1 Review and Overview

All of the reinforcement learning approaches we have covered so far in this course come from a family of algorithms known as *value estimation* methods and assume that you can obtain an exact solution to the problem of estimating parameters, at least in the limit (i.e., with infinite data). This review assumes fluency in the terminology of RL and MDPs.

1.1 Objectives

There are two objectives in value estimation methods: solving the *prediction problem* and solving the *control problem*.

Prediction problem. We have encountered the prediction problem under two other names: *policy evaluation* and *value function estimation*, where the value function may refer to the state-value function (i.e., v) or the action-value function (i.e., q).

Learning functions. All machine learning problems involve estimating the value of some parameters. For RL, the parameters of interest are the values: v or q. We often refer to these as "functions," which may be an unfamiliar way of thinking about learning parameters of a model to some. We do this because it is a compact way of talking about the set of v's and q's. It turns out that, for every policy π, there is a unique v for every state and a unique q for every valid state-action pair. One way of phrasing this is that there is a *mapping* from every state (or state-action pair) to a real number. Since there is only one possible number associated with each state or state-action pair *for a given policy* π, and since every state must have an associated number, we can say that this mapping meets the criteria of a *function*. We sometimes denote the function with a capital letter for convenience, but this practice may vary. Note that random variables, which are also functions, are also often denoted with capital letters.

Control problem. The control problem refers to the process of learning the best action to take in a state. We have encountered the control problem in the context of bandits, where the *only* design choice we face is which action to take (since there is only one state), and under the names *policy improvement*, *policy iteration*, and *value iteration*. We tend to emphasize the control problem less because (1) the sample or time complexity is dominated by the same search process used in the prediction problem and (2) in nearly every case, the algorithm for the control problem looks like the prediction problem, plus the policy update rule:

$$\pi(s) \leftarrow \arg\max_a \left(\mathbb{E}[q^\pi(s,a)]\right)$$

We are particularly interested in learning our parameters via update rules that can be applied as new data come in. Whatever the unit of iteration (i.e., the outer loop of the algorithm), we want to be able to update our estimate of the parameters using just (a) any prior estimates of the parameters and (b) quantities available to us within some easily-accessible window of experience. Our objective is to refine our estimates each time the outer loop is executed. This means that if we need to terminate our search early, we will still have made some progress on our parameter estimation.

1.2 Mathematics Review

All of the problems we encounter in this course are *discrete control problems*: the state space is discrete and often finite. Reinforcement learning developed in tandem with another disciplines known as *control theory*, which emphasizes the *continuous control problem*. Continuous control is "easier" in the sense that we can deploy *convex optimization* to solve most problems. Discrete state spaces must often resort to *combinatorial optimization*, which can be NP-hard. Fortunately, the Bellman equations express a recurrence relation that allows us to update our estimates of parameters incrementally, meaning that we can always terminate early with an approximate solution.

While the mathematics that underpin the theoretical results of this course are deep and may be unfamiliar, we do not focus on optimality, just on derivations. Derivations are important to understand because they tell a story of where the equations you use come from. All of the derivations we cover rely on a small subset of mathematics you have seen before: discrete probability theory and algebraic manipulation.[1]

1.2.1 Discrete Probability Theory

A *random variable* is a function from the outcome of an experiment (an "event") to a real number. A *probability rule* is mapping from an event (in the set-theoretic sense) to a number between 0 and 1. For example, when there is stochasticity (i.e., randomness) in the environment, q is a random variable: it maps an event (being in state s and taking action a) to a real number (the q-value). When our policy is stochastic, we treat it as a probability law: $\pi(a \mid s)$, which gives the probability of taking action a in state s. Note that we only really want to treat π as a probability law while we are learning. Once we have found the optimal action to take for every state, we will fix π so that $\pi^*(s) = \arg\max_a(\mathbb{E}[q^*(s,a)])$.[2]

Definitions. When deriving identities, I may use any of the following definitions. If you are not familiar with any of these, please ask! Note that all of the probability laws we work with obey the Kolmogorov axioms; these three axioms should be intuitive at this point, but for completeness, I've reproduced them here:

1. There are no negative probability events,
2. The probabilities of all events add up to one, and
3. The probability of any subset of events is equal to the sum of the probabilities of the individual events.

conditional probability distribution The probability laws where some random variable's outcome has been observed, expressed as $P(X \mid Y)$. For the purposes of this course, you may find it helpful to think about this expression as having observed some value of Y and mentally rewrite it as $P(X \mid Y = y_0)$ for some existentially quantified output of Y, denoted here as

[1] By "algebraic manipulation" I mean the algebra you learned prior to college, not the theory of groups, rings, etc.
[2] Recall that we use the asterisk to denote that this is the best, or optimal, policy and optimal q value.

y_0.[3] All conditional probability distributions obey the Kolmorgorov axioms. Conditional probability distributions are often represented as a path through a tree, where each event or random variable on the right side of the condition symbol corresponds to a split along the path.

joint probability distribution The probability laws for more than one event happening *jointly*: $P(X_1, X_2, \ldots, X_n)$, where each X is a random variable. Joint probability distributions are often represented as the sum of leaves in a tree.

unbiased estimator And estimator is a function that estimates a parameter. The first estimator that students are often exposed to is sample average as an estimate of population mean. This estimator also happens to be *unbiased*, that is, as you collect more and more data, it converges to the population mean (i.e., is equal to the parameter in expectation). However, you may not be interested in unbiased estimators such as sample average because the parameters they estimate may not be important. For example, population mean is sensitive to outliers, so maybe you want something that gives better predictive accuracy, but does not converge to the population parameter. Unbaised estimators often have high variance in their predictions, so they are not always desired. We see several examples of biased estimators throughout RL.

Important Identities. There are several important identities you will need to know to follow the derivations:

- *Law of Total Probability.* Since in this course we focus on random variables, rather than sets, we use the definition of the Law of Total Probability for random variables:

$$P(X) = \sum_{y \in \mathcal{Y}} P(X, Y = y)$$

  $\mathcal{Y}$ is called the *support* of the random variable Y and refers to the codomain of Y, but you can think of it as the set of values Y can take on.

- *Definition of Conditional Probability.* Conditional probability is actually defined in terms of joint probability:

$$P(X \mid Y) = \frac{P(X, Y)}{P(Y)}$$

  While it can sometime be helpful to think of conditional probabilities as sequential events, from the definition above you can see that there need not be a temporal component.

- *Expectation.* Expectation is a function[4] of random variables, denoted $\mathbb{E}$. Since we can define new random variables in terms of other random variables, we arrive at the general expression

[3] Some would say that all probability laws are conditional probability laws, since, for example, they condition on the possibility of chance, that reality is real, that gravity works, etc. If we can never change the values of a particular random variable we are conditioning on (i.e., the stuff to the right of |), then we drop it.

[4] Well, technically it's a *functional* (the math term) or *higher order function* (the computer science term), since it takes another function as its argument (since a random variable is a function). You'll notice that expectation is written with brackets instead of parentheses (i.e., $\mathbb{E}[\cdot]$ instead of $\mathbb{E}(\cdot)$. This is a notational convenience that signals that it is a functional.

for expectation via $Y = g(X)$ for some function g:

$$\mathbb{E}[Y] = \mathbb{E}[g(X)] = \sum_{x \in \mathcal{X}} g(X) \cdot P(X = x)$$

Expectation always produces a constant. This fact is often leveraged in proofs and derivations. We often refer to this constant as the *expected value* of the input.

- *Conditional Expectation.* Conditional expectation is similar defined, but we now condition on some other variable. Let X, $Y = g(X)$ and Z be random variables. Then:

$$\mathbb{E}[Y \mid Z] = \mathbb{E}[g(X) \mid Z] = \sum_{x \in \mathcal{X}} g(X) \cdot P(X = x \mid Z)$$

Conditional expectation is itself a random variable, but for the purposes of this course, we will treat it analogously to marginal expectation (i.e., regular old expectation). In practice this means I will often give you a specific value for Z.

1.2.2 Properties of Summations and Other Linear Maps

This course expresses many important identities in terms of summations. Summation has special properties because it is a *linear map*. It turns out that expectation also shares these properties, and they are critical for many of the derivations we do. We won't get into the most general expression for linear maps, since it also relies on defining what + means! The important piece to know is that the following properties hold for both summation and expectation:

$$\sum_{i=0}^{n} a x_i = a \sum_{i=0}^{n} x_i \quad \text{and} \quad \mathbb{E}[aX] = a\mathbb{E}[X]$$

$$\sum_{i=0}^{n} (x_i + y_i) = \left(\sum_{i=0}^{n} x_i\right) + \left(\sum_{i=0}^{n} y_i\right) \quad \text{and} \quad \mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$$

We will use these identities many times through our derivations. There are a handful of other tricks we will apply, all which are simple but easily forgotten:

- Sometimes we want the first term of a sum; we need to increment the lower bound:

$$\sum_{i=0}^{n} x_i = x_0 + \sum_{i=1}^{n} x_i \tag{1}$$

- Sometimes we want the final term of a sum; we need to increment the upper bound:

$$\sum_{i=0}^{n} x_i = x_n + \sum_{i=0}^{n-1} x_i \tag{2}$$

- Sometimes we want a term to appear in an expression because it can then be grouped into an expression that can be rewritten in some desirable way. For example, if we want to introduce

Z because $X + Z$ corresponds to some known expression, we can introduce Z by both adding and subtracting it:

$$X + Y = X + Y + (Z - Z) = (X + Z) + Y - Z \tag{3}$$

We use this "trick" repeatedly in our derivations of incremental update rules, when we have some ratio:

$$\begin{aligned}
\frac{\sum_{i=1}^{n-1} f(i)}{\sum_{i=1}^{n} f(i)} &= \frac{\left(\sum_{i=1}^{n-1} f(i)\right) + f(n) - f(n)}{\sum_{i=1}^{n} f(i)}, && \text{the "trick"} \\
&= \frac{(\sum_{i=1}^{n} f(i)) - f(n)}{\sum_{i=1}^{n} f(i)}, && \text{move } f(n) \text{ into summation} \\
&= \frac{\sum_{i=1}^{n} f(i)}{\sum_{i=1}^{n} f(i)} - \frac{f(n)}{\sum_{i=1}^{n} f(i)} \\
&= 1 - \frac{f(n)}{\sum_{i=1}^{n} f(i)}
\end{aligned}$$

- Similarly, we sometimes want a multiplicative factor (e.g., $\frac{1}{n}$) to appear for mathematical convenience, and can introduce it by multiplying by an expression equal to one:

$$X = \frac{n}{n} X = n \left(\frac{1}{n} X \right) \tag{4}$$

All other derivation "tricks" in the text arise from the definitions of quantities defined in the text. For example, there are serveral derivations that simple rewrite some expression on the right-hand side of an equation to have the same form as the expression on the left-hand side of the equation.

2 Core Classic RL Algorithms

This section of the course covers the most "famous" RL algorithms, which purport to "show off" what is so special about RL: temporal difference (TD) learning, SARSA, and Q-learning. It can be helpful to think of these algorithms as more nuanced or refined versions of the techniques we have already seen. Note that Sutton and Barto introduce with bandits the idea that most update rules can be described using the template:

$$\text{NewEstimate} \longleftarrow \text{OldEstimate} + \text{StepSize}[\underbrace{\text{Target} - \text{OldEstimate}}_{\text{Error}}]^{5} \tag{5}$$

All value-function-based RL algorithms focus on improving the definitions of Target and StepSize. In some cases, Target and StepSize have "natural" definitions that follow from definitions for unbiased estimators. However, unbiased estimators are often not good predictors, furthermore we may be working in contexts where we'd like to more heavily weigh recent experience. Figure 2 gives an overview of the update rules we have seen so far.

[5] Note that the brackets here do not mean anything special, as they did for expectation; I merely reproduced the notation used in Sutton and Barto, which used brackets for grouping.

Bandits

$$q_{n+1} = q_n + \frac{1}{n}[r - q_n]$$

Dynamic Programming

$$V^{\pi}_{k+1}(s_t) = \sum_{a \in A} \big(r(s_t, a)\pi(a|s_t)\big) + \gamma \cdot V^{\pi}_{k}(s_{t+1})$$

$$Q^{\pi}_{k+1}(s_t, a_t) = r(s_t, a_t) + \gamma \cdot \sum_{a \in A} \big(Q^{\pi}_{k}(s_{t+1}, a)\pi(a \mid s_t)\big)$$

Monte Carlo (general)

$$V^{\pi}_{n+1}(s_t) = V^{\pi}_{n}(s_t) + \frac{1}{n}[G_n(s_t) - V^{\pi}_{n}(s_t)]$$

$$Q^{\pi}_{n+1}(s_t, a) = Q^{\pi}_{n}(s_t, a) + \frac{1}{n}[G_n(s_t) - Q^{\pi}_{n}(s_t, a)]$$

Monte Carlo (importance sampling)

$$V^{\pi}_{n+1}(s_t) = V^{\pi}_{n}(s_t) + \frac{W_n}{C_n}[G_n(s_t) - V^{\pi}_{n}(s_t)]$$

$$Q^{\pi}_{n+1}(s_t, a) = Q^{\pi}_{n}(s_t, a) + \frac{W_n}{C_n}[G_n(s_t) - Q^{\pi}_{n}(s_t, a)]$$

Where $G_n(s_t)$, W_n, and C_n are all computing using the behavior policy b, rather than the target policy, π.

Figure 1: Update rules for prior learning approaches

2.1 Bandits

Recall that for bandits, there is only one state, and the only choice we are free to make is which action to take (we ignore contextual bandits for now, since they do not fundamentally change our formalism).

Recall that the update rule for bandits is simply a rearrangement of our expression for updating the running average of rewards resulting from taking some action a. We store the running average of each a in some table Q and the total count of each action taken in some table C. Then the average reward for the next round (i.e., the next time we take action a) is

$$Q[a] = \frac{Q[a]C[a] + r}{C[a] + 1}$$

If this expression is confusing, I encourage you to write out a string of numbers and compute the running average as each element comes in.

Since referring to table lookups is very operational and in the weeds of low-level details, we tend to prefer expressing $Q[a]$ as just q when the action is known and clear. Also, since we recompute q as each new datum comes in, we can index our current estimate of q by the total number of data

seen so far. This is how we arrive at the preferred representation:

$$\begin{aligned} q_{n+1} &= \frac{q_n(n-1)+r}{n} \\ &= q_n - \frac{q_n}{n} + \frac{r}{n} \\ &= q_n + \frac{r}{n} - \frac{q_n}{n} \\ &= q_n + \frac{1}{n}\left[r - q_n\right] \end{aligned}$$

There are several ways to arrive at this quantity, all equivalent to recomputing the average of rewards seen so far, for a particular action. Nearly all update rules are derived in an analogous manner.

We can see how each term maps to the template:

NewEstimate:	q_{n+1}	StepSize:	$\frac{1}{n}$
OldEstimate:	q_n	Target:	r

Recall that in the bandits videos, we discussed how one could replace $\frac{1}{n}$ with some α; if $\alpha > \frac{1}{n}$, then we will move our estimate of q_{n+1} more than if $\alpha = \frac{1}{n}$. If α scales with n, but is always greater than $\frac{1}{n}$, then we effectively weigh more recent rewards more than older rewards.

2.2 Dynamic Programming

Dynamic programming is the one case where we do not follow the template, since its estimation procedure is an estimate for a different reason from the other methods. The other other methods we've seen are estimate that we expect to converge as we collect more data, leveraging the law of large numbers. For dynamic programming, we are computing value functions exactly, but they are incremental sums of (possibly discounted) rewards, and so the estimation here is more about "not being there yet."

Let's look at the update rules for DP:

$$V^{\pi}_{k+1}(s_t) = \sum_{a \in A} \big(r(s_t, a)\pi(a|s_t)\big) + \gamma V^{\pi}_{k}(s_{t+1})$$
$$Q^{\pi}_{k+1}(s_t, a_t) = r(s_t, a_t) + \gamma \sum_{a \in A} \big(Q^{\pi}_{k}(s_{t+1}, a)\pi(a \mid s_t)\big)$$

Recall that by definition, the value of a state (i.e., $v^{\pi}(s)$) is the sum of discounted rewards from following the policy in that state until the end of the episode:

$$\sum_{i=0}^{T} \gamma^i r(s_i, a_i), \quad \text{when } \pi(a_i|s_i) = 1.0$$
$$\sum_{i=0}^{T} \gamma^i \mathbb{E}[r(s_i, a_i)] = \sum_{i=0}^{T} \gamma^i \sum_{a \in A} r(s_i, a)\pi(a|s_i), \quad \text{otherwise compute the expected reward}$$

For convenience, we will assume that the policy is deterministic, but when it is not, we can replace any instance of the reward function with the expected reward.

We can take the expression of the sum of discounted rewards and pull out the first term, noting that it is equivalent to the reward obtained in the current state, plus the discount factor times the value of the next state:

$$V^\pi(s_t) = r(s_t, a_t) + \gamma V^\pi(s_{t+1}) \tag{6}$$

This expression can be obtained using only the background in Section 1.2, and is covered in the lecture videos. I its derivation is not immediately obvious, I encourage you to try to produce it, and if you cannot, to come see me.

Note that because V is a single number, it "hides" whether the environment is stochastic. When we have stochasticity in the environment, the value of being in a given state at time t must be the expected reward plus the expected value of all possible transitions (think of it as averaging over all possible trajectories, starting in state s_t). When the environment is deterministic, the s_{t+1} in the above expression can only refer to one state, but when there is stochasticity, we can "pun" on V as both a lookup table (function) and a random variable (function):

$$\begin{aligned} V^\pi(s_t) &= r(s_t, a_t) + \gamma \mathbb{E}[V^\pi] \\ &= r(s_t, a_t) + \gamma \sum_{(s,s') \in \mathcal{S} \times \mathcal{S}} V^\pi(s') P(s'|s, a) \end{aligned}$$

Now, the dynamic programming update rule just iteratively adds discounted rewards. The key here is to update the next estimate of $V^\pi(s_t)$ using the old estimate of $V^\pi(s_{t+1})$:

$$V^\pi_{k+1}(s_t) = r(s_t, a_t) + \gamma V^\pi_k(s_{t+1})$$

That is, $V_k(s_{t+1})$ corresponds to the *current* estimate of the value of the *next* state. When the environment is stochastic, you will explicitly compute the expected value by considering all possible transitions and their probabilities.

We can see how this method builds up the estimate of the parameter *not* by sampling, but rather by stepping through each possible trajectory. Even when there is stochasticity in the agent or environment, we do not sample and can compute the expected values exactly, because we have access to the transition probabilities (i.e., we have a model of the environment). The general form of the update rule (Eq. 5) does not apply.

2.3 Monte Carlo Methods

For Monte Carlo methods, we use a policy to actually *sample trajectories* from the environment, compute returns, and average the observed state-values or action-values.

Recall the definition of a return for an episode that terminates at time T:

$$G_t = \sum_{i=t}^{T-1} \gamma^{i-t} r(s_i, a_i)$$

such that if we observe a trajectory

$$s^0\ a^1\ 1\ s^1\ a^1\ 1\ s^1\ a^2\ 2\ s^2\ a^2\ 3\ s^\infty$$

then the undiscounted (i.e., $\gamma = 1$) return for the entire trajectory (G_0) is:

$$1 + 1 + 2 + 3 = 7$$

We might also be interested in the first-visit return for a particular state. We scan the sequence to find the index of the first occurrence of that state in the trajectory and then compute the return at that index. For example, suppose we are interested in the return for s^1. First we find our first occurrence of s^1:

$$s^0\ a^1\ 1\ \underset{\uparrow}{s^1}\ a^1\ 1\ s^1\ a^2\ 2\ s^2\ a^2\ 3\ s^\infty$$

Then we sum up the discounted rewards that we scan *after* identifying the state of interest:

$$s^0\ a^1\ 1\ s^1\ a^1\ \underbrace{\mathbf{1}\ s^1\ a^2\ \mathbf{2}\ s^2\ a^2\ \mathbf{3}}_{1^0\cdot\mathbf{1}+1^1\cdot\mathbf{2}+1^2\cdot\mathbf{3}=6}\ s^\infty$$

Since our aim is to take a sample of returns, we need a way to refer to a collection of trajectories. Let $\mathcal{H} = \{H_i, \ldots, H_n\}$ be a set of n trajectories. In this handout, we will refer to returns both by their index (i.e., G_t^i refers to the return at time t in the i^{th} trajectory) and as a function of their state (i.e., $G^i(s)$ is the return at the first occurrence of state s in trajectory i).

Monte Carlo methods use returns to estimate values. If we have the ability to drop an agent into an arbitrary state $s \in \mathcal{S}$, then we can just loop over each state, dropping the agent into each state, and forward-simulating. We can run our agents until our estimates converge. We might decide ahead of time to take n sample trajectories and then say:

$$\hat{V}^\pi(s) = \frac{1}{n}\sum_{i=1}^{n} G^i(s)$$

This estimate can be decomposed into an update rule that follows the template of Eq. 5. Rather than deciding n *a priori*, we want to continually query for new trajectories on an as-need basis:

$$
\begin{aligned}
\hat{V}^{\pi}_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} G^i(s) \\
&= \frac{G^n(s) + \sum_{i=1}^{n-1} G^i(s)}{n}, && \text{Eq. 2} \\
&= \frac{G^n(s)}{n} + \frac{\frac{n-1}{n-1}\cdot\sum_{i=1}^{n-1} G^i(s)}{n}, && \text{Eq. 4} \\
&= \frac{G^n(s)}{n} + (n-1)\frac{\frac{1}{n-1}\cdot\sum_{i=1}^{n-1} G^i(s)}{n}, && \text{regroup} \\
&= \frac{G^n(s)}{n} + (n-1)\frac{\hat{V}^{\pi}_n(s)}{n}, && \text{substitute} \\
&= \frac{G^n(s)}{n} + \frac{n}{n}\hat{V}^{\pi}_n(s) - \frac{\hat{V}^{\pi}_n(s)}{n}, && \text{distribute} \\
&= \hat{V}^{\pi}_n(s) + \left(\frac{G^n(s)}{n} - \frac{\hat{V}^{\pi}_n(s)}{n}\right), && \text{regroup} \\
&= \hat{V}^{\pi}_n(s) + \frac{1}{n}\left(G^n(s) - \hat{V}^{\pi}_n(s)\right), && \text{factor}
\end{aligned}
$$

Now we have an incremental update rule where:

NewEstimate:	V_{n+1}	StepSize:	$\frac{1}{n}$
OldEstimate:	V_n	Target:	$G^n(s)$

If we cannot drop the agent into an arbitrary state, we will need to sample from the eligible start states and have the agent explore its environment. In the general there are better methods for learning a value function than straight Monte Carlo methods. However, there will be some cases where we cannot interact directly with the environment. Fortunately, we can use a form of Monte Carlo sampling called *importance sampling* to compute the value functions of interest.

Importance Sampling. Importance sampling is an *off-policy* learning algorithm that is most often used in cases where it is very costly to train a new agent by interacting with the environment directly. The idea is to use historic log data from *a different* agent (often called the *behavior policy*) in order to solve the prediction and control problems for *this agent* (often called the *target policy*).

In order to compute this quantity, we must compute the *importance ratio* for each trajectory, starting at the first occurrence of s_t, *in addition* to computing the relevant return for each trajectory. Let t be the index of the first occurrence of s in H_i:

$$\rho^i(s_t) = \prod_{j=t}^{T-1} \frac{\pi(a_j \mid s_j)}{b(a_j \mid s_j)}$$

The lecture videos and text derive the formulae and provide explanations, so we will not go into detail here. Note that the importance ratio is often referred to with a subscript, which corresponds to the time at which the state of interest was observed, in a similar manner to returns. When we use

importance ratios in incremental update rules, we refer to them as W_n. We will be most interested in *weighted* importance sampling, whose update rules map to the quantities:

NewEstimate:	V_{n+1}	StepSize:	$\frac{W_n}{C_n}$
OldEstimate:	V_n	Target:	$G^n(s)$

While at first blush, it appears the that the only difference between importance sampling and general MC methods is the step size, we want to point out that the returns are obtained from the behavior policy (i.e., $G^n(s)$ is from the n^{th} trajectory obtained from following the behavior policy).

Example. Suppose we have some target policy:

$$\pi(a \mid s) = \begin{cases} 1/2, & \text{if } a = a^1 \\ 1/6, & \text{otherwise} \end{cases}$$

and some behavior policy:

$$b(a \mid s) = 1/|\mathcal{A}|$$

and we observed some trajectories from b over the state and action sets:

$$\mathcal{S} = \{s^0, s^1, s^2, s^\infty\}$$
$$\mathcal{A} = \{a^1, a^2, a^3, a^4\}$$

$$H_1 = s^0\ a^1\ 1\ s^1\ a^1\ 1\ s^1\ a^2\ 2\ s^2\ a^2\ 3\ s^\infty$$
$$H_2 = s^0\ a^1\ 1\ s^2\ a^4\ 3\ s^2\ a^2\ 3\ s^\infty$$
$$H_3 = s^0\ a^2\ 0\ s^0\ a^2\ 0\ s^0\ a^1\ 1\ s^2\ a^3\ 3\ s^2\ a^2 3\ s^\infty$$

Table 1 shows the batch computation for $\hat{V}_3^\pi(s^2)$.

We can also compute $\hat{V}_3^\pi$ incrementally. There are two ways to think about the incremental update for MC methods. The first is very intuitive and builds off the derivations of the update rules from earlier methods. However, it is still multi-pass (we will sweep the trajectory multiple times). The second way to think about incremental updates is a single-pass over each trajectory, which involves some creative accounting.

First we derive an update rule. Let ρ^i and G^i be respectively the importance ratio and return for the first visited state s in trajectory H_i. Then:

$$H_1 = s^0\ a^1\ 1\ s^1\ a^1\ 1\ s^1\ a^2\ \underbrace{2\ s^2\ a^2}_{t=3}\ 3\ s^\infty$$

$G_3^1(s^2) = 3$ $\qquad$ $\rho^1(s^2) = \frac{1/6}{1/4} = \frac{2}{3}$

$$H_2 = s^0\ a^1\ \underbrace{1\ s^2\ a^4}_{t=1}\ 3\ s^2\ a^2\ 3\ s^\infty$$

$G_1^2(s^2) = 3 + 3 = 6$ $\qquad$ $\rho^2(s^2) = \frac{1/6 \cdot 1/6}{1/4 \cdot 1/4} = \frac{4}{9}$

$$H_3 = s^0\ a^2\ 0\ s^0\ a^2\ 0\ s^0\ a^1\ 1\ s^2\ a^3\ \underbrace{3\ s^2\ a^2}_{t=4} 3\ s^\infty$$

$G_4^3(s^2) = 3$ $\qquad$ $\rho^3(s^2) = \frac{1/6}{1/4} = \frac{2}{3}$

$$\hat{V}(s^2) = \frac{3 \cdot \frac{2}{3} + 6 \cdot \frac{4}{9} + 3 \cdot \frac{2}{3}}{\frac{2}{3} + \frac{4}{9} + \frac{2}{3}} = 3.75$$

Table 1: Illustration of how to compute the estimated state value for s^2 with importance sampling. All trajectories are sampled from the behavior policy.

$$\begin{aligned}
V_{n+1}(s) &= \frac{\sum_{i=1}^{n} \rho^i G^i}{\sum_{i=1}^{n} \rho^i} \\
&= \frac{1}{\sum_{i=1}^{n} \rho^i} \left(\sum_{i=1}^{n} \rho^i G^i \right) && \text{factor for legibility} \\
&= \frac{1}{\sum_{i=1}^{n} \rho^i} \left(\rho^n G^n + \sum_{i=1}^{n-1} \rho^i G^i \right) && \text{Eq. 2} \\
&= \frac{1}{\sum_{i=1}^{n} \rho^i} \left(\rho^n G^n + \frac{\sum_{i=1}^{n-1} \rho^i}{\sum_{i=1}^{n-1} \rho^i} \sum_{i=1}^{n-1} \rho^i G^i \right) && \text{Eq. 4} \\
&= \frac{1}{\sum_{i=1}^{n} \rho^i} \left(\rho^n G^n + \left(\sum_{i=1}^{n-1} \rho^i \right) \left(\frac{1}{\sum_{i=1}^{n-1} \rho^i} \right) \sum_{i=1}^{n-1} \rho^i G^i \right) && \text{factor} \\
&= \frac{1}{\sum_{i=1}^{n} \rho^i} \left(\rho^n G^n + \left(\sum_{i=1}^{n-1} \rho^i \right) \left(\frac{\sum_{i=1}^{n-1} \rho^i G^i}{\sum_{i=1}^{n-1} \rho^i} \right) \right) && \text{group} \\
&= \frac{1}{\sum_{i=1}^{n} \rho^i} \left(\rho^n G^n + \left(\sum_{i=1}^{n-1} \rho^i \right) V_n(s) \right) && \text{substitute def. of V} \\
&= \frac{\rho^n G^n}{\sum_{i=1}^{n} \rho^i} + V_n(s) \frac{\sum_{i=1}^{n-1} \rho^i V_n(s)}{\sum_{i=1}^{n} \rho^i} && \text{distribute} \\
&= \frac{\rho^n G^n}{\sum_{i=1}^{n} \rho^i} + V_n(s) \left(1 - \frac{\rho^n}{\sum_{i=1}^{n} \rho^i} \right) && \text{Eq. 3 (adding } \rho^n) \\
&= V_n(s) + \frac{\rho^n}{\sum_{i=1}^{n} \rho^i} \left(G^n - V_n(s) \right) && \text{factor and arrange}
\end{aligned}$$

This equation gives us a way to update our estimates of $V_n(s)$ as new trajectories come in (where

n refers to the estimate of the value after encountering the nth trajectory). Using the returns and importance ratios from Table 1:

$$
\begin{aligned}
V_0(s) &= 0 \\
V_1(s) &= 0 + \frac{2/3}{0 + 2/3}(3 - 0) = 3 \\
V_2(s) &= 3 + \frac{4/9}{2/3 + 4/9}(6 - 3) = 3 + \frac{4/9}{10/9}(6 - 3) = 4.2 \\
V_3(s) &= 4.2 + \frac{2/3}{10/9 + 2/3}(3 - 4.2) = 3.75
\end{aligned}
$$

This incremental update rule increments over *trajectories*. Each importance ratio ρ^i is the likelihood ratio of the data under the two policies, which means it involves multiply by potentially many terms, and means that for each state, we will repeat computation. The book provides another form of the incremental update rule where the increments are over *steps* instead (i.e., proportional to $T \times n$ time steps, rather than n trajectories).

In order to implement the new incremental update, the book defines two new quantities: W_n and C_n, which correspond to ρ^n and $\sum_{i=1}^{n} \rho^i$, except we allow them to range over *all* time steps/states, rather than fixing them to the set of ratios and returns for s. Both the prediction algorithm and control algorithm move backwards through the trajectory.

2.4 Temporal Difference Learning

TD(0)

$$
\begin{aligned}
V^{\pi}_{n+1}(s_t) &= V^{\pi}_{n}(s_t) + \alpha \left[r(s_t, a_t) + \gamma V^{\pi}_{n}(s_{t+1}) - V^{\pi}_{n}(s_t) \right] \\
Q^{\pi}_{n+1}(s_t, a) &= Q^{\pi}_{n}(s_t) + \alpha \left[r(s_t, a) + \gamma Q^{\pi}_{n}(s_{t+1}, a_{t+1}) - Q^{\pi}_{n}(s_t, a) \right]
\end{aligned}
$$

TD(n)

$$
\begin{aligned}
V_{t+n}(s_t) &= V_{t+n-1}(s_t) + \alpha [G_{t:t+n} - V_{t+n-1}(s_t)] \\
Q_{t+n}(s_t, a) &= Q_{t+n-1}(s_t, a) + \alpha [G_{t:t+n}(s_t) - Q_{t+n-1}(s_t, a)],
\end{aligned}
$$

where $a_t = a$ for $G_{t:t+n}(s_t)$ and $0 \leq t < T$.

Figure 2: Update rules for the prediction problem (value function estimation) for the temporal difference (TD) family of algorithms.

Temporal difference (TD) learning is often synonymous with reinforcement learning itself. TD learning actually refers to a family of methods where the update rule for the prediction problem uses partially computed sums (à la dynamic programming) and samples/forward simulates (à la Monte Carlo methods).

For TD learning, we begin to shift away from our focus on the prediction problem to focus more heavily on the control problem. Recall that the control algorithms for most of our prior methods

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\textit{terminal}) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 3: Prediction algorithm for TD(0) from page 120 of Sutton and Barto, 2nd ed.

simply combined our action-value estimation with an update to the policy (with the exception of on-policy Monte Carlo control). As a result, we will focus more heavily on action-values (q), rather than state-values (v), since action-values help us decide what action to take.

2.4.1 TD prediction

Our TD update rule for state-values is:

$$V^\pi_{n+1}(s_t) = V^\pi_n(s_t) + \alpha \left[\underbrace{R_{t+1} + \gamma V^\pi_n(s_{t+1})}_{\hat{V}^\pi_n(s_t)} - V^\pi_n(s_t) \right]$$

Note that the highlighted expression $R_{t+1} + \gamma V^\pi_n(s_{t+1})$ looks just like the state function decomposition of Equation 6; we might be tempted to replace it with the expression $V^\pi_n(s_t)$. There are two problems with this:

1. our error term (the stuff inside the brackets) would be zero and we would never learn!, and
2. $R_{t+1} + \gamma V^\pi_n(s_{t+1})$ may not actually equal $V^\pi_n(s_t)$.

(2) follows from the fact that we are using our current estimate of $V^\pi_n(s_{t+1})$ as part of our target, and because s_{t+1} is *a state we actually observed*, whereas $V^\pi_n(s_t)$ effectively encapsulates a running average of all discounted rewards seen so far. Thus, $R_{t+1} + \gamma V^\pi_n(s_{t+1})$ is really more of an estimate of $V^\pi_n(s_t)$, which we denote as $\hat{V}^\pi_n(s_t)$.

Recall that we generally refer to the expression Target - OldEstimate as the error. Since TD learning is actually a family of methods, we give it a special name and symbol, known as *TD error*:

$$\delta_t = R_{t+1} + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$$

Note that TD error is indexed at t, rather than n. We have been using n to refer to the increment in our outer loop, which may be over entire episodes (i.e., trajectories), or subsequences of trajectories. When we operate strictly over single state transition, we use t. Since the TD error is always defined in terms of a state-to-state transition, this is why it is indexed at t. We can plot δ over time to track our progre

Now let's look at how to compute some of these quantities. Let $\mathcal{S}$ and $\mathcal{A}$ be the state and action sets, respectively, and let H_i be the i^{th} trajectory we've collected.[6]

$$\mathcal{S} = \{s^0, s^1, s^2, s^\infty\}$$
$$\mathcal{A} = \{a^1, a^2, a^3, a^4\}$$

$$H_1 = \underbrace{s^0\ a^1}\ \underbrace{1\ s^1\ a^1}\ \underbrace{1\ s^1\ a^2}\ \underbrace{2\ s^2\ a^2}\ \underbrace{3\ s^\infty}$$

$$H_2 = \underbrace{s^0\ a^1}\ \underbrace{1\ s^2\ a^4}\ \underbrace{3\ s^2\ a^2}\ \underbrace{3\ s^\infty}$$

$$H_3 = \underbrace{s^0\ a^2}_{t=0}\ \underbrace{0\ s^0\ a^2}_{t=1}\ \underbrace{0\ s^0\ a^1}_{t=2}\ \underbrace{1\ s^2\ a^3}_{t=3}\ \underbrace{3\ s^2\ a^2}_{t=4}\underbrace{3\ s^\infty}_{t=5}$$

First we initialize every value to zero, as we did for dynamic programming:

$$V_0^\pi(s) = 0, \forall s \in \mathcal{S}$$

Recall that for Monte Carlo methods, although all of the algorithms involved initializing the state-value function for all states, we did not strictly *need* to know the full state set ahead of time, because we ensured our policies doing the exploration were soft. As a result, we knew that we would visit every state, in the limit. We will discuss the subtleties of the prediction problem for action-value functions and its implications for control in the next section.

Now that we have initialized our state-value function, we take our first step, grabbing the state and action at time 0 and the reward and state at time 1. Let α be equal to $1/10$ and γ be $1/2$:

$$H_1 = \mathbf{s}^0\ a^1\ \mathbf{1}\ \mathbf{s}^1\ a^1\ 1\ s^1\ a^2\ 2\ s^2\ a^2\ 3\ s^\infty$$
$$V_1^\pi(\mathbf{s}^0) = V_0^\pi(\mathbf{s}^0) + \frac{1}{10}\left[\mathbf{1} + \frac{1}{2}V_0^\pi(\mathbf{s}^1) - V_0^\pi(\mathbf{s}^0)\right]$$
$$= 0 + \frac{1}{10}\left[\underbrace{1 + \frac{1}{2}\cdot 0 - 0}_{\delta_0 = 1}\right]$$
$$V_1^\pi(\mathbf{s}^0) = \frac{1}{10}$$

Now let's progress through the next few steps.

[6]Note that for TD methods, we do not need to collect trajectories. However, we are expressing them here in order to compare them with other methods.

$$H_1 = s^0\ a^1\ 1\ \mathbf{s^1}\ a^1\ \mathbf{1}\ \mathbf{s^1}\ a^2\ 2\ s^2\ a^2\ 3\ s^\infty$$
$$V_2^\pi(\mathbf{s^1}) = V_1^\pi(\mathbf{s^1}) + \frac{1}{10}\left[\mathbf{1} + \frac{1}{2}V_1^\pi(\mathbf{s^1}) - V_1^\pi(\mathbf{s^1})\right]$$
$$= 0 + \frac{1}{10}\left[\underbrace{1 + \frac{1}{2}\cdot 0 - 0}_{\delta_0=1}\right]$$
$$V_2^\pi(s^1) = \frac{1}{10}$$

$$H_1 = s^0\ a^1\ 1\ s^1\ a^1\ 1\ \mathbf{s^1}\ a^2\ \mathbf{2}\ \mathbf{s^2}\ a^2\ 3\ s^\infty$$
$$V_3^\pi(\mathbf{s^1}) = V_2^\pi(\mathbf{s^1}) + \frac{1}{10}\left[\mathbf{2} + \frac{1}{2}V_2^\pi(\mathbf{s^2}) - V_2^\pi(\mathbf{s^1})\right]$$
$$= \frac{1}{10} + \frac{1}{10}\left[\underbrace{2 + \frac{1}{2}\cdot 0 - \frac{1}{10}}_{\delta_0=19/10}\right]$$
$$V_3^\pi(\mathbf{s^1}) = \frac{29}{100}$$

$$H_1 = s^0\ a^1\ 1\ s^1\ a^1\ 1\ s^1\ a^2\ 2\ \mathbf{s^2}\ a^2\ \mathbf{3}\ \mathbf{s^\infty}$$
$$V_4^\pi(\mathbf{s^2}) = V_3^\pi(\mathbf{s^2}) + \frac{1}{10}\left[\mathbf{3} + \frac{1}{2}V_3^\pi(\mathbf{s^\infty}) - V_3^\pi(\mathbf{s^2})\right]$$
$$= 0 + \frac{1}{10}\left[\underbrace{3 + \frac{1}{2}\cdot 0 - 0}_{\delta_0=3}\right]$$
$$V_4^\pi(\mathbf{s^2}) = \frac{3}{10}$$

Now that we have reached the terminal state, we move on to the next trajectory:

$$H_2 = \mathbf{s^0}\ a^1\ \mathbf{1}\ s^2\ a^4\ 3\ s^2\ a^2\ 3\ s^\infty$$
$$V_5^\pi(\mathbf{s^0}) = V_5^\pi(\mathbf{s^0}) + \frac{1}{10}\left[\mathbf{1} + \frac{1}{2}V_4^\pi(\mathbf{s^2}) - V_4^\pi(\mathbf{s^0})\right]$$
$$= \frac{1}{10} + \frac{1}{10}\left[\underbrace{1 + \frac{1}{2}\cdot\frac{3}{10} - \frac{1}{10}}_{\delta_0=5/4}\right]$$
$$= \frac{9}{40}$$

And so forth.

n-step TD for estimating $V \approx v_\pi$

Input: a policy π
Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n
Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$
All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_{\tau:\tau+n})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha \left[G - V(S_\tau)\right]$
  Until $\tau = T - 1$

Figure 4: Prediction algorithm for n-step TD learning form page 144 of Sutton and Barto, 2nd ed.

Now, in the examples, we were operating over trajectories. However, the appeal of TD methods is precisely that you *do not have to fetch the full trajectory*; you only need to grab the next state. Figure 3 shows this can be done with minimal change from how we normally estimate the value function: by initializing the first state, and simply updating at the *end* of the algorithm, rather than the *beginning*.

Basic TD learning can be thought of as setting the Target to be a partial return over a trajectory subsequence of size one. If we denote a partial return from time t to some time $t + n$, for basic TD learning, $n = 0$. You might wonder why n is *not* 1 for basic TD learning. This is a byproduct of how the n-step TD learning algorithm for prediction is defined.

Figure 4 depicts the algorithm for the n-step prediction problem. It may appear complicated, since it is accounting for all cases, but at a high level, it:

1. Steps through the environment, collecting the sliding window of states and rewards until it hits the appropriate window size.
2. Computes the return for the current window (which may be shorter than n, if we hit the end of an episode).
3. Adds the properly discounted current estimate of the next state after the sliding window (otherwise, we we would definitely not have the proper estimate of the value function1)
4. Applies the update rule.

Using the same parameters as before, let's do a few iterations through the 2-step TD prediction problem informally. Let's start with the first trajectory:

$$H_1 = \underbrace{s^0\ a^1}_{t=0}\ \underbrace{1\ s^1\ a^1}_{t=1}\ \underbrace{1\ s^1\ a^2}_{t=2}\ \underbrace{2\ s^2\ a^2}_{t=3}\ \underbrace{3\ s^\infty}_{t=T=4}$$

We end up calling the inner loop once before we execute the second if-statement (i.e., we are on the 2nd iteration, at time $t = 1$ when this happens):

	$t=0$	$t=1$
Reward at $t+1$	1	1
State at $t+1$	s^1	s^1
τ at t	-1	0

Note that the initial state (s^0) is *not* stored.

Now we execute the second if-statement. τ is zero at this point. The partial return is just the discounted sum of rewards from the elements of the queue:

$$\begin{aligned} G_2 &= \gamma^{1-0-1} R_1 + \gamma^{2-0-1} R_2 \\ &= R_1 + \gamma R_2 \\ &= 1 + \frac{1}{2} \cdot 1 = \frac{3}{2} \end{aligned}$$

Since we have not yet hit the terminal state, we need to update the return with the appropriately discounted current estimate of the value of the next state after the end of window:

$$G_2 = \frac{3}{2} + \frac{1}{2} V_2(s^1)$$

Since this is the first iteration of the loop, $V_2(s^1)$ is whatever we initialized it to be. Let all state values be initialized to zero. Then our Target is 11/10.

Now we can update our estimate of $V_2(s^0)$:

$$\begin{aligned} V_2(s^0) &= V_2(s^0) + \frac{1}{10}\left[G_2 - V_2(s^0)\right] \\ &= 0 + \frac{1}{10}\left[\frac{3}{2} - 0\right] \\ &= \frac{3}{20} \end{aligned}$$

Now let's take another step through the algorithm. We continue to treat the indexing scheme of τ as a queue, shifting along the values from before and taking the next step:

	$t=1$	$t=2$
Reward at $t+1$	1	2
State at $t+1$	s^1	s^2
τ at t	0	1

We update our return again, combining the two updates into one, since we have not yet reached the terminal state:

$$\begin{aligned} G_3 &= \gamma^{2-1-1} R_2 + \gamma^{3-1-1} + R_3 + \gamma^2 V(s_3) \\ &= R_2 + \gamma R_3 + 0 \\ &= 1 + \frac{1}{2} \cdot 2 \\ &= 2 \end{aligned}$$

We now update the value function again:

$$\begin{aligned} V_3(s^1) &= V_2(s^1) + \alpha \left[G_3 - V_2(s^1) \right] \\ &= 0 + \frac{1}{10} \left[2 - 0 \right] \\ &= \frac{1}{5} \end{aligned}$$

And so forth.

2.4.2 TD control.

SARSA and Q-learning are both TD methods for the control problem. You may sometimes hear TD learning contrasted with SARSA and Q-learning when the technique under discussion has the properties of TD learning, but is not SARSA, nor Q-learning.

SARSA. There are many variants of SARSA, but they all take their name from the fact that they use the state (S) and action (A) at the current time step, plus the reward (R), state (S), and action (A) at the next time step.

The general SARSA control problem uses the action-value update rule from Figure 2.4, but where we always compute the action actually taken. That is, rather than computing the general case:

$$Q^{\pi}_{n+1}(s_t, a) = Q^{\pi}_n(s_t, a) + \alpha \left[r(s_t, a) + \gamma Q^{\pi}_n(s_{t+1}, a_{t+1}) - Q^{\pi}_n(s_t, a) \right]$$

we always compute

$$Q^{\pi}_{n+1}(s_t, a_t) = Q^{\pi}_n(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma Q^{\pi}_n(s_{t+1}, a_{t+1}) - Q^{\pi}_n(s_t, a_t) \right]$$

such that a_t is the action we actually observed the agent taking, and where π is some policy that can explore.

Note that the "policy improvement" within the SARSA algorithm is implicit. We have thus far been thinking about policies as strictly encapsulated in an Agents interface, but in the SARSA example, Because we always fix the action to be the observed one, as we build our policy we must be careful to ensure that adequately balances exploration and exploitation. Thus, using a simple epsilon-greedy policy is a good approach. Once we have converged, or have run for an adequate

number of steps, we can simply output a policy such that the agent always takes the action that has the highest action-value. Recall that we denote an optimal policy like so: π^*. Since we have wrapped together the prediction problem with the control problem, we will drop the super scripted π for the remainder of the discussion:

$$Q_{n+1}(s_t, a_t) = Q_n(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma Q_n(s_{t+1}, a_{t+1}) - Q_n(s_t, a_t) \right]$$

Recall that in on-policy Monte Carlo methods, we allowed agents to guide their own search, but the cost is that it may take a long time for state-values to converge. If we use an epsilon-greedy method of balancing exploration and exploitation within SARSA, then we will have a similar problem for the action-values here.

Fortunately, because we know the probability of an agent taking an action in any given state, we can replace the term $\gamma Q_n^\pi(s_{t+1}, a_{t+1})$ (i.e., the next state and then the epsilon-greedy chosen action) with the expected action value: $\gamma \mathbb{E}[Q_n(s_{t+1}, a_{t+1} \mid s_{t+1})] = \gamma \sum_{a \in \mathcal{A}} Q(a, s_{t+1}) \pi(a \mid s_{t+1})$.

If we assume we are using an epsilon-greedy policy to choose our actions, and let a^* be the greedy value, then expected action value for the next state is:

$$\frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} Q(a, s_{t+1}) + (1 - \epsilon) Q(a^*, s_{t+1})$$

Now we have the update rule for expected SARSA:

$$Q_{n+1}(s_t, a_t) = Q_n(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma \left(\frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} Q_n(a, s_{t+1}) + (1 - \epsilon) Q_n(a^*, s_{t+1}) \right) - Q_n(s_t, a_t) \right]$$

Unsurprisingly, we can also implement an n-step variant of SARSA and expected SARSA. The n-step control problem via SARSA follows the same algorithm as for the n-step predication problem, except:

1. The input policy is expected to be epsilon-greedy, and
2. We take the next action, if running regular SARSA, and compute the expected action value if running expected SARSA, and
3. The update rule is for the action value, not the state value.

Otherwise the algorithm proceeds in the same manner.

Example. Let's compute an iteration of SARSA, expected SARSA, 2-step SARSA, and expected 2-step SARSA. Suppose that all four learning algorithms are set to have the same random seed, so that the transition function produces the the same next state for the same input across the four agents. Let $\gamma = \frac{1}{2}, \alpha = \frac{1}{10}$, and $\epsilon = \frac{3}{5}$. Let the action value table at time n be:
There is no particular significance to these numbers (and in fact with fractional parameter settings, they are almost certain impossible numbers). Note that in the general case, we would *not* expect the action value table to be the same for every method. We use the same table (Table 2) here for convenience. Table 3 gives the update for step $n + 1$.

	s^0	s^1	s^2	s^∞
a^1	3	1	1	0
a^2	3	1	0	0
a^3	1	2	1	0
a^4	1	1	0	0

$Q_n(s_i, a_i)$

	s^0	s^1	s^2	s^∞
a^1	1	1	1	0
a^2	0	2	3	0
a^3	1	1	3	0
a^4	1	1	3	0

$r(s_i, a_i)$

Table 2: Left: the action-value table at time n. Right: lookup table for the reward function.

Q-learning. Q-learning refers to a family of off-policy TD control algorithms. Like SARSA, there are many variants. For Q-learning, rather than computing the general update rule for action values in temporal difference learning,

$$Q_{n+1}(s_t, a) = Q_n(s_t, a) + \alpha \left[r(s_t, a) + \gamma Q_n(s_{t+1}, a_{t+1}) - Q_n(s_t, a) \right]$$

we instead compute

$$Q_{n+1}(s_t, a) = Q_n(s_t, a) + \alpha \left[r(s_t, a) + \gamma \max_a Q_n(s_{t+1}, a) - Q_n(s_t, a) \right],$$

Note that the only difference between SARSA and Q-learning is that for Q-learning, we compute the action-value of interest by assigning the action that produces the max action-value to the second action in our algorithm (i.e., the bold "A" in SARS**A** is the action that generates the max value, not an action chosen by an arbitrary policy).

But what about importance sampling? We can use importance sampling for TD methods by taking our n-step SARSA algorithm, computing the return for the *behavior policy*, and then multiplying the TD error by the importance ratio.

Double Learning. Both SARSA and (especially) Q-learning prefer the optimal next action; due to how iterative updates work, this means that they will start their search for an optimal policy by preferring high reward state-action pairs. While all of RL promises to converge to optimal values and policies eventually, the preference for high rewards early (and thus high q-values early) causes most of these algorithms to produce overly *optimistic* estimates (the book touches on this with the discussion of Maximization Bias in the section on Double Q-learning).

Double learning essentially maintains two value estimates for every value being estimated. You can think of this as being equivalent to running two control problems in parallel and averaging their results at each step (although the algorithm in the book is sequential).

We will not cover double learning here. Optimistic value estimates are not always undesirable; they may cause faster convergence, depending on the shape of the reward function or the method used to update the policy (this is especially true for deep learning). The algorithm is a small modification to the Q-learning algorithm of Figure 7.

Standard SARSA

$$Q^{\pi}_{n+1}(s_t, a_t) = Q^{\pi}_n(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma Q^{\pi}_n(s_{t+1}, a_{t+1}) - Q^{\pi}_n(s_t, a_t) \right]$$

Expected SARSA

$$Q^{\pi}_{n+1}(s_t, a_t) = Q^{\pi}_n(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma \left(\frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} Q^{\pi}_n(a, s_{t+1}) + (1-\epsilon) Q^{\pi}_n(a^*, s_{t+1}) \right) - Q^{\pi}_n(s_t, a_t) \right]$$

k-step SARSA (Since we have been using n for iterating over data, let's use k for the step size here)

$$Q_{n+1}(s_t, a_t) = \begin{cases} Q_n(s_t, a_t), & \text{if } t < k \\ Q_n(s_t, a_t) + \alpha \left[G_{t:t+k} - Q_n(s_t, a_t) \right], & \text{otherwise} \end{cases}$$

$$\text{where } G_{t:t+k} = \sum_{i=t}^{t+k-1} \gamma^{i-t} r(s_i, a_i) + \gamma^k Q_n(s_{t+k}, a_{t+k})$$

k-step Expected SARSA

$$Q_{n+1}(s_t, a_t) = \begin{cases} Q_n(s_t, a_t), & \text{if } t < k \\ Q_n(s_t, a_t) + \alpha \left[G_{t:t+k} - Q_n(s_t, a_t) \right], & \text{otherwise} \end{cases}$$

$$\text{where } G_{t:t+k} = \sum_{i=t}^{t+k-1} \gamma^{i-t} r(s_i, a_i) + \gamma^k \left(\frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} Q^{\pi}_n(a, s_{t+1}) + (1-\epsilon) Q^{\pi}_n(a^*, s_{t+1}) \right)$$

Q-learning

$$Q^{\pi}_{n+1}(s_t, a) = Q^{\pi}_n(s_t, a) + \alpha \left[r(s_t, a) + \gamma \max_a Q^{\pi}_n(s_{t+1}, a) - Q^{\pi}_n(s_t, a) \right]$$

In all cases listed here, the update rule for the policy is:

$$\pi_{n+1}(a \mid s_t) = \begin{cases} 1.0, & \text{if } n+1 \text{ is the final iteration and } a = \arg\max_a Q_{n+1}(s_t, a) \\ 0.0, & \text{If } n+1 \text{ is the final iteration and } a \neq \arg\max_a Q_{n+1}(s_t, a) \\ 1-\epsilon & \text{if } n+1 \text{ is not the final iteration and } a = \arg\max_a Q_{n+1}(s_t, a) \\ \epsilon & \text{if } n+1 \text{ is not the final iteration and } a \neq \arg\max_a Q_{n+1}(s_t, a) \end{cases}$$

Note that the update rule for double learning is slightly different (i.e., it is the average of the two estimates).

Figure 5: Update rules for the control problem (finding the optimal value function) for the temporal difference (TD) family of algorithms.

Solving for $Q_{n+1}(\mathbf{s}^0, \mathbf{a}^1)$; $\mathbf{r}_{t+1} = 1$ $s_{t+1} = \mathbf{s}^1$, unless otherwise noted.

Standard SARSA

Let the internal policy choose the greedy action. Then $a_{t+1} = \mathbf{a}^3$.

$$\begin{aligned}
&= Q_0(\mathbf{s}^0, \mathbf{a}^1) + \alpha \left[\mathbf{r}_1 + \gamma Q_0(\mathbf{s}^1, \mathbf{a}^3) - Q_0(\mathbf{s}^0, \mathbf{a}^1)\right] \\
&= 3 + \frac{1}{10}\left[1 + \frac{1}{2} \cdot 1 - 3\right] \\
&= \frac{57}{20} = 2.85
\end{aligned}$$

Expected SARSA

$$\begin{aligned}
&= Q_n(\mathbf{s}^0, \mathbf{a}^1) + \alpha \left[\mathbf{r}_1 + \gamma \left(\frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} Q_n(\mathbf{s}^1, a) + (1-\epsilon) Q_n(\mathbf{s}^1, a^3)\right) - Q_n(\mathbf{s}^0, \mathbf{a}^1)\right] \\
&= 3 + \frac{1}{10}\left[1 + \frac{1}{2}\left(\frac{3}{5} \cdot \frac{1}{3}(1+1+2+1) + \frac{2}{5} \cdot 2\right) - 3\right] \\
&= 3.21
\end{aligned}$$

Suppose we use the 2-step subsequence from trajectory H_1 for the 2-step SARSA variants:
$\mathbf{s}^0$ $\mathbf{a}^1$ 1 $\mathbf{s}^1$ $\mathbf{a}^1$ 1 $\mathbf{s}^1$ $\mathbf{a}^2$

2-step SARSA

$$\begin{aligned}
G_{n:n+1} &= R_1 + \gamma R_2 + \gamma^2 Q_n(\mathbf{s}^1, \mathbf{a}^2) \\
&= 1 + \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 1 \\
&= 1.75 \\
Q_{n+1}(\mathbf{s}^0, \mathbf{a}^1) &= Q_n(\mathbf{s}^0, \mathbf{a}^1) + \alpha \left[G_{n:n+1} - Q_n(\mathbf{s}^0, \mathbf{a}^1)\right] \\
&= 3 + \frac{1}{10}\left[1.75 - 3\right] \\
&= 3.125
\end{aligned}$$

2-step expected SARSA

$$\begin{aligned}
G_{n:n+1} &= R_1 + \gamma R_2 + \gamma^2 \left(\frac{\epsilon}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} Q_n(\mathbf{s}^1, a) + (1-\epsilon) Q_n(\mathbf{s}^1, a^3)\right) \\
&= 1 + \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot \left(\frac{3}{5} \cdot \frac{1}{3}(1+1+2+1) + \frac{2}{5} \cdot 2\right) \\
&= \frac{39}{20} \\
Q_{n+1}(\mathbf{s}^0, \mathbf{a}^1) &= Q_n(\mathbf{s}^0, \mathbf{a}^1) + \alpha \left[G_{n:n+1} - Q_n(\mathbf{s}^0, \mathbf{a}^1)\right] \\
&= 3 + \frac{1}{10}\left[\frac{39}{20} - 3\right] \\
&= 2.895
\end{aligned}$$

Table 3: Applications of SARSA variant update rules. Each row estimates the action value $Q(s^0, a^1)$ for the next time step. The action value table in 2 would be updated with each of these values.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 6: SARSA for the control problem reproduced from Sutton and Barto p. 130, 2nd ed. This algorithm estimates the action-value function (hence the name), and can be modified to return a policy by returning a map from each state to the action that has the highest value. (Note that while the algorithm is producing action-value estimates and thus looks like a prediction algorithm, the construction of this algorithm implicitly follows the optimal policy – this is the implication behind "policy derived from Q.")

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 7: Q-Learning for the control problem reproduced from Sutton and Barto p. 131