Blueprints for building reinforcement learning algorithms in customer-facing applications

Douglas Mason[1]

[1]Koyote Science, LLC *

November 2021

1 Introduction

This document outlines primary considerations for designing production-worthy reinforcement learning algorithms in customer-facing settings. We will (1) establish the notation, (2) discuss how policies are evaluated, (3) show how policies can be defined by evaluation functions alone, (4) demonstrate how we train an evaluation for our policy, even from customer interactions driven by another policy, known as off-policy training, (5) show how we can train the policy directly from customer interactions, and (6) demonstrate how we can combine both approaches for the current state-of-the-art in the field. We also (7) provide practical considerations under the hood for building our policies and evaluation functions, as well as (8) additional resources to help the reader dive more deeply into the field, as well as discover implementations that can be used right now. This document attempts to be brief but mathematically thorough. Please write to the author at douglas@koyotescience.com for questions or feedback. We provide a quick schematic of the algorithms we will cover in Figure 1.

2 Notation and policy evaluation

When designing products that take users through a sequence of actions to get to a desired goal, it's a natural instinct to look at the field of reinforcement learning to optimize your design. In this formulation, we have a state vector $\mathbf{s}$ (e.g., the current page a user is on) which may also incorporate a context vector $\mathbf{c}$ (e.g., user demographics), an action vector $\mathbf{a}$ that represents a decision that the system can take (e.g., presenting different options along the way), and a

*http://www.koyotescience.com

[2]Source: https://towardsdatascience.com/an-overview-of-classic-reinforcement-learning-algorithms-part-1-f79c8b87e5af

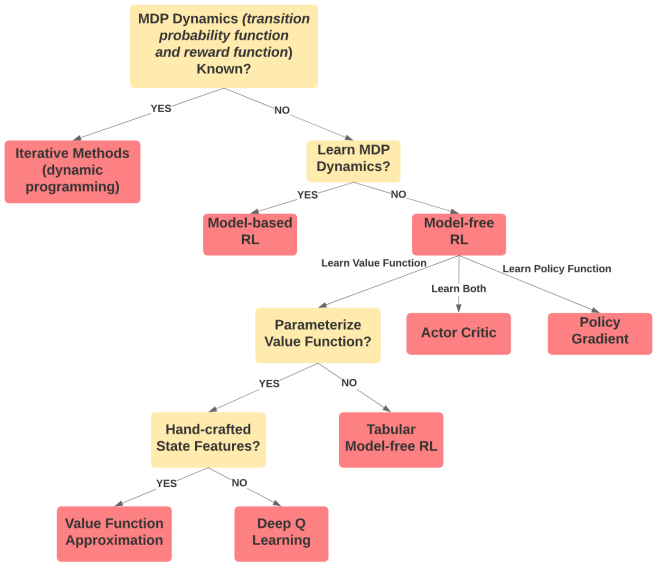

Figure 1: Schematic for the reinforcement learning approaches covered in this document.[2]

reward for taking an action at a given state $r(\mathbf{s}, \mathbf{a})$ (e.g., whether a user signs up at the end of a sign-up flow). Note that we can also define the reward without considering the action, $r(\mathbf{s})$, which is identical except that it is defined by the state you end up in to get the reward rather than the state and action you take to get it.

We are interested in learning a policy $\pi(\mathbf{a}|\mathbf{s}) \in [0, 1]$ that gives us the probability of choosing a given action $\mathbf{a}$ with a given state $\mathbf{s}$. Note that when implemented, such a function depends on both $\mathbf{a}$ and $\mathbf{s}$. The policy is normalized so that its returns for all available actions at a given state add up to one, i.e.,

$$\sum_{\mathbf{a}} \pi(\mathbf{a}|\mathbf{s}) = 1 \tag{1}$$

When learning this policy, our goal is to maximize the discounted sum of future rewards $G_t(\boldsymbol{\tau})$[3] from time step t through the problem horizon H, as we step through a given trajectory of states $\boldsymbol{\tau}$ which we write out as

$$\boldsymbol{\tau} = (\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_H, \mathbf{a}_H) \tag{2}$$

[3]The letter G is chosen for historical reasons, while $J(\pi_{\boldsymbol{\theta}}) = \mathbb{E}_{\boldsymbol{\tau} \sim \pi_{\boldsymbol{\theta}}}[G_0(\boldsymbol{\tau})]$ is used latr in Equation 27)

, and where the discount factor $\gamma \in (0, 1]$. We write this quantity out as

$$G_t(\boldsymbol{\tau}) = \sum_{t'=t}^{H} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \gamma^{t'-t} \tag{3}$$

. We can accomplish this by learning a Q-value or action-value function for each state and action, which returns the expected discounted sum of future rewards assuming we follow a policy π starting from state $\mathbf{s}$ and action $\mathbf{a}$, and can be written as

$$\begin{aligned} Q_\pi(\mathbf{s}, \mathbf{a}) &= \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \pi}[G_t(\boldsymbol{\tau}) | \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}] \\ &= \sum_{t'=t}^{H} \pi(\mathbf{a}_{t'} | \mathbf{s}_{t'}) r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \gamma^{t'-t} \end{aligned} \tag{4}$$

.Other formulations may work with a state-value function

$$\begin{aligned} V_\pi(\mathbf{s}) &= \sum_{\mathbf{a} \sim \mathcal{A}} \pi(\mathbf{a}|\mathbf{s}) Q_\pi(\mathbf{s}, \mathbf{a}) \\ &= \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \pi}[G_t(\boldsymbol{\tau}) | \mathbf{s}_t = \mathbf{s}] \\ &= \sum_{t'=t}^{H} \sum_{\mathbf{a} \sim \mathcal{A}} \pi(\mathbf{a}_{t'} | \mathbf{s}_{t'}) r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \gamma^{t'-t} \end{aligned} \tag{5}$$

which only depends on the state.

3 Deriving the policy from the evaluation

The policy can be determined by the Q-value using a greedy strategy

$$\pi(\mathbf{a}|\mathbf{s}) = \mathbb{1}\left[\mathbf{a} = \arg\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}))\right] \tag{6}$$

where $\mathbb{1}[\cdot]$ returns a 1 when the argument is true and 0 otherwise. This policy is deterministic, but we can make a stochastic policy using a variety of other strategies. In ϵ-greedy, the greedy action is chosen with a fixed probability (say, 75%), and other actions are chosen at random for the remaining 25% of the time. Another strategy is to use the softmax function

$$\pi(\mathbf{a}|\mathbf{s}) = \frac{\exp(Q(\mathbf{s}, \mathbf{a}))}{\sum_{\mathbf{a}} \exp(Q(\mathbf{s}, \mathbf{a})} \tag{7}$$

And yet another strategy is to perform gradient ascent against the Q-value (when such gradients are available) to maximize it over the action space, which is used in the "deterministic policy gradient" algorithm[13]. If we use the state-value instead of the action-value, then we need to know how actions lead to state

transitions, which is outside the scope of this article, and falls under the domain of "model-based" reinforcement learning [7]. Note that the word "model" here refers to a model of the environment ($p(\mathbf{s}_{t+1}|\mathbf{s}, \mathbf{a})$ as opposed to the models that are used for the policy or state- and action-value function approximation.

If we define the policy entirely by the state-value function, then we obtain

$$\pi(\mathbf{a}|\mathbf{s}) = \mathbb{1}\left[\mathbf{a} = \arg\max_{\mathbf{a}_t} \left(r(\mathbf{s}_t, \mathbf{a}_t) + p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)V(\mathbf{s}')\right)\right] \tag{8}$$

which requires us to also learn a model of the environment in the form of $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$. How do we learn the environment model? Given transitions observed by the environment, $p_{\text{obs.}}(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ which sum to one over all possible states $\mathbf{s}'$, we can sample transitions using any policy π by storing or learning a function approximator to

$$p(\mathbf{s}'|\mathbf{s}, \mathbf{a}) = \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi}\left[\rho(\mathbf{a}_t|\mathbf{s}_t)p_{\text{obs.}}(\mathbf{s}_{t+1}|\mathbf{s_t}, \mathbf{a}_t)\mathbb{1}\left[\mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}, \mathbf{s}_{t+1} = \mathbf{s}'\right]\right] \tag{9}$$

where the *importance sampling ratio*

$$\rho(\mathbf{a}|\mathbf{s}) = \frac{\pi_{\text{random}}(\mathbf{a}|\mathbf{s})}{\pi(\mathbf{a}|\mathbf{s})} \tag{10}$$

is the relative likelihood of choosing that action randomly according to policy π_{random} compared to the likelihood of choosing it with the policy used to sample trajectories π. When used in this context, the quantity is sometimes called the *inverse propensity weight* or *inverse propensity score* since it is defined as the inverse of the propensity of that transition under policy π, where "propensity" is used as a rarer synonym for "probability" to help distinguish it from other probabilities that are also often labeled with the Greek letter ρ.

Another strategy makes use of Bayesian models that possess the ability to sample their parameters according to their epistemic uncertainty [6], and is called Thompson sampling [10]. In this strategy, the model parameters are sampled, the Q-value is calculated from this sampled model for each available action, and the strategy chooses the action with the highest score. This approach has been shown to optimize the exploration-exploitation trade-off in bandits[10, 9]. Further developments, such as Randomized Least Squares Value Iteration, use epistemic uncertainty sampling even during the Q-learning phase[8] to encourage exploration.

Lastly, a recent and amazingly-successful strategy, called Monte Carlo Tree Search (MCTS), performs simulations during each planning stage, using a simplified policy and guided by state- and/or value-functions that are also trained. These strategies have been very successful in perfect-information, adversarial games, and fall under model-based reinforcement learning, but they are outside the scope of this overview. Read Section 8.11 of [16] as well as [14] and [15] to learn more.

4 Training the state- and value-function policy evaluators

With this notation secured, we can finally bring in the **Bellman equation**, which is a major foundation of both reinforcement learning and dynamic programming. It states that the best choices we make now can be determined greedily by assuming we continue to make the best possible choices later, and we can write it out mathematically as

$$Q^*(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \arg\max_{a'} Q^*(\mathbf{s}', \mathbf{a}') \tag{11}$$

where we assume that action $\mathbf{a}$ leads us from state $\mathbf{s}$ to state $\mathbf{s}'$. We denote Q^* as the optimal Q-value (which can then be used to define the optimal policy π^*), and any function approximation for the Q-value, using model parameters $\boldsymbol{\phi}$, as $Q_{\boldsymbol{\phi}}$ with the associated policy $\pi_{\boldsymbol{\phi}}$.

We can update the Q-value using **temporal-difference** (TD) update rules that use the fact that we can always improve our estimates of the Q-value by iterating over our trajectories, at each time step updating our function approximator of the Q-value to predict targets that increment against the old estimate. The incremental value, or target, is supplied by the difference between the reward returned by the environment for the current state and action pair ($\mathbf{s}$, $\mathbf{a}$) and the reward for the current time step predicted by our function approximator. We then multiply it by a learning rate $\alpha \in (0, 1]$ in the explicit update rules, or use the learning rate in the computed policy gradient update rules (more on this later).

Since the approximator covers the discounted sum of future rewards G_t, rather than specific rewards at each time step, we use the Bellman equation to predict the reward for the current time step as the temporal difference $G_{t+1} - G_t$ predicted by the function approximator. We can also look at higher-order approxiamtions using more than one time step, and define these quantities as

$$\begin{aligned} G_t^{(1)} &= G_{t:t+1} = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma V(\mathbf{s}_{t+1}) \\ G_t^{(2)} &= G_{t:t+2} = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma r(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) + \gamma^2 V(\mathbf{s}_{t+2}) \\ G_t^{(n)} &= G_{t:t+n} = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma r(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) + \cdots \\ &\qquad + \gamma^{n-1} r(\mathbf{s}_{t+n-1}, \mathbf{a}_{t+n-1}) + \gamma^n V(\mathbf{s}_{t+n}) \end{aligned} \tag{12}$$

where we have dropped the explicit policy markers for clarity.

Similar definitions can be made using the action-value functions instead of state-value functions. Since the action- or Q-value is preferred for policies that depend on the estimators, while the state-value is preferred for the policy gradients presented in the next section, the methods covered in this section are collectively referred to as *Q-learning*.

One more common elaboration, featured in the TD(λ) algorithm (discussed later), is to work with a weighted sum of discounted sums of future rewards

defined as:

$$G_{t:h}^{(\lambda)} = (1-\lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h} \tag{13}$$

where $0 \leq t < h \leq T$ and $0 \leq \lambda \leq 1$.

We can either update according to the current policy, which is slow, or by using counter-factuals, also known as "off-policy learning". For example, the one-step on-policy SARSA update rule for TD(0) can be written as

$$\begin{aligned} Q(\mathbf{s},\mathbf{a}) &\leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha \underbrace{[G_{t:t+1} - Q(\mathbf{s},\mathbf{a})]}_{\delta_t} \\ &\leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha \underbrace{[r(\mathbf{s},\mathbf{a}) + \gamma Q(\mathbf{s}',\mathbf{a}') - Q(\mathbf{s},\mathbf{a})]}_{\delta_t} \end{aligned} \tag{14}$$

and the one-step off-policy Q-learning update rule for TD(0) can be written as

$$\begin{aligned} Q(\mathbf{s},\mathbf{a}) &\leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha \underbrace{[\rho_{\text{target}}(\mathbf{a}'|\mathbf{s}') G_{t:t+1} - Q(\mathbf{s},\mathbf{a})]}_{\delta_t} \\ &\leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha \underbrace{\left[\rho_{\text{target}}(\mathbf{a}'|\mathbf{s}') \left(r(\mathbf{s},\mathbf{a}) + \gamma \arg\max_{a'} Q_{\text{target}}(\mathbf{s}',\mathbf{a}')\right) - Q(\mathbf{s},\mathbf{a})\right]}_{\delta_t} \end{aligned} \tag{15}$$

with the main difference that we update according the best possible next action for Q-learning, rather than just the one that happens to be taken by the current policy for SARSA. We have marked the TD error in all equations with the underbrace indicating δ_t. By extension, the n-step off-policy Q-learning update rule for TD(n) is

$$Q(\mathbf{s},\mathbf{a}) \leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha \left[\rho_{t:t+n}^{(\text{target})}(\boldsymbol{\tau}) G_{t:t+n} - Q(\mathbf{s},\mathbf{a})\right] \tag{16}$$

All TD(0) update rules add a hyperparameter $\alpha \in (0, 1]$ that determines how quickly the Q-values are updated and which must be tuned to the task at hand. There are, as usual, equivalent formulations for the state-value function $V(\mathbf{s})$.

A few new quantities appear in Equations 15 and 16, namely Q_{target}, which represents a target action-value evaluated using a different policy π_{target}, and the *importance sampling ratio* ρ between the target policy π_{target} and the training policy π defined here as

$$\rho_{\text{target}}(\mathbf{a}|\mathbf{s}) = \frac{\pi(\mathbf{a}|\mathbf{s})}{\pi_{\text{target}}(\mathbf{a}|\mathbf{s})} \tag{17}$$

. The n-step importance sampling ratio is defined as

$$\rho_{t:h}^{(\text{target})}(\boldsymbol{\tau}) = \prod_{t'=t}^{\min(h,T-1)} \frac{\pi(\mathbf{a}_{t'}|\mathbf{s}_{t'})}{\pi_{\text{target}}(\mathbf{a}_{t'}|\mathbf{s}_{t'})} \tag{18}$$

In the semi-gradient algorithms discussed below, whereby we take derivatives of the TD-error, the target function Q_{target} used to select the maximizing action and the training function Q are defined by separate models with separate sets of parameters. Why do we do this? It's absolutely possible to calculate the true gradient – it ends up being a difference between two derivatives evaluated at different state and action inputs. However, this fact actually causes gradient descent to no longer reach an equilibrium, but rather to smooth the values between the different inputs (see Section 11.5 and Example 11.2 in [16]). Moreover, keeping the two models separate has been shown to be crucial to many recent breakthroughs in the field by providing greater stability, as in the DDPG algorithm[13]. However, theoretically the target policy is often treated as the same as the training policy, so that many presentations assume ρ to be identical to one and can be ignored.

How do we define Q_{target} so that it is as close as possible to Q while avoiding numerical instabilities that can and short-circuit the system? The obvious choice is to use a frozen copy of the model made before each update is processed. If we denote the model parameters for our function approximation at update step k as $\boldsymbol{\phi}_k$, and the model parameters used to predict our targets as $\boldsymbol{\phi}_{k,\text{target}}$, then this strategy can be written as $\boldsymbol{\phi}_{k,\text{target}} \leftarrow \boldsymbol{\phi}_{k-1}$. Another is to store up targets over a given number of updates and then batch update the model at regular intervals. And yet another is to use both agents simultaneously with a linear (polyak) interpolation so that $\boldsymbol{\phi}_{k,\text{target}} \leftarrow \rho\boldsymbol{\phi}_{k,\text{target}} + (1-\rho)\boldsymbol{\phi}$ with $\rho \in [0,1]$, which is also known as "soft updating". The smoothing hyperparameter ρ should not be confused with the importance sampling ratio $\rho(\mathbf{a}|\mathbf{s})$.

We define the loss that we attempt to minimize as the negative log of the likelihood. For the normal distribution likelihood

$$\mathcal{L}(\mathbf{y}_{\text{predicted}}, \mathbf{y}_{\text{target}}) = \frac{1}{\sigma\sqrt{2\pi}} \exp[-\frac{(y_{i,\text{target}} - y_{i,\text{predicted}})^2}{2\sigma^2}] \tag{19}$$

the negative log-likelihood term in the loss, $L = -\log \mathcal{L}$, can be written as

$$L_{\text{NLL}}(\boldsymbol{\phi}, \mathcal{D}) = \frac{1}{2}\times$$
$$\mathop{\mathbb{E}}_{(\mathbf{s},\mathbf{a},r,\mathbf{s}',d)\sim\mathcal{D}} \left[\left(\underbrace{\rho_{\text{target}}(\mathbf{a}'|\mathbf{s}') \left(r(\mathbf{s},\mathbf{a}) + \gamma(1-d) \arg\max_{\mathbf{a}'} Q_{\text{target}}(\mathbf{s}',\mathbf{a}') \right)}_{\text{target}} - \underbrace{Q(\mathbf{s},\mathbf{a})}_{\text{predicted}} \right)^2 \right] \tag{20}$$

where $d = 1$ when state $\mathbf{s}'$ is terminal, σ is a nuisance parameter that can be set to 1, and where we have dropped the explicit dependence on $\boldsymbol{\phi}$. The dataset

$$\mathcal{D} = [(\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}'_0, d_0), (\mathbf{s}_1, \mathbf{a}_1, r_1, \mathbf{s}'_1, d_1), \ldots, (\mathbf{s}_H, \mathbf{a}_H, r_H, \mathbf{s}'_H, d_H)] \tag{21}$$

is known as the *replay buffer* and provides us with samples of transitions that have been previously encountered so that we can obtain accurate rewards, and

we may discard old trajectories as our estimations improve. The importance sampling ratio ρ_{target} can be ignored (assumed equal to one) if the target and training parameters are sufficiently close to each other.

We note that gradient updates of the form

$$\boldsymbol{\phi}_k = \boldsymbol{\phi}_{k-1} - \alpha \boldsymbol{\nabla}_{\boldsymbol{\phi}} L(\boldsymbol{\phi}, \mathbf{v}) \tag{22}$$

where $L(\boldsymbol{\phi}, \mathbf{v})$ is the loss function of the parameters $\boldsymbol{\phi}$ and additional values, collected into $\mathbf{v}$, can be performed using stochastic gradient descent using batch samples to evaluate the loss function, which means we can simplify Equation 20 by considering individual samples one at a time. Working with the negative log-likelihood loss, we obtain

$$\begin{aligned} \boldsymbol{\phi}^{(k+1)} &= \boldsymbol{\phi}^{(k)} - \frac{1}{2}\alpha \boldsymbol{\nabla}_{\boldsymbol{\phi}} L_{\text{NLL}}(\boldsymbol{\phi}, \mathbf{v}) \\ &= \boldsymbol{\phi}^{(k)} - \frac{1}{2}\alpha \boldsymbol{\nabla}_{\boldsymbol{\phi}} \left(U_t - Q(\mathbf{s}, \mathbf{a})\right)^2 \\ &= \boldsymbol{\phi}^{(k)} + \alpha \underbrace{\left(U_t - Q(\mathbf{s}, \mathbf{a})\right)}_{\delta_t} \boldsymbol{\nabla}_{\boldsymbol{\phi}} Q(\mathbf{s}, \mathbf{a}) \end{aligned} \tag{23}$$

where we have replaced the term in underbrace labeled "target" in Eq. 20 with the simpler term U_t which we assume doesn't depend on $\boldsymbol{\phi}$, and where we have applied the chain rule for derivatives. We've indicated the TD error δ_t in the underbrace, which has imperfect analogs to dopamin in the brain (see Section 15.4 of [16]).

The target U_t can take on many forms, giving rise to different named algorithms, including

TD(1) (a.k.a. Monte Carlo):
$U_t = G_t$

TD(0):
$U_t = G_{t:t+1}$

TD(0) off-policy:
$U_t = \rho_{\text{target}}(\mathbf{s}_t|\mathbf{a}_t) G_{t:t+1}$

TD(n):
$U_t = G_{t:t+n}$

TD(n) off-policy:
$U_t = \rho^{(\text{target})}_{t:t+n} G_{t:t+n}$

TD(λ):
$U_t = G^{(\lambda)}_{t:h}$

Dynamic Programming (DP):
$U_t = \sum_{\mathbf{a},\mathbf{s}',r} \pi(\mathbf{a}|\mathbf{s}') p(\mathbf{s}', r|\mathbf{s}_t, \mathbf{a}) \left(r(\mathbf{s}_t, \mathbf{a}) + \gamma Q_{\text{target}}(\mathbf{s}', \mathbf{a})\right)$

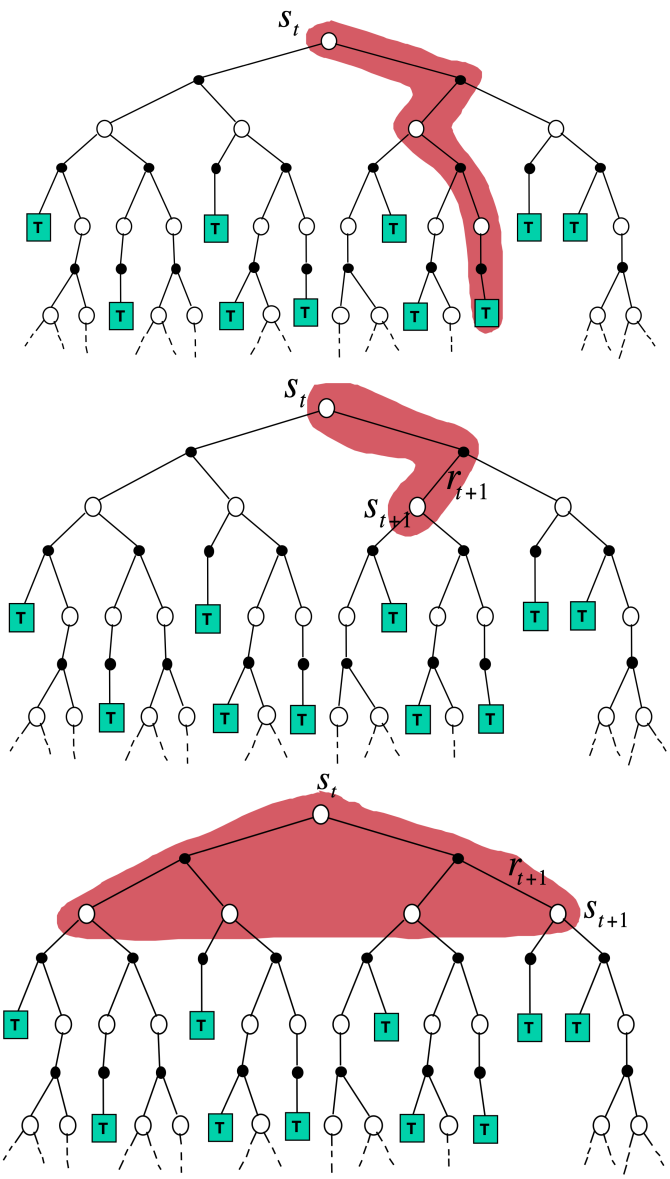

Figure 2: Backup diagrams for the following algorithms: Monte Carlo (top), TD(0) (middle), and DP (bottom)**??**

Note that the DP target requires a model of the environment $p(\mathbf{s}', r|\mathbf{s}_t, \mathbf{a})$. The back-up diagrams for these targets are shown in Figure 4

Of course, in all cases except for TD(1), U_t absolutely does depend on $\boldsymbol{\phi}$, which is why we call methods that use Equation 23 *semi-gradient* methods. Note that we often often replace $\boldsymbol{\phi}$ with $\boldsymbol{\phi}_{\text{target}}$ in our notation for the target to make this distinction, even though ideally we want to work with just one set of parameters $\boldsymbol{\phi}$. However, it is insanely hard to calculate the gradient of U_t with respect to $\boldsymbol{\phi}$, so the dependency is ignored. At this moment, we only have convergence guarantees to the local optimum for TD(1), as long as Equation 2.7 from [16] is satisfied (α is properly reduced during traing). However, semi-gradients methods lie at the heart of the current state-of-the-art algorithms such as TD(0), DQN[9], and DDPG[5] anyways.

When looking at linear functions of the form $Q_{\boldsymbol{\phi}}(\mathbf{s}, \mathbf{a}) = \boldsymbol{\phi}^\top \mathbf{x}(\mathbf{s}, \mathbf{a})$, where $\mathbf{x}(\mathbf{s}, \mathbf{a})$ is a feature vector dependent on the state and action, this is a convex optimization problem, so that the local optimum we find is also the global one. In this case, TD(1) also has convergence guarantees, but only to a point close to the optimum called the *TD fixed point*. See sections 9.3 and 9.4 in [16] for more details.

We can connect the tabular update rules in Equation 14 to the function approximation parameter update rules in Equation 23 by observing that a linear model using categorical variables encodes them as one-hot vectors, with a 1 for any element matching the category, and a 0 for all other elements. In this case, $\phi_i = Q(\mathbf{s}_i, \mathbf{a}_i)$ where the index i refers to a unique state-and-action pair. In this case, we obtain

$$\begin{aligned} \phi_i^{(k+1)} &\leftarrow \phi_i^{(k)} + \alpha(U_t - \phi_i^{(k)}) \cancelto{1}{\frac{\partial Q(\mathbf{s}_i, \mathbf{a}_i)}{\partial \phi_i}} \\ &\leftarrow \phi_i^{(k)} + \alpha(U_t - \phi_i^{(k)}) \end{aligned} \tag{24}$$

which lines up nicely with Equations 23 and 14 when $U_t = G_{t:t+1}$.

5 Training the policy directly

Some approaches learn a function approximator of the Q-value using parameters $\boldsymbol{\phi}$, and then choose the action at a given state from a pool of candidates based on their predicted Q-values. Others instead work with an explicit policy function $\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})$, dependent on parameters $\boldsymbol{\theta}$, which can either output probabilities for a discrete set of actions or which can output a single probability for a joint input $\mathbf{x}(\mathbf{s}, \mathbf{a})$. Others still can use both approaches simultaneously.

There are many reasons to desire a policy function that is separate from the state- or value-functions. For some problems, direct policy optimization outperforms the alternative, possibly because the policy may be easier to learn and require fewer model parameters than the state- and value-functions. For another, we may want to have independent control over the stocahsticity of our policy. For example, we may want to learn a deterministic policy even

though there may be multiple actions that could lead to high rewards with similar action values. Conversely, we may want to learn a stochastic policy with specified action probabilities that differ from their action values or whatever transformation we use on them. Read Section 13.1 of [16] to learn more.

To see how the loss function works for an explicit policy model, as opposed to an action- or value-function model, we first observe the expected discounted sum of returns can be written as

$$R(\boldsymbol{\tau}) = G_0(\boldsymbol{\tau}) = \sum_{t=0}^{H} \gamma^t r_t \tag{25}$$

Given a distribution of initial states $p(\mathbf{s}_0)$, we write out the probability of a trajectory for a given policy as

$$p(\boldsymbol{\tau}|\pi) = p(\mathbf{s}_0) \prod_{t=0}^{H-1} p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\pi(\mathbf{a}_t|\mathbf{s}_t) \tag{26}$$

for which the expected return is written as

$$J(\pi) = \int_{\boldsymbol{\tau}} p(\boldsymbol{\tau}|\pi)R(\boldsymbol{\tau}) = \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi}[G_0(\boldsymbol{\tau})] = \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi}[V(\mathbf{s}_0)] \tag{27}$$

where the letter J, like the letter G, is chosen for historical reasons, in this case to relate to other equations involving loss or cost function. We thus write out our policy gradient update rule as

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + \alpha \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})|_{\boldsymbol{\theta}_{k-1}} \tag{28}$$

which you'll notice looks just like Equation 22, except that it uses plus sign instead of a minus sign before α, meaning that $J = -L$ is the *negative* loss function, or the (positive) log-likelihood, for any stochastic gradient descent algorithm. The confusion around negative signs never goes away, and is something we just have to keep in mind: gradient *ascent* maximizes the reward, and gradient *descent* minimizes the loss or cost.

The last niggling detail is to numerically compute the policy gradient. Among other issues, our current formulation contains the model of the environment $p(\mathbf{s}_{t+1}|\mathbf{s}t, \mathbf{a}_t)$. For the action- and state-value functions, we used the semi-gradient approach to approximate the gradient, but in this case we can solve it exactly. This is accomplished using the proofs presented in [3] and Sections 13.1-13.3 in [16], which are summarized in the Policy Gradient Theorem (PGT):

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) = \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{H} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t|\mathbf{s}_t)\Psi_t \right] \tag{29}$$

where, like U_t earlier, Ψ_t can take on many forms, giving rise to various named algorithms. They are:

not used, but easiest to prove
$\Psi_t = G_0$

REINFORCE:
$\Psi_t = G_t$

REINFORCE with baseline:
$\Psi_t = \sum_{t'=t}^{H} \gamma^{t'-t} r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) - b(\mathbf{s}_{t'})$

time-difference (TD):
$\Psi_t = G_{t:t+n}$ for any n

Q-Actor Critic (AC):
$\Psi_t = Q(\mathbf{s}_t, \mathbf{a}_t)$

Q-Actor Critic (AC) off-policy:
$\Psi_t = \rho_{\text{target}}(\mathbf{s}_t, \mathbf{a}_t) Q(\mathbf{s}_t, \mathbf{a}_t)$

Advantage Actor Critic (A2C) :
$\Psi_t = A(\mathbf{s}_t, \mathbf{a}_t) \coloneqq Q(\mathbf{s}_t, \mathbf{a}_t) - V(\mathbf{s}_t)$

where $b(\mathbf{s})$ is any baseline function that depends only on the state $\mathbf{s}$. The baseline function allows us to reduce the variance in our gradients, since it doesn't affect the gradient due to the normalization condition (Equation 1) and the logarithm. Combining Equations 28 and **??**, we thus obtain the policy gradient ascent update rule

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{k-1} + \alpha \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{H} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t | \mathbf{s}_t) \Psi_t \right] \tag{30}$$

. One simple formula to rule over many named algorithms: REINFORCE[18], AC[2], A2C **??**, RG, and GAE [11] [4]!

The more-recent Proximal Policy Optimization (PPO) algorithm[12] has a somewhat complex-looking target but with a simple interpretation. It is

$$\Psi_t^{(\text{PPO})} = \begin{cases} \min\left(\frac{\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})}{\pi_{\boldsymbol{\theta}_k}(\mathbf{a}|\mathbf{s})}, 1+\epsilon \right) A_{\pi_{\boldsymbol{\theta}_k}}(\mathbf{s}, \mathbf{a}), & A_{\pi_{\boldsymbol{\theta}_k}}(\mathbf{s}, \mathbf{a}) > 0 \\ \min\left(\frac{\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})}{\pi_{\boldsymbol{\theta}_k}(\mathbf{a}|\mathbf{s})}, 1-\epsilon \right) A_{\pi_{\boldsymbol{\theta}_k}}(\mathbf{s}, \mathbf{a}), & A_{\pi_{\boldsymbol{\theta}_k}}(\mathbf{s}, \mathbf{a}) < 0 \end{cases} \tag{31}$$

. In other words, it is off-policy Advantage Actor Critic (A2C) algorithm but clipped so that the target never exceeds the range of

$$\Psi_t^{(\text{PPO})} \in \left[(1-\epsilon) \left| A_{\pi_{\boldsymbol{\theta}_k}}(\mathbf{s}, \mathbf{a}) \right|, (1+\epsilon) \left| A_{\pi_{\boldsymbol{\theta}_k}}(\mathbf{s}, \mathbf{a}) \right| \right] \tag{32}$$

At first, the surprising introduction of logarithm makes one ask: why $\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}$ instead of $\boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}$? To cultivate intuition for the PGT, we first note that our goal

[4] https://ai.stackexchange.com/questions/10049/why-are-lambda-returns-so-rarely-used-in-policy-gradients/10061

is to relate the gradient of an expectation over trajectories to the expectation over trajectories of a gradient, in order to make it possible to compute. This comes up because the value we are attempting to maximize, the state-value function for a given policy at the start of all trajectories made by the policy, is the expectation of action-values over all actions. Meanwhile, the policy itself is a function of individual actions. We can relate this back to the expectation over trajectories formed by the policy by observing that since

$$\begin{aligned} \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} f(\mathbf{a}_t, \mathbf{s}_t) &= \sum_{\mathbf{a}\sim\mathcal{A}} \pi(\mathbf{a}|\mathbf{s}_t) f(\mathbf{a}, \mathbf{s}_t) \\ \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} f(\boldsymbol{\tau}) &= \int_{\boldsymbol{\tau}} p(\boldsymbol{\tau}) f(\boldsymbol{\tau}) \end{aligned} \tag{33}$$

and

$$\frac{\partial f(x)}{\partial x} = f(x) \frac{\partial \log f(x)}{\partial x} \tag{34}$$

and

$$\log p(\boldsymbol{\tau}|\pi_{\boldsymbol{\theta}}) = p(\mathbf{s}_0) \sum_{t=0}^{H} \pi(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \tag{35}$$

we have

$$\begin{aligned} \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) &= \boldsymbol{\nabla}_{\boldsymbol{\theta}} \int_{\boldsymbol{\tau}} p(\boldsymbol{\tau}, \boldsymbol{\theta}) R(\boldsymbol{\tau}) \\ &= \int_{\boldsymbol{\tau}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} p(\boldsymbol{\tau}, \boldsymbol{\theta}) R(\boldsymbol{\tau}) \\ &= \int_{\boldsymbol{\tau}} p(\boldsymbol{\tau}, \boldsymbol{\theta}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\boldsymbol{\tau}, \boldsymbol{\theta}) R(\boldsymbol{\tau}) \\ &= \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} [\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log p(\boldsymbol{\tau}, \boldsymbol{\theta}) R(\boldsymbol{\tau})] \\ &= \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{H} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s}) R(\boldsymbol{\tau}) \right] \end{aligned} \tag{36}$$

where we can replace $R(\boldsymbol{\tau})$ with Ψ_t. You'll also see, as in [16], the policy gradient written as

$$\begin{aligned} \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}}) &= \boldsymbol{\nabla}_{\boldsymbol{\theta}} \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} [V(\mathbf{s}_0)] \\ &\propto \sum_{\mathbf{s}} p_{\pi_{\boldsymbol{\theta}}}(\mathbf{s}) \sum_{\mathbf{a}} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s}) Q_{\pi_{\boldsymbol{\theta}}}(\mathbf{s}, \mathbf{a}) \\ &= \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} \left[\sum_{\mathbf{a}} Q(\mathbf{s}_t, \mathbf{a}) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s}_t) \right] \\ &= \mathop{\mathbb{E}}_{\boldsymbol{\tau}\sim\pi_{\boldsymbol{\theta}}} [Q(\mathbf{s}_t, \mathbf{a}_t) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t|\mathbf{s}_t)] \end{aligned} \tag{37}$$

where we can replace $Q(\mathbf{s}_t, \mathbf{a}_t)$ with Ψ_t.

6 Combining evaluation and policy training in actor-critic algorithms

Actor-critic algorithms like the Q-Actor Critic and the Advantage Actor Critic train the value function policy estimator along with the policy itself. Among the A2C class of algorithms there are different estimators of the advantage function for the policy part of the training process (the value or policy estimator part handled according to Section 4):

MC advantage or Vanilla Policy Gradient (VPG):
$\Psi_t = G_t - V(\mathbf{s}_t)$

TD advantage
$\Psi_t = \delta_t \coloneqq G_{t:t+1} - V(\mathbf{s}_t) = r_t + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)$

n-step advantage:
$\Psi_t = G_{t:t+n} - V(\mathbf{s}_t)$ for any n

Generalized Advantage Estimator (GAE) :
$\Psi_t = \sum_{i=0}^{\infty} (\lambda\gamma)^i \delta_{t+i} = G_t^{(\lambda)} - V(\mathbf{s}_t)$

When implementing policy gradient algorithms, in practice the training process waits until the end of each episode $t = H$ to the beginning at $t = 0$ to propagate information, although for some choices we can update within an episode after a given number of time steps. (Another formulation called *eligibility traces* allows us to update immediately, but requires storing additional information and applying additional update rules, see Chapter 12 of [16].) In this setting, the REINFORCE and TD algorithms use the returns from the episode that just played out, making them on-policy algorithms, while AC and A2C use the expected returns provided by an action-value or Q-value function, requiring us to optimize the Q function in addition to the policy. In this case, the action that is "played out" is the Q-value-maximizing action, not necessarily the action that the policy would have used during the episode, making it an off-policy algorithm. We note the relationship between these algorithms and their constituent parts in Figure 6.

Actor-critic algorithms can learn more than one policy estimator and bootstrap one off the other using the Bellman equation. We drop the policy and parameter markers in the policy gradient targets (e.g., $V_{\pi_{\boldsymbol{\theta}}}$ becomes V) since we can decide to either link the action- and state-value functions using the same parameters and the relation

$$Q(\mathbf{s}, \mathbf{a}) = \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \pi_{\boldsymbol{\theta}}} [r(\mathbf{s}_t, \mathbf{a}_t) + \gamma V(\mathbf{s}_{t+1})] \tag{38}$$

or we can use different sets of parameters for both functions, or any other concoction. A popular formulation uses a multi-task network to predict both

[6] Source: `https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf`

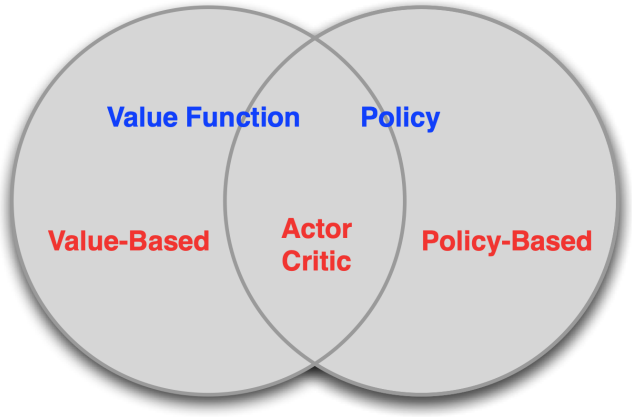

Figure 3: Venn diagram relating the value-based algorithms like Q-learning and policy-based algorithms like REINFORCE covered in this document. [6]

the action- and state-values, as well as the policy, using some shared and some separated parameters[1] by minimizing a total loss

$$L_{\text{total}} = L_{\text{actor}} + L_{\text{critic}} + L_{\text{regularizer}} \tag{39}$$

according to batch updates of Equation 22. To learn more about these various options, see [17, 3].

For the critic, which predicts action- or state-values, the loss function is a function of the difference between our predictions and the results returned from the environment, using the time-difference Bellman equation to bootstrap off our previous predictions. For the normal distribution likelihood, we can use the squared error as in

$$L_{\text{critic}} = \frac{1}{2} \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \pi_{\boldsymbol{\theta}}} \left[\left(U_t - V(\mathbf{s}_t) \right)^2 \right] \tag{40}$$

. Some variations may use the absolute error or the Huber loss.

For the actor, i.e., the policy gradient, our loss function, according to Equations **??** and 22, is

$$L_{\text{actor}} = - \mathop{\mathbb{E}}_{\boldsymbol{\tau} \sim \pi_{\boldsymbol{\theta}}} \left[\sum_{t=0}^{H} \log \pi_{\boldsymbol{\theta}}(\mathbf{a}_t | \mathbf{s}_t) \Psi_t \right] \tag{41}$$

Lastly, we can also add additional losses to our final optimized quantity, such as regularization losses to reduce the model complexity, or a self-entropy term

$$L_{\text{regularizer}} = \beta H(\pi, \mathbf{s}) = -\beta \sum_{a} \pi(\mathbf{a}|\mathbf{s}) \log \pi(\mathbf{a}|\mathbf{s}) \tag{42}$$

, weighted (multiplied) by a chosen positive or negative hyperparameter β, to encourage more or less exploration as desired. L1 and L2 regularizer losses are always available, written as

$$L_{\text{L2}} = \beta \boldsymbol{\theta}^\top \boldsymbol{\theta} = \frac{\sigma^2}{\sigma_p^2} \boldsymbol{\theta}^\top \boldsymbol{\theta} \tag{43}$$

, where β is the $L2$ regularization hyperparameter and σ_p is a noise parameter scale hyperparameter which is equivalent to the average noise parameter for the prior, where the model parameters are all set to zero, or $\boldsymbol{\theta} = \mathbf{0}$. In addition, losses that allow us to simultaneously optimize either the homoskedastic the noise parameter σ, or the heteroskedastic noise parameter $\sigma(\mathbf{a}, \mathbf{s})$, can be included, derived from the negative log of the normalization constant in a normal distribution, and which we can write as

$$L_\sigma = N_{\text{data}} \log \sigma \tag{44}$$

such that L_{L2} and L_σ balance each other out against the data fit term L_{actor} and/or the policy gradient loss L_{actor}.

7 Under the hood

Stochastic gradient descent has the benefit of only ever needing to store and compute as many components as there are parameters in the model, as opposed to least squares methods that require storage and computation over the square of the number of parameters in the model. But the parameter update rules for the action- and state-value functions can be performed using any regression technique against the target U_t, including memory-based techniques. One such method, K-nearest neighbors regression, requires prediction order complexity $\mathcal{O}(k \times N_{\text{data}} \times N_{\text{features}})$ using brute force, or $\mathcal{O}(k \times \log(N_{\text{data}}))$ using a k-d tree or ball tree data structure. Another, Gaussian process regression, requires prediction order complexity $\mathcal{O}(N_{\text{data}}^3)$, which is much higher.

For direct policy optimization, however, we are required to craft a custom loss function, which limits our techniques to methods with explicitly-defined loss functions. For example, almost all policy gradient implementations found online use neural networks, which come with infrastructure for defining a custom loss function and computing its gradient, but any algorithm that performs stochastic gradient descent over a custom loss function will also work. In special cases, the gradients can be analytically derived and programmed explicitly.

For example, if we use a linear model for action preferences $h_{\boldsymbol{\theta}}(\mathbf{s}, \mathbf{a}) = \boldsymbol{\theta}^\top \mathbf{x}(\mathbf{s}, \mathbf{a})$, where $\mathbf{x}(\mathbf{s}, \mathbf{a})$ represents our observations or features covering both the state and the action, we can use the softmax function to define the policy as

$$\pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s}) = \frac{\exp[h_{\boldsymbol{\theta}}(\mathbf{s}, \mathbf{a})]}{\sum_{\mathbf{a}' \sim \mathcal{A}} \exp[h_{\boldsymbol{\theta}}(\mathbf{s}, \mathbf{a}')]} = \frac{\exp[\boldsymbol{\theta}^\top \mathbf{x}(\mathbf{s}, \mathbf{a})]}{\sum_{\mathbf{a}' \sim \mathcal{A}} \exp[\boldsymbol{\theta}^\top \mathbf{x}(\mathbf{s}, \mathbf{a}')]} \tag{45}$$

We see that the gradients have an analytical form we can exploit to cheaply update the policy, given by

$$\frac{\partial \pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})}{\partial \boldsymbol{\theta}} = \pi_{\boldsymbol{\theta}}(\mathbf{a}|\mathbf{s})\mathbf{x}(\mathbf{s},\mathbf{a}) \left(1 - \frac{1}{\sum_{\mathbf{a}' \sim \mathcal{A}} \exp[\boldsymbol{\theta}^\top \mathbf{x}(\mathbf{s},\mathbf{a}')]}\right) \tag{46}$$

However, for more complex loss functions, it may be difficult to analytically define their derivatives, or we may prefer to automate the process to accommodate a wide range of loss functions we can't anticipate. In this case, a stand-alone automatic differentiator (like Python's *autograd*) can be used, which works with regular functions that use a special imported version of *NumPy* to help it keep track of the operations, although it can only handle certain operations and patterns that the user must be aware of. Unfortunately, problems that require any degree of simulation, or are undifferentiable, are not well suited to automatic differentiation.

While it would be possible to employ function optimization methods like SciPy 's minimize package to optimize either policy estimators or the policy itself, or to use any other non-gradient based function optimizer, these will drive the loss to its optimum over a fixed dataset and this can produce poor results. This is because, unlike in supervised learning, we do not have a fixed distribution of data, but rather the data distribution changes as the parameters change. We can see that this is the case since our actor and critic losses and their gradients are defined as expectations over trajectories produced by the policies we are optimizing, rather than sums over all possibilities. This is a major achievement of both semi-gradient and policy gradient methods, since it brings us back to earth by allowing us to sample these expectations using the most recent policy, and with importance sampling, we can account for the data distribution drift in off-policy training. However, we must still keep in mind that the loss functions we compute at any step of optimization are limited by the expectations over trajectories we had to compute using a different policy than the one we end up with because it didn't exist yet. In other words, our goal is not to optimize the loss at any given stage of training, but to optimize the rewards of the system, which the losses help us achieve at each step.

As a result, one rarely sees solutions that don't use gradient descent or require the tuning of a learning rate α hyperparameter. For exact, online methods, like Bayesian linear regression, we account for the changing data distribution of the loss function because each step of training amounts to adding new rows of data, so that we are ultimately minimizing or maximizing the average of the expectations over the history of the parameters, similar to stochastic gradient descent. It is up to numerical experiments to fine-tune α, or the decay factor we use to drop out old rows of data, to see how the different methods compare. When using neural networks with gradient descent, it is also important to scale the inputs to the standard normal where the model has the most capacity, and to be aware that the model interpolates linearly through gaps of data, and extrapolates linearly outside the domain of the standard normal. This can be addressed by either stalling training until a data sample has been made, or one

can apply batch normalization[4] as a regularization technique.

From an engineering perspective, note that the formalism for training the state- and value-functions, regardless of the target U_t, is equivalent to the formalism for *contextual multi-armed bandits* (CMABs), and both align between versions with discrete and continuous actions. While both Q-learning and bandits may train against targets compiled during a loop at the end of each episode that rewinds the previous actions, states, and rewards, only Q-learning incorporates the $\arg\max_{\mathbf{a}}$ over future steps before it is reflected in the policy being trained thanks to the Bellman equation. This allows information about successful policies to propagate back to earlier steps before they are reflected in the policy used to collect data, making training in situations where planning is relevant much faster. Since we incorporate the unique qualities of Q-learning entirely during the collection loop that passes targets to the function approximator, we can use a CMAB under the hood for both training and prediction in the collection loop. The policy function, however, requires its own engineering structure, while bandits are still useful for the Q-learning structures in actor-critic algorithms that incorporate both policy optimization and Q-learning..

Just as properties of the update algorithm made true gradient methods using the state- or action-value functions unstable, requiring the development of semi-gradient methods, certain combinations of algorithm properties can produce other instabilities. One particular combination, known as the *deadly triad*, must be particularly avoided:

1. Bootstrapping the target U_t, i.e., using any target besides Monte Carlo such as TD
2. Off-policy training
3. Non-linear function approximation

To learn more, see Section 11.3 in [16].

Using function approximation for both the policy evaluation and the policy itself necessarily implies partial observability of our environment, and while losing observations can kill our ability to learn a successful policy, the methods described here will do as well as we possibly can given the handicap. However, we can go the other way and add observations, too, and an obvious choice is to add historical data from earlier the trajectory. This can be accommodated by either encoding them as truncated and padded concatenations to the original feature vector, or by using sequential models like RNNs and LSTMs to encode the historical data into a fluid representation. Of course, as we add more expressiveness and/or features to our models, that is, as we add more parameters, we increase our model capacity but also dramatically reduce the speed at which it learns. A general rule of thumb is that by using M times more features, it takes M^2 more samples to move out of exploration into exploitation.

Anther popular option, which requires us to sample our input data before training our agents, is to expand the feature dimensionality using Gaussian Mixture Models (GMMs), which encode features as distances to different centroids

that cover the distribution of input data sample. And yet another popular option is to expand features out into a set of basis functions, where, again, a popular choice is to convolve your input with Gaussians tiled across the feature space, such that their widths touch but do not substantially overlap. For neural networks, Convolutional Neural Networks (CNN) are an even more flexible option.

Moreover, policy and action-value functions generally only work when we incorporate interaction features between the state and action features, except for decision-tree based algorithms like random forests and gradient-boosted trees, whereas neural networks are designed to learn which interactions are most useful at the cost of extensive hyperparameter tuning. Since there are so many ways to add features, we can bring down the complexity of these computations and to help our agents learn faster by applying randomized projections onto any subset of our feature space, but this also introduces additional hyperparameters to tune. We recommend randomized projections over Principle Component Analysis (PCA) since random projections to not require an input data sample before we can train our function approximators, and because there is no particular reason to expect the signal to be insensitive to the low-variance features that are filtered out in PCA.

When computing the $\arg\max_{\mathbf{a}}$ for Q-learning style algorithms, we may run into difficulties for problem spaces with continuous or large action spaces. However, we have several options at play. The first is to simply sample the action space randomly and rank the predicted Q-values for all of them. This is a popular approach in derivative-free Bayesian optimization. If the action space is continuous, another option is to apply gradient ascent, but this can be computationally expensive.

8 Additional resources

Additional resources can be found below:

- RLlib from Ray covers different algorithms and whether they can accept discrete or continuous actions, as well as additional support such as sequential models, multi-agent implementations, and multi-GPU training. See `https://docs.ray.io/en/latest/rllib-algorithms.html`
- Lilian Wang has created one of the most thorough overview of policy gradient based algorithms at `https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html`
- Spinning Up from OpenAI offers a terrific overview of reinforcement learning concepts as well as deep dives (and implementations) of some of the more advanced algorithms such as Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG), and Soft Actor-Critic (SAC). However, some of their explanations, such as around the policy gradient, can be obscure and needlessly confusing. See `https://spinningup.openai.com/en/latest/`

- David Silver's lecture slides at University College London are a legendary resource for succinct coverage of the main concepts in reinforcement learning. See `https://www.davidsilver.uk/teaching/`
- Julien Vitay provides a sprawling overview of deep reinforcement learning at `https://julien-vitay.net/deeprl/Introduction.html`

References

[1] *Actor Critic Method*. `https://keras.io/examples/rl/actor_critic_cartpole/`. Accessed: 2021-10-20.

[2] Ivo Grondman et al. "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595.

[3] *Intro to policy optimization*. `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html`. Accessed: 2021-09-29.

[4] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[5] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[6] Douglas Mason. *Gaussian processes and equivalent Bayesian linear regressions*. Sept. 2021. DOI: 10.5281/zenodo.1234. URL: `https://github.com/KoyoteScience/GP-BLR`.

[7] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. "Model-based reinforcement learning: A survey". In: *arXiv preprint arXiv:2006.16712* (2020).

[8] Ian Osband, Benjamin Van Roy, and Zheng Wen. "Generalization and exploration via randomized value functions". In: *International Conference on Machine Learning*. PMLR. 2016, pp. 2377–2386.

[9] Ian Osband et al. "Deep exploration via bootstrapped DQN". In: *Advances in neural information processing systems* 29 (2016), pp. 4026–4034.

[10] Daniel J Russo et al. "A Tutorial on Thompson Sampling". In: *Foundations and Trends® in Machine Learning* 11.1 (2018), pp. 1–96.

[11] John Schulman et al. "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438* (2015).

[12] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[13] David Silver et al. "Deterministic policy gradient algorithms". In: *International conference on machine learning*. PMLR. 2014, pp. 387–395.

[14] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[15] David Silver et al. “Mastering the game of go without human knowledge”. In: *nature* 550.7676 (2017), pp. 354–359.

[16] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[17] Lilian Weng. “Policy Gradient Algorithms”. In: *lilianweng.github.io/lil-log* (2018). URL: `https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html`.

[18] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3 (1992), pp. 229–256.