Reinforcement Learning

Value Function Updates
Value Function Updates

- Different methods for updating the value function
  - Dynamic programming
  - Simple Monte Carlo
  - Temporal differencing (TD, Q-learning, SARSA)

- Character of value function updates varies across methods
  - Different branching over updates
  - Different depth of update
Value Function Backups

- Temporal-difference learning
- Dynamic programming
- Monte Carlo
- Exhaustive search

© Manfred Huber 2014
Value Function Backups

- Wider (branching) backups allow to back up multiple outcomes at the same time
  - Requires to generate multiple (probabilistic) outcomes according to the underlying distribution
    - Can only be performed in simulation (off-line)

- Deeper backups allow to include information from further in the future
  - More precise estimate of the value of the outcome
  - Requires to track rewards over multiple time steps
N-Step TD Backups

- Using multiple steps for backup

**N-step TD Prediction**

Idea:
Look farther into the future when you do TD backup (1, 2, 3, …, n steps)

7.1. **N-step TD Prediction**

\[ G(1) = R_{t+1} + \gamma V_{S_{t+1}} \]

This makes sense because \( \gamma V_{S_{t+1}} \) takes the place of the remaining terms \( \gamma R_{t+2} u \gamma R_{t+3} u \cdots u \gamma T-t-1 R_{T} \) as we discussed in the previous chapter. Our point now is that this idea makes just as much sense after two steps as it does after one. The two-step target is

\[ G(2) = R_{t+1} + \gamma R_{t+2} u \gamma V_{S_{t+2}} \]

where now \( \gamma V_{S_{t+2}} \) takes the place of the terms \( \gamma R_{t+3} u \gamma R_{t+4} u \cdots u \gamma T-t \) as we discussed in the previous chapter. In general, the n-step target is

\[ G(n) = R_{t+1} + \gamma R_{t+2} u \gamma \cdots u \gamma n-1 R_{t+n} u \gamma n V_{S_{t+n}} \]

This quantity is sometimes called the “corrected n-step truncated return” because it is a return truncated after n steps and then approximately corrected for the truncation by adding the estimated value of the nth next state. That terminology is descriptive but a bit long. We instead refer to \( G(n) \) simply as the n-step return at time t.
N-Step TD Backups

- Different outcome utility estimates
  - 1-step outcome utility prediction:
    \[ \hat{U}_t^{(1)}(s_t) = r_t + \gamma V(s_{t+1}) \]
  - Simple Monte Carlo outcome utility:
    \[ \hat{U}_t^{(\infty)}(s_t) = \sum_{\tau=t}^{T} \gamma^{\tau-t} r_\tau \]
  - n-step outcome utility prediction:
    \[ \hat{U}_t^{(n)}(s_t) = \sum_{\tau=t}^{t+n-1} \left( \gamma^{\tau-t} r_\tau \right) + \gamma^n V_t(s_{t+n}) \]
- Corresponding value backup
  \[ \Delta V_t^{(k)}(s_t) = \alpha \left( \hat{U}_t^{(k)}(s_t) - V_t(s_t) \right) \]
N-Step TD Backups

- TD and MC converge
  - With limited data to different results
- N-step on-policy updates converge to the correct utility for on-policy exploration
  - Precision of expected value of n-step update depends on precision of estimate after the n\textsuperscript{th} step

\[
\max_s \left| E\left[ \hat{U}^{(k)}(s) \pi \right] - V^\pi(s) \right| \leq \gamma^n \max_s \left| V_t(s) - V^\pi(s) \right|
\]

- Expected error for n-step backup is at least as small as for 1-step (TD) backup and thus converges
N-Step TD Backups

- RMS for 10 episodes of 19 state problem
Multi-Horizon Backups

- Optimal \( n \) depends on data, learning rate, and the problem
  - Larger \( n \) usually require smaller learning rates
  - Off-line generated data usually benefits more from larger \( n \)
  - No a priori optimal value

- Combining multiple horizon lengths
  - Average the outcome estimate for multiple \( n \)
    \[
    \hat{U}^w(s_t) = \sum_{k \in K} w_k \hat{U}^{(k)}(s_t)
    \]
Exponentially Weighted Complex Backups

- Assign exponentially decreasing weights
  \[ w_k = (1 - \lambda) \lambda^{k-1} \]
  \[ w_T = \lambda^{T-1} \]
  - Longer horizon estimates have lower weights
  - All weights add up to 1
    - Terminal length gets complete remaining weight

- TD and MC are special cases
  - \( \lambda = 0 \) : TD
  - \( \lambda = 1 \) : Simple Monte Carlo
Exponential Averaging

- Exponentially weighted complex backups

\[ \Delta V_t(s_t) = \alpha \left( (1 - \lambda) \sum_{\tau=1}^{T-t-1} \lambda^{\tau-1} \hat{U}_t^{(\tau)}(s_t) + \lambda^{T-t-1} \hat{U}_t^{(T-t)}(s_t) \right) - V_t(s_t) \]

- Update is based on all horizons

- \( \lambda \) regulates the amount of update due to future rewards

- Also changes how much emphasis is put on representing value of training data sequence versus on following local Markov assumption
Exponential Averaging

The forward or theoretical view: We decide how to update each state by looking forward to future rewards and states.

Figure 7.5: Performance of the offline $\lambda$-return algorithm on a 19-state random walk task. A way of mixing $n$-step backups is that there is a simple algorithm—TD($\lambda$)—for achieving it. This is a mechanism issue rather than a theoretical one. In the next few sections we develop the mechanistic or backward view of eligibility traces as used in TD($\lambda$).

Example 7.2: $\lambda$-return on the Random Walk Task

Figure 7.6 shows the performance of the offline $\lambda$-return algorithm on the 19-state random walk task used with the $n$-step methods in Example 7.1. The experiment was just as in the $n$-step case except that here we varied $\lambda$ instead of $n$. Note that we get best performance with an intermediate value of $\lambda$.

Exercise 7.4 The parameter $\lambda$ characterizes how fast the exponential weighting in Figure 7.4 falls off and thus how far into the future the $\lambda$-return algorithm looks in determining its backup. But a rate factor such as $\lambda$ is sometimes an awkward way of characterizing the speed of the decay. For some purposes it...
Incremental Complex Backups

- Forward-looking n-step backups can only be performed after the future rewards are known
  - No fully incremental learning
  - Not feasible for non-episodic tasks

- Apply complex backups in parts
  - At each step apply a backup to all previous states
    - Make backups add up to the complete complex backup over time
Incremental Complex Backups

- Complex backup can be broken into components due to particular states / steps

\[
\Delta V_t (s_t) = \alpha \left( \left( (1 - \lambda) \sum_{\tau=1}^{T-t-1} \lambda^{\tau-1} \hat{U}_t^{(\tau)}(s_t) + \lambda^{T-t-1} \hat{U}_t^{(T-t)}(s_t) \right) - V_t(s_t) \right)
\]

\[
= \alpha \left( (1 - \lambda) \left( r_t + \gamma V_t(s_{t+1}) \right) 
+ (1 - \lambda) \lambda \left( r_t + \gamma r_{t+1} + \gamma^2 V_t(s_{t+2}) \right) 
+ (1 - \lambda) \lambda^2 \left( r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V_t(s_{t+3}) \right) 
+ \ldots 
+ \lambda^{T-t-1} \left( r_t + \gamma r_{t+1} + \ldots + \gamma^{T-t} V_t(s_{T-t}) \right) 
- V_t(s_t) \right)
\]
Incremental Complex Backups

- Complex backup broken by time steps

\[
\Delta V_t (s_t) = \alpha \left( r_t + \gamma V_t(s_{t+1}) - \lambda \gamma V_t(s_{t+1}) 
+ \lambda \gamma (r_{t+1} + \gamma V_t(s_{t+2}) - \lambda \gamma V_t(s_{t+2})) 
+ \ldots 
+ \lambda^{T-t-1} \gamma^{T-t-1} \left( r_{T-t} + \gamma V_t(s_{T-t}) \right) 
- V_t(s_t) \right) 
= \alpha \left( r_t + \gamma V_t(s_{t+1}) - V_t(s_t) 
+ \lambda \gamma (r_{t+1} + \gamma V_t(s_{t+2}) - V_t(s_{t+1})) 
+ \ldots 
+ \lambda^{T-t-1} \gamma^{T-t-1} \left( r_{T-t} + \gamma V_t(s_{T-t}) - V_t(s_{T-t-1}) \right) \right)
\]
Eligibility Traces

- Apply partial backups retroactively as transitions occur

\[ \Delta V_t(s_t) = \sum_{\tau=0}^{T-t-1} \Delta V_{t,\tau}^{TD}(s_t) \]

\[ \Delta V_{t,i}^{TD}(s_t) = \alpha (\lambda \gamma)^i (r_{t+i} + \gamma V_t(s_{t+i+1}) - V_t(s_{t+i})) \]

\[ = \alpha E_{t+i}(s_t) \delta_{t,i} \]

- Eligibility trace \( E \) keeps track how much a TD error should affect an earlier state
Eligibility Traces

To precisely reflect complex updates the retroactive updates would have to depend on the non-updated value function

More practical (approximately equal for small $\alpha$)

$$\Delta V_t(s_t) = \sum_{\tau=t}^{T} \Delta V_{\tau}^{TD}(s_t)$$

$$\Delta V_{t+i+1}^{TD}(s_t) = \alpha(\lambda \gamma)^i (r_{t+i} + \gamma V_{t+i}(s_{t+i+1}) - V_{t+i}(s_{t+i}))$$

$$= \alpha E_{t+i}(s_t) \delta_{t+i}$$
Eligibility Traces

- Alternative view of eligibility traces and complex backups
  - Updates to previous states reflect anticipated changes that would happen if the current state would have already been updated
    - Individual propagation assumes that the states would happen in the same sequence the next time, too
    - $\lambda$ reflects the incremental, exponentially weighted averaging constant for subsequent states
      - Addresses the fact that successor states do not follow deterministically
Temporal difference learning can be generalized with complex updates

- Single step error:
  \[ \delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t) \]

- Eligibility trace:
  \[ E_t(s) = \begin{cases} 
    \lambda \gamma E_{t-1}(s) + 1 & s = s_t \\
    \lambda \gamma E_{t-1}(s) & \text{otherwise} 
  \end{cases} \]

- Traditional TD is TD(0)
Exercise 7.4

Example 7.2:

The parameter $\lambda$ characterizes how fast the exponential weight $w$ and thus how far into the future the traces as used in TDs achieve it. This is a mechanism issue rather than a theoretical one. In the way of mixing the performance of the on-line and off-line on Random Walk. On-line performs better over a broader range of parameters.

Same 19 state random walk $R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction

On-line versus Off-line on Random Walk

Batch update

On-line update

$TD(\lambda)$
Eligibility Traces

- To use eligibility traces transition samples have to be generated consecutively.
- Equations assume update sequences that are:
  - On-policy
  - On-line (i.e. which form a consecutive state action sequence)
- Extending this to state/action value function learning requires additional considerations:
  - What if policy changes?
**SARSA(λ)**

- SARSA requires on-policy data learning and its update depends only on the action taken
  \[
  Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha (r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t))
  \]
- **SARSA(λ)**
  \[
  Q_{t+i+1}(s_t, a_t) = Q_{t+i}(s_t, a_t) + \alpha E_{t+i}(s_t, a_t) \delta_{t+i}
  \]
  - Single step error:
    \[
    \delta_t = r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)
    \]
  - Eligibility trace:
    \[
    E_t(s, a) = \begin{cases} 
    \lambda \gamma E_{t-1}(s, a) + 1 & s = s_t, a = a_t \\
    \lambda \gamma E_{t-1}(s, a) & \text{otherwise}
    \end{cases}
    \]
Q(λ)

- Q-learning includes a policy improvement operation that changes the policy
  \[ Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha (r_t + \gamma \max_b Q_t(s_{t+1}, b) - Q_t(s_t, a_t)) \]
  - Single step Q-learning error:
    \[ \delta_t = r_t + \gamma \max_b Q_t(s_{t+1}, b) - Q_t(s_t, a_t) \]

- Handling non-policy actions
  - Resetting eligibility traces
    \[ E_t(s, a) = \begin{cases} 
    \lambda \gamma E_{t-1}(s, a) + 1 & s = s_t, a = a_t, a_t \in \arg\max_a Q_t(s_t, a) \\
    1 & s = s_t, a = a_t, a_t \notin \arg\max_a Q_t(s_t, a) \\
    0 & s \neq s_t \lor a \neq a_t, a_t \notin \arg\max_a Q_t(s_t, a) \\
    \lambda \gamma E_{t-1}(s, a) & \text{otherwise}
  \end{cases} \]
Eligibility Traces

- Eligibility traces can produce instabilities when their values get too high
  - Replacing traces
    \[
    E_t(s) = \begin{cases} 
    1 & s = s_t \\
    \lambda \gamma E_{t-1}(s) & \text{otherwise}
    \end{cases}
    \]
  - Replacing and resetting can be combined
    \[
    E_t(s, a) = \begin{cases} 
    1 & s = s_t, a = a_t \\
    0 & s \neq s_t \lor a \neq a_t, a_t \notin \text{argmax}_a Q_t(s_t, a) \\
    \lambda \gamma E_{t-1}(s, a) & \text{otherwise}
    \end{cases}
    \]
Replacing Traces

RMS error at best $\alpha$

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.png}
\caption{Error as a function of $\lambda$ on a 19 state random walk task. These data are using the best value of $\alpha$ for each value of $\lambda$. The error is averaged over all 19 states and the first 100 trials of 10 different runs. Prediction or control algorithms using replacing traces are often called replace-trace methods. Although replacing traces are only slightly different from accumulating traces, they can produce a significant improvement in learning rate. Figure 7 shows the performance of conventional and replacing-trace versions of TDo$^\lambda$ont h e r a n d o m w a l k p r e d i c t i o n t a s k. O t h e r examples for a slightly more general case are given in Figure 9. Figure 7.5 shows an example of the kind of task that is difficult for control methods using accumulating eligibility traces. All rewards are zero except on entering the terminal states, which produces a reward of $r$. From each state, selecting the right action brings the agent one step closer to the terminal rewards whereas the wrong upper action leaves it in the same state to try again. The full sequence of states is long enough that one would like to use long traces to get the fastest learning. However, problems occur if long accumulating traces are used. Suppose on the first episodes at some states the agent by chance takes the wrong action a few times before taking the right action. As the agent continues the trace $Z_\lambda s$, the wrong action is selected more times than the right action was recently, but the trace $Z_\lambda s$, right $p$. The right action was more recent but the wrong action was selected more times. When reward is finally received, the value for the wrong action is likely to go up more than the value for the right action. On the next episode, the agent will be even more likely to go the wrong way many times before going right, making it even more likely that the wrong action will have the larger trace. Eventually, all of this will be corrected, but learning is significantly slowed. With replacing traces, on the other hand, this problem never occurs. No matter how many times the wrong action is selected, the trace will always be smaller for the wrong action.}
\end{figure}
Value Function Updates

- Methods for updating the value function differ in the way they propagate values
  - Eligibility traces provide a means to combine different depths updates
- Different depths provide different properties
  - Low depths provide good bootstrapping using Markov property assumption
  - High depths provide more precision but do not use local Markov assumption as efficiently
    - High depths can be better if system is not truly Markov