Reinforcement Learning

CS 480
Intro to Artificial Intelligence

MDP-based agent sketch

```
def MDP_agent(T,R,gamma,states,actions):
          Utility = Value_Iteration(T,R,gamma,states,actions)
          pi = best_policy(Utility)
          while s_cur.isTerminal()==False:
                    s_cur = sense_state()
                    a = pi[s_cur]
                    do_action(a)
```

The techniques we've discussed so far are **offline**: Policy is computed from transition function, reward function before the agent takes any actions

Requires transition function ($O(|S|^2|A|)$ memory), reward function are **known**

How can we remove these requirements?

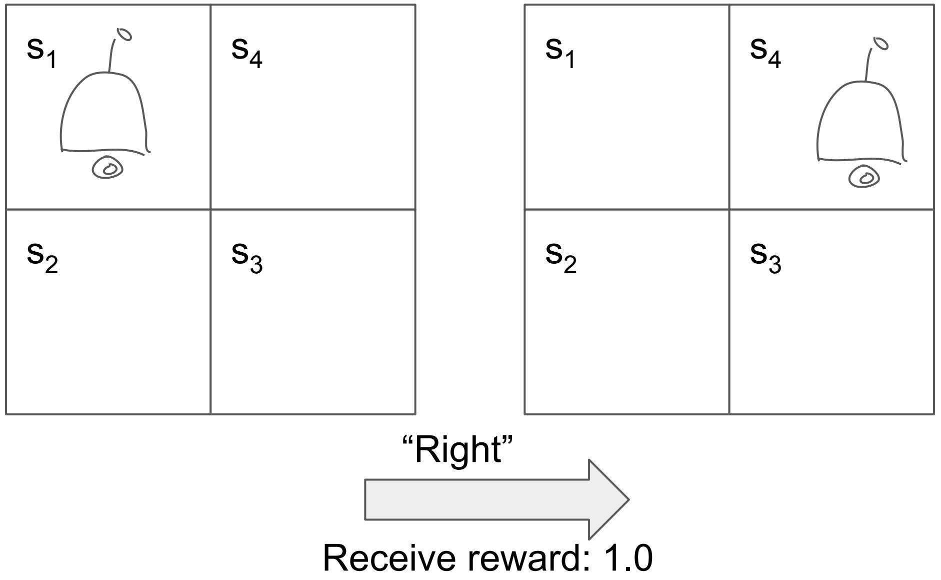
Reinforcement Learning agent

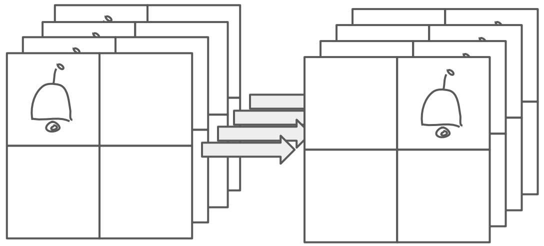
Reinforcement Learning

- Use the same utility framework
- Don’t require transition or reward functions
- Learn policy by **interacting** with the environment

Surprisingly, this works quite well, and isn’t any more complicated to implement!

Key idea is that interactions with the environment are **samples** from the transition and reward functions





Experience Tuple
(s, a, s’, r)

Two different ways of expressing Utility

Value Iteration

$$U^{\pi}(s)$$

Long-term expected discounted reward of being in state **s** and following policy π

$$\pi^*(s) = \arg\max_a \sum_{s'} p(s' \mid s, a) U^{\pi^*}(s')$$

Optimal policy in state s

Reinforcement Learning

$$Q^{\pi}(s, a)$$

Long-term expected discounted reward of being in state **s**, taking action **a**, and *then* following policy π

$$\pi^*(s) = \arg\max_a Q^{\pi^*}(s, a)$$

Optimal policy in state s

Relating $Q^{\pi^*}(s,a)$ and $U^{\pi^*}(s)$

$U^{\pi^*}(s)$ and $Q^{\pi^*}(s,a)$ are just two different ways of expressing the same utility. $U^{\pi^*}(s)$ was useful for Value/Policy Iteration, but for RL methods, $Q^{\pi^*}(s,a)$ will be easier to work with.

We can relate the two ways of writing utility in the following way

$$U^{\pi^*}(s) = \max_a Q(s,a)$$

$$Q^{\pi}(s,a) = R(s) + \gamma \sum p(s' \mid a, s) U^{\pi}(s')$$

What if R is not “per state”?

Important for project!

By the same math (just “off by one” in the sums), we can have our reward function be in terms of “taking an action in a state”

$$Q^{\pi}(s,a) = \sum_{s'} p(s' \mid s,a) \left[R(s,a,s') + \gamma U^{\pi}(s')\right]$$

$$U^{\pi^*}(s) = \max_{a} \sum_{s'} p(s' \mid s,a) \left[R(s,a,s') + \gamma U^{\pi^*}(s')\right]$$

Model-based and model-free RL

Model Based RL

Interacting with the environment produces samples of the transition function and the reward function.

Use these samples to build **explicit** models of these functions, then use VI/PI to solve the MDP

p(s'|s,a) = (# times doing "a" in s led to s')/ (# times did "a" in s)

R(s) = average reward received in s

Model Free RL

Interacting with the environment produces samples of the transition function and the reward function.

We **only** care about the transition function and reward function because they let us compute utility and from that policy

Directly compute optimal policy from samples

Method we will focus on: **Q-learning**

RL-based agent sketch

```
def RL_agent(gamma,actions,init_Q):
          cur_state = sense_state()
          cur_Q = init_Q
          while cur_state.isTerminal() == False:
                    a = argmax(cur_Q(cur_state))
                    new_state, reward = do_action(a)
                    experience_sample = (cur_state,a,new_state,reward)
                    cur_Q = update(cur_Q,experience_sample)
                    cur_state = new_state
```

An RL agent starts with some initial estimate of the utility, then iteratively improves this estimate through **experience samples** gathered by interacting with the environment.

How do we update Q(s,a) using these samples?

Updating Q from samples

Given an experience sample (s,a,s’,r)

$$Q(s,a) \leftarrow \underline{Q(s,a)} + \alpha(\underline{r + \gamma \max_{a'} Q(s',a')} - \underline{Q(s,a)})$$

Equivalently, by rearranging

$$Q(s,a) \leftarrow (1-\alpha)\underline{Q(s,a)} + \alpha(\underline{r + \gamma \max_{a'} Q(s',a')})$$

So, we update Q by blending the old Q and an improved estimate given the sample!

Where did these equations come from? Bellman again!

Important edge case for terminal states

The usual formulation of MDPs/RL is with no “terminal-states”, the agent goes on taking actions forever.

We can modify our update equations in the case where certain states stop any future actions by the agent (goal states, or failure states)

$$U(s_T) \leftarrow R(s_T)$$

$$Q(s_T, a) \leftarrow Q(s_T, a) + \alpha(r - Q(s_T, a))$$

The idea is that if s_T is terminal, the reward for states “beyond” s_T is set to zero

Picking the right action

Do we always want to pick the action that maximizes our current utility estimate?

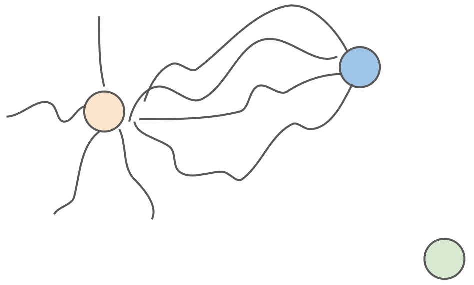
Which actions we pick influence the sequence of states we visit, which influences which utilities we update

We need a way to trade off “exploration” and “exploitation”

Picking actions greedily

```
def RL_agent(gamma,actions,init_Q):
          cur_state = sense_state()
          cur_Q = init_Q
          while cur_state.isTerminal() == False:
                    a = argmax(cur_Q(s))
                    new_state, reward = do_action(a)
                    experience_sample = (cur_state,a,new_state,reward)
                    cur_Q = update(cur_Q,experience_sample)
                    cur_state = new_state
```

Start
Medium reward
High reward

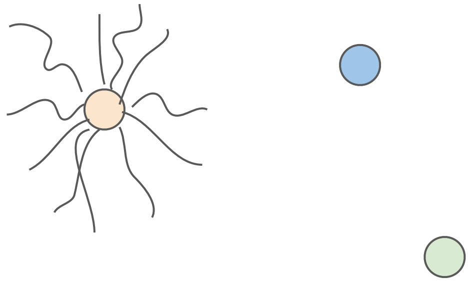


Might not find the **best** goal state. Does not explore environment

Picking actions randomly

```
def RL_agent(gamma,actions,init_Q):
          cur_state = sense_state()
          cur_Q = init_Q
          while cur_state.isTerminal() == False:
                    a = rand(actions)
                    new_state, reward = do_action(a)
                    experience_sample = (cur_state,a,new_state,reward)
                    cur_Q = update(cur_Q,experience_sample)
                    cur_state = new_state
```

Start
Medium reward
High reward

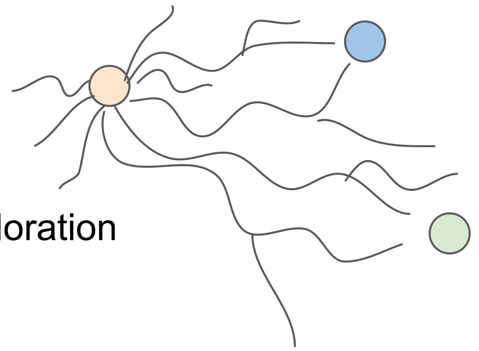


Might never see **any** goal states! Does not improve performance

Picking actions ε-greedy

```
def RL_agent(gamma,actions,init_Q,epsilon):
          cur_state = sense_state()
          cur_Q = init_Q
          while cur_state.isTerminal() == False:
                    a = argmax(cur_Q(s)) if rand()>epsilon else rand(actions)
                    new_state, reward = do_action(a)
                    experience_sample = (cur_state,a,new_state,reward)
                    cur_Q = update(cur_Q,experience_sample)
                    cur_state = new_state
```

Start
Medium reward
High reward



Balances exploring to find new better states and exploiting known good states

GLIE: Greedy in the Limit of Infinite Exploration
Guaranteed to converge to optimal π

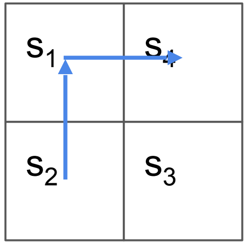
Example - stepping through Q updates (1)

α=0.5, γ=0.5

$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r+\gamma \max_{a'} Q(s',a'))$

Experience tuples

1. (s_2, Up, s_1, -0.04)
2. (s_1, Right, s_4, 1.0)



	Up	Down	Left	Right
s_1	0	0	0	**0.5**
s_2	**-0.02**	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0

Updates

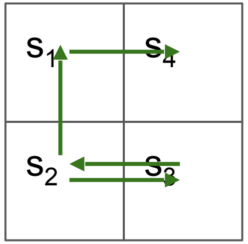
1. $Q(s_2,\text{Up}) = 0.5*0.0 + 0.5*(-0.04+0.5*0.0)$
   $= -0.02$
2. $Q(s_1,\text{Right}) = 0.5*0.0 + 0.5*(1.0+0.5*0.0)$
   $= 0.5$

Example - stepping through Q updates (2)

$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r+\gamma \max_{a'} Q(s',a'))$

Experience tuples

3. (s_2, Right, s_3, -0.04)
4. (s_3, Up, s_2, -0.04)
5. (s_2, Up, s_1, -0.04)
6. (s_1, Right, s_4, 1.0)



	Up	Down	Left	Right
s_1	0	0	0	~~0.5~~ **0.75**
s_2	~~-0.02~~ **0.085**	0	0	**-0.02**
s_3	**-0.02**	0	0	0
s_4	0	0	0	0

Updates

3. Q(s_2,Right) = 0.5*0.0 + 0.5*(-0.04+0.5*0.0) = -0.02
4. Q(s_3,Up) = 0.5*0.0 + 0.5*(-0.04+0.5*0.0) = -0.02
5. Q(s_2,Up) = 0.5*-0.02 +0.5*(-0.04+0.5*0.5) =-0.01+0.5*0.21 = 0.085
6. Q(s_1,Right) = 0.5*0.5+0.5*(1.0+0.5*0.0) = 0.25+0.5 = 0.75

Summary and preview

Wrapping up

- Reinforcement Learning takes the tools we used for solving offline MDPs and adapts them for use when the **transition** and **reward** functions are not known ahead of time
- Q-learning is a **model-free** RL algorithm that can learn the optimal policy **online**
- How the action is chosen has a big impact! (GLIE, ε-greedy)

Next time: adding even more uncertainty!