COMP90054 — AI Planning for Autonomy

9. Model-Free Reinforcement Learning: Q-Learning and SARSA

How to learn without a model

Adrian Pearce

Semester 2, 2021

Agenda

Learning Outcomes

1. Identify situations in which model-free reinforcement learning is a suitable solution for an MDP
2. Explain how model-free planning differs from model-based planning
3. Apply Q-learning and SARSA to solve small-scale MDP problems manually and program Q-learning and SARSA algorithms to solve medium-scale MDP problems automatically
4. Compare and contrast off-policy reinforcement learning with on-policy reinforcement learning

Planning and Learning

So far, the this subject, we have at looked blind/heuristic search and value/policy iteration.

- Search and value/policy iteration are what are called as *model-based* techniques. This means that we need to know the model; in particular, we have access to $P_a(s' \mid s)$ and $r(s, a, s')$.
- Q-learning and SARSA (discussed in this lecture) are *model-free* techniques. This means that we do NOT know the $P_a(s' \mid s)$ and $r(s, a, s')$.

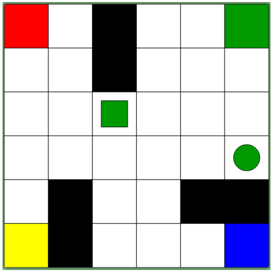
Key Question: How can we calculate a policy if we don't know the transitions and the rewards?!

Answer: We can *learn through experience* by trying actions and seeing what the results is, making this machine learning problem.

- Importantly, in model-free reinforcement learning, we do NOT try to learn $P_a(s' \mid s)$ or $r(s, a, s')$ — we learn a policy *directly*.
- There is something in between model-based and model-free: simulation-based techniques. In this cases, we have a model as a *simulator*, so we can *simulate* $P_a(s' \mid s)$ and $r(s, a, s')$ and learn a policy with a model-free technique.

Example: The Mystery Game

`https://programmingheroes.blogspot.com/2016/02/udacity-reinforcement-learning-mystery-game.html`



From this website (in Spanish, Chrome offered to translate into English when I visited): The aim of this game is to experiment how computers learn. Press keys from 1 to 6 to do actions. You need to learn what effects/outcomes the actions produce and how to win the game.

Some rewards values appears when you do the things very well or very bad. When you finish the game the phrase "You Win :)" appears in the board. Good luck!

Example: Mystery Game (continued)

What was the process you took?

- What did you learn?
- What assumptions did you use?

$\rightarrow$ Imagine how hard it is for a computer that doesn't have any assumptions or intuition!

Approaches to AI Planning and Reinforcement Learning

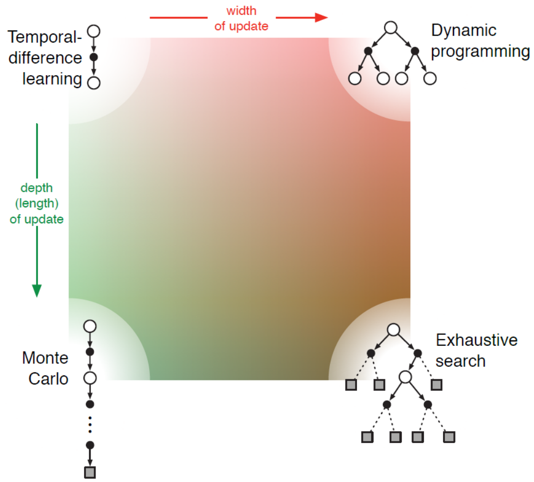


Figure from Sutton & Barto (read Section 8.13 for description of this figure)

Reinforcement Learning: The Basics

There are many different models of reinforcement learning, all with the same basis:

We execute many different *episodes* of the problem we want to solve, and from that we learn a *policy*.

- During learning, we try to learn the value of applying particular actions in particular states.
- During each episode, we need to execute some actions. After each action, we get a reward (which may be 0) and we can see the new state.

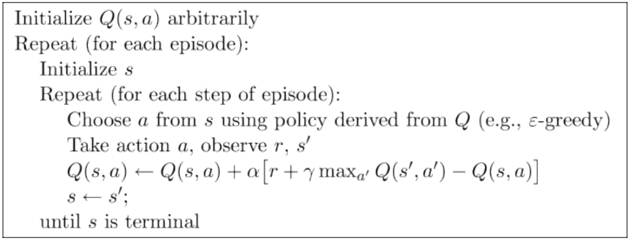
From this, we *reinforce* our estimates of applying the previous action in the previous state.

- We terminate when: (1) we run out of training time; (2) we think our policy has converged to the optimal policy (for each new episode we see no improvement); or (3) our policy is 'good enough' (when for each new episode we see only minimal improvement).

Q-Learning

Q-Learning is perhaps the simplest of reinforcement learning methods, and is based on how animals learn from their environment. The intuition is quite straightforward.

Maintain a Q-function that records $Q(s,a)$ for every state-action pair. At each step: (1) choose an action using a multi-armed bandit algorithm; (2) apply that action and receive the reward; and (3) update $Q(s,a)$ based on that reward. Repeat over a number of episodes until . . . when?



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Figure: Q-Learning Algorithm (from Sutton and Barto)

Updating the Q-function

Updating the Q-function (line 7) is where the learning happens:

$$Q(s,a) \leftarrow \underbrace{Q(s,a)}_{\text{old value}} + \overbrace{\alpha}^{\text{learning rate}} \cdot [\underbrace{r}_{\text{reward}} + \overbrace{\gamma}^{\text{discount factor}} \cdot \underbrace{\max_{a'} Q(s',a')}_{\text{estimate of optimal future value}} \overbrace{-Q(s,a)}^{\text{do not count extra } Q(s,a)}]$$

A higher learning rate α will weight more recent information higher than older information ($Q(s,a)$).

Note that we estimate the future value using $\max_{a'} Q(s',a')$, which means it *ignores* the action chosen by the policy, and instead updates based on the estimate of the best action for the updated state s'.

This is known as *off policy* learning – more on this later.

Q-functions using Q-Tables

Q-tables are the simplest way to maintain a Q-function. They are a table with an entry for every $Q(s,a)$. Thus, like value functions in value iteration, they do not scale to large state-spaces. (More on scaling in the next lecture).

Initially

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

After some training

State	**Action**			
	North	South	East	West
(0,0)	0.53	0.36	0.36	0.21
(0,1)	0.61	0.27	0.23	0.23
...				
(2,2)	0.79	0.72	0.90	0.72
(2,3)	0.90	0.78	0.99	0.81

Q-learning: Example

After some training

State	**Action**			
	North	South	East	West
(0,0)	0.53	0.36	0.36	0.21
(0,1)	0.61	0.27	0.23	0.23
. . .				
(2,2)	0.79	0.72	0.90	0.72
(2,3)	0.90	0.78	0.99	0.81

In state (2,2), the action ‘North’ is chosen and executed successfully, which would return to state (2,2) there is no cell above (2,2). Using the Q-table above, we would update the Q-value as follows:

$$\begin{aligned}
Q((2,2),N) &\leftarrow Q((2,2),N) + \alpha[r + \gamma \max_{a'} Q((2,2),a') - Q((2,2),N)] \\
&\leftarrow 0.79 + 0.1[0 + 0.9 \cdot Q((2,2),East) - Q((2,2),N)] \\
&\leftarrow 0.79 + 0.1[0 + 0.9 \cdot 0.90 - 0.79] \\
&\leftarrow 0.792
\end{aligned}$$

Using Q-functions

We iterate over as many episodes as possible, or until each episode hardly improves our Q-values. This gives us a (close to) optimal Q-function.

Once we have such a Q-function, we stop exploring and just exploit. We use *policy extraction*, which is exactly as we do for value iteration:

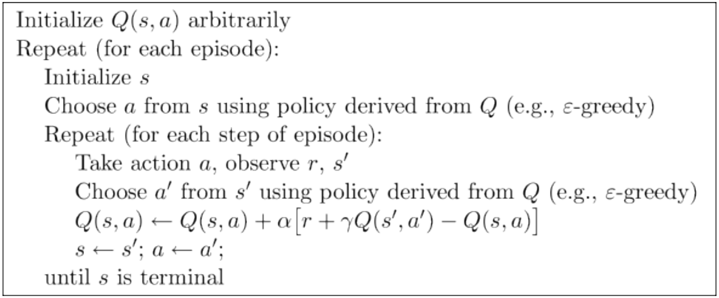
$$\pi(s) = \operatorname*{argmax}_{a \in A(s)} Q(s,a)$$

SARSA: On-Policy Reinforcement Learning

SARSA = State-action-reward-state-action

On-Policy: Instead of using $Q(s', a')$ for the best estimated future state during update, on-policy uses the actual next action to update:

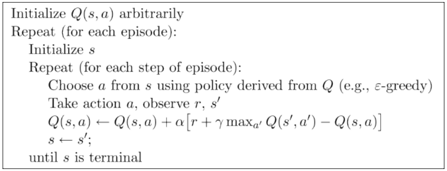
- On-policy learning estimates $Q^{\pi}(s, a)$ state action pairs, for the current behaviour policy π, whereas off-policy learning estimates the policy independent of the current behaviour.



```
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

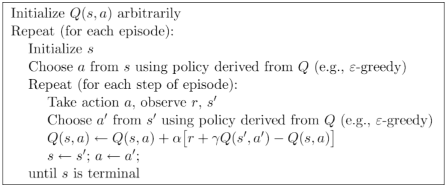
On-Policy: Uses the action chosen by the policy for the update!

Q-learning vs. SARSA



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Figure: Q-Learning Algorithm



```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma Q(s',a') - Q(s,a)\big]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

Figure: Sarsa Algorithm

Off-Policy: Ignores the action chosen by the policy, uses the best action $\mathrm{argmax}_{a'}\, Q(s',a')$ for the update!

On-Policy SARSA learns action values relative to the policy it follows, while Off-Policy Q-Learning does it relative to the greedy policy.

On-policy vs. off-policy: What is the difference here?

The difference is all in how the update happens in the loop body.

Q-learning: (1) selects an action a; (2) takes that actions and observes the reward & next state s'; and (3) updates *optimistically* by assuming the future reward is $\max_{a'} Q(s', a')$ – that is, it assumes that future behaviour will be optimal (according to its policy).

SARSA: (1) selects action a' for the *next* loop iteration; (2) in the next iteration, takes that action and observes the reward & next state s'; (3) only then chooses a' for the next iteration; and (4) updates using the estimate for the actual next action chosen – which may not be the greediest one (*e.g. it could be selected so that it can explore*).

SARSA: Example

After some training

State	**Action**			
	North	South	East	West
(0,0)	0.53	0.36	0.36	0.21
(0,1)	0.61	0.27	0.23	0.23
...				
(2,2)	0.79	0.72	0.90	0.72
(2,3)	0.90	0.78	0.99	0.81

In state (2,2), the action ‘North’ is chosen and executed successfully, which would return to state (2,2) there is no cell above (2,2). The next selected action is West. Using the Q-table above, we would update the Q-value using SARSA as follows:

$$\begin{aligned}
Q((2,2),N) &\leftarrow Q((2,2),N) + \alpha[r + \gamma Q((2,2),W) - Q((2,2),N)] \\
&\leftarrow 0.79 + 0.1[0 + 0.9 \cdot Q((2,2),W) - Q((2,2),N)] \\
&\leftarrow 0.79 + 0.1[0 + 0.9 \cdot 0.72 - 0.79] \\
&\leftarrow 0.7758
\end{aligned}$$

On-policy vs. off-policy: Who cares??

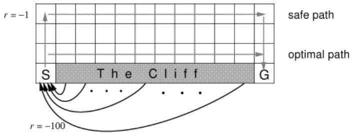
So what difference does this really make? There are two main differences:

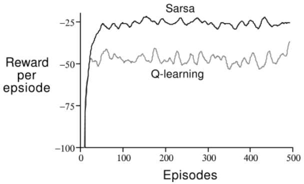
- Q-learning will converge to the optimal policy irrelevant of the policy followed, because it is *off-policy*: it uses the greedy reward estimate in its update rather than following the policy such as ϵ-greedy). Using a random policy, Q-learning will still converge to the optimal policy, but SARSA will not (necessarily).
- Q-learning learns an optimal policy, but this can be 'unsafe' or risky *during training*.

SARSA vs. Q-learning: Example

Consider the grid below. S is the start and state G receives a reward of 100. Falling off the cliff receives a reward of -100. Going to the top row receives a -1 reward. Actions are deterministic, but they are not known before learning.

Q-learning leads the optimal path along (along the edge of the cliff), but will fall off sometimes due to the ϵ-greedy action selection. SARSA learns the safe path because it is on-policy, and considers the action select method when learning. Here is a graph showing the reward per trial for both Sarsa and Q-Learning:





Rewards during training

SARSA receives a higher average reward *per trial* than Q-Learning, because it falls off the cliff less in later episodes. However, Q-learning learns the *optimal* policy.

Demo of the cliff example: `https://studywolf.wordpress.com/2013/07/01/reinforcement-learning-sarsa-vs-q-learning/`

On-policy vs. off policy: Why do we have both?

Imagine a reinforcement learning agent that manages resources for a cloud-based platform and we have no prior data to inform a policy.

- On-policy learning is more appropriate when we want to optimise the behaviour of an agent who learns *while operating in its environment*.
  We would need to operate our cloud platform to get data. As such, if the average reward *per trial* is better using on-policy, this would give us better overall outcomes than off-policy learning, because the 'trials' are not practice – they actually influence how much money we make.
- Off-policy learning is more appropriate when we have the luxury of training our agent offline before it is put into operation.

  If we could run our reinforcement learning algorithm in a simulated environment before deploying (and we had reason to believe that simulated environment was accurate), off-policy learning would be better because its optimal policy could be followed.

In short: use on-policy reinforcement learning for online learning, and off-policy learning for offline learning.

Example: if we used reinforcement learning for traffic light optimisation, we would use on-policy reinforcement learning, because we could not train it before hand given that the policy influences the behaviour of drivers.

Q-learning Examples in Action

In action: solving the cliff example using Q-learning with ϵ-greedy:

`https://www.youtube.com/watch?v=ppALjH0kYPE`

The source code for this:

`https://github.com/alecKarfonta/Gridworld`

Worked example: a complete worked example of using Q-learning to calculate the optimal path:

`http://www.mnemstudio.org/path-finding-q-learning-tutorial.htm`

Summary

If we know the MDP:

- **Offline**: Value Iteration, Policy Iteration,
- **Online**: Monte Carlo Search Tree and friends.

If we do *not* know MDP:

- **Offline**: Reinforcement Learning
- **Online**: Monte Carlo Tree Search and friends.

Once you've got your pacman Q-learning working in python (optional assessment and bonus exercise in the workshop), you can test it on all the environments on OpenAI, a Toolkit for developing and testing reinforcement learning algorithms:

`https://gym.openai.com/`

Applications of Reinforcement Learning

- Checkers (Samuel, 1959)
  first use of RL in an interesting real game
- (Inverted) Helicopter Flight (Ng et al. 2004)
  better than any human
- Computer Go (AlphaGo 2016)
  AlphaGo beats Go world champion Lee Sedol 4:1
- Atari 2600 Games (DQN & Blob-PROST 2015)
  human-level performance on half of 50+ games
- Robocup Soccer Teams (Stone & Veloso, Reidmiller et al.)
  World's best player of simulated soccer, 1999; Runner-up 2000
- Inventory Management (Van Roy, Bertsekas, Lee & Tsitsiklis)
  10-15% improvement over industry standard methods
- Dynamic Channel Assignment (Singh & Bertsekas, Nie & Haykin)
  World's best assigner of radio channels to mobile telephone calls
- Elevator Control (Crites & Barto)
  (Probably) world's best down-peak elevator controller
- Many Robots
  navigation, bi-pedal walking, grasping, switching between skills, . . .
- TD-Gammon and Jellyfish (Tesauro, Dahl)
  World's best backgammon player. Grandmaster level

Reading

- Chapter 6: Temporal-Difference Learning of *Reinforcement Learning: An Introduction (Second Edition)*, 2020 [*Sutton and Barto*]

  Available at:

  `http://www.incompleteideas.net/book/RLbook2020.pdf`

  Content: Great entry level book to Reinforcement Level written by the founders of the field.

- *Slides about Approximate Q-learning for PacMan*

  Available at:

  `https://www.cs.swarthmore.edu/~bryce/cs63/s16/slides/3-25_approximate_Q-learning.pdf`

  Content: Great technique if you want to use reinforcement learning for the competition!

- *Deep Q-learning for Atari*

  Available at:

  `http://www.davidqiu.com:8888/research/nature14236.pdf`

  Content: Convolutional Neural Networks (NN) to estimate $Q(s, a)$. The input for the NN is the state, and the output is the esimated reward for each action.