6.0: Intro

TD Learning is a combination of:

- MC methods (can learn from experience without knowing the model). No bootstrapping bc the final return is known.
- Dynamic programming (update estimate based on other learned estimates). Bootstrapping method.

Each error is proportional to the change over the time of the prediction, that is, to the temporal differences in predictions.

TD Learning:

learns value functions by bootstrapping - using current estimates to under other estimates.

6.1: TD Prediction

MC waits until the return following the visit is known and then uses it as a target for $V(S_t)$. Every-visit MC:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right], \text{ where}$$

- G_t is the actual return following time t
- α = a constant step-size parameter.

TD methods need only to wait until the next time step. At $t+1$, we use the observed reward R_{t+1} and the estimate $V(S_{t+1})$ to create a target:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{target}} - V(S_t)\right]. \quad (6.2)$$

Instead of using G_t (actual return @ time t)

This method is called TD(0) or one-step TD because it's a special case of TD(λ) which will be discussed in ch 12 (eligibility traces) and ch 7 (n-step bootstrapping).

Tabular TD(0) for estimating v_π // prediction

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

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

Because TD(0) bases its update on an existing estimate (as opposed to the real value), we say it's a bootstrapping method, like DP. From this chapter we know:

$$v_\pi(s) \doteq \mathbb{E}[G_t \mid S_t = s] \tag{6.3}$$

$$= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s]$$

$$= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \tag{6.4}$$

Roughly speaking, MC uses an estimate of (6.3) as a target, DP uses an estimate of 6.4. MC uses an estimate bc the expected value is not known, and DP uses an estimate because $v_\pi(S_{t+1})$ is not known, and we use $V(S_{t+1})$ instead. The TD target is an estimate for both reasons.

- It samples the expected values in (6.4).
- And it uses the current estimate V instead of v_π.
- The value estimate for the state node at the top of the backup diagram is updated on the basis of one sample transition from it to the immediate following state.
- Sample updates differ from expected updates of DP methods in that they are based on a single sample successor instead of a complete distribution over all possible successors.
- The quantity in brackets in (6.2) is a sort of error, measuring the difference btwn the estimated value of $V(S_t)$ and the better estimate $R_{t+1} + \gamma V(S_{t+1})$, and is called the TD error:

$$\delta_t \doteq \{R_{t+1} + \gamma V(S_{t+1})\} - V(S_t) \quad \text{// TD error.}$$

The TD error is the error made at that time.
δ_t is available at time step $t+1$.

If the array V does not change during the episode, then the MC error can be written as a sum of TD errors:

$$G_t - V(S_t) = R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1})$$
$$= \delta_t + \gamma(G_{t+1} - V(S_{t+1}))$$
$$= \delta_t + \gamma\delta_{t+1} + \gamma^2(G_{t+2} - V(S_{t+2}))$$
$$= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(G_T - V(S_T))$$
$$= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}(0-0).$$
$$= \boxed{\sum_{k=t}^{T-1} \gamma^{k-t}\delta_k}$$

This identity is not exact if V is updated during the episode (as in TD(0)), but if the step size is small it may hold approximately.

6.2: Advantages of TD prediction methods

- they do not require a model of the environment.
- online, fully incremental updates (no delay, no waiting for the end of the episode like in MC).
- For a fixed policy π, TD(0) has been shown to converge to V_π (discussed in Ch 9).
- TD methods usually converge faster than MC methods.

Example:

State A, current estimates are:

$V(A) = 10$

$V(B) = 5$.

You take an action, get reward 2, and end up in state B. Assume $\gamma = 0.9$, TD target becomes:

TD target: $R_{t+1} + \gamma S_{t+1} = 2 + (0.9)(5) = 6.5$

TD error: $6.5 - 10 = -3.5$

$V(A) \longleftarrow 10 + 0.1(-3.5) = 9.65$.

6.3: Optimality of TD(0)

Batch Updating

Say we have a limited amount of experience. In this case a common approach is to present the experience repeatedly until the method converges. Given an approximate value function V, the increment in $V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$ is computed for every time step t at which a nonterminal state is visited, but the value function is only changed once, by the value of the increments. Then all the experience is presented again with the new value function to produce a new overall increment, and so on until convergence. We call this batch updating because updates are made only after processing each complete batch of training data.

Convergence:

under batch updating, TD(0) converges deterministically to a single answer given that α is sufficiently small. The constant-α MC also converges, but to a different answer. Understanding those differences will help understand the differences btwn the 2 methods.

certainty-equivalence estimate

These examples help explain why TD(0) converge faster than MC.

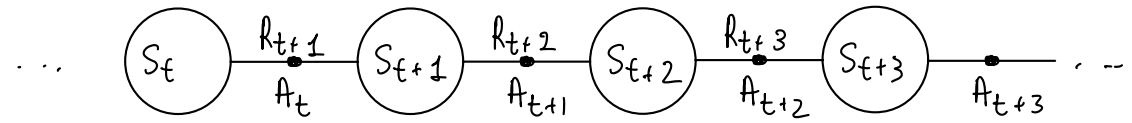
- In batch form, it computes the true certainty-equivalence estimate.
- Non-batch TD(0) may be faster than MC because it is moving towards a better estimate (even if it's not getting all the way there).

The certainty-equivalence estimate is in some sense an optimal solution, but it is almost never feasible to compute it directly. If $n = |\mathcal{S}|$ is the number of states, then forming the maximum likelihood estimates takes $O(n^2)$ memory and computing the corresponding value function takes $O(n^3)$ computational steps. It's nice to see that TD can approximate this in $O(n)$, and in problems with a large state space, TD methods can be the only way to approximate the certainty equivalence solution.

6.4: SARSA: On-policy TD control

Here we present an **on-policy TD control method.**

First, we **learn an action value function**, rather than a state-value function. For an on-policy method, we must **estimate $q_\pi(s,a)$** for the current policy behavior π for all s and all a. We'll use the same TD method as above for learning v_π. Here's an episode:



- previous section: we considered transitions from state-action pairs to states and learned the **values of states.**
- Now: we consider **transitions from state-action pairs to state-action pairs**, and **learn the values of state-action pairs.**

Formally, the two cases are identical, meaning the theorems assuring convergence of state value under TD(0) also holds for action values:

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

This update is done after every transition from a non-terminal state S_t. **If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1}) = 0$.**
This rule makes use of five elements: S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, hence the algorithm name.

On-policy control algorithm:
we continually estimate q_π for the behavior policy π and at the same time change π towards greediness wrt q_π.

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

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

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

Here, we try to learn action values:

$$Q_\pi(s, a) = \mathbb{E}\left[G_t \mid S_t = s, A_t = a, \text{following policy } \pi \right]$$

$$G_t = R_{t+1} + \gamma G_{t+1}$$

$$\Rightarrow Q_\pi(s, a) = \mathbb{E}\left[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a, \text{following } \pi \right]$$

// Bellman Eq for Action Values under π

$\Rightarrow$ update Rule:

$$Q(S_t, A_t) \leftarrow \underbrace{Q(S_t, A_t)}_{\text{current estimate}} + \alpha \Big[\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{target}} - Q(S_t, A_t) \Big]$$

Convergence:
convergence properties of SARSA depends on the nature of the policy dependence to Q. Sarsa converges with probability 1 to an optimal policy and action-value function as long as all the state-value pairs are visited an infinite amount of time and the policy converges in the limit to the greedy policy.

6.5: Q-learning: Off-Policy TD Control

Q-learning learns the optimal Q-function Q^*.

Recall the Bellman Optimality Equation:

$$Q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right]$$

// notice the max instead of following policy π.

The Q-learning update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[\overbrace{R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')}^{\text{target}} - Q(S_t, A_t)\right]$$

different from SARSA!

$\max_{a'} Q(S_{t+1}, a')$ instead of $Q(S_{t+1}, A_{t+1})$

The target assumes taking the best action onward, regardless of the policy. This reevaluates $Q(S_t, A_t)$ in setting of the best actions instead of the policy actions.

Here, the learned value function Q directly approximates q_*, independent of the policy being followed.

The policy effect is to determine which state-action pairs are visited and updated.

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 $Q(\text{terminal}, \cdot) = 0$

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

This means:

- Your behavior policy (what actions you actually take) can be anything.
- Your target policy (what you're learning) is always optimal.

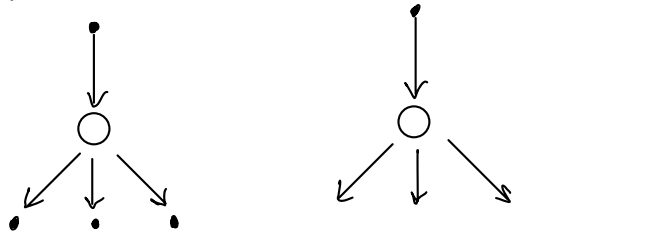
Convergence: All is required for convergence is that all state-action pairs continue to be visited and updated.
Thus, as long as your behavioral policy visits all state-actin pairs infinitely, Q-learning converges to Q_*.

6.6 : Expected Sarsa :

Same as Q-learning except that instead of taking the max over the next actions, we use the expected value, taking into account how likely each action is under the current policy.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma E\left[Q(S_{t+1}, A_{t+1}) \mid S_{t+1} \right] - Q(S_t, A_t) \right]$$
$$= Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_{a'} \pi(a' \mid S_{t+1}) \cdot Q(S_{t+1}, a') - Q(S_t, A_t) \right]$$

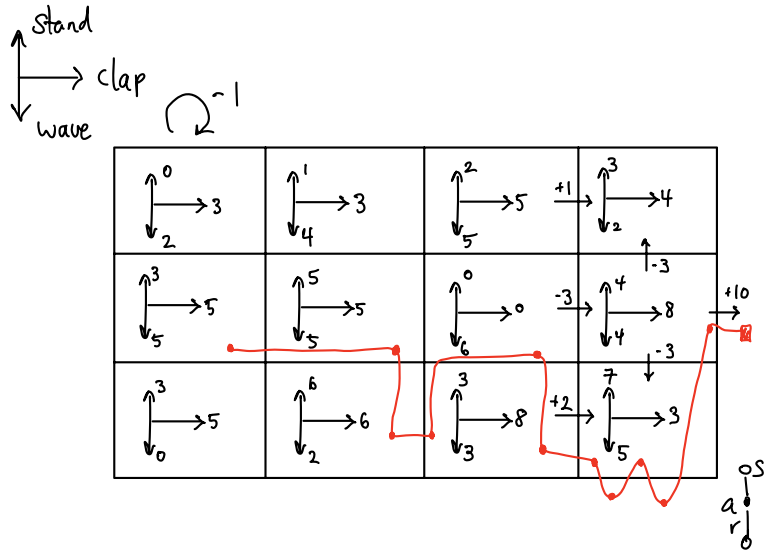
Given the next state S_{t+1}, this algorithm moves deterministically in the same direction that Sarsa moves in expectation (hence the name).



Q - learning Expected Sarsa.

It is more complex than Sarsa but removes the variance due to the random selection of A_{t+1}. Expected Sarsa might use a policy different from the target policy π to generate behavior, becoming a off-policy algorithm. If π is the greedy policy while behavior is more exploratory, then Expected Sarsa is Q-learning ($\pi(a \mid s) = 1$ for argmax action, 0 for all else).

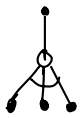
SARSA vs Q-learning: $\alpha = 0.1$, $\gamma = 1$, ties: favor move →, $\epsilon = .75$



SARSA: on policy

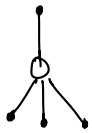
5 →	↓ 5.1	↓ 6.2	8.2 →
	5.7 →	↑ 3.3 → 7.9	↑ 7.1 ↓ 4.9 → 5.01

Q-learning: off policy



5 →	↓ 5.1	↓ 6.2	8.2 →
	6.2 →	↑ 3.3 → 8.1	↑ 7.1 ↓ 5.1 → 5.19

Expected SARSA: can be either.



5 →	↓ 5	↓ 5.95	8.2 →
	5.95 →	↑ 3 → 7.55	↑ 6.9 ↓ 4.95 → 4.90375

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[\text{Target} - Q(s,a)\right]$$

The difference among the 3 algorithms is what the target is:

SARSA: $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

Q-learning: $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$

Expected SARSA: $R_{t+1} + \gamma E_\pi\left[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}\right]$

$$= R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1})\, Q(S_{t+1}, a)$$

6.7: Maximization Bias and Double Learning

All the control algorithms so far involve maximization in the construction of their target policy:

- in q-learning we use greedy policy given the current action values.
- In sarsa, the policy is often ε-greedy.

Maximization Bias:

In these algorithms, a max is implicit used over the estimated values, which can lead to a significant positive bias. To see this, consider a single state s in which for all actions the true value $q(s,a)$ is zero but whose estimated values $Q(s,a)$ are uncertain and thus distributed some above and some below zero. The max will always be positive, when the max over the true values is zero. It's a maximization bias.

How to avoid maximization bias: double learning

One way to view the problem is that it is due to using the same samples (plays) both to determine the maximizing action and to estimate its value. Suppose we divide the plays into 2 sets and learn 2 independent estimates, $Q_1(a)$ and $Q_2(a)$, each an estimate of the true value $q(a)$.

- Action Selection: use Q_1 to determine the maximizing action $A^* = \arg\max_a Q_1(a)$

  // we select an best action of one model

- Action Eval: use Q_2 to provide an estimate of its values $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$

// but we obtain the value of previously selected "best" action using another model, to prevent selecting a biasedly high value for that "best" action.

This estimate will be unbiased in the sense that $E[Q_2(A^*)] = q(A^*)$. we can reverse both to produce another unbiased estimate $Q_1(A^*) = Q_1(\arg\max_a Q_2(a))$, and we have double learning.

Double Q-learning

The idea of double learning extends naturally to algorithms for full MDPs. For example, double Q-learning divides the time steps in two, and with probability 0.5 the update is:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q_2\left(S_{t+1}, \max_a Q_1(S_{t+1}, a)\right) - Q_1(S_t, A_t)\right] \qquad (6.10)$$

and with probability 0.5 the update is reversed, updating Q_2 instead.

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q_1\left(S_{t+1}, \max_a Q_2(S_{t+1}, a)\right) - Q_2(S_t, A_t)\right]$$

- The behavior policy can use both estimates (an ε-greedy policy could be base on the average or sum of Q_1 and Q_2).
- There are also double versions of Sarsa and Expected Sarsa.

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 $Q(\textit{terminal}, \cdot) = 0$

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

Why this works:
Q_1 and Q_2 have independent noise.
If Q_1 thinks action A looks great (due to positive noise), Q_2's estimate of action A is independent.

Final policy:
Average the Q functions: $\pi(s) = \operatorname{argmax}_a \left[Q_1(s,a) + Q_2(s,a) \right]$,
or use one of them as they should converge.

6.8: Games, afterstates, and other special cases

In chapter 1, we presented a TD algorithm for learning Tic-Tac-Toe. It learned something more like a state-value function, the player has the option to play actions, but in tic-tac-toe we evaluate after the player has taken the action. It's not an action-value either because we know the state of the game after the action is played. Let's call these afterstates and the value functions over these afterstate value-functions.

We can use these to valuate values more efficiently (in tic-tac-toe, two separate state-actions produce the same after state so we can just learn once on the afterstate and not on the two action-states).

6.9: Summary