Part II: Approximate Solution Methods

...have been instrumental in recent RL successes
Reminder: we can use supervised learning to approximate value function

For example: use a deep convolutional neural network:
9.8 Summary

- Function approximation is useful for generalizing from one state to the next, especially if there are many states.
- Connection to supervised learning: treat backup as supervised training example.
- Objective often: minimize MSVE for on-policy distribution.
- Many supervised learning techniques can be used.
- Linear function approximation theoretically best understood.
- Nonlinear function approximation with deep neural networks (“deep reinforcement learning”)
Chapter 10: On-Policy Control with Approximation

- Idea: extend to control by approximating action-value function:

\[
\hat{q}(s, a, \theta) \approx q^*(s, a) \quad \theta \in \mathbb{R}^n
\]

- on-policy control: semi-gradient SARSA algorithm
- episodic case
- average reward formulation for continuing case
10.1 Episodic Semi-gradient Control

- approximate: \( \hat{q}(s, a, \theta) \approx q_*(s, a) \)

- gradient descent:
  \[
  \theta_{t+1} = \theta_t + \alpha \left[ U_t - \hat{q}(S_t, A_t, \theta_t) \right] \nabla \hat{q}(S_t, A_t, \theta_t)
  \]

- one-step SARSA update target:
  \[
  \theta_{t+1} = \theta_t + \alpha \left[ R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \theta_t) - \hat{q}(S_t, A_t, \theta_t) \right] \nabla \hat{q}(S_t, A_t, \theta_t)
  \]

- For control with discrete set of actions: calculate \( \hat{q}(S_t, a, \theta_t) \) for every action and determine the greedy one. Then, e.g., use epsilon greedy action selection: \( A_t^* = \arg\max_a \hat{q}(S_t, a, \theta_t) \)
CHAPTER 10. ON-POLICY CONTROL WITH APPROXIMATION

Action-value prediction is

\[ \phi_{t+1} = \phi_t + \gamma \hat{R}_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \phi_t) - \hat{q}(S_t, A_t, \phi_t) \hat{q}(S_t, A_t, \phi_t) \]

(10.1)

For example, the update for the one-step Sarsa method is

\[ \phi_{t+1} = \phi_t + \gamma R_{t+1} + \hat{q}(S_{t+1}, A_{t+1}, \phi_t) - \hat{q}(S_t, A_t, \phi_t) \hat{q}(S_t, A_t, \phi_t) \]

(10.2)

We call this method episodic semi-gradient one-step Sarsa. For a constant policy, this method converges in the same way that TD(0) does, with the same kind of error bound (9.14).

To form control methods, we need to couple such action-value prediction methods with techniques for policy improvement and action selection. Suitable techniques applicable to continuous actions, or to actions from large discrete sets, are a topic of ongoing research with as yet no clear resolution. On the other hand, if the action set is discrete and not too large, then we can use the techniques already developed in previous chapters. That is, for each possible action \( a \) available in the current state \( S_t \), we can compute \( \hat{q}(S_t, a, \phi_t) \) and then find the greedy action \( A^\star_t = \arg\max_a \hat{q}(S_t, a, \phi_t) \).

Policy improvement is then done (in the on-policy case treated in this chapter) by changing the estimation policy to a soft approximation of the greedy policy such as the \( \varepsilon \)-greedy policy. Actions are selected according to this same policy. Pseudocode for the complete algorithm is given in the box.

**Example 10.1: Mountain–Car Task**

Consider the task of driving an underpowered car up a steep mountain road, as suggested by the diagram in the upper left of Figure 10.1. The difficulty is that gravity is stronger than the car's engine, and even at full throttle the car cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope on the left. Then, by...
Example: Mountain Car

- Problem: under-powered car needs to drive up a steep hill
- States: positions, velocities
- Actions: full forward, full reverse, zero
- Rewards: -1 on every step until episode ends
- Dynamics (simplified physics):

\[
x_{t+1} = \text{bound}\left[x_t + \dot{x}_{t+1}\right]
\]

\[
\dot{x}_{t+1} = \text{bound}\left[\dot{x}_t + 0.001A_t - 0.0025 \cos(3x_t)\right]
\]

-1.2 \leq x_{t+1} \leq 0.5

-0.07 \leq \dot{x}_{t+1} \leq 0.07
Mountain Car Continued

- start in random locations with zero velocity
- use tile coding for function approximation
- 8 tilings with width 1/8 of the allowed intervals, asymmetrical offsets

Figure 10.2: Learning curves for semi-gradient Sarsa with tile-coding function approximation on the Mountain Car example.

Figure 10.1 shows what typically happens while learning to solve this task with this form of function approximation. Shown is the negative of the value function (the cost-to-go function) learned on a single run. The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur even though the exploration parameter, \( \epsilon \), was 0. This can be seen in the middle-top panel of the figure, labeled "Step 428". At this time not even one episode had been completed, but the car has oscillated back and forth in the valley, following circular trajectories in state space. All the states visited frequently are valued worse than unexplored states, because the actual rewards have been worse than what was (unrealistically) expected. This continually drives the agent away from wherever it has been, to explore new states, until a solution is found.

Figure 10.2 show several learning curves for semi-gradient Sarsa on this problem, with various step sizes.

**Exercise 10.1**

Why have we not considered Monte Carlo methods in this chapter?

**10.2 n-step Semi-gradient Sarsa**

We can obtain an \( n \)-step version of episodic semi-gradient Sarsa by using an \( n \)-step return as the update target in the semi-gradient Sarsa update equation (10.1).

The \( n \)-step return immediately generalizes from its tabular form (7.4) to a function approximation form:

\[
G_n(t) = R_{t+1} + R_{t+2} + \cdots + R_{t+n-1} + n \hat{q}(S_{t+n}, A_{t+n}, \epsilon_{t+n})
\]

with

\[
G_n(t) = G_t \quad \text{if} \quad t + n \leq T,
\]

as usual. The \( n \)-step update equation is

\[
\epsilon_{t+n} = \epsilon_{t+n-1} + \gamma \hat{q}(S_t, A_t, \epsilon_{t+n}) - \hat{q}(S_t, A_t, \epsilon_{t+n})
\]

with

\[
\hat{q}(S_t, A_t, \epsilon_{t+n}) = \frac{1}{t} \sum_{i=0}^{n-1} R_{t+i} + \gamma \hat{q}(S_{t+i}, A_{t+i}, \epsilon_{t+i})
\]

This data is actually from the "semi-gradient Sarsa(\( \gamma \))" algorithm that we will not meet until Chapter 12, but semi-gradient Sarsa behaves similarly.
Cost-to go function: $- \max_a \hat{q}(s, a, \theta)$
10.3 Average Reward: A New Problem Setting for Continuing Tasks

- third setting next to episodic and discounted settings
- applies to continuing settings, but no discounting necessary
- quality of a policy is defined as average rate of reward when following this policy:

\[
\eta(\pi) = \lim_{T \to \infty} \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}[R_t \mid A_{0:t-1} \sim \pi]
\]

\[
= \lim_{t \to \infty} \mathbb{E}[R_t \mid A_{0:t-1} \sim \pi],
\]

\[
= \sum_s d_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s, r'|s, a)r
\]

\[
d_\pi(s) = \lim_{t \to \infty} \Pr\{S_t = s \mid A_{0:t-1} \sim \pi\} \quad \text{“ergodicity”}
\]
Note: on-policy distribution is also the steady-state distribution

\[ \sum_s d_\pi(s) \sum_a \pi(a|s, \theta)p(s'|s, a) = d_\pi(s') \]

Define returns in terms of differences between rewards and average reward (“differential return”):

\[ G_t = R_{t+1} - \eta(\pi) + R_{t+2} - \eta(\pi) + R_{t+3} - \eta(\pi) + \cdots \]
Differential Value Functions and Bellman Equations

- Compared to previous version(s): remove all discount factors and replace all rewards by differences between reward and true average reward:

\[
v_\pi(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a) \left[ r - \eta(\pi) + v_\pi(s') \right],
\]

\[
q_\pi(s, a) = \sum_{r,s'} p(s', r|s, a) \left[ r - \eta(\pi) + \sum_{a'} \pi(a'|s')q_\pi(s', a') \right],
\]

\[
v_*(s) = \max_a \sum_{r,s'} p(s', r|s, a) \left[ r - \eta(\pi) + q_*(s') \right], \text{ and}
\]

\[
q_*(s, a) = \sum_{r,s'} p(s', r|s, a) \left[ r - \eta(\pi) + \max_{a'} q_*(s', a') \right]
\]
Differential TD errors

- for learning of state values and state-action values:
  \[
  \delta_t = R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \theta) - \hat{v}(S_t, \theta), \quad \text{and}
  \]
  \[
  \delta_t = R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_t, \theta) - \hat{q}(S_t, A_t, \theta)
  \]
  should be \(A_{t+1}\)

- most algorithms and theoretical results carry through to this average reward setting

- Example update for semi-gradient SARSA:
  \[
  \theta_{t+1} = \theta_t + \alpha \delta_t \nabla \hat{q}(S_t, A_t, \theta_t)
  \]
Differential Semi-gradient SARSA for Control

Differential Semi-gradient Sarsa for Control

Input: a differentiable function \( \hat{q} : S \times A \times \mathbb{R}^n \rightarrow \mathbb{R} \)
Parameters: step sizes \( \alpha, \beta > 0 \)

Initialize value-function weights \( \theta \in \mathbb{R}^n \) arbitrarily (e.g., \( \theta = 0 \))
Initialize average reward estimate \( \bar{R} \) arbitrarily (e.g., \( \bar{R} = 0 \))
Initialize state \( S \), and action \( A \)

Repeat (for each step):
   Take action \( A \), observe \( R, S' \)
   Choose \( A' \) as a function of \( \hat{q}(S', \cdot, \theta) \) (e.g., \( \varepsilon \)-greedy)
   \( \delta \leftarrow R - \bar{R} + \hat{q}(S', A', \theta) - \hat{q}(S, A, \theta) \)
   \( \bar{R} \leftarrow \bar{R} + \beta \delta \)
   \( \theta \leftarrow \theta + \alpha \delta \nabla \hat{q}(S, A, \theta) \)
   \( S \leftarrow S' \)
   \( A \leftarrow A' \)
Example: An Access-Control Queuing Task

- $k$ servers, customers of four different priorities arrive at a single cue.
- Customers pay 1, 2, 4, or 8 reward depending on their priority
- At each time step: customer at the head of cue is either accepted (assigned to one of the servers) or rejected (for zero reward)
- Queue never empties; priorities are random
- Customer automatically rejected if no free server
- Busy server becomes free with probability $p$
Solution for tabular version

- state: number of free servers and priority of customer at the head of the cue
- action: accept or reject
- Solution found by semi-gradient SARSA (k=10, p=0.06, alpha=0.01, beta=0.01, epsilon=0.1, initial estimate of average reward was zero)
- What might the optimal policy look like?
AVERAGE REWARD: A NEW PROBLEM SETTING FOR CONTINUING TASKS

In either case, on the next time step the next customer in the queue is considered. The queue never empties, and the priorities of the customers in the queue are equally randomly distributed. Of course a customer can not be served if there is no free server; the customer is always rejected in this case. Each busy server becomes free with probability $p$ on each time step. Although we have just described them for definiteness, let us assume the statistics of arrivals and departures are unknown. The task is to decide on each step whether to accept or reject the next customer, on the basis of his priority and the number of free servers, so as to maximize long-term reward without discounting.

In this example we consider a tabular solution to this problem. Although there is no generalization between states, we can still consider it in the general function approximation setting as this setting generalizes the tabular setting. Thus we have differential action-value estimate for each pair of state (number of free servers and priority of the customer at the head of the queue) and action (accept or reject).

Figure 10.5 shows the solution found by differential semi-gradient one-step Sarsa for this task with $k=10$ and $p=0.06$. The algorithm parameters were $\epsilon=0.01$, $\gamma=0.99$, and $\alpha=0.1$. The initial action values and $\bar{R}$ were zero.

![Graph showing the policy and value function found by differential semi-gradient one-step Sarsa on the access-control queuing task after 2 million steps. The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for $\bar{R}$ was about 2.31.](image-url)
10.6 Summary

- adaptation of parameterized function approximation and semi-gradient descent for control
- average reward setting
- differential versions of value functions, Bellman equations, and TD errors
Chapter 11: Off-Policy Methods with Approximation

- omitted
Chapter 12: Eligibility Traces

- Eligibility traces unify TD and MC methods (recall $n$-step methods from Chapter 7, which had a similar flavor)
- Example: TD($\lambda$) algorithm (see later):
  - $\lambda=0$ : TD
  - $\lambda=1$ : MC
- Advantage of eligibility traces: elegant algorithmic mechanism
- Works via eligibility trace vector $e_t \in \mathbb{R}^n$ that mirrors the parameter vector $\theta_t \in \mathbb{R}^n$
The Idea

- **Idea (roughly):** when a component of \( \theta_t \in \mathbb{R}^n \) contributes to a state value estimate, then the corresponding component of \( e_t \in \mathbb{R}^n \) is bumped up and then slowly decays with a decay factor \( \lambda \in [0, 1] \)

- Advantage over \( n \)-step methods:
  - no need to store last \( n \) feature vectors
  - learning occurs continually and uniformly in time
  - no need to wait until end of an episode

- forward vs. backward view
also previous states/actions are given credit for current TD error: recent states/actions are given much credit and distant actions are given little credit, controlled by a decay rate $\lambda$. 

---

**Intuition**

- **Path taken**
- Action values increased by one-step Sarsa
- Action values increased by Sarsa($\lambda$) with $\lambda = 0.9$
12.1 The $\lambda$-return

- Recall from Chapter 7 the $n$-step return:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \theta_{t+n-1}), \quad 0 \leq t \leq T-n$$

- We can use any value for $n$, but we could also average multiple such returns (compound backup) as on the right with different weights.

- TD($\lambda$) is a particular way of averaging $n$-step returns.
Back-up Diagram for $\lambda$-return

\[ G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \]
Weightings in $\lambda$-return

**What happens for $\lambda=0$?**
- Weight given to the 3-step return is $(1 - \lambda)\lambda^2$
- Weight given to actual, final return is $\lambda^{T-t-1}$

**What happens for $\lambda=1$?**
- Total area = 1
- Weight decays by $\lambda$

Equation 12.3:
$$G_t = \left(1 - \lambda\right)\lambda^2$$

Exercise 12.1: The parameter $\lambda$ characterizes how fast the exponential weighting in Figure 12.2 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 is better to specify a time constant, or half-life. What is the equation relating $\lambda$ and the half-life, $\tau$, the time by which the weighting sequence will have fallen to half of its initial value?
TD(\(\lambda\))

- one of the oldest and most widely used algorithms in RL
- eligibility traces initialized to zero and incremented based on the gradient of the value estimate w.r.t. theta and decay:
  \[
  e_0 = 0, \\
  e_t = \nabla \hat{v}(S_t, \theta_t) + \gamma \lambda e_{t-1}
  \]
- TD error for state value prediction:
  \[
  \delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \theta_t) - \hat{v}(S_t, \theta_t)
  \]
- parameter vector updated based on TD error and eligibility trace:
  \[
  \theta_{t+1} = \theta_t + \alpha \delta_t e_t
  \]
Semi-gradient TD(\(\lambda\)) for estimating \(\hat{v} \approx v_{\pi}\)

Input: the policy \(\pi\) to be evaluated
Input: a differentiable function \(\hat{v} : S^+ \times \mathbb{R}^n \to \mathbb{R}\) such that \(\hat{v}(\text{terminal}, \cdot) = 0\)

Initialize value-function weights \(\theta\) arbitrarily (e.g., \(\theta = 0\))
Repeat (for each episode):
  Initialize \(S\)
  \[\mathbf{e} \leftarrow 0\] (An \(n\)-dimensional vector)
  Repeat (for each step of episode):
    . Choose \(A \sim \pi(\cdot|S)\)
    . Take action \(A\), observe \(R, S'\)
    . \(\mathbf{e} \leftarrow \gamma \lambda \mathbf{e} + \nabla \hat{v}(S, \theta)\)
    . \(\delta \leftarrow R + \gamma \hat{v}(S', \theta) - \hat{v}(S, \theta)\)
    . \(\theta \leftarrow \theta + \alpha \delta \mathbf{e}\)
    . \(S \leftarrow S'\)
  until \(S'\) is terminal
Example: 10-step random walk task

If \( \lambda = 1 \), then the credit given to earlier states falls only by \( \lambda \) per step. This turns out to be just the right thing to do to achieve Monte Carlo behavior. For example, remember that the TD error, \( T^\lambda(t) \), includes an undiscounted term of \( R_{t+1} \) that needs to be discounted, like any reward in a return, by \( \gamma^k \), which is just what the falling eligibility trace achieves. If \( \lambda = 1 \) and \( \gamma = 1 \), then the eligibility traces do not decay at all with time. In this case the method behaves like a Monte Carlo method for an undiscounted, episodic task. If \( \lambda = 1 \), the algorithm is also known as TD(1). TD(1) is a way of implementing Monte Carlo algorithms that is more general than those presented earlier and that significantly increases their range of applicability.

Whereas the earlier Monte Carlo methods were limited to episodic tasks, TD(1) can be applied to discounted continuing tasks as well. Moreover, TD(1) can be performed incrementally and on-line. One disadvantage of Monte Carlo methods is that they learn nothing from an episode until it is over. For example, if a Monte Carlo control method takes an action that produces a very poor reward but does not end the episode, then the agent’s tendency to repeat the action will be undiminished during the episode. On-line TD(1), on the other hand, learns in an \( \alpha \)-step TD way from the incomplete ongoing episode, where the \( \alpha \) steps are all the way up to the current step. If something unusually good or bad happens during an episode, control methods based on TD(1) can learn immediately and alter their behavior on that same episode.

It is revealing to revisit the 19-state random walk example (Example 7.1) to see how well TD(1) does in approximating the \( \lambda \)-line algorithm. The results for both algorithms are shown in Figure 12.6. For each \( \lambda \) value, if \( \lambda \) is selected

\[
\begin{align*}
\lambda &= 0 \\
\lambda &= 0.4 \\
\lambda &= 0.8 \\
\lambda &= 0.9 \\
\lambda &= 0.95 \\
\lambda &= 0.975 \\
\lambda &= 0.99 \\
\lambda &= 1
\end{align*}
\]

The two algorithms performed virtually identically at low (less than optimal) \( \lambda \) values, but TD(1) was worse at high \( \lambda \) values.
Forward vs. Backward View

The approach that we have been taking so far is what we call the theoretical, or forward view of a learning algorithm. For each state visited, we look forward in time to determine the correct and estimated values of each state measured at the end of the episode, $v(S_t)$ and $\hat{v}(S_t)$, respectively. We can do this for the entire episode, and the performance measure used is the estimated root-mean-squared error between the correct and estimated values of each state measured at the end of the episode, $\varepsilon = \sqrt{\frac{1}{n} \sum_{t=1}^{n} (v(S_t) - \hat{v}(S_t))^2}$. 

Repeat (for each episode):

1. Initialize value-function weights $\theta$.
2. Input: a differentiable function $\hat{v}(S)$.
3. Input: the policy $\pi$.
4. Repeat (for each step of episode):
   - Take action $A_t$.
   - Choose state $S_t$.
   - Repeat (for each state $S_t$):
     - Follow $\pi$ to $S_{t+1}$.
     - Receive reward $R_{t+1}$.
     - Update $\theta$.

For each state $S_t$ (and hence $\hat{v}(S_t)$), this update procedure is identical to the one we saw in Chapter 7. Notice, however, that in Chapter 7 we worked backward in time with $\gamma = \lambda = 0$; that is, we used (12.7) with one-step algorithms. Now we work forward in time with $\gamma = \lambda = 0$; that is, we use (12.7) with one-step algorithms. 

In the theoretical, or forward view, the update depends on the current TD error $\delta_t$. 

Thus the TD$(0)$ is:

$\text{TD}(0) = \Delta(S_t) = R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$

where $\Delta(S_t)$ is the TD error.

Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

Figure 12.5: The backward or mechanistic view. Each update depends on the current TD error and the fact that the previously visited states, as suggested by Figure 12.5, TD$(0)$ is:

$\text{TD}(0) = \Delta(S_t) = R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$

Where the TD error and the previously visited states come together, we get the update given by (12.7). 

In the theoretical, or forward view, the update depends on the current TD error $\delta_t$.

Thus the TD$(0)$ is:

$\text{TD}(0) = \Delta(S_t) = R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$

where $\Delta(S_t)$ is the TD error.

We can also think of the bootstrap error as the difference between the reward we received and the value of the state we are currently in. 

For each state $S_t$, we can look backward in time to determine the best action to take in that state, and hence also determine the best value for that state, $v(S_t)$. 

We can then use these best values to reward the agent for good decisions and penalize it for bad decisions. 

The performance measure used is the estimated root-mean-squared error between the correct and estimated values of each state measured at the end of the episode, $\varepsilon = \sqrt{\frac{1}{n} \sum_{t=1}^{n} (v(S_t) - \hat{v}(S_t))^2}$. 

Repeat (for each episode):

1. Initialize value-function weights $\theta$.
2. Input: a differentiable function $\hat{v}(S)$.
3. Input: the policy $\pi$.
4. Repeat (for each step of episode):
   - Take action $A_t$.
   - Choose state $S_t$.
   - Repeat (for each state $S_t$):
     - Follow $\pi$ to $S_{t+1}$.
     - Receive reward $R_{t+1}$.
     - Update $\theta$.

For each state $S_t$ (and hence $\hat{v}(S_t)$), this update procedure is identical to the one we saw in Chapter 7. Notice, however, that in Chapter 7 we worked backward in time with $\gamma = \lambda = 0$; that is, we used (12.7) with one-step algorithms. Now we work forward in time with $\gamma = \lambda = 0$; that is, we use (12.7) with one-step algorithms. 

In the theoretical, or forward view, the update depends on the current TD error $\delta_t$. 

Thus the TD$(0)$ is:

$\text{TD}(0) = \Delta(S_t) = R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$

where $\Delta(S_t)$ is the TD error.

We can also think of the bootstrap error as the difference between the reward we received and the value of the state we are currently in. 

For each state $S_t$, we can look backward in time to determine the best action to take in that state, and hence also determine the best value for that state, $v(S_t)$. 

We can then use these best values to reward the agent for good decisions and penalize it for bad decisions. 

The performance measure used is the estimated root-mean-squared error between the correct and estimated values of each state measured at the end of the episode, $\varepsilon = \sqrt{\frac{1}{n} \sum_{t=1}^{n} (v(S_t) - \hat{v}(S_t))^2}$. 

Repeat (for each episode):

1. Initialize value-function weights $\theta$.
2. Input: a differentiable function $\hat{v}(S)$.
3. Input: the policy $\pi$.
4. Repeat (for each step of episode):
   - Take action $A_t$.
   - Choose state $S_t$.
   - Repeat (for each state $S_t$):
     - Follow $\pi$ to $S_{t+1}$.
     - Receive reward $R_{t+1}$.
     - Update $\theta$.

For each state $S_t$ (and hence $\hat{v}(S_t)$), this update procedure is identical to the one we saw in Chapter 7. Notice, however, that in Chapter 7 we worked backward in time with $\gamma = \lambda = 0$; that is, we used (12.7) with one-step algorithms. Now we work forward in time with $\gamma = \lambda = 0$; that is, we use (12.7) with one-step algorithms. 

In the theoretical, or forward view, the update depends on the current TD error $\delta_t$. 

Thus the TD$(0)$ is:

$\text{TD}(0) = \Delta(S_t) = R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$

where $\Delta(S_t)$ is the TD error.
Summary

- $\lambda$-return: weighted average of many $n$-step returns
- $\lambda=0$: one-step TD
- $\lambda=1$: Monte Carlo
- $\text{TD}(\lambda)$
- Best performance usually for intermediate values
- forward and backward views