Reinforcement Learning

– Course notes –

November 26, 2019

Chapter 1

Value functions, Dynamic programming

1.1 The value function

The value function is a staple from the literature on dynamic programming, whether it be for discrete or continuous problems (as in control theory). It measures just how good a control u – or, in our case, a policy π – is regarding the desired target of our problem.

Definition 1 (Value function) *The value function $V^\pi : \mathcal{S} \to \mathbb{R}$ of a policy π is the expectation of the cumulative (discounted) future rewards starting from a point $s_0 = s$*

$$V^\pi(s) \coloneqq \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \,\middle|\, s_0 = s\right] \tag{1.1}$$

where the trajectory τ is generated under the policy π.

T can be a random stopping time corresponding to the first time of arrival at a so-called terminal *state. It can also be $T = \infty$ for* infinite horizon *problems.*

Remark 1 (Notational abuse) *Often, we will write the value functon at a state s_t, where it is implied we are at the t-th step in a trajectory, as*

$$V(s_t) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'}) \,\middle|\, s_t\right]$$

This is coherent with the notion of cumulative (discounted) future rewards which defined the value function (1.1), but not with the notation. In the finite horizon setting, a more correct notation would be to write $V(t, s_t)$ and make the dependence on the starting time t of the discounting explicit.

This notion can be generalized to the case where the rewards are generated by the transitions (s_t, a_t, s_{t+1}) rather than the (state, action) couple:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t, s_{t+1}) \,\middle|\, s_0 = s\right]$$

Under a deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ and associated decision rule $d^\pi(s) = \pi(s)$, the dynamic programming principle leads to a dynamic programming equation

called the **Bellman equation**:

$$V^\pi(s) = r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s, \pi(s), s') V^\pi(s') \tag{1.2}$$

which is a fixed-point condition.

The Bellman equation can be used to evaluate a policy π and compute its associated value function V^π.

Its fixed-point structure can be reformulated in terms of an operator on a function space, called the **Bellman operator**:

$$\mathcal{T}^\pi v(s) \coloneqq r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s, \pi(s), s') v(s')$$

Remark 2 *We have the following, possible generalizations (see Sutton's book [1, chap. 3, 4] for further details):*

- *for stochastic policies* $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+$, *the sum in eq.* (1.2) *becomes*

$$\sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} \pi(s, a) p(s, a, s') V^\pi(s')$$

- *in non-discrete state spaces, the sum can be replaced by an integral with respect to a measure* $p(s, \pi(s), \mathrm{d}s')$
- *if the rewards are given for transitions as* $r(s, a, s')$, *we introduce* $r(s, a) = \sum_{s' \in \mathcal{S}} r(s, a, s')$, *and the Bellman equation can be rewritten*

$$V^\pi(s) = \sum_{s' \in \mathcal{S}} r(s, \pi(s), s') + \gamma p(s, \pi(s), s') V^\pi(s')$$

1.1.1 The optimal value function

Solving a Markov Decision Process involves finding an **optimal** policy π^* that will maximize the expected rewards in the long run when starting from a given state s_0 (or distribution $s_0 \sim p$).

Definition 2 (Optimal policy and value function) *Given a set of policies* Π, *the* **optimal value function** *satisfies*

$$V^*(s) = \max_\pi V^\pi(s) \tag{1.3}$$

for every state s. *An* **optimal policy** π^* *is one that satisfies the maximum.*

Proposition 1 (Optimal Bellman equation) *The optimal value function* V^*

obeys a dynamic programming principle, the **optimal Bellman equation**.

$$V^*(s) = \max_{a \in \mathcal{A}} \left\{ r(s,a) + \sum_{s' \in \mathcal{S}} p(s,a,s') V^*(s') \right\} \tag{1.4}$$

This is also a fixed-point condition, which can once again be expressed in terms of an operator called the **Bellman optimal operator**:

$$\mathcal{T}^* v(s) \coloneqq \max_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') v(s') \right\}.$$

Then, we have that the optimal value function satisfies the equation

$$\mathcal{T}^* V^* = V^*$$

1.1.2 The Value Iteration algorithm

Under suitable hypotheses, it can be shown that the Bellman operators $\mathcal{T}^\pi$ and $\mathcal{T}^*$ are contractions with respect to the infinity norm with Lipschitz constant γ. This leads to the following algorithm:

Algorithm 1.1: Value iteration

Input: Rewards $r(s,a)$, transitions $p(s,a,s')$, initial value proposal V_0.

foreach $k = 1, \dots, K$ **do**
  $V_k \leftarrow \mathcal{T}^* V_{k-1}$;
foreach $s \in \mathcal{S}$ **do**
  $\pi_K(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s,a,s') V_K(s') \right\}$;
return Policy π_K, value V_K;

This algorithm is especially useful for discrete state and action spaces. It can be significantly sped up if we have sparse representations of the rewards and transitions.

1.2 The Q-function

Definition 3 (State-action value function) *The state-action value function of a policy π is the function $Q^\pi \colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is defined by*

$$Q^\pi(s,a) \coloneqq \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \,\middle|\, s_0 = s, a_0 = a \right] \tag{1.5}$$

where the trajectory τ is generated under the policy π. The horizon T of the problem can be finite or infinite (T can be a stopping time).

The state-action value function Q^π has an obvious link to the value function V^π. For any state $s \in \mathcal{S}$, it holds that

$$V^\pi(s) = \mathbb{E}_{a \sim \pi(s,\cdot)} \left[Q^\pi(s,a) \right] \tag{1.6}$$

Definition 4 (Optimal action value function) *Optimal policies as defined in section 1.1.1 also share the same* **optimal action value function**

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{1.7}$$

for $s \in \mathcal{S}$, $a \in \mathcal{A}$.

We also have a link between the optimal value functions:

$$Q^*(s,a) = \mathbb{E}_\pi \left[r_t + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a \right]$$

1.3 Temporal-difference estimation – TD(0)

The real value function V^π satisfies the Bellman equation. This means that the **temporal difference error** of a good estimate $\widehat{V}^\pi$ of V^π, defined as

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

should be small.

Chapter 2

Approximate solving of Markov Decision Processes with Policy Gradients

Solving MDPs is seeking the maximizing policy of the value function. For approximate solving of MDPs, we target what could be a more general **policy performance metric**. Often, it is indeed connected to the value function

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right] = \mathbb{E}_{\tau\sim\pi}\left[R(\tau)\right] \tag{2.1}$$

where $\tau = \{s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T\}$ and $R(\tau) = \sum_{t=0}^{T}\gamma^t r_t$ is the total return of the trajectory τ. We use the shorthand r_t for the reward $r(s_t, a_t)$ – or more generally a transition reward $r(s_t, a_t, s_{t+1})$ or stochastic reward distributed as $r_t \sim p(\cdot, s_t, a_t, s_{t+1})$. The expectation J is either conditional on a given starting point s_0, or on a distribution for it.[1]

We seek to compute the maximizing policy in a parametric search space $\{\pi_\theta : \theta \in \Theta\}$:

$$\max_\theta J(\pi_\theta)$$

Iterative methods could be used if J could be computed in closed form with given reward and transition structures; if these are not given they could be estimated by Monte Carlo methods, which would be very expensive and wasteful.

Instead, we will update the estimate as we go and simulate trajectories, by iteratively updating the policy parameter θ using a gradient ascent method with an estimated gradient.

Proposition 2 (Gradient under a parametric law) *Given a set of probability models $\{P_\theta : \theta \in \Theta \subseteq \mathbb{R}^d\}$ on a set $\mathcal{X}$ and a function $f\colon \mathcal{X} \to \mathbb{R}$, we have that*

$$\nabla_\theta \mathbb{E}_{X\sim P_\theta}[f(X)] = \mathbb{E}_{X\sim P_\theta}[f(X)\nabla_\theta \log P_\theta(X)]$$

This is a useful property for deriving estimators of the derivatives in optimization problems with stochastic objectives.

Generalization to the case where f also depends on θ is straightforward.

[1] For instance, OpenAI Gym's `CartPole-v1` environment has a stochastic initial state s_0.

This can be shown either by either writing the expectation as an integral, or by a change of measures with a Radon-Nikodym derivative.

Proposition 2 allows us to write the gradient of (2.1), called the **policy gradient** as an expectation:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[R(\tau)\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(s_t, a_t)\right] \tag{2.2}$$

and we will need to derive estimations for this quantity.

There are other ways of writing the policy gradient, such as (see [1, chap. 13])

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_s\left[\sum_a Q^{\pi_\theta}(s,a)\nabla\pi_\theta(s,a)\right]$$

2.1 Monte Carlo policy gradient: the REINFORCE algorithm

The idea. The policy gradient (2.2) is an expectation which can be estimated using Monte Carlo approximation. We obtain the following estimate:

$$\widehat{\nabla_\theta J}(\pi_\theta) = \frac{1}{M}\sum_{i=1}^{M} R(\tau_i)\sum_{t=0}^{T_i}\nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{2.3}$$

This is an unbiased Monte Carlo estimate of the policy gradient. It only requires suitable regularity of the parametric policy model $\theta \longmapsto \pi_\theta$.

Remark 3 *Equation (2.3) can be used as-is for functions with simple closed-form derivatives. In an automatic differentiation framework such as* PyTorch*, we can instead get the policy gradient from a computational graph with the following pseudo-loss function:*

$$\begin{aligned}\tilde{J}(\theta) &= \frac{1}{M}\sum_{i=1}^{M} R(\tau_i)\sum_{t=0}^{T_i}\log \pi_\theta(s_t^i, a_t^i)\\ &= \frac{1}{M}\sum_{i=1}^{M}\left(\sum_{t=0}^{T_i}\gamma^t r_t^i\right)\sum_{t=0}^{T_i}\log \pi_\theta(s_t^i, a_t^i)\end{aligned} \tag{2.4}$$

We will simulate multiple trajectories (*episodes*) to perform the gradient update: this is the similar to what is done with batch, mini-batch or stochastic gradient steps. This leads to algorithm 2.1.

Algorithm 2.1: Monte Carlo Policy Gradient (REINFORCE)

Input: Arbitrary initial policy π_{θ_0}.
Output: Optimal parametric policy π_{θ^*}.

repeat
  Simulate a trajectory τ;
  $\boldsymbol{g} \leftarrow (\sum_{t=0}^{T}\gamma^t r_t)\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(s_t, a_t)$; `// policy gradient`
  $\theta \leftarrow \theta + \alpha\boldsymbol{g}$;
until *finished*;

This algorithm can be modified in several ways, by performing the gradient step at every time in the process, or on the entire batch of trajectories. Sutton and Barto [1] actually use the more fine-grained update scheme wherein the parameter θ is updated at each step t of every episode[2].

2.1.1 Variance reduction: temporal structure and baselines

Temporal structure. We can re-weigh the log-probability gradients in eq. (2.2) by exploiting the fact that, for any time t, the cumulative rewards $\sum_{t'=0}^{t-1} \gamma^{t'} r_{t'}$ from 0 to $t-1$ are measurable with respect to the trajectory up to t, $\tau_{0:t}$:

Proposition 3 *The policy gradient (2.2) can be rewritten as*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \sum_{t'=t}^{T} \gamma^{t'} r_{t'} \nabla_\theta \log \pi_\theta(s_t, a_t) \right] \tag{2.5}$$

which leads to the policy gradient estimate

$$\widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=0}^{T_i} \gamma^t G_t^i \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{2.6}$$

where[a] $G_t^i = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^i$.

[a] This quantity is an estimate of the Q-function $Q^\pi(s_t, a_t) = \mathbb{E}[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \mid s_t, a_t]$.

The following algorithm provides an efficient recursive method for computing the returns G_t^i:

Algorithm 2.2: Computing the returns

Input: The rewards $(r_t)_{0 \le t \le T}$, discount factor γ
Output: The array of discounted returns.

$R_T \leftarrow r_T$;
foreach $t \leftarrow T-1$ **to** 0 **do**
  $G_t \leftarrow r_t + \gamma G_{t+1}$;
return $(G_t)_{0 \le t \le T}$;

Baselines. Given any **baseline** function $b \colon \mathcal{S} \to \mathbb{R}$, we can rewrite the policy gradient again as

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{T} \left(\sum_{t'=t}^{T} \gamma^{t'} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(s_t, a_t) \right] \tag{2.7}$$

The resulting policy gradient estimate we get is

$$\widehat{\nabla_\theta J(\pi_\theta)} = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=0}^{T_i} \gamma^t (G_t^i - b(s_t^i)) \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{2.8}$$

[2] This allows for fully online learning.

which is an unbiased estimate.

It can be shown that the "best" baseline in terms of variance reduction b^* is the value function:

$$b^*(s_t) = \mathbb{E}_{\pi_\theta}\left[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \,\middle|\, s_t\right]$$

…which we are already trying to approximate. This suggests that we use some kind of **bootstrap** estimate for the baseline.

An obvious, nonparametric baseline is Monte Carlo estimation:

$$\widehat{b}_t = \frac{1}{M}\sum_{i=1}^{M} G_t^i$$

This baseline is an unbiased estimate: following eq. (1.6) the expectation of the Q-function estimate G_t under the policy π_θ is

$$\mathbb{E}_{a_t\sim\pi_\theta(s_t,\cdot)}\,\mathbb{E}_{\pi_\theta}\left[G_t\right] = \mathbb{E}_{a_t\sim\pi_\theta(s_t,\cdot)}\,\mathbb{E}_{\pi_\theta}\left[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \,\middle|\, s_t, a_t\right] = V^\pi(s_t)$$

2.1.2 Parametric Bootstrapping of the baseline

We define the bootstrap estimate $b(s) \coloneqq \widehat{v}_\nu(s)$ lying in a parametric search space $\{\widehat{v}_\nu : \nu \in \mathcal{V}\}$. The value parameter can be iteratively updated using gradient steps alternatively with the policy parameter θ.

For a given trajectory sample $\tau = \{s_0, a_0, r_0, \ldots\}$, introduce the mean-squared error between the forward cumulative rewards (a nonparametric estimate of the value function) and the output of the value model:

$$\mathcal{L}(\nu;\tau) = \sum_{t=0}^{T}\left(G_t - \widehat{v}_\nu(s_t)\right)^2$$

Then before each update of the policy π_θ, update the value parameter ν using either the gradient of $\mathcal{L}$.

The adapted episodic learning algorithm with an adaptive baseline is as follows:

Algorithm 2.3: REINFORCE with parametric baseline

repeat
  Simulate a trajectory τ;
  Compute the returns (G_t) of the trajectory;
  $\boldsymbol{g} \leftarrow \sum_{t=0}^{T} \gamma^t (G_t - \widehat{v}_\nu(s_t)) \nabla_\theta \log \pi_\theta(s_t, a_t)$; `// policy gradient`
  $\nu \leftarrow \nu - \beta \nabla_\nu \mathcal{L}(\nu;\tau)$; `// update value estimate`
  $\theta \leftarrow \theta + \alpha \boldsymbol{g}$; `// update policy`
until *finished*;

As before (see remark 3), this algorithm can be implemented within an automatic differentiation framework such as PyTorch by defining the right computational

graphs. The associated pseudo-loss would be

$$\tilde{J}(\theta) = \sum_{t=0}^{T} \gamma^t (G_t - \widehat{v}_\nu(s_t)) \log \pi_\theta(s_t, a_t)$$

Remark 4 *The difference between the returns and the value approximation $G_t - \widehat{v}_\nu(s_t)$ is a (biased) estimate of the temporal-difference error $\delta_t = r_t + \gamma\widehat{v}(s_{t+1}) - \widehat{v}(s_t)$. Thus, the mean-squared error $\mathcal{L}$ can be seen as an estimate of the Bellman error of the value proposal $\widehat{v}_\nu$.*

2.2 Bootstrapped returns: Actor-Critic algorithms

The idea. The enhanced REINFORCE algorithm builds estimates of the value function to compute the policy gradient: this Monte Carlo method is computationally expensive, and may still lead to high variance. To combat this, it might be a good idea to *learn* from the Monte Carlo estimates in a way that gives a consistent estimate that follows the policy gradient updates.

To achieve this, the class of **actor-critic methods** introduces a second search space for approximation of the state(-action) value function and uses **bootstrapped temporal-difference estimates** for the returns G_t. The supervised baseline of section 2.1.2 was already a first step towards this, but this time we actually use the value estimate for policy evaluation and not only for variance reduction. The subtle difference is further explained in [1, chap. 13.5].

2.2.1 Actor-Critic

Learning the policy is still done by gradient steps, using estimates of the form eq. (2.8). But this time, we replace in the objective J the Monte Carlo returns G_t, which estimate the Q-function, by a parametric estimator $\widehat{q}_\omega(s_t, a_t)$ called the **critic**. As the algorithm runs, the critic learns how to value each action suggested by the learned policy π (called the **actor**) by looking at the bootstrapped return estimates. The policy gradient becomes

$$\widehat{\nabla_\theta J}(\pi_\theta) = \frac{1}{M} \sum_{i=1}^{M} \sum_{t=0}^{T_i} \widehat{q}_\omega(s_t^i, a_t^i) \nabla_\theta \log \pi_\theta(s_t^i, a_t^i) \tag{2.9}$$

Now, we want the critic $\widehat{q}_\omega$ to minimize the **Bellman error**

$$\mathbb{E}_{\pi_\theta}\left[(r_t + \gamma\widehat{q}_\omega(s_{t+1}, a_{t+1}) - \widehat{q}_\omega(s_t, a_t))^2 \mid s_t, a_t\right]$$

because having it as small as possible will ensure that it follows the dynamic programming principle, making it a "good" estimate of the real action-state value function Q^{π_θ}. The above expectation can be estimated using the TD(0) error of $\widehat{q}_\omega$:

$$\delta_t = r_t + \gamma\widehat{q}_\omega(s_{t+1}, a_{t+1}) - \widehat{q}_\omega(s_t, a_t).$$

which we will seek to minimize for every time step or episode alternatively with the actor loss.

This leads to the following algorithm:

Algorithm 2.4: Actor-Critic

Input: Initial policy parameter θ, value parameter ω
Output: Policy π_{θ^*}, action-state value approximation $\widehat{q}_{\omega^*}$

repeat
  Simulate trajectory τ;
  foreach $t = 0, \dots, T$ **do**
    $\delta_t \leftarrow r_t + \gamma \widehat{q}_\omega(s_{t+1}, a_{t+1}) - \widehat{q}_\omega(s_t, a_t)$; // TD(0) error
  $\omega \leftarrow \omega + \beta \sum_{t=0}^{T} \delta_t \nabla_\omega \widehat{q}_\omega(s_t, a_t)$; // critic update
  $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \widehat{q}_\omega(s_t, a_t) \nabla_\theta \log \pi_\theta(s_t, a_t)$; // actor update
until *finished*;

2.2.2 Actor-Critic with baselines: Advantage Actor-Critic (A2C)

As we've seen before, a good baseline to introduce in the policy gradient is the value function – see eq. (2.7). The coefficient before the log-probability gradient in (2.2) becomes an estimate of $Q^\pi(s,a) - V^\pi(s) = \mathbb{E}_{s'}[r(s,a) + \gamma V^\pi(s') \mid s, a]$. We introduce the **advantage** function

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{2.10}$$

An unbiased estimate of A^π is the temporal difference error of the value function

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

This can be seen by taking the expectation with respect to s_{t+1} conditionally on (s_t, a_t).

This leads to the (bootstrapped) estimate, using a parametric approximation $\widehat{v}_\nu$ of the state value function:

$$\widehat{\delta}_t = r_t + \gamma \widehat{v}_\nu(s_{t+1}) - \widehat{v}_\nu(s_t).$$

Now, we have no need to learn a action-state critic $\widehat{q}_\omega$, since we have TD error estimates that can be expressed only using $\widehat{v}_\nu(s)$ – which is simpler.

The A2C algorithm is as follows:

Algorithm 2.5: Advantage Actor-Critic (A2C)

Input: Initial policy parameter θ, value parameter ν
Output: Policy π_{θ^*}, value approximation $\widehat{v}_{\nu^*}$

repeat
  Simulate trajectory τ;
  foreach $t = 0, \dots, T$ **do**
    $\delta_t \leftarrow r_t + \gamma \widehat{v}_\nu(s_{t+1}) - \widehat{v}_\nu(s_t)$; // TD(0) error
  $\nu \leftarrow \nu + \beta \sum_{t=0}^{T} \delta_t \nabla_\nu \widehat{v}_\nu(s_t)$; // critic update
  $\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \widehat{v}_\nu(s_t) \nabla_\theta \log \pi_\theta(s_t, a_t)$; // actor update
until *finished*;

Remark 5 *To leverage automatic differentiation, appropriate pseudo-losses to define a computational graph for the updates in algorithm 2.5 are*

$$\tilde{C}(\nu) = \frac{1}{M}\sum_{i=1}^{M}\sum_{t=0}^{T}\delta_t^i \widehat{v}_\nu(s_t^i) \tag{2.11a}$$

$$\tilde{J}(\theta) = \frac{1}{M}\sum_{i=1}^{M}\sum_{t=0}^{T}\widehat{v}_\nu(s_t^i)\log\pi_\theta(s_t^i, a_t^i) \tag{2.11b}$$

where the temporal-difference estimates δ_t^i must be detached from the graph.

The value update step in algorithm 2.5 (and also algorithm 2.4) can be seen as a supervised regression step, where we fit the (bootstrapped) returns $G_{t:t+1} = r_t + \gamma\widehat{v}_\nu(s_{t+1})$ to the value model predictions $\widehat{v}_\nu(s_t)$, with a mean-squared loss

$$\mathcal{L}(\nu) = \sum_{t=0}^{T}(G_{t:t+1} - \widehat{v}_\nu(s_t))^2$$

We then perform a semi-gradient update step, taking the gradient with respect to ν whilst ignoring the dependency of the return estimate $G_{t:t+1}$ on it.

Remark 6 *This means that the pseudo-loss $\tilde{C}(\nu)$ in remark 5 can be replaced by the MSE $\mathcal{L}(\nu)$.*

We create the estimate $G_{t:t+1}$ by only looking at the immediate reward and bootstrapping the rest of the value: this is called **one-step actor-critic**. It has poor sample efficiency: the return estimates ignore most of the (actual) future rewards and end up bootstrapping too much. However, it naturally allows for *online learning*.

This can be easily generalized to using more rewards for bootstrapping the returns: ***n*-step actor-critic** methods look further ahead and use estimates

$$G_{t:t+n} = \sum_{k=0}^{n-1}\gamma^k r_{t+k} + \gamma^n\widehat{v}_\nu(s_{t+n})$$

Batch actor-critic. In batch algorithms, we switch out episodic simulation with epochs of simulation-and-reset batches, where we simulate trajectories (and reset

them at terminal states) until a fixed number B of updates have been done.

Algorithm 2.6: Batch A2C

Input: Number of epochs N, batch size B

foreach $n = 1, \ldots, N$ **do**
  $s_0 \leftarrow \texttt{Reset}()$;
  foreach $i = 0, \ldots, B-1$ **do**
    $v_i \leftarrow \widehat{v}_\nu(s_i)$; `// estimate value`
    Draw action $a_i \sim \pi_\theta$;
    Get state, reward (s_{i+1}, r_i);
    `// check if we just finished a trajectory`
    if s_{i+1} *is terminal* **then**
      $s_{i+1} \leftarrow \texttt{Reset}()$;
  if s_B *is terminal* **then**
    $v_B \leftarrow 0$;
  else
    $v_B \leftarrow \widehat{v}_\nu(s_B)$;
  Compute bootstrapped advantages, returns $(\delta_i, \widehat{G}_i)$;
  $\nu \leftarrow \nu + \beta \sum_{i=0}^{B-1} \delta_i \nabla_\nu \widehat{v}_\nu(s_i)$;
  $\theta \leftarrow \theta + \alpha \sum_{i=0}^{B-1} \delta_i \nabla_\theta \log \pi_\theta(s_i, a_i)$;

The returns $\widehat{G}_i$ have to be computed taking into account when trajectories end. An efficient algorithm is

$$\widehat{G}_i = \begin{cases} r_i + \gamma \widehat{G}_{i+1} & \text{if } s_i \text{ is not terminal} \\ r_i & \text{otherwise} \end{cases}$$

for $i = 0, \ldots, B-2$ and $G_{B-1} = r_{B-1} + \gamma v_B$. The $\mathsf{TD}(0)$ errors are $\delta_i = \widehat{G}_i - v_i$.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.