CS294-40 Learning for Robotics and Control Lecture 15 - 10/28/2008

TD, Sarsa, Q-learning, TD-Gammon

Lecturer: Pieter Abbeel *Scribe: Anand Kulkarni*

1 Lecture outline

- TD(λ), Q(λ), Sarsa(λ)
- Function approximation.
- TD-gammon by Tesauro, one of the (early) success stories of reinforcement learning

2 TD Algorithm

Recall that in model-free methods, we operate an agent in an environment and build a Q-model from experience, without reference to a dynamics model. In these systems, at time t, we set the value function to be

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

This is a stochastic version of Bellman backups, where we are sampling only one possible future state. This corresponds to the state that our agent encounters in the environment. The term $[R(s_t) + \gamma V(s_{t+1})]$ is thus an approximation $\widehat{V}(s)$ to the true value function $V(s)$. The update is only "partial" in the sense that the new $V(s_t)$ is a weighted combination of the old estimate and the approximation from the current experience $\widehat{V}(s)$. Intuitively, by using only partial updates the stochasticity will average out over time, resulting in convergence to the value function.
Rearranging this expression, we have

$$V(s_t) \leftarrow V(s_t) + \alpha \underbrace{[R(s_t) + \gamma V(s_{t+1}) - V(s_t)]}_{\delta_t}$$

Similarly, the update at the next time step is

$$V(s_{t+1}) \leftarrow V(s_{t+1}) + \alpha \underbrace{(R(s_{t+1}) + \gamma V(s_{t+2}) - V(s_{t+1})}_{\delta_{t+1}}$$

Note that at the next time step we update $V(s_{t+1})$. This (crudely speaking) results in having a better estimate of the value function for state s_{t+1}. TD(λ) takes advantage of the availability of this better estimate to improve the update we performed for $V(s_t)$ in the previous step of the algorithm. Concretely, TD(λ) performs another update on $V(s_t)$ to account for our improved estimate of $V(s_{t+1})$ as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha\gamma\lambda\delta_{t+1}$$

where λ is a fudge factor that determines how heavily we weight changes in the value function for s_{t+1}. Similarly, at time $t+2$ we perform the following set of updates:

$$V(s_{t+2}) \leftarrow V(s_{t+2}) + \alpha \underbrace{[R(s_{t+2}) + \gamma V(s_{t+2}) - V(s_{t+3})]}_{\delta_{t+2}} ...$$

$$V(s_{t+1}) \leftarrow V(s_{t+1}) + \alpha\gamma\lambda\delta_{t+2}$$

$$V(s_t) \leftarrow V(s_t) + \alpha \underbrace{\gamma^2\lambda^2}_{e(s_t)} \delta_{t+2}$$

The term $e(s_t)$ is called the *eligibility vector*. In TD, we maintain the value of the eligibility vector as our agent gains more experience, and use it to update our value function according to the following algorithm:

$TD(\lambda)$

- Initialize V(s) arbitrarily and initialize $e(s) = 0$
- Iterate over t:
  - Select action a_t and execute it
  - $\delta_t \leftarrow R(s_t) + \gamma V(s_{t+1}) - V(s_t)$
  - $e(s_t) \leftarrow e(s_t) + 1$
  - For all states s:
    - $v(s) \leftarrow v(s) + \alpha e(s)\delta_t$
    - $e(s) \leftarrow \gamma\lambda e(s)$

Q-learning with λ can be carried out in a similar fashion.

3 Value function approximation

State spaces may be extremely large, and tracking all possible states and actions may require excessive space. We can generalize the TD(λ) and Q(λ) algorithms to eliminate the need to maintain an explicit table of states. Instead, we make use of an approximate value function by keeping track of an approximate state space θ, where $|\theta| < |S|$.

Let $V(s) \simeq f_\theta(s)$, where f is parameterized in θ. Now, instead of updating $V(s)$ directly, we can update θ according to one of many possible methods. One approach is to use gradient methods. We update θ in a direction that mostly reduces error on the example observed. We want to move in the direction θ that minimizes $g = \frac{1}{2}(\widehat{V}(s_t) - f_\theta(s_t))^2$. This is simply

$$-\nabla_\theta(g) = (\widehat{V}(s_t) - f_\theta(s_t)) \bigtriangledown_\theta f_\theta(s_t)$$

We perform updates according to the following rule, which retains some memory of previous states:

$$\theta_{t+1} \leftarrow \theta_t + \alpha(\widehat{V}(s_t) - f_\theta(s_t)) \bigtriangledown_\theta f_\theta(s_t)$$

Incorporating this into the TD(λ) algorithm is easy.

$Approximate - TD(\lambda)$

- Initialize V(s) arbitrarily and initialize $e(s) = 0$
- Iterate over t:
  - Select action a_t and execute it

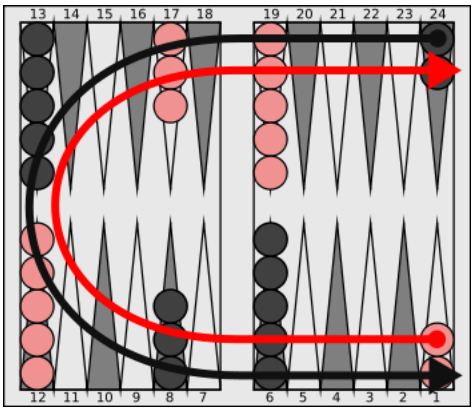
- $\delta_t \leftarrow R(s_t) + \gamma f_\theta(s_{t+1}) - f_\theta(s_t)$
- $e(s_t) \leftarrow \gamma\lambda e + \bigtriangledown_\theta f_\theta(s)$
- $\theta \leftarrow \theta + \alpha\delta e$

This approximate algorithm is quite easy to implement, often requiring no more than three or four lines. However, it is not guaranteed to converge in this general setting. [There are some convergence proofs in the case of linear function approximation.]

4 TD-Gammon (Tesaro 1992,1994,1995)

4.1 Backgammon

The goal of backgammon is to move all of your pieces, either black or red, off the board. One player moves clockwise around the board, the other counterclockwise. Each turn, a player rolls two dice and moves one or two pieces along the board a total number of spaces equal to the total on the dice. Pieces cannot be placed atop spaces with two or more opposing pieces; if only one opposing piece is present, it is captured and placed on the center bar. If any of a player's pieces are captured, they must be moved at the start of that player's turn, before any other pieces can be moved. If a player cannot move any pieces, the player must pass.



Typically, games are solved by building a game tree of the states of the board that might occur after a particular action, then doing a value function approxmation for the states at the bottom of the game tree. We then hope that we can search the tree deeply enough to evaluate the strength of future states.

Backgammon is interesting from an AI perspective due to the high branching factor in the game tree, with around 400 options at each turn. Consequently, the heuristic searches used in chess and checkers tend not to perform as well for backgammon.

4.2 Neurogammon

Tesaro's first approach used an artificial neural network to produce a value function ("Neurogammon"). He collected a large number of game configurations and dice rolls, then asked expert players what moves they would make in these states. These were labeled as the "best" actions for each of the observed state, then fed to a neural network architecture with 198 binary inputs representing various features of the game board's state (for instance, whether there were more than three opposing pieces on a particular space).

40-80 hidden layers were used in the neural network, with a transfer function in the nodes of $y = \frac{e^{\omega Tx}}{1+e^{-\omega Tx}}$. Then, the demonstrated policy θ_π was given to the neural network and trained to obtain $f_\theta(s)$. The resulting value function was applied in the usual way to determine a game strategy.

4.3 TD-Gammon

Later, Tesauro decided to try a TD approach. In TD-Gammon version 0.0, the machine was set to play against itself in random games.

The gradient rule mentioned earlier was used to update the approximate state space:

$$\theta_0 = random$$

$$\theta_{t+1} \leftarrow \theta_t + \alpha(\widehat{V}(s_t) - f_\theta(s_t)) \bigtriangledown_\theta f_\theta(s_t)$$

A simple greedy strategy was used to select actions:

$$a_t = \arg\max_a \Sigma_{s_{t+1}}(R(s_t) + \gamma V(s_{t+1}) \cdot P(s_{t+1}|s_t, a))$$

The program played against itself for 300,000 games, and remarkably, was able to match the performance of the best previous backgammon programs of the time, including, notably, Tesaro's own neural network approach. In TD-Gammon version 1.0, Tesaro combined ideas from Neurogammon and the previous version of TD-gammon. This program was able to substantially outperform all other computer programs at the time, and was able to offer serious competition to expert backgammon players. Later versions of TD-gammon applied selective 2-ply search procedures to further improve the system.

TD-Gammon remains one of the great success stories of TD(λ) learning.

5 Summary of Methods to Date

We close with a brief comparison of all the methods we've examined so far in class: model-based and model-free techniques.

	RL: Q, TD, Sarsa	Value Iteration, Policy Iteration, LP
Reward	From experience	Known in advance
Dynamics	From experience	Known in advance

Next time, we'll look at how to make reinforcement learning techniques work better through reward shaping.