Exercise 6.1 If V changes during the episode, then (6.6) only holds approximately; what would the difference be between the two sides? Let V_t denote the array of state values used at time t in the TD error (6.5) and in the TD update (6.2). Redo the derivation above to determine the additional amount that must be added to the sum of TD errors in order to equal the Monte Carlo error.

Update $V_t \to V_{t+1}$ during timestep t in the episode:

$$V_{t+1}(S_t) = V_t(S_t) + \alpha \delta_t \qquad \delta_t = \left[R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)\right]$$

Monte-Carlo error evaluation:

$$G_t - V_t(S_t) = \left[R_{t+1} + \gamma V_t(S_{t+1}) - V_t(S_t)\right] + \gamma\left[G_{t+1} - V_t(S_{t+1})\right] = \delta_t + \gamma\left[G_{t+1} - V_t(S_{t+1})\right]$$

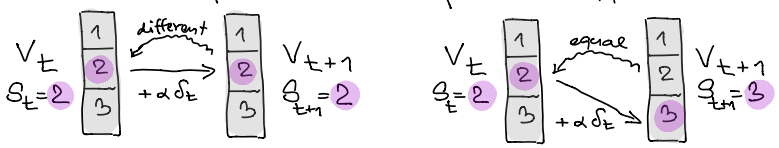
The problem now is the purple term. Ideally we'd want it to be equal to $V_{t+1}(S_{t+1})$. This is generally the case except when S_{t+1} is the same as S_t.

Then these two differ by $\alpha\delta_t$:

$$V_t(S_{t+1}) = V_{t+1}(S_{t+1}) - \alpha\delta_t \cdot \mathbb{1}(S_{t+1} = S_t)$$

← Indicator function whether two subsequent states are equal

This also makes sense intuitively. Following are diagrams of array V during subsequent timesteps for $S_{t+1} = S_t$ and $S_{t+1} \neq S_t$



So we finally have:

$$G_t - V_t(S_t) = \delta_t + \gamma\,\alpha\delta_t \cdot \mathbb{1}(S_{t+1} = S_t) + \gamma\left[G_{t+1} - V_{t+1}(S_{t+1})\right] =$$

$$= \underbrace{\delta_{t+1} + \gamma\alpha\delta_{t+1}\cdot\mathbb{1}(S_{t+2} = S_{t+1}) + \gamma\left[G_{t+2} - V_{t+1}(S_{t+2})\right]}_{\ldots}$$

$$= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k + \alpha\gamma\sum_{k=t}^{T-1}\gamma^{k-t}\delta_k \cdot \mathbb{1}(S_{k+1} = S_t)$$

The solution is equal to the fixed V one presented in the book when S_t is visited only once during the episode. Whenever S_t is visited the MC error is different because of the online updates done to V.

Exercise 6.2 This is an exercise to help develop your intuition about why TD methods are often more efficient than Monte Carlo methods. Consider the driving home example and how it is addressed by TD and Monte Carlo methods. Can you imagine a scenario in which a TD update would be better on average than a Monte Carlo update? Give an example scenario—a description of past experience and a current state—in which you would expect the TD update to be better. Here's a hint: Suppose you have lots of experience driving home from work. Then you move to a new building and a new parking lot (but you still enter the highway at the same place). Now you are starting to learn predictions for the new building. Can you see why TD updates are likely to be much better, at least initially, in this case? Might the same sort of thing happen in the original scenario?

When commuting for the first time neither method has a good estimate how long it takes to reach the parking lot and how long it takes to drive from the parking lot to the highway because these states were never visited.

During the first drive, TD update will start to give meaningful updates immediately after entering the highway. It can use the commute times obtained from previous drives and incorporate the "neighbor" experience into estimates for the new states. MC update is done only after the end of the commute. There are no updates along the way even though a significant portion of the driving is done in a "familiar territory" after entering the highway. MC update is completely independent of any previous experience and is simply equal to the actual travel time from the new states to home.

Over time as we keep driving from our new office location, TD updates will back-propagate the travel times obtained from previous commutes to the old office. Back-propagation uses accurate and nearly noise free estimates already obtained during the past year of driving on the highway to home segment. Reuse of these, already converged values, is much less random because the events like traffic jams or slow trucks are already accounted for in the values obtained in past experience and don't have to averaged again.

MC updates are collected only by averaging the travel times and will be initially much noisier. Source of the noise is the rest of the commute and all the the unusual events that can happen like the jam or the slow truck.

This situation can't happen in the original scenario because there's virtually no exploration. Both methods are learning from scratch, repeatedly going through the same sequence of states. Leveraging value of "neighbor" states in case of the TD update will not

Exercise 6.3 From the results shown in the left graph of the random walk example it appears that the first episode results in a change in only $V(\mathsf{A})$. What does this tell you about what happened on the first episode? Why was only the estimate for this one state changed? By exactly how much was it changed?

The episode must have terminated via state A. The value update for the state is (all states are initialized to 0.5):

$$V_A \leftarrow V_A + \alpha\left[R + \gamma V_T - V_A\right] = 0.5 - 0.1 \cdot \left[0 + 1.0 - 0.5\right] = 0.45$$

All the other states that were visited (could have been all except E, B and C were visited for sure) also got updated but the zero reward and identical initialization caused the TD error update to be trivial. For any state transition $X \to Y$:

$$V_X \leftarrow V_X + \alpha\left[R + \gamma V_Y - V_X\right] = 0.5 + 0.1\left[0 + 1 \cdot 0.5 - 0.5\right] = 0.5$$

Exercise 6.4 The specific results shown in the right graph of the random walk example are dependent on the value of the step-size parameter, α. Do you think the conclusions about which algorithm is better would be affected if a wider range of α values were used? Is there a different, fixed value of α at which either algorithm would have performed significantly better than shown? Why or why not?

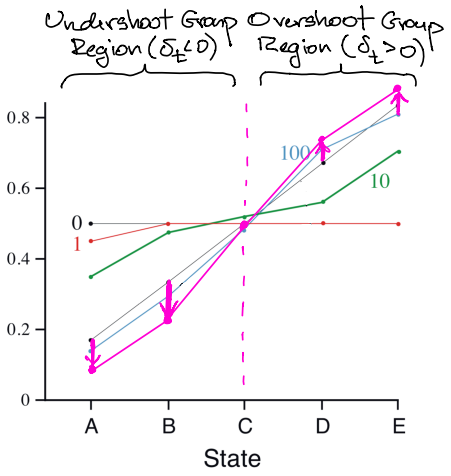
Wider range of fixed alphas wouldn't change the conclusions.

If the range would extend to higher alphas, the results would suffer from noise later in the learning process. Higher alphas generally achieve faster convergence in the initial learning phase but later on transition into a "noisy mode". The "noisy mode" happens when the algorithm is close to the optimal answer but the alpha is so large that it allows the noise from a few subsequent episode outcomes change the value of the visited states too much. The error of recently visited states jumps from positive to a negative value (or vice versa) without reducing the overall RMS. Magnitude of the "noisy mode" updates grows with alpha. Extremely high alpha value completely prevents learning of a meaningful value function because the updates to value function are so large that the noise caused by outcome of the last few episodes dominates over the average obtained from the past experience.

On the opposite side of the range, small alpha values make both algorithms converge slowly during the initial phase. They eventually reach a closer approximation of the optimal value function but small updates cause that this process takes very long time. Therefore it can't be said that lower alphas perform strictly better than larger ones. The "noisy mode" is essentially made small enough by taking a long enough time transition into it but it never completely disappears. Again, at the very extreme when alpha -> 0 no learning happens as the

**Exercise 6.5* In the right graph of the random walk example, the RMS error of the TD method seems to go down and then up again, particularly at high α's. What could have caused this? Do you think this always occurs, or might it be a function of how the approximate value function was initialized?

High alpha values initially converge quickly and perform large updates. This together with the constant initialization state values causes a natural tendency to perform a group under/over-shoot at many states close to each other at the same time. Group under/over-shooting in this particular case means simultaneously estimating smaller than optimal values of states A and B and higher than optimal values of states D and E.



After the under/over-shoot, the algorithm performs a correction in the opposite direction but for TD methods this is harder to do since the updates of the group of states are correlated to each other because they are spatially close. This coupling creates something similar to a momentum or a crowd psychology within the group that resits the correct update after under/over-shooting.

Root cause of the group under/over-shoot effect is identical initialization of all states. Updates then form two co-located groups of states. One with positive updates with optimal values above the initialization value and the second one that receives predominantly negative updates because their optimums are above the initialization value. Initially this effect causes faster convergence as the members of each group encourage positive/negative updates to their other members but it also causes the eventual RMS error rise as the under/over-shoot needs to be eventually reverted.

The effect can be prevented by using random initialization values. This will break the locality of

Exercise 6.6 In Example 6.2 we stated that the true values for the random walk example are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$, for states A through E. Describe at least two different ways that these could have been computed. Which would you guess we actually used? Why? ☑

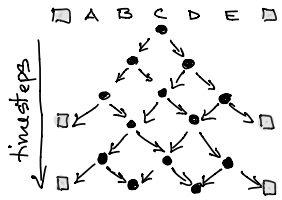
A) Solve the system of Bellman equations that turns out to be a system of linear equations because there's only one action:

$$V(s) = \max_a \sum_{\forall s'} p(s'|s,a)\big(R + \gamma V(s')\big) = \sum_{\forall s'} p(s'|s,a)\big(R + \gamma V(s')\big)$$

Which translates into the following linear algebra problem:

$$\begin{bmatrix} -1 & 1/2 & 0 & 0 & 0 & 1/2 \\ 1/2 & -1 & 1/2 & 0 & 0 & 0 \\ 0 & 1/2 & -1 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & -1 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 & -1 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} V_A \\ V_B \\ V_C \\ V_D \\ V_E \\ V_T \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1/2 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} V_A \\ V_B \\ V_C \\ V_D \\ V_E \\ V_T \end{bmatrix} = \begin{bmatrix} 1/6 \\ 1/3 \\ 1/2 \\ 2/3 \\ 5/6 \\ 0 \end{bmatrix}$$

B) Some smart shenanigans with the binomial distribution to figure out the number of ways to reach A or E times it's probability of happening summed over all timesteps.



My guess is that A) was used because B) looks like a lot of work on summing infinite series :)

**Exercise 6.7* Design an off-policy version of the TD(0) update that can be used with arbitrary target policy π and covering behavior policy b, using at each step t the importance sampling ratio $\rho_{t:t}$ (5.3). ☑

Value of state s is the expected return conditioned on following π:

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s] =$$

$$= \sum_{\forall a} \pi(a|S_t) \sum_{\forall r \forall s'} p(s', r \mid S_t, a)\big(r + \gamma V_\pi(s')\big) =$$

Unfortunately, all we have available are samples from a different policy b. So we have to somehow re-arrange the formula to contain expectations conditioned on following it

$$= \sum_{\forall a \forall s' \forall r} \pi(a|S_t) \frac{b(a|S_t)}{b(a|S_t)} p(s', r \mid S_t, a)\big(r + \gamma V_\pi(s')\big) = \mathbb{E}_b\big[\rho_{t:t}\big(R_{t+1} + \gamma V_\pi(S_{t+1})\big) \mid S_t = s\big]$$

The expectation can be estimated by averaging samples from b in the same way as the on-policy TD(0) update

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha\big[\rho_{t:t}\big(R_{t+1} + \gamma V_\pi(S_{t+1})\big) - V_\pi(S_t)\big]$$

Exercise 6.8 Show that an action-value version of (6.6) holds for the action-value form of the TD error $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$, again assuming that the values don't change from step to step. ☑

$$R^{A_t}_{t+1} + \gamma G_{t+2} - Q(S_t, A_t) = \left[R^{A_t}_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] + \gamma\left[G_{t+2} - Q(S_{t+1}, A_{t+1})\right] =$$

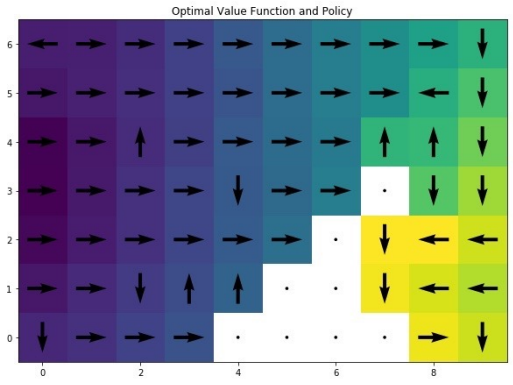
$$= \delta_t + \gamma\left[R^{A_{t+1}}_{t+2} + \gamma Q(S_{t+2}, A_{t+2}) - Q(S_{t+1}, A_{t+1})\right] + \gamma^2\left[G_{t+3} - Q(S_{t+2}, A_{t+2})\right] =$$

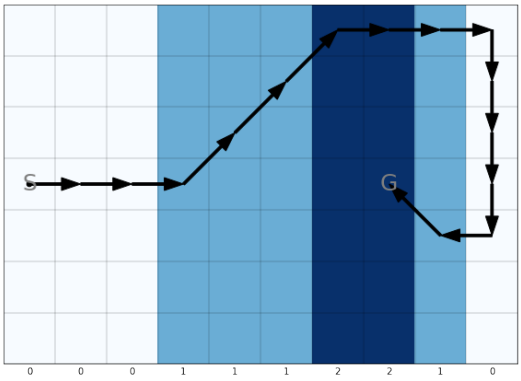
$$= \delta_t + \gamma\,\delta_{t+1} + \gamma^2\left[G_{t+3} - Q(S_{t+2}, A_{t+2})\right] =$$

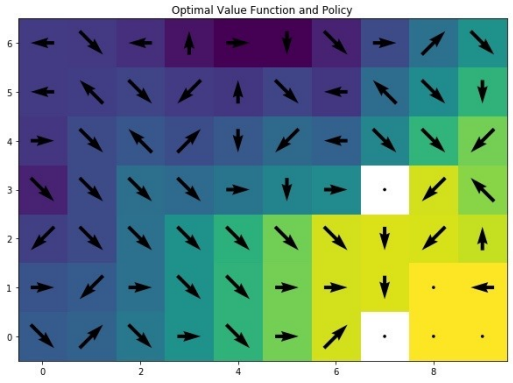
$$= \sum_{k=t}^{T-1} \gamma^{k-t}\,\delta_k$$

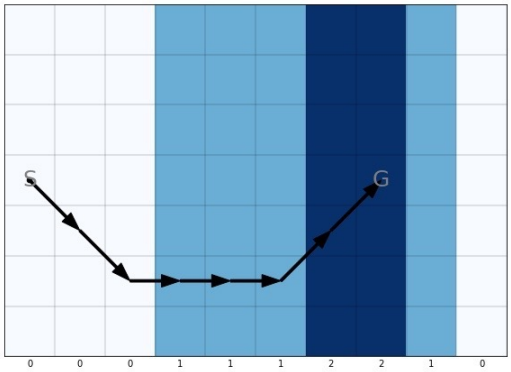
$R^{A_t}_{t+1}$ ... Reward from transition $S_t \to S_{t+1}$ conditioned on taking action A_t

Exercise 6.9: Windy Gridworld with King's Moves (programming) Re-solve the windy gridworld assuming eight possible actions, including the diagonal moves, rather than the usual four. How much better can you do with the extra actions? Can you do even better by including a ninth action that causes no movement at all other than that caused by the wind? ☑

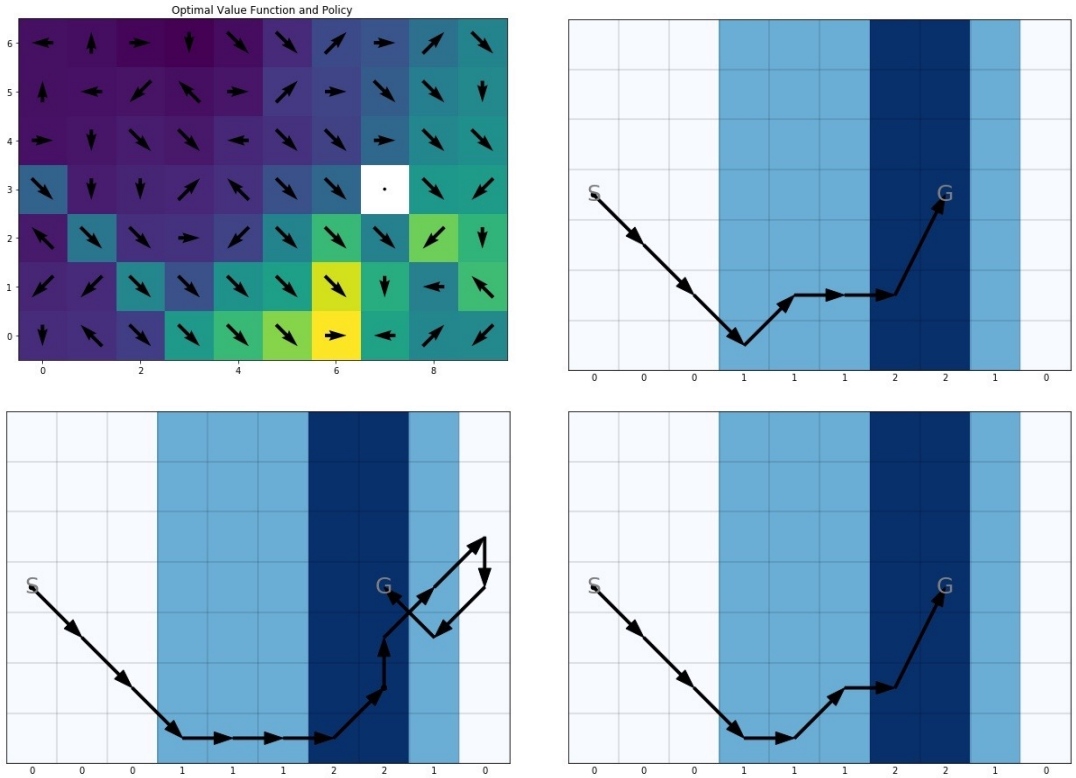








Exercise 6.10: Stochastic Wind (programming) Re-solve the windy gridworld task with King's moves, assuming that the effect of the wind, if there is any, is stochastic, sometimes varying by 1 from the mean values given for each column. That is, a third of the time you move exactly according to these values, as in the previous exercise, but also a third of the time you move one cell above that, and another third of the time you move one cell below that. For example, if you are one cell to the right of the goal and you move `left`, then one-third of the time you move one cell above the goal, one-third of the time you move two cells above the goal, and one-third of the time you move to the goal. ☑



Exercise 6.11 Why is Q-learning considered an *off-policy* control method? ☑

```
Q-Learning is off-policy because it uses different policy for action selection
(typically eps-greedy to keep exploring) and another policy for updating Q values.
Policy used for updating is implicitly built into the max function that appears in
the update rule. It can be in fact extracted out of the equation by introducing a
target policy that's greedy with respect to the current Q values.

Somewhat confusingly Q-learning update rule doesn't contain any importance sampling
ratios. The reason for that is because the updates are done only for a single time-
difference step forward. The definition of Q(s,a) conditions the first step to always
take action a, irrespective of the probability assigned to it by the behavior policy.
Therefore importance sampling doesn't play a role here. It would appear in the update
rule if we would incorporate subsequent time-steps in the rule. Subsequent steps are
not conditioned on any fixed action and the fate of the episode from this point
```

Exercise 6.12 Suppose action selection is greedy. Is Q-learning then exactly the same algorithm as Sarsa? Will they make exactly the same action selections and weight updates? ☑

Sarsa update rule and greedy policy extraction are

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

$$\pi(S_{t+1}, A_{t+1}) = \begin{cases} 1 & \text{if } A_{t+1} = \arg\max_a Q(S_{t+1}, a) \\ 0 & \text{otherwise} \end{cases} \Rightarrow A_{t+1} = \arg\max_a Q(S_{t+1}, a)$$

Combining the two yields Q-learning update rule

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

However, the action selection of Q-learning is different. Because it's off-policy can use any behavioral policy for action selection whereas Sarsa is on-policy and greedy action selection prevents any exploration. One way to view this result is that at the very end of learning when Sarsa has converged and $\varepsilon > 0$ it's updates in the limit are the same as Q-learning.

**Exercise 6.13* What are the update equations for Double Expected Sarsa with an ε-greedy target policy? ☑

Applying double learning to expected Sarsa means to have two action values Q_1 and Q_2 with their corresponding ε-greedy policies π_1 and π_2 for action selection. The evaluation of how good the next actions are is done with use of the other Q function that serves as a target of the update.

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \mathbb{E}_{\pi_1}\left[Q_2(S_{t+1}, A_{t+1}) \mid S_{t+1} \right] - Q_1(S_t, A_t) \right]$$

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha \left[R_{t+1} + \mathbb{E}_{\pi_2}\left[Q_1(S_{t+1}, A_{t+1}) \mid S_{t+1} \right] - Q_2(S_t, A_t) \right]$$

This update is equivalent to the Double Q-learning one when π_1 and π_2 both perform greedy action selection.

Exercise 6.14 Describe how the task of Jack's Car Rental (Example 4.2) could be reformulated in terms of afterstates. Why, in terms of this specific task, would such a reformulation be likely to speed convergence?

The after-states in this case correspond to the number of cars at each location after the nightly transfer of cars between the two locations. The convergence speed-up would come from the fact we'd estimate a simpler mapping.

Original problem estimates the expected revenue and car counts at each location from of pre-transfer car counts and the transfer number. After-state formulation allows for removing of one degree of freedom from this mapping - the car transfer count. The car transfer is deterministic and it's effect on the car counts is shrouded in stochasticity only by later random rental and return requests.

The simpler, two argument after-state mapping only takes in the number of cars at each location at the beginning of day and returns the expected revenue and the number of cars at each location at the end of the day. The nightly car transfer is taken out of the picture and