Reinforcement Learning

Lesson 6: Temporal-Difference Learning

Edoardo Fazzari, 2022

Overview

- TD Prediction
- Advantages of TD Prediction Methods
- Optimality of TD(0)
- Sarsa: On-policy TD Control
- Q-learning: Off-policy TD Control
- Expected Sarsa
- Maximization Bias and Double Learning

Temporal-Difference Learning

An introduction

- TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas
    - Like MC methods, TD methods can learn directly from raw experience without a model of the environment’s dynamics
    - Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap)

TD Prediction

Constant-α Monte Carlo

- Given some experience following a policy π, both TD and Monte Carlo update their estimate V of v_π for the nonterminal states S_t occurring in that experience
- Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S_t)$
- A simple every-visit MC method suitable for nonstationary environment is

Constant-α Monte Carlo

$$V(S_t) \leftarrow V(S_t) + \underbrace{\alpha}_{\text{Constant step-size parameter}}[\underbrace{G_t}_{\text{Actual return following time } t} - V(S_t)] \qquad (6.1)$$

- G_t known only at the end of the episode

TD(0)

- TD methods need to wait only until the next time step
- At time *t+1* they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$
- The simplest TF method makes the update:

TD(0) $$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \qquad (6.2)$$

- It is also called *one-step TD* as a special case of the $TD(\lambda)$ and *n*-step TD

Tabular TD(0) for estimating v_π

Pseudocode

```
Input: the policy $\pi$ to be evaluated
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 $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Some considerations

- Because TD(0) bases its update in part on an existing estimate we say that it is a *bootstrapping* method

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] && (6.3)\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s]\\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (6.4)
\end{aligned}
$$

- MC methods use an estimate of (6.3) as a target, whereas DP methods use an estimate of (6.4) as a target
- TD target is an estimate since it samples the expected values in (6.4) and it uses the current estimate *V* instead of the true V_π

Sample updates

- We refer to TD and Monte Carlo updates as *sample updates* because they involve looking ahead to a sample successor state (or state-action pair), using the blue of the successor and the reward along the way to compute a backed-up value, and then updating the value of the original state (or state-action pair) accordingly
- *Sample* updates differ from the *expected* updates of DP methods in that they are based on a single sample successor rather than on complete distribution of all possible successors

TD error

- The quantity in brackets in the TD(0) update is a sort of error
    - measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$
- This quantity is called *TD error:*

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \qquad (6.5)$$

- Notice that the TD error at each time is the error in the estimate *made at that time*

Monte Carlo error

- The Monte Carlo error can be written as a sum of TD errors:

$$
\begin{aligned}
G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \qquad \text{(from (3.9))} \\
&= \delta_t + \gamma\big(G_{t+1} - V(S_{t+1})\big) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\big(G_{t+2} - V(S_{t+2})\big) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(G_T - V(S_T)\big) \\
&= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(0 - 0\big) \\
&= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. \qquad \text{(6.6)}
\end{aligned}
$$

- This identity is not exact if *V* is updated during the episode, but if the step size is small then it may still hold approximately
- Generalizations of this identity play an important role in the theory and algorithms of TD learning

Advantages of TD Prediction Methods

Advantages

Part 1

- TD methods have an advantage over DP methods in that they do not require a model of the environment
- TD methods are implemented in an online, fully incremental fashion (advantage over MC methods)
- Some MC methods must ignore or discount episodes on which experimental actions are taken, which can greatly slow learning. TD methods are much less susceptible to these problems because they learn from each transition regardless of what subsequent actions are taken

Advantages

Part 2

- Can we still guarantee convergence to the correct answer?
    - For any fixed policy π, TD(0) has proved to converge to v_π in the mean for a constant step-size parameter if it is sufficiently small
    - And with probability 1 if the step-size parameter decreases according to the usual stochastic approximation condition (2.7)

TD and MC methods convergence

- Since they converge asymptotically to the correct predictions, which gets there first?
    - No one has been able to prove mathematically that one method converges faster than the other

- In practice TD methods have usually been found to converge faster than constant-α MC methods on stochastic tasks

Optimality of TD(0)

Batch updating

- Suppose there is available only a finite amount of experience (e.g., 10 episodes or 100 time steps)
- A common approach with incremental learning methods is to present the experience repeatedly until the method converges upon an answer
- Given an approximate value function *V* the increments specified by (6.1) and (6.2) are computed for every time step *t* at which a nonterminal state is visited, but the value function is changed only once, by the sum of all the increments
- Then all the available experience is processed again with the new value function to produce a new overall increment, and so on, until the value function converges
- We call this *batch updating* because updates are made only after processing each *batch* of training data

Convergence

- Under batch updating
    - TD(0) converges deterministically to a single answer independent of the step-size parameter, α, as long as α is chosen to be sufficiently small
    - The constant-α MC method also converges deterministically under the same conditions, but to a different answer
- Under normal updating the methods do not move all the way to their respective batch answers, but in some sense they take steps in these directions

Certainty-equivalence estimate

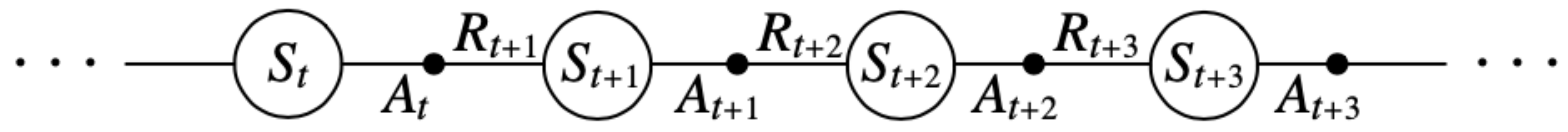
- Batch Monte Carlo methods always find the estimates that minimize mean square error on the training ste
- Batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process
    - The *maximum-likelihood estimate* of a parameter is the parameter value whose probability of generating the data is greatest
    - Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct (this is called *certainty-equivalence estimate)*
        - It is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated

Sarsa: On-policy TD Control

Sarsa

Part 1

- For an on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states *s* and actions *a*
- This can be done using essentially the same TD method described for learning v_π
- Recall that an episode consists of alternating sequence of states and state-action pairs:



Sarsa

Part 2

- We consider transition from state-action pair to state-action pair, and learn the values of state-action pairs
- The theorems assuring the convergence of state values under TD(0) also apply to the corresponding algorithm for action values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (6.7)$$

- This update is done after every transition form a nonterminal state S_t
- If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero

Sarsa

Part 3

- It is straightforward to design an on-policy control algorithm based on the Sarsa prediction method
    - As in all on-policy methods, we continually estimate q_π for the behavior policy π, and at the same time change π toward greediness with respect to q_π

- The convergence properties of the Sarsa algorithm depend on the nature of the policy's dependence on Q
    - Sarsa converges with probability 1 to an optimal policy and action-value function, under the usual condition on the step sizes (2.7), as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy ($\epsilon = 1/t$)

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

Q-learning: Off-policy TD Control

Q-learning

- Defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \qquad (6.8)$$

- The learned action-value function, Q, directly approximates q_* independent of the policy being followed
- This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs
    - Q has been shown to converge with probability 1 to q_*

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., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Expected Sarsa

Expected Sarsa

Part 1

- Consider the learning algorithm that is just like Q-learning except that instead of the maximum over next state-action pairs it uses the expected value, taking into account how likely each action is under the current policy:

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &= Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \end{aligned} \qquad (6.9)$$

- Given the next state, S_{t+1}, this algorithm moves *deterministically* in the same direction as Sarsa moves *in expectation* → *Expected Sarsa*

Expected Sarsa

Part 2

- Expected Sarsa is more complex computationally than Sarsa
- But, in return, it eliminates the variance due to the random selection of A_{t+1}
- Given the same amount of experience we might expect it to perform slightly better than Sarsa, and indeed it generally does

- In many cases Expected Sarsa can safely set $\alpha = 1$ without suffering any degradation of asymptotic performance, whereas Sarsa can only perform well in the long run at a small value of α, at which short-term performance is poor

Maximization Bias and Double Learning

Maximization bias

- All the control algorithms that we have discussed so far involve maximization in the construction of their target policies
- In these algorithms, a maximum over estimated values is used implicitly as an estimate of the maximum value, which can lead to a significant *positive bias*
    - Consider a single state *s* where there are many actions *a* whose true values, $q(s, q)$, are all zero but whose estimated values, $Q(s, a)$, are uncertain and thus distributed some above and some below zero
        - The maximum of the true values is zero, but the maximum of the estimates is positive *(positive bias)*
- We call this *maximization bias*

Are there algorithms that avoid maximization bias?

Part 1

- Consider a bandit case in which we have noisy estimates of the value of each of many actions, obtained as sample averages of the rewards received on all the plays with each action
- 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

Are there algorithms that avoid maximization bias?

Part 2

- Suppose we divided the plays in two sets and used them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(s)$
    - Each an estimate of the true value $q(a)$, for all $a \in \mathscr{A}$
- We could then use one estimate, say Q_1, to determine the maximization action $A* = \text{argmax}_a Q_1(a)$
- And Q_2 to provide the estimate of its value, $Q(A*) = Q_2(\text{argmax}_a Q_1(A))$
- This estimate will then be unbiased in the sense that

$$\mathbb{E}[Q_2(A*)] = q(A*)$$

Double learning

- We can also repeat the process with the role of the two estimates reversed to yield a second unwise estimate $Q_1(\text{argmax}_a Q_2(a))$
- This is the idea of *double learning*

- Note that although we learn two estimates, only one estimate is update on each play
- Double learning doubles the memory requirements, but does not increase the amount of computation per step

Double Q-learning

- The idea of double learning extends naturally to algorithms for full MDPs
- The double learning algorithm analogous to Q-learning, called *Double Q-learning*, divides the time steps in two (perhaps by flipping a coin on each step)
    - If the coin comes up heads, the update is

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_2(S_{t+1}, \text{argmax}_a Q_1(S_{t+1}, a)) - Q(S_t, A_t)] \tag{6.9}$$

- If the coin comes up tails, then the same update is done with Q_1 and Q_2 switched, so that Q_2 is updated

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 $\varepsilon$-greedy in $Q_1 + Q_2$
        Take action $A$, observe $R$, $S'$
        With 0.5 probabilility:
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
        else:
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
        $S \leftarrow S'$
    until $S$ is terminal
```

Bibliography:

Reinforcement Learning An Introduction (Second Edition), R. S. Sutton & A. G. Barto