Notes on Reinforcement Learning

Kian Kenyon-Dean

March 15, 2018

1 Fundamentals

The essence of Reinforcement Learning (RL) is build an *agent* that can learn how to act *optimally* (or, good enough) in some *environment* by being able to learn from its *experience* in the environment. An introduction into RL begins with an exploration into the more simple problem setting of **bandits**, continues onto **dynamic programming** (where learning can occur without "experiencing" the environment, so long as a perfect model of the environment is present). The essence of RL thus begins when knowledge truly is obtained from an agent's *experience* in the world, also known as *sampling*. We begin with a brief overview of the introductory methods and an exposition of **Finite Markov Decision Processes**, after which we can truly begin to engage with RL.

1.1 The General Update Rule

We first pose the general form of the update rule for estimating things, which is a universal formula that is applied in many situations in RL; it's usefulness will be elucidated by examples. Very generally, we are attempting to form an *estimate* E of *something*, incrementally through time. This *something* generates a *target* value U_t at each step t of incrementation. We update our *new estimate* E_{t+1} of that thing by a step-size $\alpha \in \mathbb{R}$, based on the difference between that *target* U_t and our old estimate E_t.

$$E_{t+1} \leftarrow E_t + \alpha[U_t - E_t] \tag{1}$$

Overtime, we would like for the difference between our estimate and the target to decrease, until we eventually can perfectly estimate U_t with E_t, regardless of the time step. Most of the time, we assume U_t is a stochastic function that will always randomly sample values from an unknown distribution; in this case, our estimate will not be of the function itself, but of the *expected value* of the function, $\mathbb{E}[U_t]$. How do we know if we will be able to approximate the true expected value of U_t? If we generalize our step-size α to be a function of time, $\alpha(t)$, then, stochastic approximation theory tells us that the following conditions on $\alpha(t)$ must be met if we want to be guaranteed to converge to $E = \mathbb{E}[U_t]$ (given an infinite amount of time):

$$\sum_{t=1}^{\infty} \alpha(t) = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \alpha^2(t) < \infty \tag{2}$$

In layman's terms, the first condition says the step-size must change relatively fast; the second says that the step-size cannot change too fast. For example, $\alpha_t = \frac{1}{t}$ satisfies these conditions; it also expresses the definition of the incremental version of an *equally-weighted average*. This would be a useful condition for a *stationary problem* to be satisfied. However, if the environment changes over time (i.e., if $U_t(\cdot)$ samples from different distributions depending on t), this is not

a desirable condition. Keeping $\alpha(t) = \alpha$ to be a constant value does not satisfy these conditions, but this is actually desirable if the problem is *non-stationary* in this way. This defines what is called the *exponential recency-weighted average*, where new experiences are more valued than past experiences when forming our estimate.

1.2 A Brief Overview of Dichotomies in RL

There are many different strains of RL, and often times an algorithm arises from selecting an option from these dichotomies, where different algorithms and problem formulations are more appropriate for certain settings than others.

- ◇ *Exploration* vs *Exploitation*: do we seek to explore the possible benefits of going to other states, or should we exploit our current knowledge of the values of states in order to maximize the reward we expect to obtain?
- ◇ *Continuing* vs *Episodic*: does a task end after a finite number of steps or does it go on forever? This dichotomy lends itself to different formulations of algorithms, but they can be generalized with clever use of notation. It turns out that discounting is not necessary in the continuing setting, where instead *average reward* is used to compare against.
- ◇ *Global* vs *Sampling*: are we attempting to learn based on complete knowledge of the MDP? If so, then we need not learn from experience, rather we can just solve the system of equations. However, in most problems we do not possess the scope of the MDP, or it is much to large; this fact necessitates the use of *sampling*, of using an agent's experience in the MDP to learn and make judgements about the MDP and the best way to act within it. DP vs MC and TD.
- ◇ *Prediction* vs *Control*: given a policy π - a way of choosing actions within an MDP - we may want to know, *how good is this policy*? We may also want to know, *how can we make this policy better, if it is not the best*? State value functions vs state-action value functions.
- ◇ *Bootstrapping* vs *Not bootstrapping*: we are approximating the values of states and/or state-action pairs. Do we make these approximations by factoring in our current approximations in addition to whatever reward we obtain, or do we only factor in the reward we obtain without biasing ourselves to our previous approximations? DP and TD vs MC.
- ◇ *On-* vs *Off-policy*: do we learn about the policy generating behavior, or do we learn about another policy while following the behavior of a different policy? SARSA vs Q-learning.
- ◇ *1-Step* vs *N-step*: when approximating values, do we seek to update after only one step, or do we want to wait n-steps before updating? The former will be faster but more biased, while the latter will be slower but more "true" in the sense of adhering more to the true expected returns of the MDP. TD(0) and DP vs n-step TD and MC.
- ◇ *Forward-view* vs *Backward-view*: are we going to define our value approximations by waiting until we see the results of our actions, or will we retroactively define them based on hindsight looking at our previous actions and the current state we are in as a result of them? n-step TD and λ-return TD vs TD(λ) and MC.
- ◇ *Tabular* vs *Function Approximation*: will we seek to treat each state (or state-action pair) completely separately, or will we attempt to generalize states such that an update based on one state will affect the values for other similar states?
- ◇ *Model-based* vs *Model-free*: do we know the dynamics of the MDP or not? Should we attempt to model the dynamics directly based on our experience in the MDP?

2 Multi-Arm Bandits

The characteristic feature of bandits is that they assume an environment where only a single action can be taken, where actions taken do not affect the state of the environment. In other words, they are single-step episodic tasks. Whenever an action is taken, a reward will be given, some scalar value. The objective is thus to figure out which of the k actions is the *best* to take; i.e., which action will maximize the amount of reward I get over time, my expected reward?

Formally, we are given a set of actions $a \in \mathcal{A}$, and some unknown stochastic reward function $r(a) \in \mathbb{R}$. Whenever an action a is selected, we will receive a reward value of $r(a)$. Therefore, if we seek to maximize the expected reward we get, we really want to be able to approximate $r(a), \forall a$. Let us declare our approximation of $r(a)$ to be $q(a)$, which will not approximate the function itself, but rather the expected value of the function. The true expectation of the reward is captured by the ideal approximation $q^*(a)$, represented as an expectation of the reward function over time steps t with actions taken being A_t:

$$q^*(a) = \mathbb{E}[r(A_t)|A_t = a] \tag{3}$$

How might we approximate $q^*(a)$? Well, we will approximate it by attempting to learn from experience, by pulling the arms, seeing what happens, and adjusting our estimate of the reward for those actions accordingly. The following equation defines the update rule, given some action a is taken at time t, and a step-size function $\alpha(t)$:

$$q_{t+1}(a) \leftarrow q_t(a) + \alpha(t)[r(a) - q_t(a)] \tag{4}$$

This update is sufficient for all bandit problems. The real question that arises is *how to select an action*? If we want to maximize the accumulated reward, then we must balance *exploration* and *exploitation*. On the one hand, we want solid estimates of the values of actions; on the other hand, we want to maximize our accumulated reward, which means exploiting our current knowledge and selecting the action we think is best. There are several methods for performing action selection and balancing this trade-off.

Epsilon-Greedy Let the *greedy* action be defined as the action that has the maximal approximate expected value; i.e., $A_t = \text{argmax}_a\, q(a)$. If our values are correct, then naturally we'd exploit our knowledge and only perform this action. However, if we are not confident about our values, then we want to occasionally test other actions in order to better approximate their expected values. Thus, with a probability ϵ, we will select an action randomly.

Upper-Confidence-Bound (UCB) UCB selects actions for the purpose of exploration with a bit more sophistication than the above method. We will instead randomly select actions based on how good we think they are, in addition to accounting for the amount of time that has passed, and the number of times we have selected that action $N_t(a)$:

$$A_t = \underset{a}{\text{argmax}} \left(q(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right) \tag{5}$$

Here, $c > 0$ controls the degree of exploration we pursue. Additionally, if an action hasn't been selected yet (i.e., $N_t(a) = 0$) then we will take that action. The square root term measures the uncertainty or variance in our estimate of $q(a)$.

Gradient Bandit In gradient bandit algorithms, we are not concerned with approximating the values of actions; rather, we seek to model a numerical *preference* for taking an action a at time t, $H_t(a)$. Using this, we define our first notion of a *policy*, $\pi_t(a)$, which models the probability that we take an action at that time. Gradient bandits define their policy using a softmax distribution:

$$\pi_t(a) = \frac{\exp H_t(a)}{\sum_{b \in \mathcal{A}} \exp H_t(b)} \tag{6}$$

The learning algorithm is based on stochastic gradient ascent, with updates based on if the action selected at time t is $a = A_t$, where other, non-selected actions b are updated slightly differently. We assume a reference or baseline reward $\overline{R}_t$ is defined; often it can be updated and maintained with an update-based averaging of the rewards obtained over time:

$$\begin{aligned} H_{t+1}(a) &\leftarrow H_t(a) + \alpha(R_t - \overline{R}_t)(1 - \pi_t(a)) \qquad (a = A_t) \\ H_{t+1}(b) &\leftarrow H_t(b) - \alpha(R_t - \overline{R}_t)\pi_t(b) \qquad\quad (b \neq A_t) \end{aligned} \tag{7}$$

3 Finite MDPs

A Finite Markov Decision Process is defined by $\{\mathcal{A}, \mathcal{S}, P(\cdot) \in \mathbb{R}^{|\mathcal{S}|}, r(\cdot) \in \mathbb{R}, \gamma\}$; P and r are understood as function mappings, as we see below:

- $\diamond$ $\mathcal{A}$: the set of all m possible actions a that can be taken.
- $\diamond$ $\mathcal{S}$: the set of all n states s the environment can be in.
- $\diamond$ $P(s, a) \in \mathbb{R}^n_{\Pr}$: a function that maps from state-action tuples to a distribution vector over states. Namely, $\forall s, a$, we have $P(s, a) = \Pr\{s' | s, a\}$, a vector of transition probabilities to all states s', given that we were in state s and took action a.
- $\diamond$ $r(s, a) \in \mathbb{R}$: a function that maps from state-action tuples to a real-valued reward. Oftentimes, this is a stochastic, non-stationary function; it may thus sometimes be necessary to indicate it by the current time step t in such cases. Most of the time, it is unknown and we will seek to approximate it by attempting to model its expected value.
- $\diamond$ γ: the discount factor (will be elucidated later).

Any agent that seeks to act in the world of our MDP will need to select actions at each time step t. Let the selected action at time step t be denoted A_t, and let the state the agent is in at time step t be denoted S_t. Our agent seeks to maximize reward over time, called the return G. This is done by maximizing the *expected return*; the lower equation models the expected return starting at time step t, ending at some time step T (which is finite for episodic task, and infinite for continuing tasks):

$$\begin{aligned} \mathbb{E}[G] &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t)] \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1 - \gamma} \\ G_t &= \sum_{k=t}^{T} \gamma^{k-t} r(S_k, A_k) \end{aligned} \tag{8}$$

Above we pose the expected return with respect to the absolute maximum possible return, based on the maximal reward value that can be attained, $R_{max} = \max_{s,a} r(s, a)$. Of course we now must determined which actions we will take, and which states we will find ourselves in. This brings us to the concept of a **policy**.

3.1 A Policy in an MDP

A policy $\pi(s)$ is a function mapping from $\mathbb{R}^n \to \mathbb{R}^m_{\text{Pr}}$, from states to a probability distribution over actions. Most of the time, however, we will use the policy to denote a specific probability of selecting an action a in state s, indicated as $\pi(a|s) \in \mathbb{R}$; this is only for notational convenience, it simply indicated selecting the value in the probability vector corresponding to action a; thus, we have $\sum_{a \in \mathcal{A}} \pi(a|s) = 1$. The following questions immediately arise upon introducing this concept of a policy:

- ◇ **Prediction:** If an agent follows policy π within the MDP, what is the expected return it will achieve? I.e., what is $\mathbb{E}_\pi[G]$?
- ◇ **Control:** What is the optimal policy π^* that obtains the maximum possible expected return for the MDP? I.e., what is $\text{argmax}_\pi \mathbb{E}_\pi[G]$?

We model expected return by posing the question within a slightly more specific, more useful setting. We pose the question: what is the expected return an agent will get by following π *given that it starts in state* s? This is denoted as the ***value*** *of a state under* π, $v_\pi(s)$.

3.2 Bellman Equations

Given knowledge of the MDP, the prediction question is answered with complete certainty using the recursive definition induced by the **bellman equation**. It is naturally recursive since the value of state must be based on how the policy will act in the states following it.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \sum_{a \in \mathcal{A}} \pi(a|s) \Bigg(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_\pi(s') \Bigg) \tag{9}$$

The control problem becomes answered by first posing the action-value function $q_\pi(s,a)$, which models the expected return from following policy π, *after taking some action* a *in state* s, where a is not necessarily the *greedy* action induced by the policy. We define this now, but its role for control will be seen later on.

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_\pi(s') \tag{10}$$

We have thus defined the bellman equations for value functions and action-value functions in a finite MDP. It is more elucidating to pose them in their linear algebraic formulations, upon which we will immediately recognized that the bellman equation actually defines a linear system of n equations with exactly n unknowns, whereupon an exact solution to the prediction problem will be determined.

Let us define the following two constructs induced by policy π: the expected[1] state reward vector $\mathbf{r}_\pi \in \mathbb{R}^n$, and the state transition matrix $\mathbf{P}_\pi \in \mathbb{R}^{n \times n}$. Below we indicate the values stored in these objects at the indices induced by the state in parentheses.

$$\begin{aligned} \mathbf{r}_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) r(s,a) \\ \mathbf{P}_\pi(s,s') &= \sum_{a \in \mathcal{A}} \pi(a|s) P(s'|s,a) \end{aligned} \tag{11}$$

[1]Note that this means that we assume $r(s,a)$ returns the *expected* reward from using action a in state s; an abuse of notation, where a truly articulate notation would be $\mathbb{E}[r(s,a)]$, which we do not use for the sake of notational brevity.

The matrix form of the bellman equation for the state-value function defines vector $\mathbf{v}_\pi \in \mathbb{R}^n$:

$$\begin{aligned} \mathbf{v}_\pi(s) &= \mathbf{r}_\pi(s) + \gamma \sum_{s' \in \mathcal{S}} \mathbf{P}_\pi(s, s') \mathbf{v}_\pi(s') \\ \mathbf{v}_\pi &= \mathbf{r}_\pi + \gamma \mathbf{P}_\pi \mathbf{v}_\pi \end{aligned} \tag{12}$$

Since $\mathbf{v}_\pi$ is unknown, we have thus defined an "easily solvable" linear system of equations, where the direct analytical solution for $\mathbf{v}_\pi$ is:

$$\begin{aligned} \mathbf{v}_\pi - \gamma \mathbf{P}_\pi \mathbf{v}_\pi &= \mathbf{r}_\pi \\ (\mathbf{I}_n - \gamma \mathbf{P}_\pi) \mathbf{v}_\pi &= \mathbf{r}_\pi \\ \mathbf{v}_\pi &= (\mathbf{I}_n - \gamma \mathbf{P}_\pi)^{-1} \mathbf{r}_\pi \end{aligned} \tag{13}$$

4 Dynamic Programming

Dynamic programming assumes the environment's dynamics are completely known. The linear systems formulation of the prediction problem is oftentimes much too expensive since it involves a matrix inverse of a large $n \times n$ matrix. Instead, we can iteratively perform *policy evaluation* for π, or prediction, directly using the bellman equations for state-value functions:

$$v_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \left(r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) v_k(s') \right) \tag{14}$$

This iterative algorithm is guaranteed to converge to the true v_π as long as $\gamma < 1$ or if the task is guaranteed to terminate. This is called an *expected update* because the update is based on all the successor states of s, all possible one-step transitions. This is why the backup diagram for dynamic programming looks like a tree that extends from one state to all possible next states.

We can now define *policy improvement*, a piece to the control algorithm that seeks to discover the optimal policy for the MDP. We will use our definition of the action-value function $q_\pi(s, a)$, which models the expected value of a state given that we take action a, and then follow π afterwards. We can thus define the *policy improvement theorem*:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s), \ \forall s \implies v_{\pi'}(s) \geq v_\pi(s), \ \forall s \tag{15}$$

If this property is satisfied for a policy π', then π' is strictly better than π. Note that this is defined with respect to the *true* value functions.

We define the greedy policy to be $\pi'(s) = \text{argmax}_a \, q_\pi(s, a)$. We now define *policy iteration*, which will converge to the optimal policy of the MDP.

1. Randomly initialize π
2. Use *policy evaluation* to compute $v_\pi(s) \ \forall s$
3. For each $s \in \mathcal{S}$:
   - ◇ Store a_{old} to be $\pi(s)$
   - ◇ Update $\pi(s) \leftarrow \text{argmax}_a \, q_\pi(s, a)$
   - ◇ If $a_{old} \neq \pi(s)$, then we have not reached the optimal policy
4. If we have not reached the optimal policy, go back to 2
5. Otherwise, we converged to the optimal policy $\pi = \pi^*$ and the optimal value function

4.1 Value Iteration

While the policy iteration algorithm works in theory, it is extremely expensive due to the sheer quantity of policy evaluations it must do. We can optimize this significantly by combining policy improvement and policy evaluation by basing our values on a greedy policy's action, rather than a stochastic policy's distribution over actions. We thus define:

$$v_{k+1}(s) \leftarrow \max_a r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_k(s') \tag{16}$$

If we run this iteration until the difference between updates is arbitrarily small, we will obtain the optimal value function, and the optimal policy will be defined as the greedy policy over this value function. This works due to its use of the greedy action with respect to itself; it updates its values, which then updates its policy, which then will in turn update the values again, until convergence is reached. This is a specific instance of *generalized policy iteration* (GPI).

5 Monte Carlo Methods

The characteristic features of Monte Carlo methods (MC) are: (1) they learn from sample experience, meaning they directly learn values without a model; and, (2) they do not bootstrap, their value estimates are based on real returns, not on their other value estimates.

MC performs the prediction problem very simply. Given π, generate an episode, starting from S_0 all the way to the final state S_T using actions obtained with π. Then, update the values of the states in reverse by accumulating the return at each step ($G \leftarrow G + r(s_t, a_t)$) and update $v(S_t) \leftarrow \text{average}(G, PreviousReturns(S_t))$. As the number of episodes approaches infinity, this method is guaranteed to converge to the true v_π, since each return is an IID estimate of v_π.

The real difficulty with MC is in estimating action-values for the purpose of control. This is because exploration can only really be imposed in the beginning of MC, where we use the method of exploring starts. One way is to specify a GPI method, where we alternate between evaluation and improvement episodically. That is, not only do we update the action-values in an analogous way as with state-values (above), but also, after each update to the value of state S_t, additionally update the policy at S_t to be $\pi(S_t) = \text{argmax}_a\, q(S_t, a)$. If each state-action pair in $\mathcal{S} \times \mathcal{A}$ has a nonzero probability of being selected to start from, we will be guaranteed to converge to the optimal policy (in the limit to infinity).

5.1 On-Policy MC Control

Exploration can be imposed to learn more about the true values of the current policy (and then improve it) by using ϵ-soft policies, where the greedy action of the policy at state S_t, A_t^* is given probability $\pi(A_t^*, S_t) = 1 - \epsilon + \frac{\epsilon}{m}$, and all other actions a in S_t receive probability $\pi(a, S_t) = \frac{\epsilon}{m}$. This is the simple way to ensure some sort of exploration in MC throughout its trajectory without relying on exploring starts. Its disadvantage is that it weights all other actions equally, while some may be known to be much worse than others; so, we'd instead like to take actions proportionally to their estimated values, which can be done using *Boltzmann exploration* (not discussed here).

5.2 Off-Policy MC Prediction and Control

In *off-policy learning*, we seek to learn the values for a *target policy* π, given behavior generated by a *behavior policy* b. To do this, the *coverage* property must be satisfied: $\pi(a|s) > 0 \implies b(a|s) > 0$. Oftentimes, π is the deterministic greedy policy with respect to q, and b is a stochastic and exploratory policy, such as ϵ-greedy.

Off-policy methods require the use of *importance sampling* to be able to approximate the values of π. We declare the following importance-sampling ratio:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{17}$$

Given this ratio, we have two ways to approximate v_π. Let $\mathcal{T}(s)$ be the set of all time steps in which state s is visited, $T(t)$ the first time of termination following time t, and G_t the return after t up through $T(t)$. We thus define *ordinary* and *weighted importance sampling*.

$$\begin{aligned} v(s) &= \frac{1}{|\mathcal{T}(s)|} \sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t \\ v(s) &= \frac{1}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}} \sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t \end{aligned} \tag{18}$$

Ordinary IS is unbiased, but has unbounded variance, meaning it is quite finicky to use in practice. Weighted IS is biased, but has dramatically lower variance, making it much preferred and easier to use, despite its inherent bias.

The action-value update using IS is defined below, from which we can easily define prediction and control algorithms. Let $f(\rho_{t:T-1})$ be a function that either converts ρ to weighted or ordinary importance sampling. Note that, if an action a is selected by b such that $\pi(S_t, a) = 0$, then learning is halted for that episode, since ρ will be equal to zero from then down to the beginning of time.

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + f(\rho_{t:T-1})[G - q(S_t, A_t)] \tag{19}$$

6 Temporal-Difference Learning

TD learning is a central idea for RL. It is based on *bootstrapping* and *sampling*, combining the two elements of DP and MC. The general update rule for 1-step TD (or, TD(0)) is:

$$v(S_t) \leftarrow v(S_t) + \alpha[r(S_t, A_t) + \gamma v(S_{t+1}) - v(S_t)] \tag{20}$$

This is directly related to the Bellman equation for an MDP, where the values are recursively defined with respect to the values of other states. However, while the Bellman equation (and DP) work with a sweep over all possible next states, TD only works with the next state generated by experience, since it uses sampling and experience to learn and does not assume a model of the MDP. TD(0) is distinct from MC in terms of what it is theoretically trying to do. MC attempts to minimize the mean-squared error on the training set (its experience); TD(0), on the other hand, finds estimates that would be exactly correct for the maximum-likelihood model of the Markov process.

6.1 SARSA: On-policy Control

Given that our policy has generated the current state and action and the next state and action (a quintiple, state-action-reward-state-action), we define the SARSA update as follows, where after an update the policy is then derived from q (possibly using ϵ-greedy):

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[r(S_t, A_t) + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)] \tag{21}$$

A variant of this algorithm, called *Expected Sarsa*, augments the update target to be as follows. This eliminates the variance imposed by the random selection of A_{t+1}:

$$r(S_t, A_t) + \gamma \sum_a \pi(a|S_{t+1}) q(S_{t+1}, A_{t+1})$$

6.2 Q-Learning: Off-policy Control

This method directly approximates the optimal action-value function of the MDP, independent of the behavior policy. We derive the behavior policy directly from q after each update, possibly using ϵ-greedy.

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[r(S_t, A_t) + \gamma \max_a q(S_{t+1}, a) - q(S_t, A_t)] \tag{22}$$

6.3 N-Step TD

We now generalize TD to account for the next n-steps, rather than just one step look-ahead. Below we define the complete MC return of an episode that ends at time step T, G_t; the one-step TD return, $G_{t:t+1}$; and, the generalized n-step TD return $G_{t:t+n}$:

$$\begin{aligned} G_t &= \sum_{k=t}^{T} \gamma^{k-t} r(S_k, A_k) \\ G_{t:t+1} &= r(S_t, A_t) + \gamma v_t(S_{t+1}) \\ G_{t:t+n} &= \left(\sum_{k=t}^{n-1} \gamma^{k-t} r(S_k, A_k) \right) + \gamma^n v_{t+n-1}(S_{t+n}) \end{aligned} \tag{23}$$

Note that, if $t + n >= T$, then $G_{t:t+n} = G_t$. The only difference between this and TD(0) is the definition of the update target; in all the above algorithms, simply replacing the update target with this $G_{t:t+n}$ is sufficient for converting them to N-step algorithms. Note that an off-policy algorithm naturally needs to factor in the importance sampling ratios, if we are not using Q-learning. N-step TD factors in more experience and has to wait longer before updating values; this wait occurs because this approach encompasses the *forward view* of RL, that values should be updated with respect to what will happen in the future.

6.4 TD(λ)

TD(λ) generalizes n-step TD to a nicer, more elegant algorithm. While n-step TD is equivalent to basic one-step TD when $n = 1$, and MC when $n = T$, TD(λ) offers a continuous formulation: TD($\lambda = 0$) is equivalent to one-step TD, and TD($\lambda = 1$) is equivalent to MC. This algorithm is more rigorously defined in the context of function approximation, whereas until now we have been working in the tabular setting of RL. While tabular RL is in fact a specific instance of function approximation (where feature vectors are simply one-hot encoding of states), it is necessary to first elucidate function approximation before exploring this variant of TD.

7 Function Approximation

Let $\mathbf{x}(S_t) = \mathbf{x}_t$ be a d-dimensional feature vector encoding of a state S_t. There are many ways to extract features from a state, but for now we assume this method is already defined. The purpose of FA is to reduce the computational complexity of RL, since $d << n$; additionally, FA allows us

to generalize from experience in certain states to approximate the values of similar states. We are considered with the case of linear FA in this section, where we seek to learn a weight vector $\mathbf{w}$ such that the value of a state is defined:

$$v(S_t, \mathbf{w}) = \mathbf{w}^T \mathbf{x}_t \tag{24}$$

Note that the gradient of this value function with respect to the weights is simply the features; i.e., $\nabla v(S_t, \mathbf{w}) = \mathbf{x}_t$. The general form of stochastic gradient descent for optimizing the linear value function in RL is the following, where specific algorithms augment this update by offering a definition of the update target U_t:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[U_t - v(S, \mathbf{w})]\mathbf{x}_t \tag{25}$$

MC defines $U_t = G_t$ the return at the end of the episode. n-step TD defines $U_t = G_{t:t+n}$, and specifically 1-step TD defines $U_t = r(S_t, A_t) + \gamma v_t(S_{t+1}, \mathbf{w})$. When U_t includes a bootstrapped target, as with TD, we call this method a *semi-gradient* algorithm since it is not based on true samples generated from the distribution, which is what is defined by *stochastic gradient descent* (SGD). While MC is true SGD, it is typically much slower than TD since it takes so long to update its values.

7.1 Closed-Form Solution to Linear TD(0)

With the update law defined above, consider a weight vector update at time $t + 1$:

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\big(r(S_t, A_t) + \gamma \mathbf{w}_t^T \mathbf{x}_{t+1} - \mathbf{w}_t^T \mathbf{x}_t\big)\mathbf{x}_t \\ &= \mathbf{w}_t + \alpha\big(r(S_t, A_t)\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T \mathbf{w}_t\big) \\ &= \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t) \end{aligned} \tag{26}$$

Where $\mathbf{b} \in \mathbb{R}^d$ and $\mathbf{A} \in \mathbb{R}^{d \times d}$ are linear operators defined as the expectations of the quantities they replace in the equation above them. With this fact, we find the closed form solution to the *TD fixed point* weight vector, which TD(0) ultimately converges to:

$$\begin{aligned} \mathbf{w} &= \mathbf{w} + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}) \\ \mathbf{0} &= \mathbf{b} - \mathbf{A}\mathbf{w} \\ \mathbf{w} &= \mathbf{A}^{-1}\mathbf{b} \end{aligned} \tag{27}$$

7.2 TD(λ)

We can now fully define TD(λ). This algorithm encompasses a *backward-view* of RL, where updates to the weight vector occur at each step of the algorithm, rather than waiting for a certain amount of steps to be pursued. An *eligibility trace* vector $\mathbf{z}_t = \gamma\lambda \mathbf{z}_{t-1} + \mathbf{x}_t$ is defined, where $\mathbf{z}_{-1} = \mathbf{0}$. With TD error defined with the specific one-step TD error, but we will find that this algorithm is not one-step due to its use of eligibility traces. The error is defined as $\delta_t = r(S_t, A_t) + \gamma v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w}_t)$. The updates for the entire algorithm become:

$$\begin{aligned} &\mathbf{z} \leftarrow \gamma\lambda \mathbf{z} + \mathbf{x}_t \\ &\delta \leftarrow r(S_t, A_t) + \gamma v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w}) \\ &\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z} \end{aligned} \tag{28}$$

The eligibility trace captures exactly the amount a state has contributed to the current trajectory and thus the current error/reward attained. Over time, the vector accumulates all the states visited in the trajectory, causing each update to update the value function's weights with respect to more and more of the states considered.

The forward view using n-step λ returns redefines the return update target with the equation to horizon h below. This is simply a way to augment n-step methods to account for not just the last step, but to factor in all other steps k between 1 and n by a proportion factoring $(1-\lambda)\lambda$. n-step methods with FA suffer from the fact that they have to store all weight vectors between time t and $t+n$, and is neither interesting nor on par with the strength of the the *backward-view*.

$$\begin{aligned} G^{\lambda}_{t:h} &= (1-\lambda)\sum_{n=1}^{h-t-1}\lambda^{n-1}G_{t:t+n} + \lambda^{h-t-1}G_{t:h} \qquad 0 \le t < h \le T \\ =G^{\lambda}_{t:t+k} &= v(S_t,\mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1}(\gamma\lambda)^{i-t}(r(S_{i+1},A_{i+1}+\gamma v(S_{i+1},\mathbf{w}_i) - v(S_i,\mathbf{w}_{i-1})) \end{aligned} \tag{29}$$

7.3 Action-value Function Approximation

FA with action-values $q(s,a)$ is nearly equivalent to the state-value FA already described, except that it becomes necessary to define a feature vector representing both states and actions combined. The methods for SARSA, Q-learning, and the rest can all be extended to this case with obvious rewrites of their functions in terms of function approximation. Control is implicitly performed whenever the weight vector is updated since it operates over the approximate state-action value function and thus can modify the (ϵ-)greedily selected action at each step, unlike in prediction where a policy is assumed and the weight vector is only updated to modify the value of states under the fixed policy.