Sutton Barto Notes

Maximillian Guo
mguo@college.harvard.edu

2021

Contents

4 Chapter 4: Dynamic Programming

DP refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a MDP.

We can easily obtain optimal policies once we've found the optimal value functions v_* or q_* that satisfy Bellman optimality equations:

$$v_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \tag{1}$$
$$= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{2}$$

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \tag{3}$$
$$= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{4}$$

4.1 Policy Evaluation

Recall the definition:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{5}$$

Policy evaluation solves this iteratively:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{6}$$

4.2 Policy Improvement

Theorem 4.1 (Policy Improvement)**.** *Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$:*

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{7}$$

Then, the policy π' must be as good as, or better than, π. For all $s \in \mathcal{S}$,

$$v'_\pi(s) \geq v_\pi(s). \tag{8}$$

Definition 4.1 (Policy Improvement)**.** The process of making a new policy that improves upon the original policy by making it greedy with respect to the value function of the original policy:

$$\pi'(s) = \text{argmax}_a q_\pi(s, a) \tag{9}$$

4.3 Policy Iteration

Definition 4.2 (Policy Iteration)**.** Policy iteration is repeated policy evaluation and policy improvement.

This is guaranteed to converge on the optimal policy in a finite MDP (with a finite number of policies).

Definition 4.3 (ε-soft Policy)**.** Probability of selecting each action in each state s is at least $\varepsilon/|\mathcal{A}(s)|$.

4.4 Value Iteration

Policy iteration requires, in each step, the itself-iterative process of Policy Evaluation. We can truncate this step after jut one sweep $\rightarrow$ value iteration.

The update becomes:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{10}$$

Note that this is just turning the Bellman optimality equation into an update rule!

4.5 Asynchronous Dynamic Programming

Major drawback so far - they involve operations over the entire state set of the MDP. What about large state sets?

Asynchronous DP $\rightarrow$ update values of states in any order, using whatever values are available. The algorithm does not need to get locked into a hopelessly long sweep before making progress to improve a policy.

Asynchronous algorithms in general $\rightarrow$ easier to intermix computation with real-time interaction.

4.6 Generalized Policy Iteration (GPI)

GPI: general idea of letting policy evaluation and policy improvement interact, independent of details of the two processes.

4.7 Efficiency of Dynamic Programming

Out of all methods for solving MDP's, DP methods are quite efficient. The time it takes to find an optimal policy is polynomial in the number of states and actions.

4.8 Summary

All of the DP methods use *bootstrapping*: they update estimates on the basis of other estimates.

5 Chapter 5: Monte Carlo Methods

We now do not assume complete knowledge of the environment. Instead, we learn from actual experience.

5.1 Monte Carlo Prediction

- *first-visit MC method*: estimates $v_\pi(s)$ as the average of the returns following first visits to s.
- *every-visit MC method*: averages the returns following all visits to s.

Both methods converge to true values.
Monte Carlo methods do not bootstrap, since estimates for one state do not build upon estimates of other states.

5.2 Monte Carlo Estimation of Action Values

If a model is not available, useful to estimate action values (state-action pairs).
Complication: What if not all state-action pairs are visited? Need to maintain exploration:

- *Exploring Starts*: specifying that episodes start in a state-action pair, and every pair has nonzero probability of being selected.

5.3 Monte Carlo Control

Control $\rightarrow$ estimate approximate optimal policies. Policy improvement using action-value functions:

$$\pi(s) = \text{argmax}_a q(s, a) \tag{11}$$

5.4 Monte Carlo Control without Exploring Starts

- *On-policy* methods: evaluate or improve the policy that is used to make decisions
- *Off-policy* methods: evaluate or improve a policy different from that used to generate the data.
- *soft* policy: $\pi(a|s) > 0$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$.
- ε-soft policies remove the assumptions about exploring starts
- ε-greedy policies among the best ε-soft policies.

5.5 Off-Policy Prediction via Importance Sampling

How can we learn about an optimal policy while behaving according to an exploratory one? Use two policies.

- *Target policy* $\rightarrow$ policy being learned about.
- *Behavior policy* $\rightarrow$ policy used to generate behavior.

Prediction problem $\rightarrow$ wish to estimate v_π or q_π, but we only have episodes following policy $b \neq \pi$.

- Coverage assumption: $\pi(a|s) > 0 \implies b(a|s) > 0$.

Importance sampling $\rightarrow$ estimating expected values under one distribution given samples from another.

- Given starting state S_t, the probability of subsequent state-action trajectory under any policy π is:

$$P(A_t, S_{t+1}, A_{t+1}, \ldots, S_T | S_t, A_{t:T-1} \sim \pi) = \prod_{k=1}^{T-1} \pi(A_k|S_k) p(S_{k+1}|S_k, A_k) \tag{12}$$

- The relative probability of a trajectory under the target and behavior policies is:

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{13}$$

  since the $p(S_{k+1}|S_k, A_k)$ terms cancel.

- $v_b(s) = E[G_t|S_t = s]$. Transforming this, we have: $E[\rho_{t:T-1}G_t|S_t = s] = v_\pi(s)$.

- Let $\mathcal{J}(s)$ denote the set of all time steps in which state s is visited. Let $T(t)$ denote the first time of termination following time t, and G_t denote the return after t up to $T(t)$.

- *ordinary importance sampling*:

$$V(s) = \frac{\sum_{t\in\mathcal{J}(s)} \rho_{t:T(t)-1}G(t)}{|\mathcal{J}(s)|} \tag{14}$$

  This is unbiased but variance is unbounded.

- *weighted importance sampling*:

$$V(s) = \frac{\sum_{t\in\mathcal{J}(s)} \rho_{t:T(t)-1}G_t}{\sum_{t\in\mathcal{J}(s)} \rho_{t:T(t)-1}} \tag{15}$$

  This is biased but variance is low (preferred). Bias falls asymptotically to zero.

5.6 Incremental Implementation

From Chapter 2 methods, on-policy and ordinary importance sampling off-policy methods can be implemented incrementally.

This chapter derives a incremental way of implementing weighted importance sampling.

5.7 Off-policy Monte Carlo Control

Find greedy policy π_*, using ε-soft behavior policy b.

6 Chapter 6: Temporal-Difference Learning

Like DP, TD methods bootstrap and update estimates based in part on other learned estimates.

Like MC, TD methods learn directly from experience without a model.

6.1 TD Prediction

Constant-α MC (waits until episode finishes):

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{16}$$

TD(0) (waits just one step):

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1} - V(S_t))) \tag{17}$$

TD error:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1} - V(S_t)) \tag{18}$$

6.2 Advantages of TD Prediction Methods

TD is implemented in an online, incremental fashion (don't need to wait until the end of the episode).

For a fixed policy π, TD(0) is proved to converge to v_π.
No one knows yet mathematically whether MC or TD converges faster. In practice, TD often converges faster.

6.3 Optimality of TD(0)

batch updating: updates are made only after processing each complete batch of training data.

Under batch updating, TD(0) converges deterministically to a single answer, as long as α is sufficiently small.

- Batch MC - finds estimates that minimizes mean square error on training set
- TD(0) - finds estimates that would be exactly correct for the maximum likelihood model of the Markov process
  - *certainty-equivalence* $\rightarrow$ equivalent to assuming that estimate of the underlying process was known with certainty rather than approximated

6.4 SARSA: On-policy TD Control

Corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{19}$$

Continually estimate q_π and at the same time change π toward greediness with respect to q_π. If we use ε-greedy, SARSA converges wp 1 to optimal policy as long as all state-action pairs are visited an infinite number of times and policy converges in the limit to the greedy policy.

6.5 Q-learning: Off-policy TD Control

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

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

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

Figure 1: Q-learning Algorithm

6.6 Expected SARSA

Instead of taking the maximum over next state-action pairs, it uses the expected value. Given the next state S_{t+1}, this algorithm moves deterministically in the same direction that SARSA moves in expectation.

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

Note that Expected Sarsa can be used on-policy or off-policy. If it is used off-policy with an ε-greedy behavior policy and π is the greedy policy, then it is exactly Q-learning.

6.7 Maximization Bias and Double Learning

A maximum over estimated values used as an estimate of a maximum value can lead to significant positive bias.
Problem: using the same samples both to determine the maximizing action and to estimate its value. *Double learning* - divide the plays in two sets and use them to learn two independent estimates.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

Figure 2: Double Q-learning Algorithm

There are also double versions of SARSA and Expected SARSA.

6.8 Games, Afterstates, and Other Special Cases

For example, in tic-tac-toe, there are many state-action pairs that lead to the same *afterstate* - so it is more efficient to represent afterstate value functions.

6.9 Summary

7 Chapter 7: n-step Bootstrapping

7.1 n-step TD Prediction

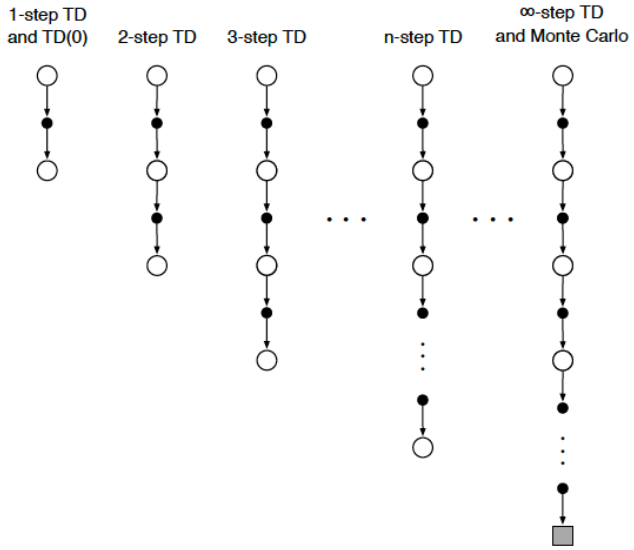

Figure 3: Backup Diagrams

Monte-Carlo updates using the complete return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T.$$

One-step updates (TD):

$$Gt : t+1 = R_{t+1} + \gamma V_t(S_{t+1})$$

Arbitrary n-step update, using n-step return:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}).$$

7.2 n-step SARSA

Much more can be learned in a single episode if many of the actions leading to a nice state are strengthened.

7.3 n-step Off-policy Learning

Uses importance sampling

7.4 Per-decision Methods with Control Variates

7.5 Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm

Generalization of the Expected SARSA return.

7.6 A Unifying Algorithm: n-step $Q(\sigma)$

8 Chapter 8: Planning and Learning with Tabular Methods

8.1 Models and Planning

- *distribution models* - produce a description of all possibilities and their probabilities
- *sample models* - produce one possibility sampled according to the probabilities.
- model is used to simulate the environment and produce simulated experience.
- *planning* - any process that takes a model as input and produces or improves a policy for interacting with the model

Unified state space planning methods go:

$$\text{model} \longrightarrow \text{simulated experience} \longrightarrow \text{values} \longrightarrow \text{policy}$$

8.2 Dyna: Integrated Planning, Acting, and Learning

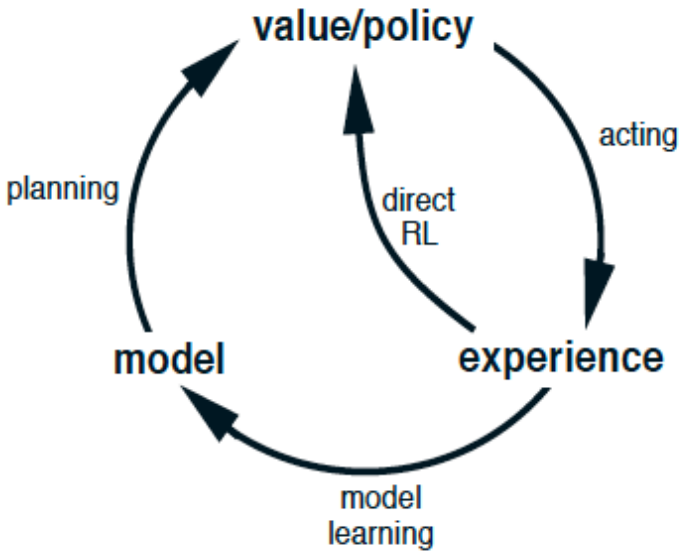

Figure 4: Dyna-Q architecture

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 5: Dyna-Q algorithm

Using the loop from planning, the learning curve for Dyna-Q could converge must faster than direct RL.

8.3 When the Model is Wrong

When model is incorrect, planning process is likely to compute a suboptimal policy.
Simple heuristics are often effective.
E.g. Dyna-Q+ agent that keeps track how many time steps have elapsed since every state-action pair was tried. The more time has elapsed, a greater bonus is given to these simulated experiences to encourage exploration.

8.4 Prioritized Sweeping

Not all state-action pairs will update at the same rate. We can keep a priority queue of every state-action pair whose estimated value would change nontrivially if updated, and use this during the planning portion. This increases the speed at which optimal solutions are found in maze tasks (often by factor of 5 to 10).

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow policy(S,Q)$
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Model(S,A) \leftarrow R, S'$
- (e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.
- (f) if $P > \theta$, then insert S,A into $PQueue$ with priority P
- (g) Loop repeat n times, while $PQueue$ is not empty:
  - $S, A \leftarrow first(PQueue)$
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
  - Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:
    - $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
    - $P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
    - if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

Figure 6: Prioritized Sweeping

8.5 Expected vs. Sample Updates

Expected updates take a long time to update one state-action pair!

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r|s,a)[r + \gamma \max_{a'} Q(s',a')]$$

Sample update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a'} Q(S',a') - Q(s,a)]$$

Sample updates seem to be superior in the case of problems with large stochastic branching factors and too many states to be solved exactly.

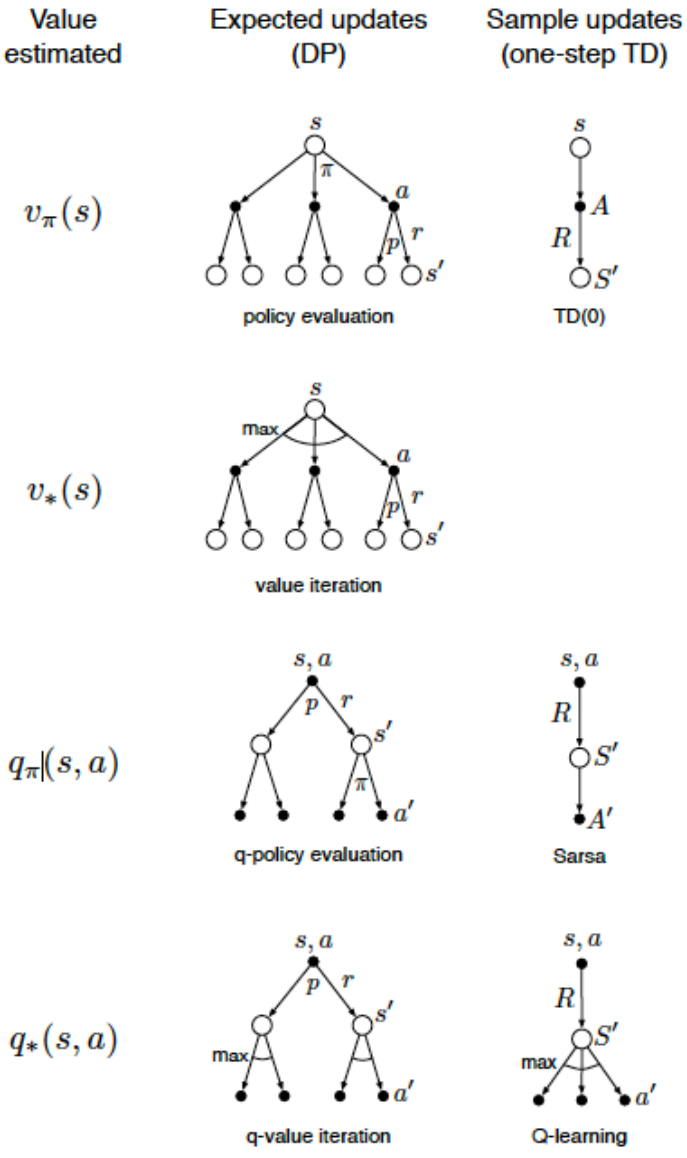

Figure 7: Various backup diagrams

8.6 Trajectory Sampling

Two ways of distributing updates:

- Exhaustive sweeps (e.g. DP): problematic on large tasks, may not be even enough time to complete one sweep.
- Trajectory sampling: start in some start state and simulate until terminal state. (on-policy distribution of updates)

(e.g. if you study positions in chess, you would study those that arise in real games and not random positions of chess pieces)

8.7 Real-time Dynamic Programming

RTDP - asynchronous DP algorithm, updates based on on-policy trajectory sampling using value-iteration as opposed to exhaustive sweeps.
In general, need optimal partial policy - optimal for relevvant states but can be undefined for irrelevant (e.g. unreachable) states.

8.8 Planning at Decision Time

8.9 Heuristic Search

8.10 Rollout Algorithms

8.11 Monte Carlo Tree Search

8.12 Summary

9 Chapter 9: On-policy Prediction with Approximation

9.1 Value-function approximation

Up until now, each update for a state has not affected other states. Now we can use the input-output examples of states and updates/values in supervised learning.
Not all machine learning methods work well - we need function approximation methods that are able to handle nonstationary target functions.

9.2 The Prediction Objective

Natural objective function, the mean square value error:

$$\bar{VE}(\mathbf{w}) = \sum_{s \in \S} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

gives a rough measure of how much approximate values differ from true values. $\mu(s)$ is a weighting function (how much we care about state s).
Sometimes it is only possible to find a local optimum for VE, especially with nonlinear methods.

9.3 Stochastic-gradient and Semi-gradient Methods

Monte Carlo generalization - can update values based on stochastic gradient descent.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t, \mathbf{w})\big]\nabla\hat{v}(S_t, \mathbf{w})$

Figure 8: Gradient Monte Carlo

Guaranteed to find a locally optimal solution for decreasing α.

We don't get the same guarantees with bootstrapping methods, since the target depends on current value of $\mathbf{w}$. Called semi-gradient methods. Still might work though.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$|
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

Figure 9: Semi-gradient Method

9.4 Linear Methods

Corresponding to every state s is a real-valued vector $\mathbf{x}(s) = (x_1(s), \ldots, x_d(s))^\intercal$, and the linear approximation of the state-value function is:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\intercal \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

$\mathbf{x}(s)$ is the feature vector representing s.
With linear function approximation, SGD converges to a global optimum, while semi-gradient TD(0) converges to a local optimum.
At a steady state for semi-gradient TD(0), we have:

$$E[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t)$$

where

$$\mathbf{b} = E[R_{t+1}x_t] \in \mathbb{R}^d$$

and

$$\mathbf{A} = E[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\intercal] \in \mathbb{R}^{d\times d}$$

If this method converges, it converges to the TD fixed point:

$$\mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b}$$

9.5 Feature Construction for Linear Methods

Must choose appropriate features for a given task (a way of incorporating prior domain knowledge). Examples:

- Polynomials
- Fourier series
- Coarse Coding - circles on a two dimensional plane, each state is a point that is within some circles and not within others (binary).
- Coarse coding with tiles
- Radial Basis Functions - generalization of Coarse Coding to continuous valued features.

9.6 Selecting Step-Size Parameter Manually

9.7 Nonlinear Function Approximation

(Neural networks and some basic theory)

9.8 Least-Squares TD

(LSTD) does not compute the TD fixed point iteratively, but instead directly computes the matrix inverse. It is more expensive computationally but more data efficient.

9.9 Memory-based Function Approximation

So far, we discussed parametric methods.
Nonparametric methods include memory based function approximation methods, which simply save the training examples in memory. Whenever a value estimate is needed, they'll retrieve the memory and compute a value estimate.

For example:

- Nearest neighbors methods
- Locally weighted regression

9.10 Kernel-based Function Approximation

Function that assigns weights in memory based methods is known as the kernel. For $k : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$, $k(s, s')$ is the weight given to data s' in its influence on answering queries about s.

Kernel functions numerically express how relevant knowledge about any state is to any other state.

Kernel regression:

$$v'(s, \mathcal{D}) = \sum_{s' \in D} k(s, s') g(s')$$

For example, kernel regression with the kernel:

$$k(s, s') = \mathbf{x}(s)^\intercal \mathbf{x}(s')$$

produces the same approximation that a linear parametric method would. Instead of constructing features for linear parametric function approximators, can instead construct kernel functions directly (the "kernel trick").

9.11 Looking Deeper at On-policy Learning: Interest and Emphasis

10 Chapter 10: On-policy Control with Approximation

10.1 Episodic Semi-gradient Control

Now we approximate an action-value function $\hat{q} \approx q_\pi$, which is parametrized with weight vector $\mathbf{w}$. Th general gradient descent update is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t).$$

For one-step SARSA, we have:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[R_{t+1} - \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla\hat{q}(S_t, A_t, \mathbf{w}_t).$$

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Figure 10: Episodic Semi-gradient Sarsa

10.2 Semi-gradient n-step SARSA

10.3 Average Reward: A new Problem Setting for Continuing Tasks

10.4 Deprecating the Discounted Setting

10.5 Differential Semi-gradient n-step SARSA

10.6 Summary

11 Chapter 11: Off-policy Methods with Approximation

12 Chapter 12: Eligibility Traces

13 Chapter 13: Policy Gradient Methods

14 Chapter 14: Psychology

15 Chapter 15: Neuroscience

16 Chapter 16: Applications and Case Studies

17 Chapter 17: Frontiers