COMP90054 — AI Planning for Autonomy

9. n-step reinforcement learning Learning quicker using look-aheads

Adrian Pearce

THE UNIVERSITY OF MELBOURNE

Semester 2, 2021

Agenda

1. Motivation
2. n-step learning: TD(λ)
3. Combining MCTS and TD (Not examinable, but very interesting!)

Learning Outcomes

1. Manually apply n-step reinforcement learning approximation to solve small-scale MDP problems given a set of
2. Design and implement n-step reinforcement learning to solve medium-scale MDP problems automatically
3. Argue the strengths and weaknesses of n-step reinforcement learning

Discounted Future Rewards (again)

When calculating a discounted reward over a trace, we can re-write as:

$$
\begin{aligned}
G_t &= r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \ldots \\
&= r_1 + \gamma(r_2 + \gamma(r_3 + \gamma(r_4 + \ldots)))
\end{aligned}
$$

If G_t is the value received at time-step t, then $G_t = r_t + \gamma G_{t+1}$

In TD(0) methods such as Q-learning and SARSA, we do not know G_{t+1} when updating $Q(s, a)$, so we estimate using bootstrapping:

$$G_t = r_t + \gamma \cdot V(s_{t+1})$$

That is, the reward of the entire future from step t is estimated as the reward at t plus the estimated (discounted) future reward from $t + 1$. $V(s_{t+1})$ is estimated using the maximum expected return (Q-learning) or the estimated value of the next action (SARSA).

This is a *one-step return*.

Truncated Discounted Rewards

However, we can estimate a two-step return:

$$G_t^2 = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2})$$

or three-step return:

$$G_t^3 = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3})$$

or n-step returns:

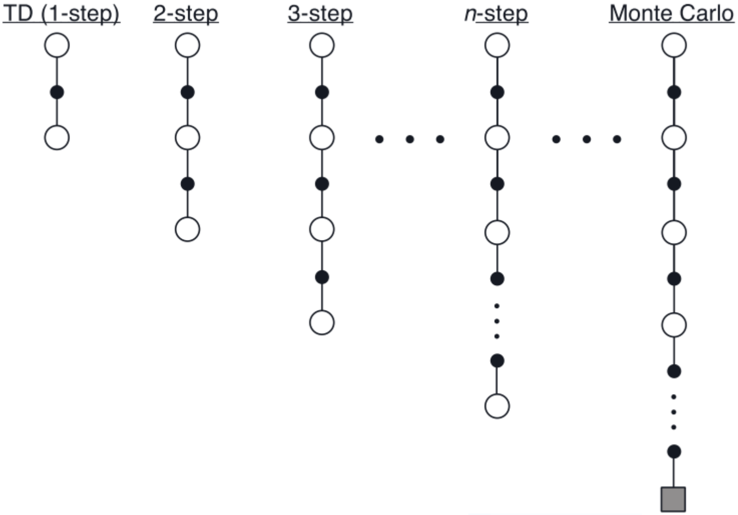
$$G_t^n = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \gamma^n V(s_{t+n})$$

In this above expression G_t^n is the full reward, *truncated* at n steps, at time t. The basic idea is that we do not update the Q-value immediately after executing an action: we wait n steps and update it based on the n-step return.

If T is the termination step and $t + n > T$, then we just use the full reward.

In Monte-Carlo methods, we go all the way to the end of an episode. Monte-Carlo Tree Search is one such Monte-Carlo method, but there are others that we do not cover.

Different Levels Truncated Rewards



Updating the Q-function

The update rule for the Q-function is then different. First, we need to calculate the truncated reward for n steps, in which τ is the time step that we are updating for:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} r_i$$

This just sums the rewards from time step $\tau + 1$ until either n steps ($\tau + n$) or termination of the episode (T), whichever comes first. For n-step SARSA, we have:

Then update:

$$\begin{aligned}&\text{If } \tau + n < T \text{ then } G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})\\&Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]\end{aligned}$$

The first line just adds the future expect reward if we are not at the end of the episode (if $\tau + n < T$).

But! It's not so simple

While conceptually this is not so difficult, we have to modify our algorithms quite a bit, because at each step, n-step return uses a reward from the future.

For the first $n-1$ steps of the any episode, we do not update Q at all.

Also, we have to continue updating $n-1$ steps after the end of the episode.

This leads to the algorithm on the following slide. All of the changes, except for the three lines immediately after 'if $\tau \geq 0$', just manage the n-steps.

Computationally, this is not much worse than 1-step learning. We need to store the last n states, but the per-step computation is small and uniform for n-step, just as for 1-step.

n-step SARSA

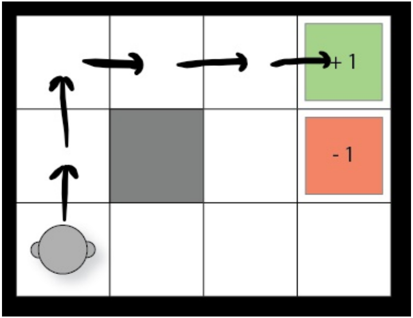
n-step Sarsa for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to a fixed given policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod n

```
Repeat (for each episode):
    Initialize and store S_0 ≠ terminal
    Select and store an action A_0 ~ π(·|S_0)
    T ← ∞
    For t = 0, 1, 2, ... :
    |   If t < T, then:
    |       Take action A_t
    |       Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
    |       If S_{t+1} is terminal, then:
    |           T ← t + 1
    |       else:
    |           Select and store an action A_{t+1} ~ π(·|S_{t+1})
    |   τ ← t − n + 1   (τ is the time whose estimate is being updated)
    |   If τ ≥ 0:
    |       G ← Σ_{i=τ+1}^{min(τ+n,T)} γ^{i−τ−1} R_i
    |       If τ + n < T, then G ← G + γ^n Q(S_{τ+n}, A_{τ+n})        (G_{τ:τ+n})
    |       Q(S_τ, A_τ) ← Q(S_τ, A_τ) + α [G − Q(S_τ, A_τ)]
    |       If π is being learned, then ensure that π(·|S_τ) is ε-greedy wrt Q
    Until τ = T − 1
```

Exercise

Consider our simple 2D navigation task, in which we do not know the probability transitions nor the rewards. Initially, the reinforcement learning algorithm will be required to search randomly until it finds a reward. Propagated this reward back n-steps will be helpful.



Assuming $Q(s, a) = 0$ for all s and a, if we (finally) traverse the episode the labelled episode, what will our Q-function look like for a 5-step update with $\alpha = 0.5$ and $\gamma = 0.9$?

Exercise (continued)

We only receive a reward in the last action, and all other actions give an immediate reward of 0 until then:

$$
\begin{aligned}
G &\leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} r_i \\
G_1 &\leftarrow 0 + \gamma^1 \cdot 0 + \ldots + \gamma^4 \cdot 1 \\
&\leftarrow 0.9^4 \cdot 1 \\
&\leftarrow 0.6561
\end{aligned}
$$

So, we update the Q-value for the state $(0, 2)$, which is 5 steps back:

$$
\begin{aligned}
Q((0,2),E) &\leftarrow Q((0,2),E) + \alpha[G_1 - Q((0,2),E)] \\
&\leftarrow 0 + 0.5[0.6561 - 0] \\
&\leftarrow 0.32805
\end{aligned}
$$

Exercise (continued)

The table below compares 1-step vs. 5-step SARSA for the trace above. In 1-step SARSA, reaching the reward only informs the state from which it is reached. Whereas for 5-step, it informs the previous five steps. Then, in the next episode, there is more chance of encountering a non-zero state, so which will again inform the five steps instead of just one. The rewards 'spread' throughout the Q-table faster.

With 1-step learning

State	**Action**			
	North	South	East	West
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
...				
(1,2)	0	0	0	0
(2,1)	0	0	0	0
(2,2)	0	0	0.45	0
(2,3)	0	0	0	0
...				

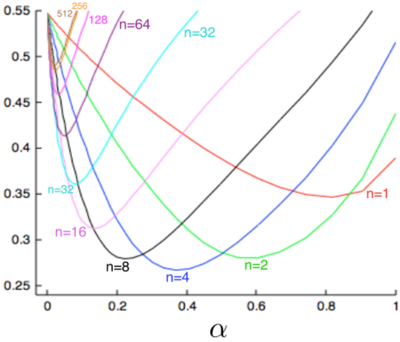
With 5-step learning

State	**Action**			
	North	South	East	West
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0.2953	0
...				
(1,2)	0	0	0.3281	0
(2,1)	0.405	0	0	0
(2,2)	0	0.3645	0.45	0
(2,3)	0	0	0	0
...				

Simple experiment: Random Walk

Consider the following simple deterministic Markov reward process:

The following shows results from a series of experiments in varying α and n. The y-axis shows the root mean-squared error:



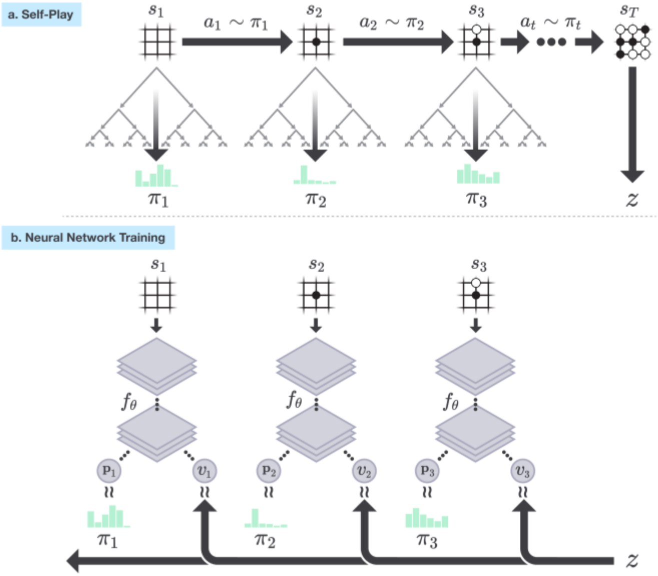
$n = 1$ is TD(0), while larger n are closer to Monte-Carlo methods. Note that the 'in between' parameters perform best in this example.

Combining MCTS and Reinforcement Learning: AlphaGo Zero (Not examinable)

AlphaGo Zero (or more accurately its predecessor AlphaGo) made headlines when it beat Go world champion Lee Sodol in 2016. It uses a combination of MCTS and (deep) reinforcement learning to learn a policy. A simple overview:

1. It uses a deep neural network to estimate the Q-function. More accurately, it gives an estimate of the probability of selecting action a in state s ($P(a|s)$), and the *value* of the state ($V(s)$), which represents the probability of the player winning from s.
2. It is trained via *self-play*.
3. At each move, AlphaGo Zero:
   1. Executes an MCTS search using UCB $Q(s,a) + P(s,a)/1 + N(s,a)$, which returns the probabilities of playing each move.
   2. The neural network is used to guide the MCTS by influencing $Q(s,a)$.
   3. The final result of a simulated game is used as the reward for each simulation.
   4. After a set number of MCTS simulations, the best move is chosen for self-play.
   5. Repeat steps 1-4 for each move until the self-play game ends.
   6. Then, feedback the result of the self-play game to update the Q function for each move.

AlphaGo is Best Summarised Using This Figure



Reading

- (Chapter 7 n-step Bootstrapping & Chapter 12 Eligibility Traces) of *Reinforcement Learning: An Introduction (Second Edition)* [*Sutton and Barto*, 2020]
  Available at:

  `http://www.incompleteideas.net/book/RLbook2020.pdf`
- *Mastering the Game of Go without Human Knowledge* from DeepMind.
  Available at:

  `https://deepmind.com/research/publications/mastering-game-go-without-human-knowledge` *(select Download link for pdf)*