, collect time steps until terminal state is reached and then set $T = t+1$, while $t < T$.
2. $\tau = t-n+1$, and $\tau \geq 0$, then calculate the return following the above equation, creating a buffer to store all ρ_t ratios as well. After calculating the return, use the standard n-step TD update to update $V_{\tau+n-1}(S_\tau, A_\tau) \to V_{\tau+n}(S_\tau, A_\tau)$.
3. Recalculate the target policy with new value function.

When using action-values, the ρ_t at the beginning is not used since we are now learning the pair (S_t, A_t), and for all intents $p(A_t|S_t) = 1$, so we shift the ratio time by 1. The book describes the n-step on-policy return ending at horizon h for expected SARSA:

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h} \tag{122}$$
$$= R_{t+1} + \gamma R_{t+2} + ...\gamma^{n-1} R_{t+n} + \bar{V}_{h-1}(S_h) \tag{123}$$

We can now rewrite to have an off-policy form with control variates. If $\rho_t = 0$, we want the multiplier to keep the same update, i.e $(1-\rho_{t+1})Q_{h-1}(S_{t+1}, A_{t+1})$ otherwise, we want to update with the continued return estimate. The off-policy n-step return with control variates is then:

$$(S_t, A_t) \to (R_{t+1}, S_{t+1}, A_{t+1}), G_{h:h} = \bar{V}_{h-1}(S_h) \tag{124}$$
$$G_{t:h} = R_{t+1} + \gamma(\rho_{t+1} G_{t+1:h} + \bar{V}_{h-1}(S_{t+1}) - \rho_{t+1} Q_{h-1}(S_{t+1}, A_{t+1})) \tag{125}$$
$$\tag{126}$$

When $h < T$, the final return estimate is $G_{h:h} = Q_{h-1}(S_h, A_h)$, and if $h \geq T \implies G_{T-1:h} = R_T$.

Exercise 7.6: The expected value of this new return:

$$\mathbb{E}_\pi[G_{t:h} | S_t = s, A_t = a] \tag{127}$$
$$= R_{t+1} + \gamma(\mathbb{E}_\pi[\rho_{t+1} G_{t+1:h}] + \bar{V}_{h-1}(S_{t+1}) - \mathbb{E}_\pi[\rho_{t+1} Q_{h-1}(S_{t+1}, A_{t+1})] \tag{128}$$
$$= R_{t+1} + \gamma(G_{t+1:h} + \sum_a \pi(a'|S_{t+1}) Q_{h-1}(S_{t+1}, A_{t+1}) - \sum_a \pi(a|S_{t+1}) Q_{h-1}(S_{t+1}, A_{t+1})) \tag{129}$$
$$= R_{t+1} + \gamma G_{t+1:h} \tag{130}$$

Exercise 7.7:

1. Same as always, set up hyperparameters and $Q(s, a)$ table, as well as buffers for S, A, R.
2. Start rolling out episodes with behavior policy b, while $t < T$. If we find a terminal state, then $T = t + 1$.
3. Backup stage: set $\tau = t - n + 1$. If $\tau \geq 0$, then we perform a reverse loop from $h \to t, h = \tau + n - 1$ - if $h \geq T$, then start at $T - 1$ instead, and use the return equation in 7.14 to keep updating the values, until we get $G_{t:h}$, then perform the TD error update for action-value n-step.

Exercise 7.8:

$$G_{t:h} = \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1 - \rho_t)V(S_t) \tag{131}$$
$$\delta_t = R_{t+1} + \gamma V(S_{t+1})) - V(S_t) \tag{132}$$
$$= \rho_t R_{t+1} - \rho_t V(S_t) + \gamma \rho_t G_{t+1:h} + V(S_t) + \rho_t \gamma V(S_{t+1}) - \rho_t \gamma V(S_{t+1}) \tag{133}$$
$$= \rho_t \delta_t + \gamma \rho_t G_{t+1:h} + V(S_t) - \rho_t \gamma V(S_{t+1}) \tag{134}$$
$$= \rho_t \delta_t + \gamma \rho_t(\rho_{t+1} \delta_{t+1} + \gamma \rho_{t+1} G_{t+2:h} + V(S_{t+1}) - \rho_{t+1} \gamma V(S_{t+2})) + V(S_t) - \rho_t \gamma V(S_{t+1}) \tag{135}$$
$$= \sum_{k=t}^{\min(h,T)-1} \gamma^{k-t} \rho_k \delta_k + V(S_t) \tag{136}$$

Exercise 7.9: Going to write down the expected SARSA TD error again:

$$\delta_t = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \tag{137}$$
$$= R_{t+1} + \gamma \bar{V}(S_{t+1}) - Q(S_t, A_t) \tag{138}$$

Now we need to write out the off policy N-step general:

$$G_{t:h} = R_{t+1} + \gamma \bar{V}(S_{t+1}) + \gamma \rho_{t+1} G_{t+1:h} - \gamma \rho_{t+1} Q(S_{t+1}, A_{t+1}) + Q(S_t, A_t) - Q(S_t, A_t) \tag{139}$$
$$= \delta_t + \gamma \rho_{t+1} G_{t+1:h} - \gamma \rho_{t+1} Q(S_{t+1}, A_{t+1}) + Q(S_t, A_t) \tag{140}$$
$$= \delta_t + \gamma \rho_{t+1}(\delta_{t+1} + \gamma \rho_{t+2} G_{t+2:h} - \gamma \rho_{t+2} Q(S_{t+2}, A_{t+2}) \tag{141}$$
$$+ Q(S_{t+1}, A_{t+1})) - \gamma \rho_{t+1} Q(S_{t+1}, A_{t+1}) + Q(S_t, A_t) \tag{142}$$
$$= \delta_t + \sum_{k=t+1}^{\min(h,T)-1} \gamma^{k-t} \rho_k \delta_k + Q(S_t, A_t) \tag{143}$$

N-step Tree Backup Algo: We can also try doing off-policy learning without IS, similar to Q-learning and Expected SARSA. The only difference is that those methods were the 1-step, and the *tree-backup* algorithm is the *n-step* extension. The backup diagrams for n-step algorithms usually just follow a single path until the S_{t+n} node, where we then perform some weighting over all A_{t+n}. The tree-backup now also accounts the estimated values of the dangling action nodes hanging off the side at each level.

The only action-value estimates that contribute are all the *leaf-nodes* of the tree, so the actual actions and path taken do not participate in the backup. The probabilities of the actual actions taken still contribute, since the k-th level of the tree will follow the action probabilities taken from $t : t+k-1$.

The detailed equations are as follows: for the one-step return, it is:

$$G_{t:t+1} = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) \tag{144}$$

$$G_{t:t+2} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})Q_{t+1}(S_{t+1}, a) + \tag{145}$$

$$\gamma\pi(A_{t+1}|S_{t+1})(R_{t+2} + \gamma \sum_a \pi(a|S_{t+2})Q(S_{t+2}, a)) \tag{146}$$

$$= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})Q_{t+1}(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:t+2} \implies \tag{147}$$

$$G_{t:t+n} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})Q_{t+1}(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:t+n} \tag{148}$$

If we keep normal bounding conditions, so that $0 \leq t < T, n \geq 2, G_{T-1:t+n} = R_T$. This is the return target, and then it is used with the normal n-step SARSA update rule:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{149}$$

Exercise 7.11:

$$G_{t:t+n} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})Q(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:t+n} \tag{150}$$

$$= Q(S_t, A_t) + R_{t+1} - Q(S_t, A_t) + \gamma \sum_a \pi(a|S_{t+1})Q(S_{t+1}, a) + \tag{151}$$

$$\gamma\pi(A_{t+1}|S_{t+1})[R_{t+2} - Q(S_{t+2}, A_{t+2}) + \gamma\bar{V}(S_{t+2}) + \gamma\pi(A_{t+2}|S_{t+2})(...] \tag{152}$$

$$= Q(S_t, A_t) + \delta_t + \gamma\pi(A_{t+1}|S_{t+1})[\delta_{t+1} + \gamma\pi(A_{t+2}|S_{t+2})(\delta_{t+2} + ...)] \tag{153}$$

$$= Q(S_t, A_t) + \sum_{k=t}^{min(t+n-1, T-1)} \delta_k \prod_{i=t+1}^{k} \gamma\pi(A_i|S_i) \tag{154}$$

Unifying Algo: n-step Q(σ): N-step SARSA, N-step expected SARSA and N-step tree backup all have extremely similar tree diagrams, where the only differences are where we choose to take the expectations. Expected SARSA does it at the last state, SARSA doesn't do it all, and tree-backup does it at every transition. n-step Q(σ) subsumes and unifies all three algorithms, by introducing a time-variate σ_t variable, where if $\sigma_t = 0$, we perform an expectation on that timestep, and when $\sigma_t = 1$, we perform a normal off-policy sampling update by calculating an importance ratio.

We develop the update equations for this new algo by beginning with the n-step tree backup update:

$$G_{t:h} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})Q_{h-1}(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:h} \tag{155}$$

$$= R_{t+1} + \gamma\bar{V}_{h-1}(S_{t+1}) - \gamma\pi(A_{t+1}|S_{t+1})Q_{h-1}(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:h} \tag{156}$$

$$= R_{t+1} + \gamma\pi(A_{t+1}|S_{t+1})(G_{t+1:h} - Q_{h-1}(S_{t+1}, a)) + \gamma\bar{V}_{h-1}(S_{t+1}) \tag{157}$$

This form is the exact same as N-step SARSA with control variates, but we swapped in the action probability instead of the ρ_{t+1}. So now the tree-backup algo and the N-step SARSA update have the exact same form, and we use the σ_{t+1} variable to linearly interpolate between the two:

$$= R_{t+1} + \gamma(\sigma_{t+1}\rho_{t+1} + (1 - \rho_{t+1})\pi(A_{t+1}|S_{t+1}))(G_{t+1:h} - Q_{h-1}(S_{t+1}, a)) + \gamma\bar{V}_{h-1}(S_{t+1}) \tag{158}$$

After we calculate the return, we can use the earlier N-step off-policy SARSA update equation, but without importance-sampling ratios $\rho_{t+1:t+n}$ since they are now incorporated in the return calculation:

$$Q_{t+n-1}(S_t, A_t) = Q_{t+n}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)] \tag{159}$$

Chapter Summary: N-step methods expand on the ideas of the TD methods in Chapter 6, showing that we can go to many more steps, as long as we are careful about our algorithms and timestep checks, as well as the return calculations. The algos just need to carefully encode the delay of n-time steps before updating, as well as more computation (that scales with n) to calculate return targets. N-step methods also require more memory buffers to store different information, although this can be alleviated by modular indexing.

Planning/Learning Tabular Methods

The book follows a nice pattern of first presenting algorithms that seem wholly separate, like MC and TD, and then showing higher level frameworks like N-step where MC and TD are just extremes. Chapter 8 is about going a level even higher now to unify the model-based, (DP and search methods) with the model-free methods, like MC and TD. Model-based focuses primarily on planning, while model-free focuses on learning, but the core goal of both algorithms is the computation of value functions, by looking ahead to future events, computing a backed-up value and improving an estimate.

Models

Models of an environment are anything that an agent can use to predict how the environment will respond to its actions, so given a (s, a) it will predict (s', r). Distribution models give the entire probability / sampling range of next states and rewards, while sample models give only a single sample of one. Models are used to mimic/simulate experience, usually at a much lower cost compared to actually interacting with the environment.

Planning - the computational process of taking a model as input and producing/improving a policy interacting with the environment. *state-space planning* is a search through the state space for optimal policies / paths, usually by improving a value function over states. *plan-space planning* is a search over the space of plans, like evolutionary methods. A key idea is that all state-space planning methods follow a structure: computing value functions to improve the policy, and this computation involves update / backup equations through simulated experience.

The first union we see between planning and learning algorithms is that learning equations can be plugged in at the update/backup step of a planning algo, since learning only requires experience as input, simulated or real. Another important aspect of planning is the benefits of small, incremental steps, to ensure quick changes in direction and little wasted computations, while interleaving acting and learning.

Dyna

When performing online planning alongside learning, there are a few interesting dynamics that come into play between planning and learning. Both of these require computation, so it becomes a decision problem about how to divide resources and when to prioritize what. There are several types of learning that occur in Dyna-Q for online planning agents:

1. Model-learning: using real experience to improve the world model
2. Direct RL - using real experience to directly learn better values/policies (learning)
3. Indirect RL - improving the world model and then using planning algorithms to update the value/policy function

Experience is collected by the value / policy function acting with the environment. Indirect methods are often more sample-efficient, but direct methods are simpler and aren't dragged down by biases in model design.

Dyna-Q has planning, acting, model-learning and direct RL. The planning method (indirect RL method) is the random-sample one-step tabular Q-planning method, where random sample experience is collected from the model and the update is Q-learning (max off-policy). Direct RL is simply one-step tabular Q-learning. Model learning is tabular and assumes a deterministic (and small enough to be tabular) environment. Conceptually all four stage are happening simultaneously, but on serial computers they are done sequentially.

Dyna-Q has the best of both worlds, by integrating learning and planning, operating both on real and simulated experience - this is similar to the idea of sleep time compute. While the agent is acting according to its most updated value functions and learning, planning can also happen in parallel, so the agent is always reacting based on its most recent sensory information while still continuously planning in the background. The model-learning process keeps going too, creating tight feedback loops that converge much faster than normal learning.

Exercise 8.1: The n-step bootstrapping models would perform better than $n = 0$, but the core issue is still apparent - that Dyna on the second episode can learn 50x times more compared to a $n = 50$ learning method. Planning can happen in parallel alongside learning, so the convergence is much faster, compared to the serial updates that will happen in N-step. N-step helps because it will produce almost equivalent amounts of updates with real experience, though.

When the Model is Wrong: The deterministic tabular assumptions of the model are idealized, and often many factors can cause a model to be wrong and misapproximate the environment dynamics. Then the planning process computes suboptimal policies. Usually, these can be caught when planning with the suboptimal process, since the model will optimistically follow a suboptimal path and correct predictions. Greater difficulties arise when the environment doesn't become worse, but *better* - the model might continue to take its old path, since it hasn't become worse, and fail to find the better, more optimal path without exploration bonuses. Simple exploration encouraging heuristic, like Dyna-Q+ in the book, often help with this.

Exercise 8.2: The exploration bonus helps to explore / consider states faster, so the planning agent could find the hole in the wall quicker in both stages. Actions that were never tried were allow to be in the planning step to increase state-space coverage. The exploration bonus in the shortcut maze example allows it to continue to try new states, even after it found the left opening.

Exercise 8.3: It narrowed slightly because the Dyna-Q+ with exploration bonus and more coverage converges faster, but as the timesteps progress Dyna-Q's world model and planning catches up - when it goes to infinity both would converge to the same performance.

Exercise 8.5: The model should handle stochastic environments by now also storing the transition probability over all possible next states and rewards pairs, and then when performing the update in the planning step, we should perform a weighted sum over the (s, a, r, s') pairs and then multiply by the step size. It will perform poorly in changing environments because now the transition probabilities, like when a path gets completely blocked, need to all be reset. In order to handle changing environments, we should maybe keep an EMA of the probabilities of different states? While ensuring they still sum to 1.

Prioritized Sweeping:

Often times, it makes a lot more sense to have some intentional selection of state-action pairs for planning rather than a fully random search, especially at the beginning of training runs. It doesn't make sense to select states where none of the transitions have been updated yet - the target and updates will equate to zero and no learning is performed.

To expand beyond the ideas of the "goal state" in mazes, we want to extend to general reward functions. Here, if the value of a certain state changes, then the values of any of the "predecessor states" with possible transitions into this state should also have their values updated. Furthermore, in stochastic environments we should weight the magnitude of this value change alongside the transition probability to get a priority score. We then maintain a priority queue of state-action pairs ranked by these priority scores, and pick the first one. When the top pair in the queue is updated, the effect on all the other states is calculated, alongside priorities, and these are inserted into the queue with a new priority, and so forth.

Extensions to stochastic environments: Instead of updating the model with a sample update, which we could do in deterministic environments, we would need to perform an expected updates, meaning we keep track of all possible next states and their probabilities so far. This would be easier in a DP situation where the transition models are known - in the case where they aren't this would require running count averages which could explode memory. Expected updates are also inefficient because they need to take into account low, near-zero probability states in updates. Sample updates are usually faster even in stochastic environments because it breaks up the long chains of trajectory computation into the immediate state backup.

In "small state backups", we update along a single transition like a sample update, but prioritized based on the probability of the transition. While prioritized sweeping is a *backward-focusing* algorithm, there are also forward-focusing algorithm, which focus on states according to how easily they can be reached from frequently visited states under the current policy. Again, all these state-space planning methods still focus on the computation and backup of the value function.

Expected vs Sample Updates

One-step updates vary among these three binary dimensions:

1. State vs action-values - most methods have both versions of estimating and updating these value functions

2. Whether they estimate value of optimal policy or arbitrary policy - the difference here is whether the (Bellman) value update equations take $\max_a$ for v_*, or $\max_{a'} q_*(s', a')$ for q_*.
3. Expected vs sample updates (DP vs TD(0)): expected consider all possible events that can happen, while sample updates focus on the presently surfaced transition. Some differences - for $q_*(s, a)$, expected is q-value iteration, which is (DP) value iteration with maxes over next actions, while sample would be Q-learning - bootstrapping and taking expectations over next actions.

Intuitively, expected samples are expected to perform better than sample updates, because they account for all the states and transitions, and yield better estimates since they are uncorrupted by sampling error, at the expense of more computation. If we define b as the branching factor, the number of possible next states s', then expected updates are around b times more computation than a sample update. If there is enough time / computation to make an expected update, the resulting estimate is better than b sample updates.

However this is all problem dependent. Usually this requires some form of comparative analysis to understand for a given unit of computational effort, whether a few expected updates is better than b times as many sample updates, across states. The book performed a small experiment, but the main message is that because many states are getting updated simultaneously for sample updates, there is a network effect where backed up states provide better estimates for their predecessors - sample updates are superior on problems with large stochastic branching factors.

Exercise 8.6: It would strengthen the case for sample updates - since the distribution is highly skewed towards only some actions now, the action space effectively shrinks and sample updates would even better approximate expected updates, since there are less actions to worry about. Because sample updates in the case of $|A(s)| = 1$ are equivalent to expected updates.

Trajectory Sampling:

This section concerns / compares two ways of distributing updates to state-action pairs. The first one is the DP approach, which performs sweeps through the state-space, updating every single pair once per sweep. This is often expensive and unnecessary - as long as we ensure in the limit each state is visited an infinite number of times, convergence is guaranteed.

Trajectory Sampling is the second option, where we sample state/state-action pairs according to a distribution, like the on-policy distribution, simply by interacting with the model and following the current policy. Naturally, one is just simulating explicit individual trajectories and performing updates based on these sample transitions. This is the best middle ground compared to trying to enumerate the distribution, which would be near equivalent in computational cost to policy evaluation (prediction), and is elegant in that it handles the nonstationary dynamics of the on-policy distribution.

Running a few experiments, with stochastic branching factors and some random rewards for each transition show how on-policy does better than its counterparts. All the graphs showed that on-policy initially performs better, and much better for higher state counts and lower branching factors, but eventually uniform state-action pair sampling catches up. This makes sense - if there are many states and a small branching factor, then there aren't as many possible paths and sampling according to the on-policy will allow the smaller path space to be quickly covered. However, in the limit uniform sampling catches up, because on-policy will continue to explore previously trodden paths without switching it up.

Exercise 8.7: The scallop shape, is because of the $b = 1$ branching factor - because this reduces the state-action pair state space to only the amount of states, this allows it to be easily searchable, which is why it first learns so rapidly. Once all the states have been visited a few times, on policy can't explore other states besides the ones it is biased towards, so performance bottoms. Uniform sampling has a slightly better learning rate, since it will still sample infrequent states and get better estimates.

Real Time Dynamic Programming

RTDP is a on-policy trajectory-sampling version of the value iteration algorithms of DP. Recall that value iteration is the interleaving of policy evaluation and improvement, using the tabular update equation that takes the max action over the probability weighted next states. RTDP is a form of *asynchronous DP*, where there are no systematic sweeps - the order is by visited states. Since RTDP can only visited states that are relevant (reachable) under some optimal policy, the policy becomes an *optimal partial policy*, only optimal for the relevant states. Finding these partial policies still requires that all the state-action pairs are visited an infinite number of times - this can be resolved with strategies like exploring starts, where RTDP converges for discounted finite MDPs.

The most interesting consequence of RTDP is instead for problems where it is guaranteed to find an optimal partial policy without visiting every state - it holds for undiscounted episodic MDP tasks with absorbing goal states that generate zero reward - at every step or a real / simulated trajectory, RTDP selects a greedy action and applies the expected value-iteration update. It can also perform multi-step lookahead searches and do arbitrary updates on those ahead states.

The concrete conditions for convergence are:

1. Initial value of every goal state is zero
2. There exists at least one policy that guarantees that a goal state will be reached with probability 1 from any start state (in the policy space)
3. All rewards of transitions from non-goal states are strictly negative
4. All the initial values are equal to greater than their optimal values (which is just setting all of them to 0, since all returns are negative)

These tasks are examples of *stochastic optimal path problems*, stated in terms of cost minimization of different paths in the path space from start to finish. Examples in the book showed RTDP is up to 2x more efficient than conventional DP, because as the policy becomes more optimal, the value iteration updates focus on the relevant state subset, encouraging convergence.

Decision Time Planning Methods

Algorithms like DP and Dyna are examples of *background planning*, since the planning is not focused on the current state, and instead does sweeps across the entire tabular search space in order to improve the value functions. Decision-time planning is when planning begins and is completed after encountering each new state S_t, as a computational process that selects a single action A_t, so it focuses only on a single state. Decision time planning is much more ephemeral - the same pattern of simulated experience to updates/values to policies is still apparent, but since the values and policies are specific to the curren state, they are often discarded after the current action selection step. DTP is most useful in scenarios where low latency actions are not required, like chess.

Heuristic Search: This is the classical state-space planning method in AI, where for each state encountered, we first enumerate huge decision trees. The approximate value functions are evaluated at the leaf nodes, and backed up towards the current state. Once estimates are computed, we choose the best action with the highest value, and then discards all the backed-up values. Heuristic search can be viewed as an extension of the UCB and ϵ-greedy search methods described so far in the book, just taking it beyond a single step. Searching deeper also gives better action selections - with a perfect model and an imperfect action-value function, a deeper search (usually) yields better policies. Heuristic search is also considered so effective because it focuses all the memory and computational resources on the current decision.

Heuristic search and the search tree that results from it can also be converted into the familiar RL one-step paradigm - by performing individual one-step updates from bottom-up, this will achieve the same overall updates as depth-first heuristic search.

Rollout Algorithms: These are Decision-Time planning algorithms based on Monte Carlo Control applied to simulated trajectories, beginning at the current environment state. They estimate action values for a given policy by averaging returns of many simulated trajectories that start with an action and then follow a policy. An example for backgammon is to rollout out a bunch of randomly generated sequences of dice rolls from a given position, and estimate the state-action values from the current position, playing till the game's end.

Motivation behind rollout algorithms: following the policy improvement theorem again, if two policies are identical, π, π', except that $\pi'(s) = a \neq \pi(s)$ for some states, if $q_\pi(s, a) \geq v_\pi(s)$, then $\pi' \geq \pi$ and the strict inequality follows. So rollout algorithms always maximize $q_\pi(s, a)$ and improve over the rollout policy, like one-step policy-iteration. The performance of the improved policy will then depend on the properties of hte rollout policy and the rankings of the actions of the MC estimates. Rollout policies have a lot of factors to consider, since they need to meet strict time constraints, and we need to consider the length / computational expenditure of trajectories, as well as the number of actions that need to be evaluated. This is partially alleviated by running many rollouts on parallel processors, and truncating simulated trajectories, as well as pruning candidate actions that are unlikely to turn out to be best.

Monte Carlo Tree Search: This is a rollout algorithm that is enhanced with some "memory", by accumulating value estimates from MC simulations toward more highly-rewarding trajectories. It does this by extending initial portions of trajectories that were ranked highly from earlier simulations, by maintaining a core tree that it selectively grows. This tree is the state-action pairs, with a few-step look ahead that are most likely to reached in a few steps, and then it incrementally grows this tree by adding state nodes that have promising simulated trajectories.

Outside of the central tree, a less optimal, faster rollout policy is used, but inside the tree we can use a value-informed *tree policy*, like the ϵ-greedy policy. Each iteration of a basic MCTS follows these four steps, i.e for each state:

1. Selection: starting at the root node, a tree policy based on action values attached to edges of the trees traverse the tree to select a leaf node.

2. Expansion: On some iterations, the tree gets expanded from the selected leaf node by adding child nodes that represent unexplored actions.

3. Simulation: From the selected node, simulation of a complete episode is run by a rollout policy, so the result is a segmented Monte Carlo trial with actions selected by the tree policy and then the rollout policy.

4. Backup: Using the returns generated/averaged over many simulated episodes to initialize the action values attached to edges of tree traversed by the tree policy.

MCTS continues to execute these four steps, starting at the root node, until some constraint is met, then chooses an action based on some metric. Then we go to the next state, where MCTS can reuse descendants of the old tree that are also descendants of this new state. MCTS at its core is a decision time planning algorithm based on Monte Carlo control applied to simulations starting from the root-state, so it is a rollout algorithm benefiting from online, incremental, sample-based value estimation and policy improvement. By updating the action-value estimates of the root tree's edges, then traversing using an intentional policy, MCTS focuses on initial tree segments which are common to high-return trajectories. By incrementally expanding the tree, MCTS grows a lookup table storing partial action-value function estimates, which allow it to have some past experience to guide exploration while avoiding global approximation of an action-value function, best of both worlds.

Summary:

Planning always requires a model of the environment, whether that be a distribution model, which is what DP uses for expected updates, or a sample model, which is what algorithms like on-policy TD and MC control utilize. The latter is much easier to obtain. Planning and learning algorithms are extremely similar, and can be integrated with each other as they both involve backing up value estimate incrementally - their only difference is the sources of experience. The chapter touches on several different tweakable vectors of state-space planning methods:

1. The size / number of planning updates. Dyna-Q allows to plan for n steps.

2. Distribution of the search: Focus of search matters, Dyna-Q+, Dyna-Q-Action+ enable exploration bonuses during the planning stage, while prioritized sweeping focuses backward on predecessors of states with large TD(0) error. On-policy trajectory sampling performs updates and focuses on trajectory segments the policy is very likely to visit, and a version of this, Real-Time DP is an on-policy trajectory sampling version of value iteration, which under the right conditions converges much faster than conventional sweep-based DP.

Part 1 Summary: Dimensions

There are many different dimensions that are involved when exploring the solution space of RL problems - right now limited to tabular updates:

1. Expected vs sample updates - DP vs TD-0, and then other mixtures of sample and expected updates mixed in

2. Depth of search - one-step TD(0) to rolling out the entire episode to estimate returns

3. On-policy vs off-policy - importance sampling

4. Action-values vs state-action values
5. Action selection/exploration - encouraging exploration while maintaining greediness
6. Real vs simulated experience
7. Sychronous vs Asychronous updates
8. Location of updates / timing of updates - Which state action pairs should be updated and when?
9. Memory for updates - how long should we store the updated value function estimates? extremes are from tabular updates which keep it for the lifetime of the run, and Decision time planning methods that discard all memory after new state

Part II: Approximate Solution Methods

Chapter 9: On-Policy Prediction with Approximation

Function approximation can be used to estimate the state-value function of on-policy data, approximating from experience generated using a known policy π, where $v_\pi(s) \approx \hat{v}(s, \mathbf{w})$ with weight vector. Extending RL to function approximation makes it applicable to partially observable problems. Because we are now using functions where the magnitude of the weight vector is much smaller than the size of the state set, changes to the function will affect estimates of many states, so updating s needs to generalize to many other states that have also changed. When choosing supervised learning / regression methods as the approximators, we want models that are efficient incremental learners and can handle non-stationarity well.

The Prediction Objective Because of how large the state space is, and the considerable smaller weight space, we need to be selective about which state values we care about being correct, so we specify a distribution over the states that weights the error, giving us a total mean square value error:

$$\mu(s) \geq 0, \sum_s \mu(s) = 1, \bar{VE}(\mathbf{w}) = \sum_s \mu(s)[v_\pi(s) - \hat{v}(s, w)] \tag{160}$$

A nice example $\mu(s)$ is the fraction of time spent in s. $\mu(s)$ is the on-policy distribution, and for continuing tasks the on-policy distribution is the stationary distribution under π. For episodic tasks, it is slightly more nuanced since now the distribution depends on the initial states of the episodes + the number of time steps spent on average in state s in a single episode. If $h(s)$ is the prob we start in state s for an episode, or if transitions are made from state $\bar{s}$ to s, the total time is: $\eta(s) = h(s) + \gamma * \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s}, a)$. This system of equations is solved for expected number of visits to $\eta(s)$, and $\mu(s)$ is the normalization of this value across all states.

Stochastic-gradient and semi-gradient methods

SGD for value approximation is: select a state using some heuristic search, then adjust the weight vector by a small amount in the direction that reduces error:

$$w_{t+1} = w_t - \frac{1}{2}\alpha \nabla [v_\pi(S_t) - \hat{v}(S_t, w_t)]^2 = w_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t) \tag{161}$$

When the step size follows the Robbins-Monro stochastic approximation conditions, the SGD method is guaranteed to converge to a **local optimum**. In cases where $v_\pi(S_t)$ is not known exactly, we can use noisy estimates U_t, and as long as U_t is an unbiased estimate with $\mathbb{E}[U_t|S_t = s] = v_\pi(s)$, then w_t is guaranteed to converge to a local optimum. For example, MC target $U_t = G_t$ is an unbiased estimate of $v_\pi(S_t)$, at the expected limit, so a Gradient MC algorithm for estimation would be to simulate infinite episodes, then generate an episode using π, and loop for each step of the episode collecting returns and performing SGD.

When the targets are bootstrapped and not independent of w_t weight vector, they are biased estimates and this is not true GD. Bootstrapping methods take into account the effect of changing the weight vector w_t on the estimate - this is the error term, but ignore the effect on the target's response to shifts in

w, and these are partial gradients we call them *semi-gradient methods.* These still have their advantages, like faster learning and they enable learning to be continual and online instead of waiting for return calculations. A common semi-gradient method is semi-gradient TD(0), with $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$ as the target.

State aggregation is a simple generalization of function approximation where states are grouped together, with one component of the weight vector for each group, so that a natural partitioning in the weight vectors occurs. The value of a state is then estimated as it's group component, and when we perform GD update, only that component is updated.

Linear Methods

The approximate function in this special case is just a linear function of the weight vector: $\hat{v}(s, w) = w^T x(s)$. The SGD update reduces to $w_{t+1} = w_t + \alpha[U_t - \hat{v}(S_t, w_t)]x(S_t)$. The nice thing about the linear case is that there is only one optimum, (or in degenerate cases, one set of equally good optima), and any method that converges to a local optimum is guaranteed to converge to the global optimum. The gradient Monte Carlo algorithm converges to the global optimum, but the semi-gradient TD(0) method will converge to a point near the local optimum, which is a special case. The update each time t is

$$w_{t+1} = w_t + \alpha(R_{t+1} + \gamma w_t^T x_{t+1} - w_t^T x_t)x_t \tag{162}$$

$$= w_t + \alpha(R_{t+1}x_t - x_t(x_t - \gamma x_{t+1})^T w_t) \tag{163}$$

Where we rearranged the dot product and moved the x_t term in. Once the system reaches steady states, for any given w_t the expected next weight vector is written as: $\mathbb{E}[w_{t+1}|w_t] = w_t + \alpha(b - Aw_t)$, where $b = \mathbb{E}[R_{t+1}x_t] \in R^d, A = \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T)$. If the system converges, it converges to the weight vector w_{TD}, which is a fixed point: $b - Aw_{TD} = 0 \implies b = Aw_{TD} \implies w_{TD} = A^{-1}b$. So solving this system of $Aw_{TD} = b$ gives the point linear semi-gradient TD(0) will converge to. There are some properties that assure convergence of the linear TD(0) algorithm, by first rewriting the expected update equation as:

$$\mathbb{E}[w_{t+1}|w_t] = (I - \alpha A)w_t + \alpha b \tag{164}$$

In this situation, b almost acts like a bias term, and only A is important to convergence. Specifically, A must be positive definite - if any diagonal element of A is negative, then $(I - \alpha A)$ for the element will be positive and w_{t+1} will tend towards divergence. Positive definiteness is also required so that A is nonsingular. In the continuing case with $\gamma < 1$, the A matrix is written as:

$$A = \mathbb{E}[x_t(x_t - \gamma x_{t+1})^T] = \sum_s \mu(s) \sum_a \pi(a|s) \sum_{r,s'} p(r, s'|s, a)x(s)(x(s) - \gamma x(s'))^T \tag{165}$$

$$= \sum_s \mu(s) \sum_{s'} p(s'|s)x(s)(x(s) - \gamma x(s'))^T \tag{166}$$

$$= \sum_s \mu(s)x(s)(x(s) - \gamma \sum_{s'} p(s'|s)x(s'))^T \tag{167}$$

$$= \sum_s \mu(s)x(s)x(s)^T - \gamma \sum_s \mu(s)x(s)[\sum_{s'} p(s'|s)x(s')]^T \tag{168}$$

$$= X^T DX - \gamma X^T DPX = X^T D(I - \gamma P)X \tag{169}$$

In order to determine positive definiteness, $D(I - \gamma P)$ needs to be positive definite. P is the transition matrix of probabilities $p(s'|s)$ from s to s' under policy π. The book shows the proof of its positive definiteness, which requires two other theorems, and showing that all of its columns sum to nonnegative numbers. Further, at the TD fixed point it has been proven in the continuing case that the VE error is within a bounded expansion of the lowest possible error:

$$\bar{VE}(w_{TD}) \leq \frac{1}{1 - \gamma} \min_w \bar{VE}(w) \tag{170}$$

This provides a nice upper bound, and since $\gamma \approx 1$ usually, the expansion factor can be quite large, with a large potential loss in asymptotic performance using the semi-gradient TD(0) approach. Other convergence requirements are the usual Robbins-Monro conditions on the step size + the states are updated according to the on-policy distribution. For other update distributions, **bootstrapping methods + function approximation can diverge to infinity**.

Other methods like semi-gradient n-step TD algorithms is also analogously an extension of semi-gradient TD(0), and is the natural extension of the tabular n-step TD algorithm in Chapter 7. The pseudocode is:

1. Perform appropriate initialization, make $\hat{v}$ a differentiable function, and initialize value function weights arbitrarily.
2. Start looping and collecting data on-policy until we get to the terminal timestep, then record it. Every loop, check if the timestep n steps back is a valid timestep to be updated, then perform one gradient step.

The only difference with this semi-gradient method and the tabular is that the update equation uses a gradient instead of a tabular / expected-value update.

Exercise 9.1: Tabular methods are a special case where the feature vectors are binary masks, so that for a given state, we mask all states beside the current one and then perform a fetch into the table.

Feature Construction for Linear Models:

Features construction is an important way of giving more knowledge to the approximation models, by encoding important / relevant aspects of the problem as different feature vectors. However, a limitation of the linear form is that it cannot take into account interaction / mixing between features, such as the presence of feature i only being good when feature j is absent. It needs to create more feature vectors for different combinations of underlying state dimensions in the problem in order to learn fully.

Polynomials

Polynomials are a good starting point, since many states of problems are expressed as numbers, like in pole balancing. If a state can be represented by k different numbers, then for this k-dimensional state space, we can construct an order-n polynomial basis feature set, with each order-n polynomial x_i written as:

$$x_i(s) = \prod_{j=1}^{k} s_j^{c_{t,j}}, c_{t,j} \in \{0, 1..n\} \tag{171}$$

So each individual features $x_i(s)$ is written as a product of all the state components raised to different powers, giving $(n+1)^k$ different features. Higher order polynomials allow for more accurate approximations - we can think of them like Taylor series, but the state space grows exponentially. It is better to select a subset of them for function approximation, using prior beliefs about the nature of the function to be approximated, and some automated selection methods for polynomial regression.

Exercise 9.2: The number of features is defined like this because for each $s_i, 1 \leq i \leq k$, each of them can be raised to any of the $n+1$ powers, and multiplied k ways with the other components.

Exercise 9.3: $n = 2$, $c_{i,j} = \{(0,0), (1,0), (0,1), (1,1), (2,0), (2,1), (1,2), (2,2)\}$

Fourier Basis

We can also use another series / function-based approximation, by using the Fourier series. The Fourier basis is simple to use, just variations of the periodic sinusoids, and any function can be approximated to some degree with enough terms.

In the 1D case, the Fourier series representation of a 1D function with period τ is a function as a linear combination of periodic sinusoids with periods that evenly divide τ. If instead, in the more realistic case, we are interested with aperiodic functions over bounded intervals, then we should calculate Fourier basis features with τ set to the length of the interval. The approximation is just one period of the periodic linear combination. By setting τ to twice the interval and restricting to the first half, then you can just use the cosine features, since any even function can be approximated by enough cosine features, and we're restricting the half-interval from $[-\tau/2, \tau/2] \to [0, \tau/2]$.

By setting $\tau = 2$, the one-dimensional order-n Fourier cosine basis has $n+1$ features, $x_i(s) = \cos(i\pi s), s \in [0,1]$. In the multidimensional generalization, if each state s corresponds to a vector of k numbers, with $s_i \in [0,1]$. The ith feature of the order-n Fourier cosine basis in the k-dimensional space is written: $x_i(s) = \cos(\pi s^T c^i)$, with each coefficient vector c^i being a k-length vector with integer components between $[0, n]$, so these define the $(n+1)^k$ feature vectors, with the effect of assigning a feature frequency along that dimension. So we switched from powers in the polynomial basis to frequencies along different state dimensions in the Fourier basis. When multiple c^i_j components are nonzero, this represents interactions between two state variables, with the values determining frequencies and the ratios giving directions.

For applications, it is recommended to have different step-size parameters for each feature, like for each x_i setting $\alpha_i = \alpha / \sqrt{(c^i_1)^2 + ..(c^i_k)^2}$. Fourier cosine features + SARSA provide good performance, but don't do well approximating discontinuities, because of its attempts to approximate with lower frequency functions. The number of features also grows exponentially - but with smaller $k \leq 5$, we can modulate n so that all of the order-n Fourier features are used, making selection of features automatic. For higher dim spaces, we need to use prior beliefs about the functiob being approximated. Because Fourier features are non-zero over most of the entire state space (due to sinusoids), they represent global properties of states, and it is difficult to find local properties.

Coarse Coding

Coarse coding is using some sort of shapes that cover the state space, and then creating a binary feature vector to indicate the shapes that a state is contained in. If we assume linear GD function approximation, then the allocation of circles, as well as their size and density across the space determine the generalization of the features. If one state gets trained, the weights of the circles inside are affected, as well as the states inside the union of those circles. Smaller circles have narrower generalization, while elongated circles will have asymmetric generalization. However, receptive field shape's only tends to have a strong effect on generalization, but little effect on asymptotic solution quality.

Tile Coding

In tile coding, the receptive fields of the features are grouped into partitions of the state space. It holds a similar concept to coarse coding, where state features are generated by observing the tiles the state is contained in. The case of one tiling is equal to state aggregation. To get the strengths of coarse codings, this requires overlapping tilings, offset by a fraction of the tile width, and the feature vector has one component for each tile in each tiling - for 4 tiles on a 4 x 4 tile board, there are 64 components.

A nice advantage of tile coding is that due to the partition-based nature, the overall number of features active for each state is fixed, so the step-size parameter α can be set in an easy, intuitive way. If one chose $\alpha = 1/10n$, then the estimate for the trained state moves one-tenth of the way to the target, and neighboring states will be moved less, proportional to the number of states they have in common. This is how generalization across state occurs, and the choice of how to offset the tiles is affects generalization.

If the tiles are offset uniformly in each dimension, then different states can generalize in qualitatively different ways, but the pattern of generalization is noticeably diagonal. This affect can be avoided by performing asymmetrical (unequal in different dimensions) offset tilings, which have better generalization patterns centered on the training states. In any case, tilings are always offset from each other by a fraction of tile width in each dimension, and the fundamental unit is w/n, w = tile width, n = number of tilings. From studies, a good choice of displacement factor for dimension k is the first odd integers $(1, ..2k-1)$, with n set to power of 2 greater than $4k$.

Tiling strategies are also nuanced - one has to pick a number of tiles and the shape of tiles (with the requirement that they are partitions). Square tils generalize roughly equally in every dimension, because of the equal length. Tiles that are elongated along one dimension will promote generalization along that dimension, since more points are included, while in other dimensions that are denser and thinner, discrimination will be promoted in those denser, thinner regions. Diagonal tilings promote generalization along one diagonal.

A trick for reducing memory requirements is *hashing* - by collapsing many tiles into much smaller sets of tiles. One tile might consist of four subtiles, all disjoint noncontiguous regions. Hashing provides little loss in performance, because high resolution is often only needed in a fraction of the state space, and it also frees us from the curse of dimensionality because memory requirements are not exponential in the number of dimensions.

Exercise 9.4: Denser tilings on this dimension, so that we have more discrimination between values in this dimension, so lots of dense, thin stripes.

Radial Basis functions

RBFs are generalization of coarse coding techniques to continuous-valued functions, by allowing the feature to be in the interval $[0,1]$, allowing more flexibility. The typical RBF is the Gaussian response: $x_i(s) = \exp(-||s_i - c||^2/2\sigma_i^2)$. RBF's primary advantage is approximate functions that are smooth and differentiable, but practically they show no real advantage, and degrade in higher state dimensions, where the edges of tiles (which binary features ONLY care about) are are more important.

Selecting step-size parameters manually

SGD methods usually require manual tuning of the α step-size parameter. If we return to the tabular case, a step size of $\alpha = 1$ completely eliminates the sample error, but we want to learn slower than this to avoid overestimating from one sample state. In general, if $\alpha = 1/\tau$, then the tabular estimate will reach the mean of its targets, with more recent targets having greater effects after about τ state visits. Function approximation is not as clear, because now the states aren't enumerated. Instead, a good rule of thumb is $\alpha = (\tau\mathbb{E}[x^T x])^{-1}$, where x is a randomly sampled feature vector from the input distribution. This method works best when the feature vectors do not vary greatly.

Exercise 9.5: We should use Eq 9.19:

$$E[x^T x] = \sum_{i=1}^{98} x_i^T x_i \tag{172}$$

We assume the tilings are uniformly sampled, but we separate them according to which dimensions they are along:

$$\sum_{i=1}^{56} x_i^2 p(x_i) + 2\sum_{j=1}^{21} p(x_j)x_j^2 \tag{173}$$

$$= \sum_{i}^{56} \frac{1}{8} * 1 + 2\sum_{j}^{21} \frac{6}{21} * 1 = 8 + 12 = 20 \tag{174}$$

$$\alpha = (10 * 20)^{-1} \tag{175}$$

The 1/8 comes from the fact that each feature, when inside its own dimension has a uniform 1/tile chance to be selected due to the partitioning scheme. The 6/21 comes from the pair combinations.

Exercise 9.6:

$$\alpha = (\tau\mathbb{E}[x^T x])^{-1} \tag{176}$$

$$w_{t+1} = w_t + (x(S_t)^T x(S_t))^{-1}(U_t - \hat{v}(S_t, w_t))\nabla\hat{v}(S_t, w_t) \tag{177}$$

$$= w_t + (x(S_t)^T x(S_t))^{-1}(U_t - w_t^T x(S_t))x(S_t)^T \tag{178}$$

$$= w_t + (x(S_t)^T x(S_t))^{-1}U_t - w_t = \tag{179}$$

Nonlinear Function Approximation

A lot of this chapter is just explaning recent advancements (up to 2016) in ANs. NN can learn using TD errors to learn value functions, or using a policy-gradient algo (Chapter 13).

Least-Squares TD

As established before, TD(0) with linear function approximation converges asymptotically to the fixed point $w_{TD} = A^{-1}b$. Instead of computing this solution iteratively, we could try computing estimates of A, b and then directly computing the fixed point - this is Least-Squares TD. It forms the estimates:

$$\hat{A}_t = \sum_{k=0}^{t-1} x_k(x_k - \gamma x_{k+1})^T + \epsilon I, \hat{b}_t = \sum_{k=0}^{t-1} R_{k+1}x_k \tag{180}$$

So normal semi-gradient TD(0) computes the expectation iteratively, until we get to the fixed point, while LSTD directly estimates the expectation value from collected data. $\epsilon > 0$ is some small value to ensure A is invertible. This method is more data efficient than semi-gradient TD(0), but requires more computation - the outer product to update A is a matrix update + memory to hold A is $O(d^2)$. Calculating $\hat{A}^{-1}$ is also costly but we can use the *Sherman-Morrison formula*, which decomposes a sum of outer products into $O(d^2)$ computations:

$$\hat{A}_t^{-1} = (\hat{A}_{t-1} + x_{t-1}(x_{t-1} - \gamma x_t)^T)^{-1} \tag{181}$$

$$= \hat{A}_{t-1}^{-1} - \frac{\hat{A}_{t-1}^{-1} x_{t-1}(x_{t-1} - \gamma x_t)^T \hat{A}_{t-1}^{-1}}{1 + (x_{t-1} - \gamma x_t)^T \hat{A}_{t-1}^{-1} x_{t-1}} \tag{182}$$

This allows the update to be done in $O(d^2)$ computation as well, but this is still much larger than $O(d)$ for semi-gradient TD. Whether the better data efficiency of LSTD is worth it depends on other parts of the problem as well - LSTD also doesn't require a step-size parameter, which is nice, but this parameter is replaced by ϵ, which can be finnicky to tune. No step-size parameter also means it never forgets, which is good in some problem settings, but problematic in settings like RL + GPI, where the target policy changes often. LSTD's full algorithm (compacted):

1. Initialize $\hat{A}^{-1} = \epsilon^{-1} I, b = 0$
2. For each episode, initialize a start state and loop for each step of the episode: take action A and observe R, S' pair.
3. Calculate an intermediary value: $v = \hat{A^{-1}}^T (x - \gamma x')$, $\hat{A}^{-1} = \hat{A}^{-1} - (\hat{A}^{-1} x) v^T / (1 + v^T x)$ (this is really just because it gets reused). Update the $\hat{b} = b + Rx$, and then update w with the fixed point equation.

Memory-Based Function Approx

So far, only parametric approaches to approximating value functions have been discussed, where the learning algorithm adjusts its parameters (of a functional form) over the problem's entire state space. Each update adjusts the parameters, and then the training example can be discarded.

Memory-based methods are different: they save training examples in memory as they arrive without updating any parameters. Then whenever a query's state value estimate is needed, a set of similar examples / states are retrieved from memory, and then a value estimate is computed. These are *nonparametric* methods, where the function form is determined by the training examples and not some class of functions. Classic example are k-means, k-nearest-neighbors with different weighting averages based on distances or other similarity metrics. *Locally weighted regression* is similar, fitting a surface to the values of the set (maybe a hyperplane or Gaussian distribution) by means of a parametric approximation method that minimizes a weighted error measure, again using distances as weights. The value returned is the evaluation of this local approximate surface at the query state. Memory-based methods allow an agent's experience to have an immediate affect on value estimates in the neighborhood of the current state, instead of having to incrementally adjust parameters of a global approximation. So updates can be must quicker and intentional.

Memory-based methods also address the curse of dimensionality, since for a k-dimensional space each example requires some memory proportional to k, not exponential. In order to improve search and example fetching latency in larger sets, special data structures like the k-d tree recursively split a k-dimensional space into regions arranged as nodes of a binary tree.

Kernel-Based Function Approximation

Kernel functions are a strict generalization of most of the methods presented in this chapter, and especially memory-based methods. They use a kernel function $k : S \times S \to \mathbb{R}$ to measure similarity between states. Kernel regression is a memory-based method that computes kernel weighted averages of the targets of all examples stored in memory, assigning results to query states. A common kernel is the Gaussian RBF, where the centers of the kernels are the examples in the dataset, with some type of parameter for widths.

Kernel trick: Popular trick in ML, where if we have some high d-dimensional feature vector $x(s)$, the linear parametric regression setting can be recast to the kernel regression setting by setting $k(s, s') = x(s)^T x(s')$. Performing kernel regression with this kernel function produces the same approximation that a linear parametric method would learn. This is a nice trick - that means not all, but some kernel function can be expressed as inner products of higher dimensional feature vectors. For these cases, instead of performing a complex, error-prone parametric linear approximation that works directly in the higher dimensions. By using only the set of stored training examples and the kernel function, we can achieve the same results at vastly lower levels of complexity.

Looking deeper at On-policy learning:

So far the algorithms have only treated the states encountered equally, as if they are all equally important, but in most cases some states are more interesting than others. In discounted episodic problems, earlier states may be more interesting than later states that are discounted away. If we treat all states encountered equally, we are updating the on-policy distribution which has stronger theoretical results. We can generalize this to many on-policy distributions, in the sense that they are all encountered in trajectories, but the trajectories vary in initialization.

We introduce a new, nonnegative scalar measure, a random variable I_t called interest, set in any causal way - could be an autoregressive variable or based on model parameters. The visited state distribution is defined as the distribution of states encountered while following the target policy, weighted by the interest for a given state-action pair at each timestep. Another value is the *emphasis* M_t, which multiplies the learning update:

$$w_{t+n} = w_{t+n-1} + \alpha M_t[G_{t:t+n} - \hat{v}(S_t, w_{t+n-1})]\nabla\hat{v}(S_t, w_{t+n-1}) \tag{183}$$
$$M_t = I_t + \gamma^n M_{t-n}, M_t = 0, t < 0 \tag{184}$$

Notice how the emphasis is an EMA of the interest. For the Monte Carlo case, $n = T - t$, and $M_t = I_t$, and we perform all recursive updates at the end of an episode.

Summary

This chapter emphasizes the need for powerful function approximation methods for policy evaluation / prediction, such as parameterized function approximation, where the $|w| < |\mathcal{S}|$. We use supervised learning methods + SGD alongside an update equation, treating each state as a training example. Algorithms like n-step semi-gradient TD are powerful approximators, but are not true gradient methods because the target itself relies on the weight vectors, but this is ignored when computing the gradient, which treats the target as a constant.

Linear function approximation is where semi-gradient methods perform well, and this requires choosing good feature spaces. Some options were the polynomial and Fourier bases, as well as using coarse and tile coding on the receptive fields. LSTD is a variant of TD learning that skips the iterative optimization and instead directly estimates the values for $w_{TD} = A^{-1}b$, but it requires a $O(w^2)$ computations because of outer products and iterative matrix updates.

Exercise 9.7: Single semilinear unit, with weight w and then a logistic nonlinearity, using the sigmoid function, so that

$$\hat{v}(x, w) = \frac{1}{1 + \exp(-w^T x)} \tag{185}$$
$$\nabla\hat{v}(x, w) = \frac{x}{(1 + \exp(-w^T x))^2} \tag{186}$$
$$w_{t+1} = w_t + \alpha[p - \hat{v}(x, w)] * \frac{x}{\hat{v}(x, w)^2} \tag{187}$$

Exercise 9.8 Using cross entropy loss:

$$w_{t+1} = w_t - \alpha\nabla[\log\hat{v}(S_t, w_t)p + (1-p)(1 - \log\hat{v}(S_t, w_t)] \tag{188}$$
$$= w_t - \alpha\frac{\nabla\hat{v}(S_t, w_t)}{\hat{v}(S_t, w_t)}[p - (1-p)] = w_t - \alpha\frac{x(s)}{\hat{v}(S_t, w_t)}[2p - 1] \tag{189}$$

On-policy Control with Approximation

This chapter extends the ideas of approximation to the control problem, approximating $\hat{q}(s, a, w) \approx q_*(s, a)$, still restricting to the on-policy case. In the episodic the case, the extension from policy evaluation to control is straightforward, but in the continuing case we need to re-examine how discounting is used to define optimal policies. The episode case thankfully doesn't need discounting.

Episodic Semi-gradient Control

Extension from chapter 9 is straightforward, the target is now just based off the current S_t, A_t, is any approximation of $q_\pi(S_t, A_t)$, like Monte Carlo or n-step Sarsa. Episodic semi-grdient one-step SARSA's update equation is then:

$$w_{t+1} = w_t + \alpha[U_t - \hat{q}(s_t, a_t, w_t)]\nabla \hat{q}(s_t, a_t, w_t) \tag{190}$$

We need to combine this policy prediction method now with some sort of policy improvement and action selection scheme. If the action set is discrete we can use previous policy improvement ideas and compute $A_{t+1} = \arg\max_a \hat{q}(S_{t+1}, a, w)$. Policy improvement is then performed by changing the policy to be an ϵ-greedy policy with respect to the current $\hat{q}$.

Semi-gradient N-step SARSA

This is just the n-step, episodic semi-gradient SARSA, with the n-step return as the target instead. The n-step return immediately generalizes from its tabular form to a function approximation form by just subbing in the function approximation symbol.

Exercise 10.1: Monte Carlo control methods with function approximation would mean that we simulate a trajectory, maybe ϵ-greedy with respect to Q, and then we compute backed up returns. Then for every visit to S_t, A_t, we set $Q(S_t, A_t)$'s target as the average of simulated returns and perform a gradient update. I think it's reasonable to not give pseudocode, because MC methods are inherently simulation based - they aren't optimizing for a loss function and making incremental updates, they're just performing averages and then setting the values. On the Mountain Car task, the MC methods would probably get stuck, because it needs to get to a terminal state in order to compute G_t, but that requires it to go to worse states (the left side) to accelerate enough to get to the right side. If we use an on-policy strategy the agent will fail to explore and probably get stuck in the valley - maybe using off-policy with an ϵ-greedy policy could work, but with very slow convergence.

Exercise 10.2: Same structure as semi-gradient SARSA, except that we compute an expected sum over the S_{t+1}, A_{t+1} to compute $\bar{V}(S_{t+1})$, and the target is now $R_{t+1} + \gamma \bar{V}(S_{t+1})$.

Exercise 10.3: Large N has more variance when considering updates, since your estimates between S, S' are always one step apart, or small steps apart for small n. For large n, one you update $\hat{q}(s_\tau, a_\tau)$, you use the estimate based on $\hat{q}(s_{\tau+n}, a_{\tau+n})$, hasn't been updated for a while and could be a very stale estimate. Also higher n-step methods generally have higher noise.

Average Reward: For Continuing Tasks

The average reward is a third classical setting, beyond episodic and discounted settings, for formulating the goal in MDPs. It applies to continuing problems, but there is no discounting. The average reward is denoted as:

$$r(\pi) = \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \tag{191}$$

$$= \lim_{t\to\infty} \mathbb{E}[R_t | S_0, A_{0:t-1} \sim \pi] \tag{192}$$

$$= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r \tag{193}$$

The second and third equations hold if the steady-state distribution $\mu_\pi(s) = \lim_{t\to\infty} Pr\{S_t = s | A_{0:t-1} \sim \pi\}$ exists and is independent of S_0, so if the MDP is ergodic. In an ergodic MDP, the expectation of being in a state depends only on the policy (the action taken from the previous state) and the transition probabilities. Ergodicity is sufficient but not necessary to guarantee existence of the limit. Steady-state distribution also has the property that:

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s) p(s'|s, a) = \mu_\pi(s') \tag{194}$$

Taking actions according to π from steady-state means you remain in the same distribution. In the average-reward setting, returns are defined in terms of differences between rewards and average rewards, known as the differential return:

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + ... \tag{195}$$

, with *differential* value functions. These new differential forms have similar Bellman equations and notation to the corresponding discounted settings, except that we remove all γs and subtract an average reward term:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r - r(\pi) + v_\pi(s')] \tag{196}$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r - r(\pi) + \sum_{a'} \pi(a'|s') q_\pi(s',a')] \tag{197}$$

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r - \max_\pi r(\pi) + v_*(s')] \tag{198}$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r - \max_\pi r(\pi) + \max_{a'} q_*(s',a')] \tag{199}$$

We put a $\max_\pi r(\pi)$ inside the optimal Bellman value equations because we are using the maximal policies, the ones that maximize the expected reward. We also have a differential form of the TD errors:

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t) \tag{200}$$

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t) \tag{201}$$

So in all of these analogous differential equations, we essentially swapped out the discounting factor for an estimate of the average reward. The average reward serves as a baseline for which the current reward measures against, when normally it would be discounting against the return estimates, i.e the future reward. The psuedocode for differential semi-gradient one-step SARSA:

1. Set up $\hat{q}$, and algorithm parameters $\alpha, \beta > 0, \epsilon > 0$.
2. For each episode, initialize S, A, and then take action A and observe R, S', choose A' as a ϵ-greedy policy of $\hat{q}(S', \cdot, w)$.
3. Calculate delta $\delta = R_{t+1} - \bar{R} + \hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w)$
4. Update $\bar{R} = \bar{R} + \beta\delta$, and do the appropriate gradient update with delta.

A limitation of this algorithm is that it does not converge to the differential values but equal up to an arbitrary offset.

Exercise 10.4: Pseudocode for Q-learning would follow mostly the same scheme as SARSA, except that in the target the delta equation becomes $\delta = R - \bar{R} + \max_{a'} \hat{q}(s', a', w) - \hat{q}(s, a, w)$. Also, Q-learning is off-policy - we wouldn't store do $A \leftarrow A'$, we would choose the current action A as an ϵ-greedy policy with respect to the most recently updated $\hat{q}$.

Exercise 10.5: The (10.10) equation is all you need - everything also can be brought over from the previous semi-gradient TD(0) algorithm, we are just updating the delta to be multiplied in the gradient update equation.

Exercise 10.6: The average reward is 1/2. The differential value of state A is:

$$\lim_{\gamma \to 1} \sum_{t=0}^{\infty} 1/2 + \gamma(-1/2) + \gamma^2(1/2) + \gamma^3(-1/2) + .. \tag{202}$$

$$= \lim_{\gamma \to 1} \frac{1}{2}(1 + \gamma^2 + \gamma^4 + ...) - \frac{1}{2}(\gamma + \gamma^3 + ...) \tag{203}$$

$$= \lim_{\gamma \to 1} \frac{1}{2(1-\gamma^2)} - \frac{\gamma}{2(1-\gamma^2)} = \lim_{\gamma \to 1} \frac{1}{2(1+\gamma)} = \frac{1}{4} \tag{204}$$

The differential value of state B is:

$$\lim_{\gamma \to 1} \sum_{t=0}^{\infty} -1/2 + \gamma(1/2) + \gamma^2(-1/2) + \gamma^3(1/2) + .. \tag{205}$$

$$= \lim_{\gamma \to 1} -\frac{1}{2}(1 + \gamma^2 + \gamma^4 + ...) + \frac{1}{2}(\gamma + \gamma^3 + ...) \tag{206}$$

$$= \lim_{\gamma \to 1} -\frac{1}{2(1-\gamma^2)} + \frac{\gamma}{2(1-\gamma^2)} = -\frac{1}{4} \tag{207}$$

Exercise 10.7: The average reward is 1/3.
Differential value of A is:

$$\lim_{\gamma\to 1}(0-1/3)+\gamma(0-1/3)+\gamma^2(1-1/3)+\gamma^3(0-1/3)+\gamma^4(0-1/3)+\gamma^5(1-1/3)+.. \tag{208}$$

$$=\lim_{\gamma\to 1}-\frac{1}{3}(1+\gamma^3+\gamma^6+..)-\frac{1}{3}(\gamma+\gamma^4+..)+\frac{2}{3}(\gamma^2+\gamma^5+..) \tag{209}$$

$$=\lim_{\gamma\to 1}-\frac{1}{3(1-\gamma^3)}-\frac{\gamma}{3(1-\gamma^3)}+\frac{2\gamma^2}{3(1-\gamma^3)} \tag{210}$$

$$=\lim_{\gamma\to 1}\frac{2\gamma^2-\gamma-1}{3(1+\gamma)(-\gamma^2+\gamma-1)} \tag{211}$$

Exercise 10.8: The sequence of errors, starting from state A would be $2/3, -1/3, -1/3, 2/3 - 1/3, -1/3$. The sequence of δ_t would be the same as the errors plus an offset $\hat{q}(S', A', w) - \hat{q}(S, A, w)$ for each state. If the estimate now changes in response to the errors, the second error sequence will be more stable, because the $\hat{q}$ will essentially learn to correct the difference of $R - \bar{R}$, per state. The first error sequence would continue to oscillate, even at the limit because the sequence does not converge to 0. OTOH, when closer to convergence the TD(0) error will converge to 0.

Deprecating the Discounted Setting

The continuing discounted problem formulation has been useful in the tabular case, where returns from each state can be separately identified and averaged. In the approximate case though, where returns can't be uniquely assigned, it is not appropriate. If we have an infinite sequence of returns with no beginning or end, and no clearly identified states. If all the feature vectors of the states are the same, we only have the reward sequence + actions to assess performance. If we try discounting, over a sufficiently long interval, it turns out the average of the discounted returns is proportional to the average reward. For policy π, the average of the discounted returns is $r(\pi)/(1-\gamma)$, so it is essentially the average reward, and the ordering of the policies in the average discounted return is same as the average reward. Thus the discounted rate has no effect on the problem setting.

Proof: If we start with the expected discounted value from state s: $J(\pi) = \sum_s \mu_\pi(s) v_\pi^\gamma(s)$, then we can write this out with the Bellman equation:

$$J(\pi)=\sum_s \mu_\pi(s)\sum_a \pi(a|s)\sum_{s',r}p(s',r|s,a)[r+\gamma v_\pi^\gamma(s')] \tag{212}$$

$$=r(\pi)+\gamma\sum_s \mu_\pi(s)\sum_a \pi(a|s)\sum_{s',r}p(s',r|s,a)v_\pi^\gamma(s') \tag{213}$$

$$=r(\pi)+\gamma\sum_{s'}v_\pi(s')\sum_s \mu_\pi(s)\sum_a \pi(a|s)p(s'|s,a) \tag{214}$$

$$=r(\pi)+\gamma\sum_{s'}v_\pi(s')\mu_\pi(s')=r_\pi+\gamma J(\pi)=r(\pi)+\gamma r(\pi)+\gamma^2 r(\pi)+\gamma^2 J(\pi) \tag{215}$$

$$=\frac{1}{1-\gamma}r(\pi) \tag{216}$$

On the first line we used the definition of the Bellman equation, then the definition of the average reward, then we reduced over the reward sum, and used the definition of a steady-state distribution, and then reduced everything again to get the recursive definition. So discounting has no effect, and actually becomes a solution method parameter instead of a problem setting parameter.

The root issue of the discounted control setting is that approximation loses the policy improvement theorem, where it we improve the policy on some discounted value the policy overall will improve. This is because the recursive equations that were central to the improvement theorem are valid in the tabular case, where at each time step you can identify the return associated with the current indices, and estimates are 'exact'. Now with generalization over states it isn't guaranteed that this association holds, since our number of parameters is much smaller than the number of states, so everything is approximations with error, so the inequalities from the tabular policy improvement theorem are NOT guaranteed.

Differential Semi-gradient n-step SARSA

We need to find a n-step version of the TD error to generalize the previous differential semi-SARSA to n-step. The generalization is straightforward, it is just subtracting $\bar{R}$ at each timestep and then adding the $\hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1})$ estimate at the end.

Exercise 2.7: If we use the given step size, then

$$Q_{n+1} = Q_n + \beta_n[R_n - Q_n] \tag{217}$$
$$= (1-\beta_n)Q_n + \beta_n R_n = (1-\beta_n)(Q_{n-1} + \beta_{n-1}(R_{n-1} - Q_{n-1})) + \beta_n R_n \tag{218}$$
$$= (1-\beta_n)(1-\beta_{n-1})Q_{n-1} + \beta_n R_n + (1-\beta_n)\beta_{n-1}R_{n-1} \tag{219}$$
$$= Q_1 \prod_{i=0}^{n-1}(1-\beta_{n-i}) + \sum_{i=0}^{n-1}[\prod_{j=0}^{i}(1-\beta_{n-j})]\beta_{n-i}R_{n-i} \tag{220}$$

If we look at the initial bias term of Q_1, as n approaches the limit the $\beta_n \to 1$, since the $\bar{o}_n \to \alpha$, and then $1 - \beta_n \to 0$, so that the initial bias washes out.

Exercise 10.9: If we now want to implement Exercise 2.7's proposed constant step size (with a trace of one denominator) to the average reward setting, we need to just initialize β as the $\alpha/\bar{o}_n$, with the $\bar{o}_n$ being a trace of alpha that starts at 0. Then the initial bias of δ_0 gets washed out in the limit.

Summary

This section introduces the different ways we use on-policy control methods alongside function approximation. We first introduce the total-episodic setting, then move to the undiscounted, continuing setting, where we introduce a new problem setting that uses the **average reward**. We now try to maximize this average reward per timestep, alongside new differential versions of the value functions, Bellman updates and TD errors. Ranking policies through this average reward metric allows us to find the maximizing policy. The discounting formulation cannot be carried over to the approximation setting because now, at a given state, it is ambiguous what next states / actions the policy can choose.

In the tabular setting, this is all enumerable and concrete, so when we 'rollout', or set up Bellman equations, all the reward and returns can be accurately traced to corresponding states + actions. In the approximation setting, these all become distributions, so discounting isn't as useful, and in fact is equivalent to just using the average reward over a sufficiently long time of updates.

Off-policy Methods with Approximation

The general challenges of off-policy learning can be divided into two parts - one that arises in the tabular case, and one that arises only in function approximation. The first part is how to deal with the target of the update, which is found under the behavior policy and needs to be appropriately weighted for the target policy. We dealt with this by using importance sampling, which can increase variance but is needed for convergence.

The second part is dealing with the distribution of the updates, which will be from the behavior policy and not the target policy. Semi-gradient methods critically rely on on-policy update distributions for convergence and stability - two ways to deal with this are importance sampling methods again that warp the update distribution to the on-policy, guaranteeing convergence in the linear case. The other is to develop (true, not semi) gradient methods that are independent of update distribution.

Semi-gradient Methods

The extension to (linear) semi-gradient methods in the off-policy setting is straightforward - we just include the per-step importance sampling ratio in the update equation, depending on the n-step. For the one-step state-value algorithm, which is semi-gradient off-policy TD(0), the update equation becomes:

$$w_{t+1} = w_t + \alpha\rho_t\delta_t\nabla\hat{v}(S_t, w_t), \tag{221}$$
$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t), \tag{222}$$
$$\delta_t = R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t) \tag{223}$$

For action-value updates, the one-step is semi-gradient expected SARSA:

$$w_{t+1} = w_t + \alpha\delta_t\nabla\hat{q}(S_t, A_t, w_t) \tag{224}$$

$$\delta_t = R_{t+1} + \gamma\sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \tag{225}$$

$$\delta_t = R_{t+1} - \bar{R} + \sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \tag{226}$$

In the multi-step generalizations of the algorithms, both state-value and action-value algorithms involve importance sampling. The n-step semi-gradient off-policy SARSA has ratios at each timestep, so that

$$w_{t+n} = w_{t+n-1} + \alpha\rho_{t+1}..\rho_{t+n}[G_{t:t+n} - \hat{q}(S_t, A_t, w_{t+n-1})]\nabla\hat{q}(S_t, A_t, w_{t+n-1}) \tag{227}$$

$$\text{episodic} : G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ..\gamma^{n-1}R_{t+n} + \gamma^n\hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1}) \tag{228}$$

$$\text{continuing} : G_{t:t+n} = R_{t+1} - \bar{R}_t + ...R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1}) \tag{229}$$

There is also an off-policy algorithm that doesn't involve importance sampling - the n-tree backup, which is a mix of both expected SARSA, and following a policy. The path of the policy traces out a tree, where we use action-value estimates for the actions NOT taken at each timestep, while tracing out the tree. The update equation is identical to n-step semi-gradient SARSA, albeit without a $\rho_{t+1}\cdot\cdot\rho_{t+n}$, but the return estimate is now:

$$G_{t:t+n} = \hat{q}(S_t, A_t, w_{t+n-1}) + \sum_{k=t}^{t+n-1}\delta_k[\prod_{i=t+1}^{k}\pi(A_i|S_i)], \tag{230}$$

$$\delta_k = R_{t+1} + \gamma\sum_a \pi(a|S_{t+1})\hat{q}(S_{t+1}, a, \hat{w}_{t+n-1}) - \hat{q}(S_t, A_t, w_{t+n-1}) \tag{231}$$

Look at Exercise 7.11 for the derivation, but for a refresher we can write out a general recursive form:

$$G_{t:t+n} = R_{t+1} + \sum_{a\neq A_{t+1}} \pi(a|S_{t+1})q(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})G_{t+1:t+n} \tag{232}$$

We then use the camel trick to include the missing A_{t+1} in the sum, convert it to the δ_t term in expected SARSA.

Exercise 11.1: N-step TD is the state-value update equations, using the TD errors. For off-policy we need to weight the actions **starting** from time t, A_t, and ending at $A_{t+n-1} \rightarrow S_t$. So the ratios range from $\rho_t, ..\rho_{t+n-1}$. The update equation is then:

$$w_{t+n} = w_{t+n-1} + \alpha\delta_t\rho_t\cdot\cdot\rho_{t+n-1}\nabla\hat{v}(S_{t+n}, w_{t+n-1}) \tag{233}$$

$$\text{episodic} : \delta_t = R_{t+1} + \gamma R_{t+2} + ...\gamma^{n-1}R_{t+n} + \gamma^n\hat{v}(S_{t+n}, w_{t+n-1}) \tag{234}$$

$$\text{continuing} : R_{t+1} - \bar{R}_t + ...R_{t+n} - \bar{R}_{t+n-1} + \hat{v}(S_{t+n}, w_{t+n-1}) \tag{235}$$

Exercise 11.2: Because n-step $Q(\sigma)$ already accounts for ρ_t per timestep where needed inside the return estimate, the semi-gradient update is just:

$$w_{t+n} = w_{t+n-1} + \alpha\delta_t\nabla\hat{q}(S_t, A_t, w_{t+n-1}) \tag{236}$$

The return estimates for the episodic setting is the same at 7.17, expect we substitute $Q_{h-1}, \bar{V}_{h-1}$ with the function approximation versions. For the continuing case we do the same + remove discounts and use the average reward instead.

Examples of Policy Divergence

The more important issue with off-policy function approximation is that the update distribution does not match the on-policy distribution. This part consists of a bunch of simple examples showing off-policy's flaws.

Ex: State with estimated value w that only has one transition to another state with estimated value $2w$, reward 0. With off-policy semi-gradient TD(0), update equation becomes: $w_{t+1} = w_t + \alpha * 1 * (2\gamma - 1) * w_t * 1 = (1 + \alpha(2\gamma - 1))w_t$. This sequence diverges when $2\gamma - 1 > 0$, so that the multiplication term is greater than 1. Even though this two-state example is embedded in a smaller MDP, the key is that this one transition is the only one occuring without w being updated on other transitions. This is possible in off-policy transition from the behavior policy's coverage - $\rho_t = 1$ for only this state, and 0 everywhere else. Instead, for on-policy training $\rho_t = 1$ everywhere, so after leaving this transition, w continues to be updated from future states, and this value would only correctly continue to increase if there were further states with even higher values (plus discounting too).

For a complete system example of divergence, we can look at Baird's counterexample. It consists of six top states and a seventh state, with six dashed actions coming out of the seventh state to the other six states, and then 7 solid actions, going from all of the seven states to the seventh states. The b's policy distribution is a uniform state distribution, while π will only take the solid action, always staying in state 7. This is a favorable setting for linear function approximation, since the $\forall s, v_\pi(s) = 0$, so $\mathbf{w} = 0$ is an exact solution, and the feature vectors are a linearly independent set. Doing semi-gradient TD(0) and DP both diverge, but after setting the update distribution to π, it immediately converges.

The Deadly Triad

The danger of instability and divergence arises from the **deadly triad**, which is a combination of these three elements:

1. Function approximation
2. Boostrapping
3. Off-policy training

This deadly triad is actually independent of control or GPI, and arises even in the simpler policy prediction case. Furthermore, it is not due to learning or environment dynamics, since this also occurs in planning methods like off-policy semi-DP. We can go through the three and see their importance. Function approximation is the most important, since some problems are just not feasible to solve in the tabular case. Bootstrapping offers great computational and data efficiency, since we don't need to calculate returns after the end of an episode, and we can use smaller n-step values and still get similar results to MC.

Off-policy learning may not seem important now, but in the future will be very important, and basically required. In future situations where the agent learns not just a single value function/policy but thousands of them in parallel, there will be many target policies but only one behavior policy. Off-policy learning is the method through which agents can parse a single stream of experience and mold it to an arbitrary, learning target policy, since the behavior policy, like a human's experience overlaps with many of the different sub target policies we learn.

Linear Value-function Geometry

For the following discussion, treat a value function $v : S \to \mathbb{R}$ equivalently to its vector representation, which is a vector listing the values of the enumeration of all possible states in the state space. Normally this is infeasible to list out, but to develop intuitions consider the case with three states $S = \{s_1, s_2, s_3\}$ and two parameters $w = (w_1, w_2)^T$. All value-function vectors are points in the three-dimensional space, and the parameters are an alternative coordinate system over a two-dimensional subspace. In the case of linear value-function approximation, the subspace is a simple plane.

If we imagine a fixed policy π, and assume that its true value function v_π is too complex to be represented exactly as an approximation, so it sits above the w plane. How do we get the closest representable value function to it? There are multiple answers. Before tackling this, we need a measure of the distance between two value functions, so given two value functions v_1, v_2, the vector difference between them is $v = v_1 - v_2$. To measure the size of this difference vector, we can't take the Euclidean norm, since some states are more important and appear more often. We can instead use the distribution function $\mu : S \to [0, 1]$ (often the on-policy distribution) to define a new distance:

$$||v||_\mu = \sum_{s \in \mathcal{S}} \mu(s) v(s)^2 \tag{237}$$

Now for any value function v_π, we find its closest representable value function through a projection operation, where a projection operator Π minimizes the norm:

$$\Pi v = v_w, w = \arg \min_{w \in \mathbb{R}^d} ||v - v_w||_\mu^2 \tag{238}$$

This projection Πv_π is the solution asymptotically found by Monte Carlo methods, very slowly. Because we are using linear function approximators, the projection operation is linear, and is in fact a $|S| \times |S|$ matrix, which can be written as :

$$\Pi = X(X^T D X)^{-1} X^T D = A(A^T A)^{-1} A^T \tag{239}$$

X denotes the $|S| \times d$ matrix whose rows are the feature vectors $x(s)^T$, and D denotes the $|S| \times |S|$ diagonal matrix with the $\mu(s)$ on the diagonal. Using these matrices, we rewrite the squared norm of our vector in this space as $||v||_\mu = v^T D v$, and the approximate linear value function as $v_w = Xw$.

TD methods find different solutions in the linear plane. The true value function v_π is the only value function that solves the Bellman equation exactly - if an approximate value function v_w were substituted for v_π, then the difference between the right and left sides is a measure of how far v_π and v_w are, the Bellman error:

$$\delta_w(s) = (\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_w(s')] - v_\pi(s) \tag{240}$$

$$= \mathbb{E}[R_{t+1} + \gamma v_w(S_{t+1}) - v_w(S_t)|S_t = s, A_t \sim \pi] \tag{241}$$

Then the Bellman error is the expectation of the TD error. The reason this gives a difference between v_π, v_w is the RHS of the bellman equation with v_w subbed in gives an one-step approximation of $v_\pi(s)$ using $v_w(s')$, which is then compared with the current $v_w(s)$. The vector of all the Bellman errors at all states is the Bellman error vector $||\bar{\delta}_w||^2_\mu$, and the overall size of the vector, through the μ-norm, is the overall measure of the error from the value function, the mean square Bellman error: $B\bar{E}(w) = ||\bar{\delta}_w||^2_\mu$. It is not possible in general to reduce the mean square Bellman error to 0, but there is a unique minimum of w (linear).

The Bellman error vector is actually a result of applying the Bellman operator, $B_\pi : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$, defined by:

$$(B_\pi v)(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')] \tag{242}$$

So applying this on a vector representation of a value function makes $B_\pi v$ follow that equation now. The Bellman error vector for v_w is : $\bar{\delta}_w = B_\pi v_w - v_w$. When the Bellman operator is applied on a value function in the subspace, it will produce a new value function that is outside the subspace. In dynamic programming (no FA), the operator is applied repeatedly to points outside the plane, converging to the fixed point v_π.

If we instead use function approximation, after each iteration we need to project back onto the representable subspace. This is equivalent to performing a DP-like process with approximation, like TD(0), which is one-step DP with bootstrapping. In the case of projection we are interested in the projection of the Bellman error vector onto the representable space, $\Pi\bar{\delta}_w$. The size of this error vector is another measure of error in the approximate value function, called the mean square Projected Bellman error, $P\bar{B}E$. Notice that using the PBE vector and adding to our previous estimate gives:

$$v_\pi + \Pi(B_\pi v_w - v_w) = v_w + \Pi B_\pi v_w - v_w = \Pi B_\pi v_w \tag{243}$$

So the next iteration is $\Pi B_\pi v_w$. In linear FA there always exists a value function with zero PBE, the TD fixed point w_{TD} - this point is not always stable under semi-gradient and off-policy methods, and usually has a value function different from minimizing VE (projection of v_π) or BE (direct minimization of the Bellman error vector).

Gradient Descent + Bellman Error

The success of SGD has caused a lot of interest in finding good, descriptive objective functions to optimize. This section details the objective function based on the Bellman error. Before considering Bellman error, we can take a simpler cousin, the expected square of the TD error. The one-step TD error is: $\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)$, and a possible objective function is the mean square TD error:

$$T\bar{D}E(w) = xt\sum_{s\in\mathcal{S}} \mu(s)\mathbb{E}[\delta_t^2|S_t = s, A_t \sim \pi] \tag{244}$$

$$= \sum_{s\in\mathcal{S}} \mu(s)\mathbb{E}[\rho_t\delta_t^2|S_t = s, A_t \sim b] = \mathbb{E}_b[\rho_t\delta_t^2] \tag{245}$$

The last equation is needed for SGD since it can only sample and optimize based on experience (which is the behavior policy). If we substitute this in the SGD equation now, we get:

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla(\rho_t\delta_t^2) \tag{246}$$

$$= w_t - \alpha\rho_t\delta_t\nabla\delta_t = w_t + \alpha\rho_t\delta_t(\nabla\hat{v}(S_t, w_t) - \gamma\nabla\hat{v}(S_{t+1}, w_t)) \tag{247}$$

This update equation has convergence guarantees and is called the naive residual-gradient algorithm, which converges robustly but not necessarily to a desirable place, some simple MRP examples in the book show that the TDE minimizing solution does not match with the true value. We can now try extending to the mean square Bellman error, i.e the expectation of the TD error (equal in the deterministic setting). If the exact values are learned, the Bellman error is 0 everywhere, so this seems like a good objective function. Zero Bellman error is not necessarily achievable because in most interesting settings the true value function is outside of the representable plane. The update equation is:

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_\pi[\delta_t]^2) \tag{248}$$

$$= w_t - \frac{1}{2}\alpha\nabla(\mathbb{E}_b[\rho_t\delta_t]^2) \tag{249}$$

$$= w_t - \alpha\mathbb{E}_b[\rho_t\delta_t]\nabla\mathbb{E}_b[\rho_t\delta_t] \tag{250}$$

$$= w_t + \alpha(\mathbb{E}_b[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1}, w)] - \hat{v}(S_t, w))(\nabla\hat{v}(S_t, w) - \gamma\mathbb{E}_b[\rho_t\nabla\hat{v}(S_{t+1}, w)] \tag{251}$$

Notice that the $v(S_t, w)$ on both the gradient and value can be taken out since it is a constant with respect to the expectation. This is the (non-naive) *residual-gradient* algorithm. If we just the same sample values in all the expectation, the equation reduces to the naive residual-gradient algorithm - we need to take independent samples of S_{t+1} in order to maintain an unbiased sample. This is really only possible in deterministic environments, where only S_{t+1} is available and the naive is valid, or in planning methods, where we use simulated experience.

In the linear case, the residual-gradient algorithm always converges to the unique w minimizing the mean square Bellman error. However, the convergence is empirically slow, and it still converges to the wrong values in some toy experiments. There is another issue with the Bellman error in the next section.

The Bellman Error is not learnable

Learnability is the concept that some hypothesis / function can be learned within an infinite amount of experiential data. Many quantities / metrics of interest in RL are not learnable - these quantities are well defined and can be computed from the internal structure of the environment, but not computed/estimated from the observed sequence of feature vectors/actions/rewards. Another way to interpret this is given two infinite stream of identical data distributions, unlearnable quantities from identical data streams but different environment settings cannot be distinguished. A simple two state example in the book shows that the mean square value and Bellman errors are not learnable, i.e both of them are not unique functions of the data distribution, so how could they be useful objectives to learn, since we don't even know what data distribution we are optimizing for?

Thankfully, the parameter that optimizes the VE is learnable, i.e it is unique to its data distribution. One error that is always learnable is the mean square return error, given by:

Exercise 11.4

$$RE(w) = \mathbb{E}[(G_t - \hat{v}(S_t, w)^2] = \sum_{s\in\mathcal{S}}(G_t - \hat{v}(s, w)^2\mu(s) \tag{252}$$

$$= \sum_{s\in\mathcal{S}}(G_t - v_\pi(s) + v_\pi(s) - \hat{v}(s, w)^2\mu(s) \tag{253}$$

$$= \mathbb{E}[(G_t - v_\pi(S_t)^2] + \mathbb{E}[(v_\pi(s) - \hat{v}(s, w))^2] + 2\mathbb{E}[(G_t - v_\pi(s))(v_\pi(s) - \hat{v}(s, w)] \tag{254}$$

$$= C + VE(w) + 2\mathbb{E}[G_t v_\pi(s) - v_\pi(s)^2 - G_t\hat{v}(s, w) + v_\pi(s)\hat{v}(s, w)] \tag{255}$$

In the limit of the expectation, $\mathbb{E}[v_\pi(s)] = G_t$, since this is always the true value, and then the last term goes to 0. The mean square return error and the VE(w) are the same except for a nonparameterized value, so they share the same optimal parameter w^*. The BE(w) doesn't even have a learnable optimal parameter, but the other bootstrapping objectives, like the PBE and TBE are learnable and so are their optimal solutions.

Because the BE is not learnable, it cannot be estimated from feature vectors, and is limited to model-based settings, where the actual state vectors are available. The only algorithms that can minimize the BE are ones with access to the underlying MDP states, and the residual-gradient method is able to minimize BE because it can double sample from the same state.

Gradient-TD Methods

Recall that the PBE (projected bellman error) method is equivalent to just tracing out projections of the Bellman error matrix on the representable function space.

In the linear case of PBE there is always an exact solution, the TD fixed point w_{TD}, where the PBE is zero. Chapter 9 showed the online least-squares TD method, which had $O(d^2)$ memory complexity. We seek a SGD method with $O(d)$ memory complexity, with robust convergence properties. Gradient-TD methods come close, at the cost of a rough doubling of computational complexity. We begin by deriving the PBE in matrix terms:

$$PBE(w) = ||\Pi\bar{\delta}_w||_\mu^2 \tag{256}$$

$$= (\Pi\bar{\delta}_w)^T D\Pi\bar{\delta}_w = \delta_w^T \Pi^T D\Pi\delta_w \tag{257}$$

$$\Pi^T D\Pi = DX(X^TDX)^{-T}X^TDX(X^TDX)^{-1}X^TD = DX(X^TDX)^{-1}X^TD \tag{258}$$

$$\implies \delta_w^T DX(X^TDX)^{-1}X^TD\delta_w = (X^TD\delta_w)^T(X^TDX)^{-1}(X^TD\delta_w) \tag{259}$$

$$\nabla PBE(w) = 2\nabla[X^TD\bar{\delta}_w]^T(X^TDX)^{-1}(X^TD\delta_w) \tag{260}$$

In order for this to be SGD, we need to sample a value whose expectation is $\nabla PBE(w)$. If we take the behavior policy distribution as μ, then :

$$X^TD\delta_w = \sum_s \mu(s)x(s)\delta_w(s) = \mathbb{E}_b[\delta_t x_t] = \mathbb{E}_\pi[\rho_t\delta_t x_t] \tag{261}$$

$$\nabla[X^TD\delta_w]^T = \nabla\mathbb{E}[\rho_t\delta_t x_t]^T = \mathbb{E}[\rho_t\nabla\delta_t^T x_t^T] \tag{262}$$

$$= \mathbb{E}[\rho_t\nabla(R_{t+1} + \gamma w^T x_{t+1} - w^T x_t)^T x_t^T] = \mathbb{E}[\rho_t(\gamma x_{t+1} - x_t)x_t^T] \tag{263}$$

$$X^TDX = \sum_s \mu(s)x(s)x(s)^T = \mathbb{E}[x_t x_t^T] \tag{264}$$

If we now substitute these into our large equation:

$$\mathbb{E}[\rho_t(\gamma x_{t+1} - x_t)x_t^T](\mathbb{E}[x_t x_t^T])^{-1}\mathbb{E}_\pi[\rho_t\delta_t x_t] \tag{265}$$

We now have three separate expectations to compute - there are several ways of doing this to try to keep it computationally inexpensive. The main idea is to estimate two of the three expectations separately and store them, and then sample the third in SGD to produce the final unbiased estimate. Gradient-TD methods implement this, by estimating storing the **product** of the last two factors, since the product is a d-vector, cheaper than storing a matrix. The value we want to learn is $v \approx \mathbb{E}[x_t x_t^T]^{-1}\mathbb{E}[\rho_t\delta_t x_t]$, which is actually the solution to linear regression, least-squares, where we are learning $\rho_t\delta_t$ from the feature vectors.

The standard SGD method is to minimize the projection error $(v^T x - \rho_t\delta_t)^2$, with an update rule that is also augmented by an importance ratio, since we are still sampling from b:

$$v_{t+1} = v_t + \beta\rho_t(\delta_t - v_t^T x_t)x_t \tag{266}$$

After we have estimates of the product vector v_t, we can use this estimate to perform the actual learning process. There are two variants of the learning updates called GTD2 and TD(0) with gradient correction (TDC). Both of these involve two learning processes, a primary one for w and a secondary for v, where the secondary is assumed to move much faster. The logic of the primary learning process also depends on the assumption that the secondary is much faster and is around asymptotic value - this dependence is called a *cascade*, and these two process convergence proofs are called *two-time scale proofs*, with conditions on the step sizes like $\beta \to 0, \alpha/\beta \to 0$.

We can derive out GTD first:

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla PBE(w) \tag{267}$$

$$= w_t - \alpha\mathbb{E}[\rho_t(\gamma x_{t+1} - x_t)x_t^T]\mathbb{E}[x_t x_t^T]^{-1}\mathbb{E}[\rho_t\delta_t x_t] \tag{268}$$

$$= w_t + \alpha\mathbb{E}[\rho_t(x_t - \gamma x_{t+1})x_t^T]v_t \tag{269}$$

$$\approx w_t + \alpha\rho_t(x_t - \gamma x_{t+1})x_t^T v_t \tag{270}$$

The last step is the sampling step - if we perform the end dot product first, then this entire algorithm has $O(d)$ complexity in any memory movement / storage.

TDC is GTD but performing a few more analytic steps before substituting in v:

$$w_{t+1} = w_t - \alpha \mathbb{E}[\rho_t(\gamma x_{t+1} - x_t)x_t^T]\mathbb{E}[x_t x_t^T]^{-1}\mathbb{E}[\rho_t \delta_t x_t] \tag{271}$$
$$= w_t + \alpha(\mathbb{E}[\rho_t x_t x_t^T] - \gamma \mathbb{E}[\rho_t x_{t+1} x_t^T])\mathbb{E}[x_t x_t^T]^{-1}\mathbb{E}[\rho_t \delta_t x_t] \tag{272}$$
$$= w_t + \alpha(\mathbb{E}[\rho_t \delta_t x_t] - \gamma \mathbb{E}[\rho_t x_{t+1} x_t^T]\mathbb{E}[x_t x_t^T]^{-1}\mathbb{E}[\rho_t \delta_t x_t]) \tag{273}$$
$$= w_t + \alpha(\mathbb{E}[\rho_t \delta_t x_t] - \gamma \mathbb{E}[\rho_t x_{t+1} x_t^T]v_t) \tag{274}$$
$$\approx w_t + \alpha \rho_t(\delta_t x_t - \gamma x_{t+1} x_t^T v) \tag{275}$$

This algorithm is also $O(d)$ memory complexity if the dot product is done first.

Emphatic-TD Methods

Emphatic TD methods are the second major strategy used to obtain a cheap and efficient off-policy method with function approximation. Linear semi-gradient TD methods require on-policy update distributions for efficient and stable updates, which is related to the positive definiteness of the matrix A, i.e the solution to $w_{TD} = A^{-1}b$, as well as on-policy sampling. In off-policy learning, we reweight state transitions using importance sampling, yet the state distribution is the behavior policy's.

In order to amend the mismatch, we recall ideas from Chapter 9 about interest and emphasis of certain states - if we reweight the states, emphasizing some and de-emphasizing others, the distributions of updates will return to the on-policy distribution. There are actually many on-policy distributions, and any of them is sufficient to guarantee stability. For an undiscounted episodic problem, if the episode's state transitions are all due to the target policy, then the state distribution is the on-policy distribution.

In the discounting setting, we use the idea of psuedo-termination - termination that does not affect the sequence of state transitions, but affects the learning process. Specifically, in a continuing discounted task with $\gamma = 0.9$, we consider that with probability 0.1 the process terminates on every time step and then immediately restarts in the new state.

The one-step Emphatic-TD algorithm is given by:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t) \tag{276}$$
$$w_{t+1} = w_t + \alpha M_t \rho_t \delta_t \nabla \hat{v}(S_t, w_t) \tag{277}$$
$$M_t = \gamma \rho_{t-1} M_{t-1} + I_t \tag{278}$$

Notice that each update is weighted by the current emphasis, which itself is an EMA of the current interest at time t and the discounting factor * importance ratio.

Reducing Variance

Off-policy learning is significantly higher variance than on-policy, due to the distribution mismatch between target and behavior policies. The key motivation of off-policy learning then, is to utilize this variance to enable generalization to a vast number of closely-related (in states visited and actions chosen), but not identical policies in this policy space.

Controlling variance is especially important in off-policy importance sampling methods. Although the expectations of ρ_t are 1, their actual values vary wildly, and successive ratios are uncorrelated, so their running products still expect to 1 yet their variances compound. These ratios multiply onto the step size in SGD, which is problematic, because of the occasional large step, which hurts SGD convergence. Setting the step size small to prevent this can lead to extremely slow learning as well.

Some methods to combat this are using momentum as a first-order adaptive mechanism, Polyak-Ruppport averaging as another form of 'remembering' the step-size / ratios, or creating more fine-grained step size settings for different components of the parameter vector. Another complementary strategy is to codesign the target/behavior policies to never be too dissimilar and create large importance sampling ratios.

Summary

This chapter introduces off-policy learning and the nuanced complexities that make it much harder in function approximation than tabular. The very first section addressed the first issue of setting the correct targets for gradient learning, by including appropriate importance sampling ratios. The rest of the chapter addresses the deadly triad - how combining bootstrapping, function approximation and off-policy learning leads to extremely instable algorithms.

Some proposed fixes to the deadly triad were to perform SGD directly on the Bellman error, but this shows to be ineffective since the resulting minimum is not always correct, and the gradient of the mean Bellman error itself is not learnable. Gradient-TD methods then perform SGD on the projected Bellman error, which has a learnable minimum and gradient. This converges, but extremely slowly and at the cost of a second parameter and learning process.

Eligibility Traces

We have actually already seen eligibility traces throughout the book - they are a basic mechanism of RL. Any TD method, like Q-learning/SARSA, can be combined with eligibility traces to obtain a more general method. They also unify and generalize TD and MC methods, with MC methods ($\lambda = 1$) at one end, and TD ($\lambda = 0$) at the other.

The basic mechanism of traces is that there is a short term memory vector z_t that mirrors the long-term weight vector - when a component of w_t participates in estimating a value, the corresponding component of z_t is increased and begins to decay. Learning then occurs in that component of $w_t[i]$ if a nonzero TD error occurs before $z_t[i]$ falls to zero, and the trace decay parameter $\lambda \in [0, 1]$ decides this.

Utilizing eligibility traces shows the dual side of implementing RL algorithms. *Forward view* formulations of updates use the next n rewards and n steps in the future to update the current state - we can often achieve nearly the same updates with an algorithm using the current TD error and the eligibility trace to distribute update weightings.

λ-return

Chapter 7 defined the n-step return to go alongside the n-step update algorithms, which are the first n rewards from the timestep + the estimate, all with appropriate discounting. We now introduce *compound updates*, which average different n-step target returns, like $C = \frac{1}{2}G_{t:t+3} + \frac{1}{2}G_{t:t+5}$. The target for the compound update is valid as long as all the weights are positive and sum to 1. Note that the length of calculation for the target of the compound update always depends on the longest length, so it should be controlled accordingly. The TD(λ) algorithm uses a compound update as its target:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{279}$$

Note this is an infinite sum, so we put a correction multiplicand in front. In the case of $\lambda = 0$, this simplifies to TD(0), and when $\lambda = 1$ this is the Monte Carlo update.