Dynamic Programming and Reinforcement Learning

Week 3b: Temporal Difference Algorithms & Q-Learning

Rainer Schlosser und Alexander Kastius

Enterprise Platform and Integration Concepts

29.04.21

Recap

BI, VI, PI, ADP

- **Backward Induction (BI)**: For finite horizon MDPs, make use of the knowledge about the horizon
- **Value Iteration (VI), Policy Iteration (PI):** For infinite horizon MDPs, make use of full knowledge about the process. Events, state transitions, reward function etc. are known to the developer
- **Approximate Dynamic Programming (ADP):** Still assumes full knowledge, but prioritizes states by their occurrence in the simulation

Finite Horizon vs. Infinite Horizon

- **Finite Horizon MDPs:** Have a time *T* after which the process ends. Knowledge about this can drastically improve solution time by using backward induction
- **Infinite Horizon MDPs:** Have no fixed length, which makes BI impossible. Require either VI, PI or ADP to be solved successfully.

Problematic Process Mechanics

Can we model and solve a game like chess right now?

Problematic Process Mechanics

- Many systems hide their behavior from us
- Usually, we don't know our customers' buying behavior in pricing problems or the other players' reaction in chess
- Thus, we cannot exactly determine $P(i, a, s)$ or $r(i, a, s)$ or $\Gamma(i, a, s)$ or even what the event set I actually consists of

Learning by Simulation

- In many real world problems we have data to observe instances of our decision process
- An airline can observe past sales
- A chess player can keep track of many games

- From now on, we assume that we have some way of **generating a process trajectory of infinite length**

$$(s_0, a_0, r_0, \dots s_t, a_t, r_t, \dots)$$

- Within such a process trajectory, we have the *observed* discounted reward after reaching state s_t, $t \geq 0$:

$$G_t = \sum_{k \geq 0} \gamma^k r_{k+t}$$

Learning from Process Trajectories

t	0	1	2	3	4	…
s	s_0	s_1	s_2	s_3	s_4	…
a	a_0	a_1	a_2	a_3	a_4	…
r	r_0	r_1	r_2	r_3	r_4	…
G_t	$r_0 + \gamma G_1$	$r_1 + \gamma G_2$	$r_2 + \gamma G_3$	$r_3 + \gamma G_4$	G_4	…

Can we compute G_t given a trajectory?

What if the trajectory has finite length?

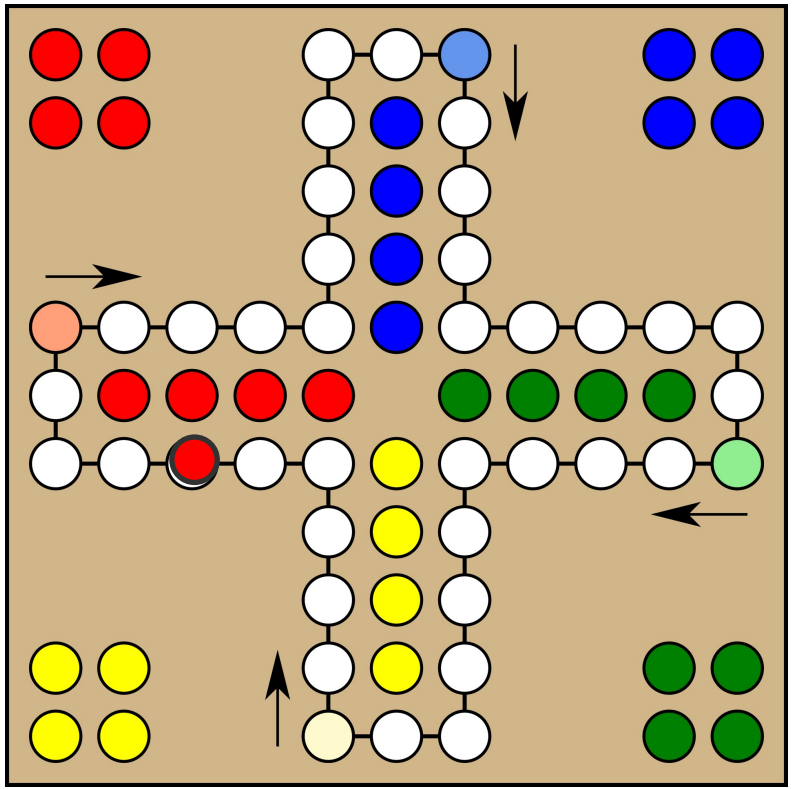
What is the meaning of $E\,[G_t \mid s_t = s]$?

“Infinite” Markov Processes with a Sink State

What’s going to happen here?

One token, one player, infinite throws?

Assume reward is 1 if a stone reaches its final place, 0 otherwise.



“Infinite” Markov Processes with a Sink State

- Idea: Keep an estimate of a state’s value, observe the trajectory and update the value accordingly.
- Simplification: We assume our process has a **sink state**, after which no further rewards occur.
- Notice: This is different from having a process with *fixed length*. **We do not know after how many steps we will reach the final state.**

t	0	…	T	$T+1$	…
s	s_0	…	s_T	s_T	…
a	a_0	…	a_T	?	…
r	r_0	…	r_T	0	…

Monte-Carlo Estimation

1. Observe the process until the final state is reached at some step T (which might have a different value in each observed process trajectory!)
2. Iterate through the process instance, compute G_t
3. For each observed state $s_t, 0 \leq t \leq T$ set:
$$V(s_t) \leftarrow G_t$$
4. Repeat from 1. with new process trajectory.

Problem?

s	$V(s)$
0	0
1	4
2	2
3	3
...	...

Monte-Carlo Estimation

1. Observe the process until the final state is reached at some step T (which might have a different value in each observed process trajectory!)
2. Iterate through the process instance, compute G_t for all $t = 0,1,\dots,T$, given $G_T = r_T$
3. For each state $s_t, t = 0,1,\dots,T$, $0 < \boldsymbol{\eta} < \mathbf{1}$ set:
   $$V(s_t) \leftarrow \boldsymbol{\eta}(\boldsymbol{G_t} - \boldsymbol{V(s_t)}) + \boldsymbol{V(s_t)}$$
4. Repeat from 1. with new process trajectory.

s	$V(s)$
0	0
1	4
2	2
3	3
…	…

Another Problem?

Works only if we know that a final state occurs at some point, otherwise we can never compute G_t

Temporal Difference Learning

Remember:

$$G_t = r_t + \gamma G_{t+1}$$

Do we have an estimate for G_{t+1}?

Yes!

We can reuse the estimate of the value of s_{t+1}:

$V^{(\pi)}(s_{t+1}) = E(\, G_{t+1} \mid s_{t+1} = s, a_t = \pi(s_t))$ under a policy π

Temporal Difference Learning

Recursive value function definition:

$$V_t^{(\pi)}(s) = E\left(r_t + \gamma V_{t+1}^{(\pi)}(s_{t+1}) | a_k = \pi(s_k), s_t = s\right)$$

- After observing s_t and performing a_t, we can observe r_t and s_{t+1}
- Given all 4 variables (s_t, a_t, r_t, s_{t+1}), we can compute a sample of the value that we would like to estimate
- This can be combined with the partial update logic of the monte-carlo approach leading us to TD(0):

$$V(s_t) \leftarrow \eta(\boldsymbol{r_t} + \boldsymbol{\gamma V(s_{t+1})} - \boldsymbol{V(s_t)}) + V(s_t)$$

- We can now incrementally learn the values of a given policy by measuring the difference between the estimated and the realized reward

Policy-Derivation

The method we use to estimate a policies value does not immediately allow us to improve this policy.

Why?

Because we do not know anything about state transitions. This prevents us from performing policy improvement based on the learned value function.

Solution Options:

(a) Learn the state transitions explicitly

(b) Learn state-action-values instead

Q-Value

$$Q^{(\pi)}(s,a) = E(G_t | a_k = \pi(s_k); k > t, s_t = s, a_t = a)$$

Which is equal to:

$$Q^{(\pi)}(s,a) = E\left(r_t + \gamma V^{(\pi)}(s_{t+1}) | a_k = \pi(s_k); k > t, s_t = s, a_t = a\right)$$

Which in turn is equal to:

$$Q^{(\pi)}(s,a) = E\left(r_t + \gamma Q^{(\pi)}(s_{t+1}, \pi(s_{t+1})) |\ s_t = s, a_t = a\right)$$

Question:

How does the Q-Value help us at this point?

SARSA

1. Observe s_t, choose a_t according to the current policy
2. Observe r_t, s_{t+1}, choose a_{t+1} according to the current policy
3. Update the Q-value estimate:

$$Q_t(s_t, a_t) \leftarrow \eta_t(r_t + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)) + Q_t(s_t, a_t)$$

4. Repeat from 1. with each new transition, reduce η_t over time (for example $\eta_t = \frac{1}{t}$)

	$s^{(1)}$	$s^{(2)}$	$s^{(3)}$	$s^{(4)}$	…
$a^{(1)}$	1.2	3.2	2.2	3.2	
$a^{(2)}$	1.1	3.1	6.2	5.2	
$a^{(3)}$	1.4	4.2	1.7	6.7	
$a^{(4)}$	2.8	0.2	4.3	0.2	
…					

What is our policy?

SARSA (2)

- The greedy policy: $\pi_t(s) = \underset{a \in A}{\operatorname{argmax}}\, Q_t(s,a)$ leads to a problem, similarly to ADP, in that we might end up not revisiting all state-action which prevents us from converging to the true Q-values.
- Revisiting all possible combinations of s and a in endless time is a convergence-criterion that we need to fulfill to ensure SARSA converges to the actual Q-values of the optimal policy
- We can re-introduce ϵ-greedy for that, setting $\epsilon = \frac{1}{t}$
- Under those circumstances SARSA is guaranteed to converge

Q-Learning

1. Observe s_t, choose a_t according to the current policy
2. Observe r_t, s_{t+1}, choose a_{t+1} according to the current policy
3. Update the Q-value estimate:
   $$Q_t(s_t, a_t) \leftarrow \eta_t(r_t + \gamma \max_{\boldsymbol{a \in A}} \boldsymbol{Q_t}(\boldsymbol{s_{t+1}}, \boldsymbol{a}) - Q_t(s_t, a_t)) + Q_t(s_t, a_t)$$
4. Repeat from 1. with each new transition, reduce η_t over time (for example $\eta_t = \frac{1}{t}$)

Policy at each point in time can be ϵ-greedy. Convergence if guaranteed if all combinations of s and a are revisited in endless time.

What is the difference of the Q-values in comparison to SARSA? Are we learning something different here?

On- vs. Off-Policy Learning

SARSA:

- Tuples are generated by the policy that we want to learn the values for
- Future estimation of a Q-value still depends on the policy
- The requirements for the policy forces us to generate the tuples in the specified order using the most current iteration of the policy
- If the policy used to generate the tuples is different, the values will change
- **This style of algorithm is called on-policy**

Q-Learning

- The tuples can be generated by **any** policy at any time, the generated Q-values will be the same
- The only requirement to ensure convergence: every combination of s and a that is visited repeatedly in endless time
- **This style of algorithm is called off-policy**

Recall

MDPs with Sink

Introduce a sink state to our MDPs which allows computation of G_t.

Monte Carlo

Use the knowledge about the sink state to learn $V(s)$ without any game knowledge.

SARSA

Use the knowledge about the recursivity of $\mathrm{Q}(s, a)$ to learn the Q-values and derive a policy from that.

QL

Improves SARSA by shortening the learning process with off-policy learning.

Schedule

- **3.5.: Problems with Q-Learning**
- 6.5.: Artificial Neural Networks
- 10.5.: Deep Q-Networks
- 17.5.: DQN Extensions (Rainbow)
- 20.5.: Policy Gradients (1)
- 27.5.: Policy Gradients (2)
- 31.5.: Project Assignments
- 3.6.: Project Assignments
- 7.6.: Project Work/Support
- 10.6.: Project Work/Support
- 14.6.: Project Work/Support
- 17.6.: Project Work/Support
- 21.6.: Project Work/Support
- 24.6.: Project Work/Support
- 28.6.: Project Work/Support
- 31.6.: Project Work/Support
- 5.7.: Project Work/Support
- 8.7.: Project Work/Support
- 12.7.: Final Presentations
- 15.7.: Final Presentations
- 31.8.: Final Documentation