

2D5362 Machine Learning

Reinforcement Learning

MIT GALib

⌘ Available at <http://lancet.mit.edu/ga/>

⌘ download galib245.tar.gz

```
gunzip galib245.tar.gz
```

```
tar -xvf galib245.tar
```

```
cd galib245
```

```
make
```

⌘ or access my files at

[/afs/nada.kth.se/home/cvap/hoffmann/Research/galib245](http://afs.nada.kth.se/home/cvap/hoffmann/Research/galib245)



Reinforcement Learning

- ⌘ Reinforcement Learning Problem
- ⌘ Dynamic Programming
- ⌘ Monte-Carlo Methods
- ⌘ Temporal Difference Learning



Control Learning

Learning to choose actions

- ⌘ Robot learning to dock to a battery station
- ⌘ Learning to choose actions to optimize a factory output
- ⌘ Learning to play Backgammon

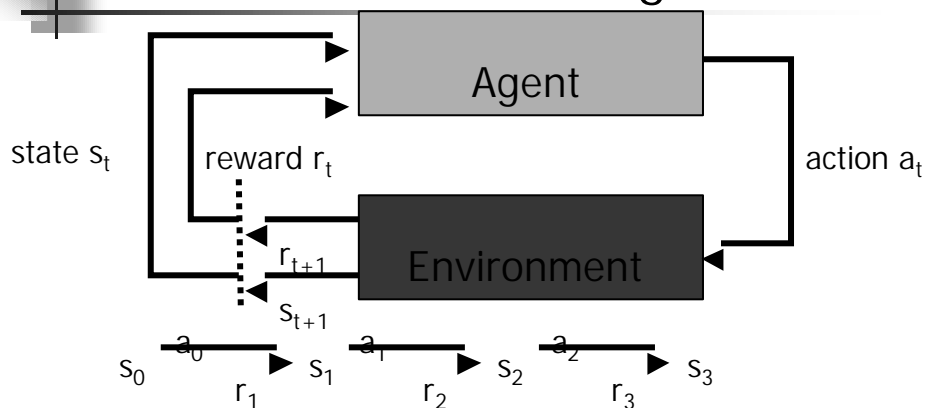
Problem characteristics:

- ⌘ Delayed reward
- ⌘ No direct feedback (error signal) for good and bad actions
- ⌘ Opportunity for active exploration
- ⌘ Possibly that state is only partially observable

Learning to play Backgammon

- ⌘ Immediate reward
 - ⌘ +100 win
 - ⌘ -100 loose
 - ⌘ 0 for all other actions/states
- ⌘ Trained by playing 1.5 million games against itself (Tesauro [1995])
- ⌘ Now approximately equal to the best human player

Reinforcement Learning Problem



Goal: Learn to choose actions a_t that maximize future rewards $r_1 + \gamma r_2 + \gamma^2 r_3 + \dots$, where $0 < \gamma < 1$ is a discount factor

Markov Decision Process (MDP)

- ⌘ Finite set of states S
- ⌘ Finite set of actions A
- ⌘ At each time step the agent observes state $s_t \in S$ and chooses action $a_t \in A(s_t)$
- ⌘ Then receives immediate reward r_{t+1}
- ⌘ And state changes to s_{t+1}
- ⌘ Markov assumption : $s_{t+1} = \gamma(s_t, a_t)$ and $r_{t+1} = r(s_t, a_t)$
 - ⌘ Next reward and state only depend on *current* state s_t and action a_t
 - ⌘ Functions $\gamma(s_t, a_t)$ and $r(s_t, a_t)$ may be non-deterministic
 - ⌘ Functions $\gamma(s_t, a_t)$ and $r(s_t, a_t)$ not necessarily known to agent

Learning Task

Execute actions in the environment, observe results and

- ⌘ Learn a policy $\pi_t(s, a) : S \rightarrow A$ from states $s_t \in S$ to actions $a_t \in A$ that maximizes the expected reward : $E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots]$ from any starting state s_t
- ⌘ $0 < \gamma < 1$ is the discount factor for future rewards
- ⌘ Target function is $\pi_t(s, a) : S \rightarrow A$
- ⌘ But there are no direct training examples of the form $\langle s, a \rangle$
- ⌘ Training examples are of the form $\langle \langle s, a \rangle, r \rangle$

State Value Function

Consider deterministic environments, namely $\gamma(s,a)$ and $r(s,a)$ are deterministic functions of s and a .

For each policy $\pi(s,a) : S \rightarrow A$ the agent might adopt we define an evaluation function:

$$V^\pi(s) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

where r_t, r_{t+1}, \dots are generated by following the policy π from start state s

Task: Learn the optimal policy π^* that maximizes $V^\pi(s)$

$$\pi^* = \operatorname{argmax}_\pi V^\pi(s), \forall s$$

Action Value Function

- State value function denotes the reward for starting in state s and following policy π

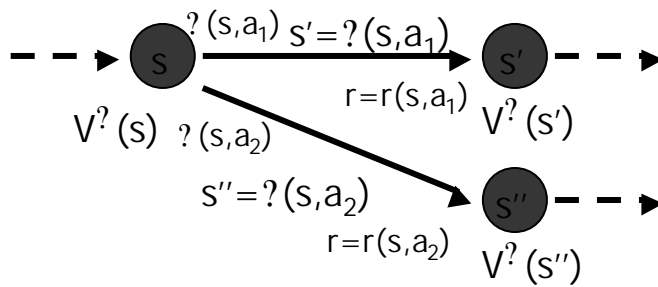
$$V^\pi(s) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- Action value function denotes the reward for starting in state s , taking action a and following policy π afterwards.

$$Q^\pi(s,a) = r(s,a) + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots = r(s,a) + \gamma V^\pi(\pi(s,a))$$

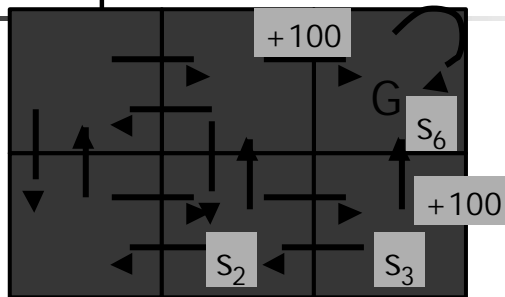
Bellman Equation (Deterministic Case)

$$\begin{aligned} V^{\pi}(s) &= r_t + \gamma V^{\pi}(s_{t+1}) \\ &= \sum_a \pi(s,a) (r(s,a) + \gamma V^{\pi}(s')) \end{aligned}$$



Set of $|s|$ linear equations, solve it directly or by policy evaluation.

Example



G : terminal state, upon entering G agent obtains a reward of $+100$, remains in G forever and obtains no further rewards

Compute $V^{\pi}(s)$ for equi-probable policy $\pi(s,a)=1/|a|$
 $V^{\pi}(s_3) = \frac{1}{2} V^{\pi}(s_2) + \frac{1}{2} (100 + V^{\pi}(s_6))$

Iterative Policy Evaluation

Instead of solving the Bellman equation directly one can use *iterative policy evaluation* by using the Bellman equation as an update rule.

$$V_{k+1}^{\pi}(s) = \sum_a \pi(s,a) (r(s,a) + \gamma V_k^{\pi}(\pi(s,a)))$$

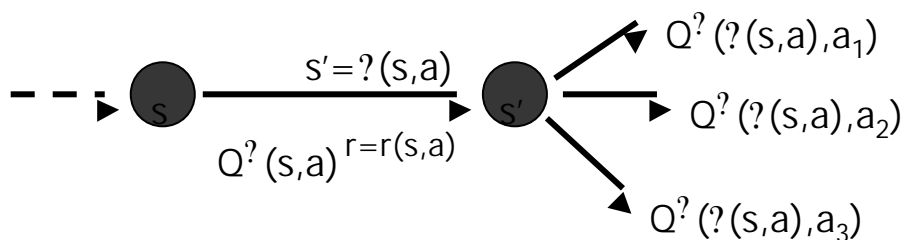
The sequence V_k^{π} is guaranteed to converge to V^{π}

$\gamma=0.9$

$V_{50}^{\pi}=52$	$V_{50}^{\pi}=66$	$V_{50}^{\pi}=0$
$V_{50}^{\pi}=49$	$V_{50}^{\pi}=57$	$V_{50}^{\pi}=76$

Bellman Equation (Deterministic Case)

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{a'} \pi(\pi(s,a),a') Q^{\pi}(\pi(s,a),a')$$

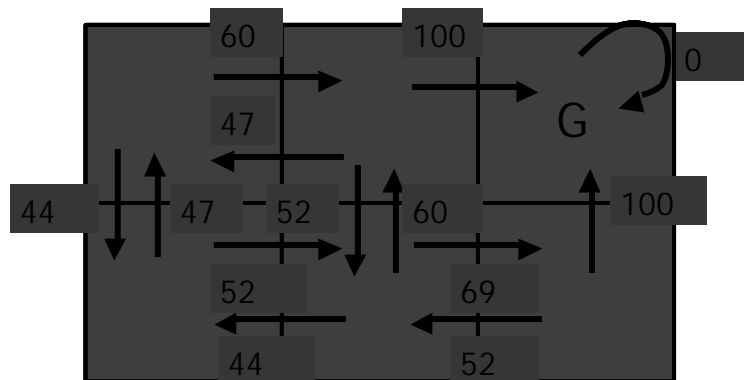


Iterative Policy Evaluation

- Bellman equation as an update rule for action-value function:

$$Q_{k+1}^{\pi}(s,a) = r(s,a) + \gamma \sum_{a'} \pi(a'|s,a) Q_k^{\pi}(s,a')$$

$\gamma=0.9$



Optimal Value Functions

- $V^*(s) = \max_{\pi} V^{\pi}(s)$
- $Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$

Bellman optimality equations

- $V^*(s) = \max_a Q^*(s,a)$
 $= \max_a (r(s,a) + \gamma V^*(\pi(s,a)))$
- $Q^*(s,a) = r(s,a) + \gamma V^*(\pi(s,a))$
 $= r(s,a) + \gamma \max_{a'} Q^*(\pi(s,a),a')$

Policy Improvement

- Suppose we have determined the value function V^{π} for an arbitrary deterministic policy π .
- For some state s we would like to know if it is better to choose an action a ($\pi(s)$).
- Select a and follow the existing policy π afterwards gives us reward $Q^{\pi}(s,a)$
- If $Q^{\pi}(s,a) > V^{\pi}$ then a is obviously better than $\pi(s)$
- Therefore choose new policy π' as

$$\pi'(s) = \operatorname{argmax}_a Q^{\pi}(s,a) = \operatorname{argmax}_a r(s,a) + \gamma V^{\pi}(\pi(s,a))$$

Example

$\gamma = 1$

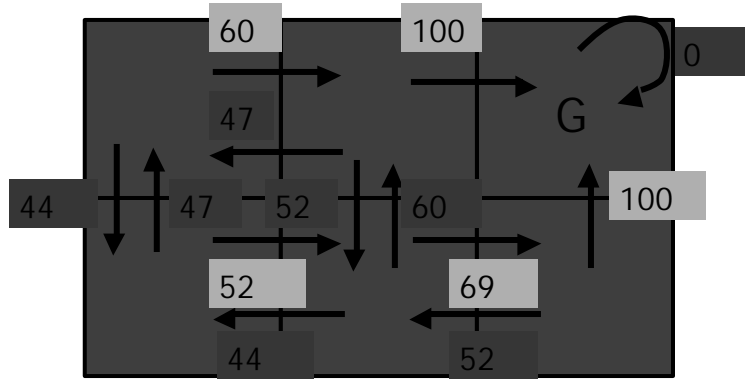
		$r=100$	
	$V^{\pi} = 63$	$V^{\pi} = 71$	$V^{\pi} = 0$
$\pi(s,a) = 1/ a $	$V^{\pi} = 56$	$V^{\pi} = 61$	$V^{\pi} = 78$
			$r=100$

$$\pi'(s) = \operatorname{argmax}_a r(s,a) + \gamma V^{\pi}(\pi(s,a))$$

$V^{\pi'} = 90$	$V^{\pi'} = 100$	$V^{\pi'} = 0$
$V^{\pi'} = 81$	$V^{\pi'} = 90$	$V^{\pi'} = 100$

Example

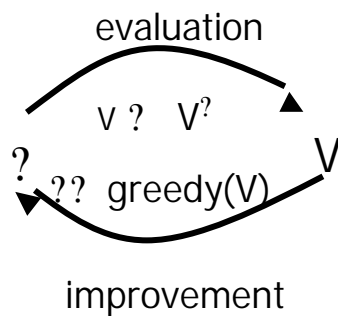
$$\pi'(s) = \operatorname{argmax}_a Q^\pi(s, a)$$



Generalized Policy Iteration

Intertwine *policy evaluation* with *policy improvement*

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} V^{\pi_2} \dots \xrightarrow{I} \pi^* \xrightarrow{E} V^{\pi^*}$$



Value Iteration (Q-Learning)

- ✗ Idea: do not wait for policy evaluation to converge, but improve policy after each iteration.

$$V_{k+1}(s) = \max_a (r(s,a) + \gamma V_k(s,a))$$

or

$$Q_{k+1}(s,a) = r(s,a) + \gamma \max_{a'} Q_k(s,a')$$

Stop when $\sum_s |V_{k+1}(s) - V_k(s)| < \epsilon$

or $\sum_{s,a} |Q_{k+1}(s,a) - Q_k(s,a)| < \epsilon$

Non-Deterministic Case

- ✗ State transition function $P(s',a)$ no longer deterministic but probabilistic given by

$$P(s'|s,a) = \Pr\{s_{t+1}=s' | s_t=s, a_t=a\}$$

Transition probability that given a current state s and action a the next state is s' .

- ✗ Reward function $r(s,a)$ no longer deterministic but probabilistic given by

$$R(s',s,a) = E\{r_{t+1} | s_t=s, a_t=a, s_{t+1}=s'\}$$

- ✗ $P(s'|s,a)$ and $R(s',s,a)$ completely specify MDP.

Bellman Equation (Non-Deterministic Case)

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) [R(s',s,a) + \gamma V^*(s')]$$

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a) [R(s',s,a) + \gamma V^*(s')]$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) [R(s',s,a) + \gamma \max_{a'} Q^*(s',a')]$$

Bellman optimality equations:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a) [R(s',s,a) + \gamma V^*(s')]$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a) [R(s',s,a) + \gamma \max_{a'} Q^*(s',a')]$$

Value Iteration (Q-Learning)

$$V_{k+1}^*(s) = \max_a \sum_{s'} P(s'|s,a) [R(s',s,a) + \gamma V_k^*(s')]$$

or

$$Q_{k+1}^*(s,a) = \sum_{s'} P(s'|s,a) [R(s',s,a) + \gamma \max_{a'} Q_k^*(s',a')]$$

Stop when $\sum_s |V_{k+1}^*(s) - V_k^*(s)| < \epsilon$

or $\sum_{s,a} |Q_{k+1}^*(s,a) - Q_k^*(s,a)| < \epsilon$

Example

$P(s' s,a) = 0$	$P(s' s,a) = (1-p)/3$	$P(s' s,a) = 0$
$P(s' s,a) = (1-p)/3$	s →	$P(s' s,a) = p + (1-p)/3$

- Now assume that actions a are non-deterministic, with probability p agent moves to the correct square indicated by a , with probability $(1-p)$ agent moves to a random neighboring square.

Example

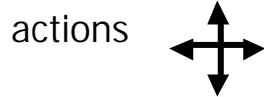
Deterministic optimal value function

$V^* = 90$	$V^? = 100$	$V^? = 0$
$V^? = 81$	$V^? = 90$	$V^? = 100$

Non-deterministic optimal value function $p=0.5$

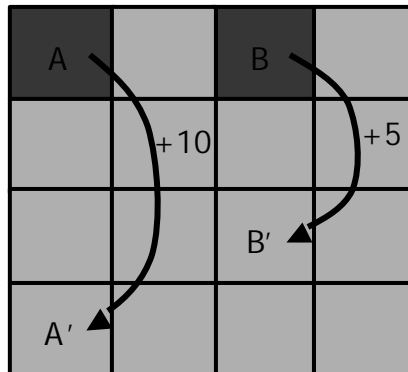
$V^* = 77$	$V^? = 90$	$V^? = 0$
$V^? = 71$	$V^? = 80$	$V^? = 93$

Homework



Reward :

- +10 for A? A'
- +5 for B? B'
- 1 for falling off the grid
- 0 otherwise



Infinite horizon no terminal state

1. Compute $V^?$ for equi-probable policy
2. Compute V^* and Q^* using policy or value iteration
3. Compute V^* and Q^* using policy or value iteration but assume that with $1-p=0.3$ the agent moves to a random neighbor state

Reinforcement Learning

- ✗ What if the transition probabilities $P(s'|s,a)$ and reward function $R(s',s,a)$ are unknown?
- ✗ Can we still learn $V(s)$, $Q(s,a)$ and identify an optimal policy $\pi^*(s,a)$?
- ✗ The answer is yes. Consider the observed rewards r_t and state transitions s_{t+1} as training samples drawn from the true underlying probability functions $R(s',s,a)$ and $P(s'|s,a)$.
- ✗ Use approximate state $V(s)$ and action value $Q(s,a)$ functions

Monte Carlo Method

- ⌘ Initialize:
 - ⌘ π policy to be evaluated
 - ⌘ $V(s)$ an arbitrary state-value function
 - ⌘ $\text{Returns}(s)$ an empty list, for all $s \in S$
- ⌘ Repeat forever
 - ⌘ Generate an episode using π
 - ⌘ For each state s appearing in the episode:
 - ⌘ R return following the first occurrence of s
 - ⌘ Append R to $\text{Returns}(s)$
 - ⌘ $V(s) \leftarrow \text{average}(\text{Returns}(s))$

Monte Carlo Method

$$V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$$

$$\text{where } R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$$

r_t is the observed reward after time t and α is a constant step-size parameter

$V^t = 30$	$V^t = 60$	$V^t = 0$
$V^t = 30$	$V^t = 40$	$V^t = 70$

\uparrow \rightarrow \rightarrow

$$V(s_t) \leftarrow 30 + 0.1 [0 + 0.9 \cdot 0 + 0.9^2 \cdot 100 - 30] = 35.1$$

Temporal Difference Learning

⌘ Monte-Carlo:

⌘ $V(s_t) \leftarrow V(s_t) + \alpha [R_t - V(s_t)]$

⌘ target for $V(s) : E \{R_t \mid s_t=s\}$

Must wait until the end of the episode to update V

⌘ Temporal Difference (TD):

⌘ $V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$

⌘ target for $V(s) : E \{r_{t+1} + \gamma V(s_{t+1}) \mid s_t=s\}$

TD method is *bootstrapping* by using the existing estimate of the next state $V(s_{t+1})$ for updating $V(s_t)$

TD(0) : Policy Evaluation

Initialize:

⌘ π policy to be evaluated

⌘ $V(s)$ an arbitrary state-value function

Repeat for each episode

⌘ Initialize s

⌘ Repeat for each step of episode

⌘ a action given by π for s

⌘ Take action a , observe reward r , and next state s'

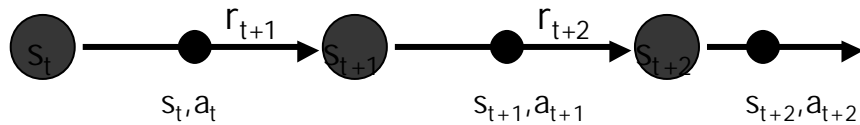
⌘ $V(s) \leftarrow V(s) + \alpha [r + \gamma V(s') - V(s)]$

⌘ $s \leftarrow s'$

⌘ Until s is terminal

TD(0): Policy Iteration

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$



The update rule uses a quintuple of events $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, therefore called SARSA.

Problem: Unlike in the deterministic case we can not choose a completely greedy policy $\pi(s) = \max_a Q(s, a)$, as due to the unknown transition and reward functions $\gamma(s, a)$ and $r(s, a)$ we can not be sure if another action might eventually turn out to be better.

ϵ -greedy policy

ϵ Soft policy: $\pi(s, a) > 0$ for all $s \in S, a \in A(s)$

Non-zero probability off choosing every possible action

ϵ ϵ -greedy policy: Most of the time with probability $(1-\epsilon)$ follow the optimal policy

$$\pi(s) = \max_a Q(s, a)$$

but with probability ϵ pick a random action:

$$\pi(s, a) = \frac{\epsilon}{|A(s)|}$$

ϵ Let $\epsilon \rightarrow 0$ go to zero as $t \rightarrow \infty$ for example $\epsilon = 1/t$ so that ϵ -greedy policy converges to the optimal deterministic policy

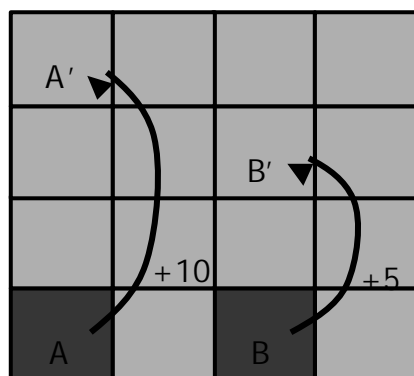
SARSA Policy Iteration

Initialize $Q(s,a)$ arbitrarily:

Repeat for each episode

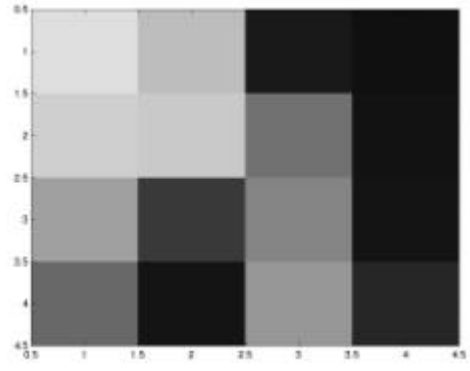
- ⌘ Initialize s
- ⌘ Choose a from using ϵ -greedy policy derived from Q
- ⌘ Repeat for each step of episode
 - ⌘ Take action a , observe reward r , and next state s'
 - ⌘ $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma Q(s',a) - Q(s,a)]$
 - ⌘ $s \leftarrow s', a \leftarrow a'$
- ⌘ Until s is terminal

SARSA Example



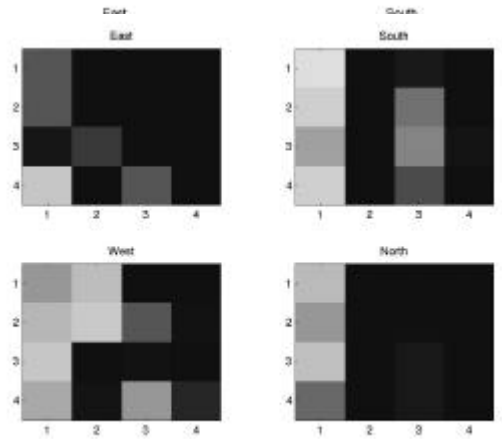
SARSA Example $V(s)$

$V(s)$ after 100, 200, 1000, 2000, 5000, 10000 SARSA steps



SARSA Example $Q(s,a)$

$Q(s,a)$ after 100, 200, 1000, 2000, 5000, 10000 SARSA steps



Q-Learning (Off-Policy TD)

- ≈ Approximates the optimal value functions $V^*(s)$ or $Q^*(s,a)$ independent of the policy being followed.
 - ≈ The policy determines which state-action pairs are visited and updated
- $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

Q-Learning Off-Policy Iteration

Initialize $Q(s,a)$ arbitrarily:

Repeat for each episode

- ≈ Initialize s
- ≈ Choose a from s using ϵ -greedy policy derived from Q
- ≈ Repeat for each step of episode
 - ≈ Take action a , observe reward r , and next state s'
 - ≈ $Q(s,a) \leftarrow Q(s,a) + \alpha [r + \max_{a'} Q(s', a') - Q(s,a)]$
 - ≈ $s \leftarrow s'$
- ≈ Until s is terminal

TD versus Monte-Carlo

So far TD uses one-step look-ahead but why not use 2-steps or n-steps look-ahead to update $Q(s,a)$.

≈ 2-step look-ahead

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+2}, a_{t+2}) - Q(s_t, a_t)]$$

≈ N-step look-ahead

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t)]$$

≈ Monte-Carlo method

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{N-1} r_{t+N} - Q(s_t, a_t)]$$

(compute total reward R^T at the end of the episode)

Temporal Difference Learning

Drawback of one-step look-ahead:

≈ Reward is only propagated back to the successor state (takes long time to finally propagate to the start state)



Initial V :

$$V(s) = 0 \quad V(s) = 0 \quad V(s) = 0 \quad V(s) = 0 \quad V(s) = 0$$

After first episode:

$$V(s) = 0 \quad V(s) = 0 \quad V(s) = 0 \quad V(s) = 0 \quad V(s) = ? * 100$$

After second episode:

$$V(s) = 0 \quad V(s) = 0 \quad V(s) = 0 \quad V(s) = ??? * 100 \quad V(s) = ? * 100 + ..$$

Monte-Carlo Method

Drawback of Monte-Carlo method

- ⌘ Learning only takes place after an episode terminated
- ⌘ Performs a random walk until goal state is discovered for the first time as all state-action values seem equal
- ⌘ It might take long time to find the goal state by random walk
- ⌘ TD-learning actively explores state space if each action receives a small default penalty

N-Step Return

Idea: blend TD learning with Monte-Carlo method

Define:

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$$

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$$

...

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

The quantity $R_t^{(n)}$ is called the n-step return at time t.

TD(?)

- ⌘ n-step backup : $V_t(s_t) \leftarrow V_t(s_t) + \alpha [R_t^{(n)} - V_t(s_t)]$
- ⌘ TD(?) : use average of n-step returns for backup

$$R_t^{(n)} = (1-\alpha) \sum_{n=1}^{\infty} \alpha^{n-1} R_t^{(n)}$$

$$R_t^{(n)} = (1-\alpha) \sum_{n=1}^{T-t-1} \alpha^{n-1} R_t^{(n)} + \alpha^{T-t-1} R_t$$

(if s_T is a terminal state)

The weight of the n-step return decreases with a factor of α

TD(0): one-step temporal difference method

TD(1) : Monte-Carlo method

Eligibility Traces

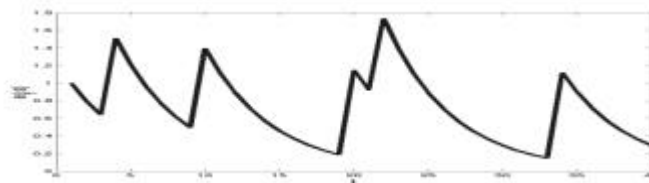
Practical implementation of TD(?):

With each state-action pair associate an eligibility trace $e_t(s,a)$

On each step, the eligibility trace for all state-action pairs decays by a factor α and the eligibility trace for the one state and action visited on the step is incremented by 1.

$$e_t(s,a) = \alpha e_{t-1}(s,a) + 1 \quad \text{if } s=s_t \text{ and } a=a_t$$

$$= \alpha e_{t-1}(s,a) \quad \text{otherwise}$$



On-line TD(?)

Initialize $Q(s,a)$ arbitrarily and $e(s,a)=0$ for all s,a :

Repeat for each episode

- ⌘ Initialize s,a
- ⌘ Repeat for each step of episode
 - ⌘ Take action a , observe r, s'
 - ⌘ Choose a' from s' using policy derived from Q (ϵ -greedy)
 - ⌘ $\delta = r + \gamma Q(s',a') - Q(s,a)$
 - ⌘ $e(s,a) = e(s,a) + \delta$
- For all s,a :
 - ⌘ $Q(s,a) = Q(s,a) + \gamma \delta e(s,a)$
 - ⌘ $e(s,a) = \gamma e(s,a)$
 - ⌘ $s = s', a = a'$
- ⌘ Until s is terminal

Function Approximation

- ⌘ So far we assumed that the action value function $Q(s,a)$ is represented as a table.
- ⌘ Limited to problems with a small number of states and actions
- ⌘ For large state spaces a table based representation requires large memory and large data sets to fill them accurately
- ⌘ Generalization: Use any supervised learning algorithm to estimate $Q(s,a)$ from a limited set of action value pairs
 - ⌘ Neural Networks (Neuro-Dynamic Programming)
 - ⌘ Linear Regression
 - ⌘ Nearest Neighbors

Function Approximation

- Minimize the mean-squared error between training examples $Q_t(s_t, a_t)$ and the true value function $Q^*(s, a)$

$$\sum_{s \in S} P(s) [Q^*(s, a) - Q_t(s_t, a_t)]^2$$

- Notice that during policy iteration $P(s)$ and $Q^*(s, a)$ change over time
- Parameterize $Q^*(s, a)$ by a vector $\theta = (\theta_1, \dots, \theta_n)$ for example weights in a feed-forward neural network

Stochastic Gradient Descent

- Use gradient descent to adjust the parameter vector θ in the direction that reduces the error for the current example

$$\theta_{t+1} = \theta_t + \alpha [Q^*(s_t, a_t) - Q_t(s_t, a_t)] \nabla_{\theta} Q_t(s_t, a_t)$$

- The target output q_t of the t -th training example is not the true value of Q^* but some approximation of it.

$$\theta_{t+1} = \theta_t + \alpha [v_t - Q_t(s_t, a_t)] \nabla_{\theta} Q_t(s_t, a_t)$$

- Still converges to a local optimum if $E\{v_t\} = Q^*(s_t, a_t)$ if $\alpha > 0$ for $t \rightarrow \infty$