Lecture 8

Cezar Ionescu
06/07/2019

DEPARTMENT FOR CONTINUING EDUCATION
UNIVERSITY OF OXFORD

Administrative

- Homework from 29/06/2019 due now!
- Please complete and hand in the declarations of authorship.

Questions?

Solution to homework from lecture 7

$$y_1(v_{11}, v_{21}) = f(x_1 * v_{11} + x_2 * v_{21}) = -2$$

$$y_2(v_{11}, v_{21}) = f(x_1 * v_{12} + x_2 * v_{22}) = 2$$

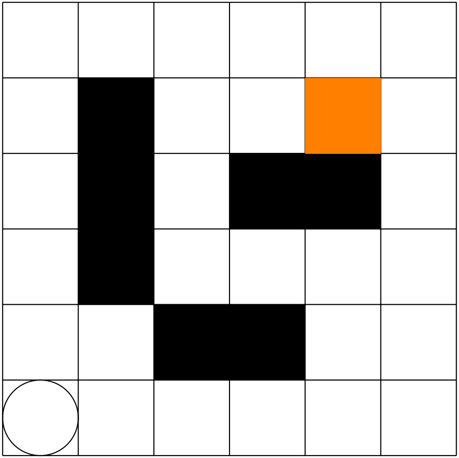
$$o(w_1, w_2, y_1, y_2) = f(y_1 * w_1 + y_2 * w_2) = 2$$

$$E\ (w_1, w_2, v_{11}, \ldots) = (t - o)^2$$

$$\begin{aligned}
\partial E / \partial v_{11} &= E'(o) * \partial o / \partial v_{11} \\
&= 2*(t - o)*(-1) * \partial o / \partial v_{11} \\
&= 2*(-2)*(-1)*f'(z)*\partial g / \partial v_{11} \\
&= 4*1*\partial (y_1 * w_1 + y_2 * w_2) / \partial v_{11} \\
&= 4 * \partial (y_1 * w_1 + y_2 * w_2) / \partial y_1 * \partial y_1 / \partial v_{11} \\
&= 4 * w_1 * f'(z) * \partial (x_1 * v_{11} + x_2 * v_{21}) / \partial v_{11} \\
&= 4 * x_1 = -4
\end{aligned}$$

Dynamic programming

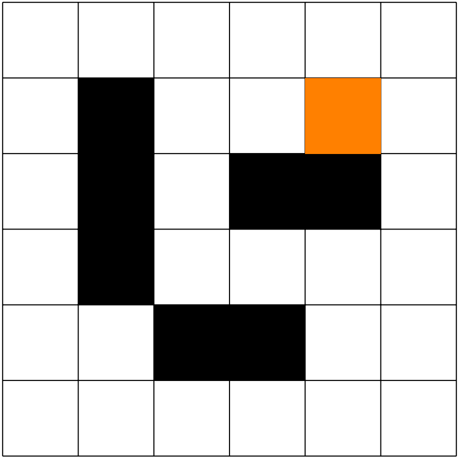
A maze



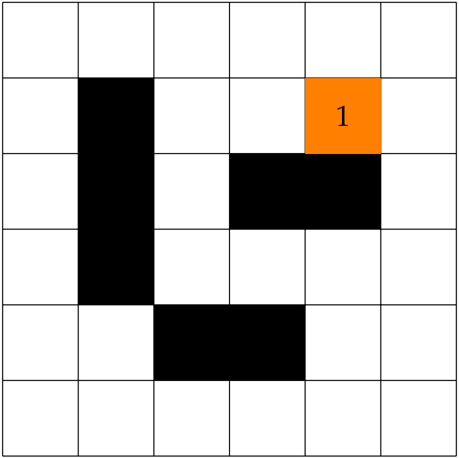
Value map

-4	-3	-2	-1	0	-1
-5		-1	0	1	0
-6		-2			-1
-7		-3	-4	-3	-2
-8	-9			-4	-3
-9	-8	-7	-6	-5	-4

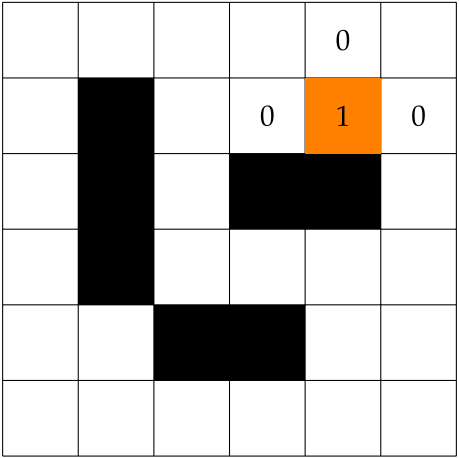
Dynamic programming



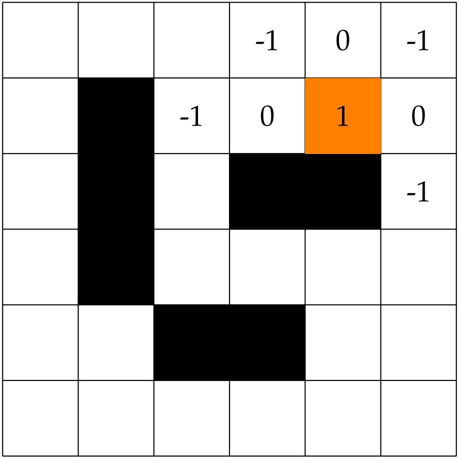
Dynamic programming



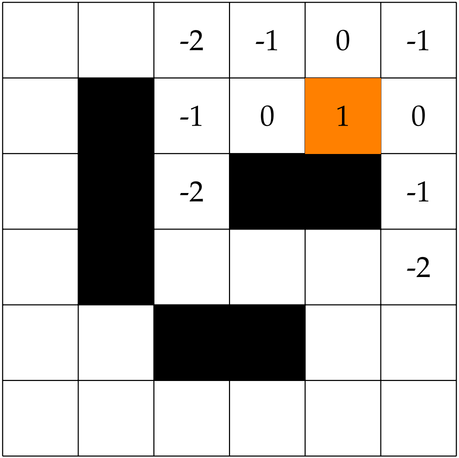
Dynamic programming



Dynamic programming



Dynamic programming



Dynamic programming

	-3	-2	-1	0	-1
		-1	0	1	0
		-2			-1
		-3		-3	-2
					-3

Dynamic programming

-4	-3	-2	-1	0	-1
		-1	0	1	0
		-2			-1
		-3	-4	-3	-2
				-4	-3
					-4

Dynamic programming

-4	-3	-2	-1	0	-1
-5		-1	0	1	0
		-2			-1
		-3	-4	-3	-2
				-4	-3
				-5	-4

Dynamic programming

-4	-3	-2	-1	0	-1
-5		-1	0	1	0
-6		-2			-1
		-3	-4	-3	-2
				-4	-3
			-6	-5	-4

Dynamic programming

-4	-3	-2	-1	0	-1
-5		-1	0	1	0
-6		-2			-1
-7		-3	-4	-3	-2
				-4	-3
		-7	-6	-5	-4

Dynamic programming

-4	-3	-2	-1	0	-1
-5		-1	0	1	0
-6		-2			-1
-7		-3	-4	-3	-2
-8				-4	-3
	-8	-7	-6	-5	-4

Dynamic programming

-4	-3	-2	-1	0	-1
-5		-1	0	1	0
-6		-2			-1
-7		-3	-4	-3	-2
-8	-9			-4	-3
-9	-8	-7	-6	-5	-4

Optimal policy

If we have the optimal value map, defining the optimal policy is easy:

> *Choose the action that results in the greatest sum of reward now and value in the next state.*

Choose `pol(s)` that maximises

```
Reward(s, a) + Value(sys(s, a))
```

where `sys : (State, Action) ~> State` is the function that tells us what happens depending on the current state and chosen action.

Review

Ingredients:

```
sys : (State, Action) -> State
rew : (State, Action) -> ℝ
pol : State -> Action
s₀ ∈ State -- given
```

Problem: Find `pol` that maximises Σ_0^n `rew(sᵢ, pol(sᵢ))`, where

$s_{i+1} = sys(s_i, a_i)$

Value function

```
Val : ℕ -> State -> ℝ
```

$\texttt{Val}_i(s) = \max_{pol} \Sigma_i^n \; \texttt{rew}(s_i, \texttt{pol}(s_i))$

Note:

$\texttt{Val}_0(s_0)$ is the maximal value for the entire problem.

Bellman equation

If `pol(s) = a`^{opt} the optimal action in state `s`, then

$\texttt{Val}_i(\texttt{s}) = \texttt{Rew}(\texttt{s}, \texttt{a}^{opt}) + \texttt{Val}_{i+1}(\texttt{sys}(\texttt{s}, \texttt{a}^{opt}))$

Therefore

$\texttt{Val}_i(\texttt{s}) = \max_a (\texttt{Rew}(\texttt{s}, \texttt{a}) + \texttt{Val}_{i+1}(\texttt{sys}(\texttt{s}, \texttt{a})))$

This is called *Bellman's equation*.

Direct adaptive optimal control

Model of the problem

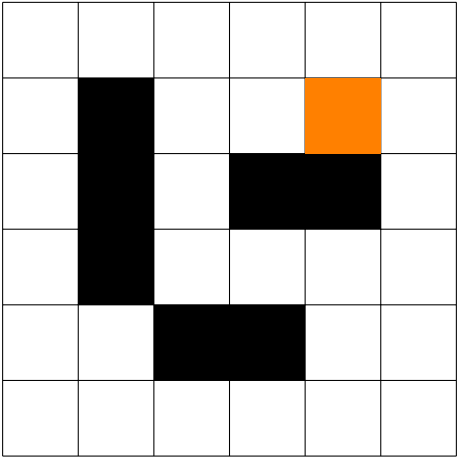
If we do not know the exact map of the maze, or the location of the goal, we cannot apply dynamic programming. We need an *adaptive method*. Alternatives:

- explore the maze and make a map of it, find the goal, and then apply dynamic programming (*indirect* method);
- learn the optimal value map directly! (*direct* method).

Reinforcement learning is *direct adaptive optimal control*.

Value iteration

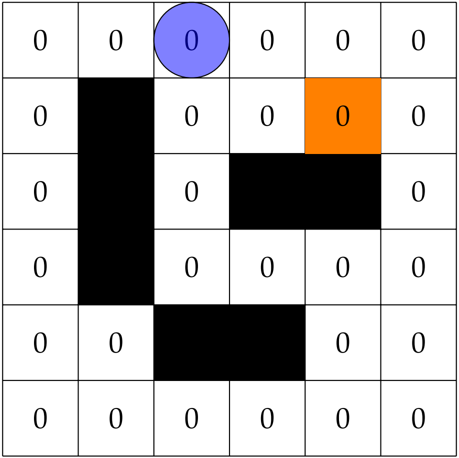
Start with a randomly generated value map

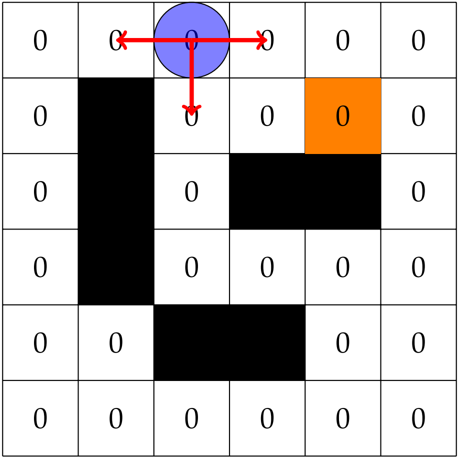


Start with a randomly generated value map

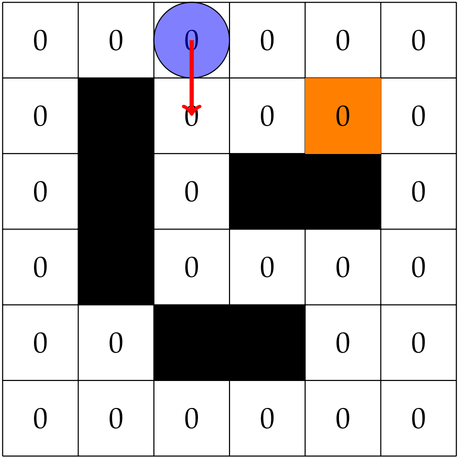
0	0	0	0	0	0
0		0	0	0	0
0		0			0
0		0	0	0	0
0	0			0	0
0	0	0	0	0	0

Alternatives

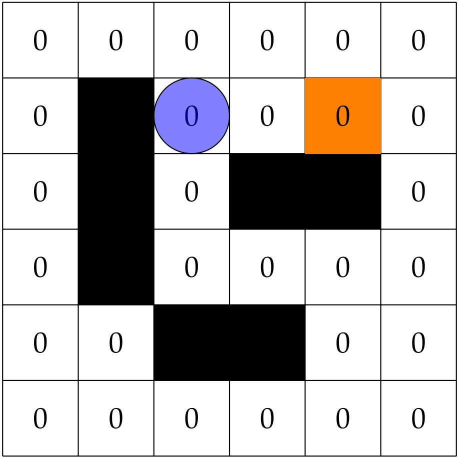



0 0 0 0 0 0
0 0 0 0 0
0 0 0
0 0 0 0 0
0 0 0 0
0 0 0 0 0 0

Choose an action



Make a move



Changing the current value function

If we had the optimal value function, then

```
Val((3,6)) = Rew((3,6), ↓) + Val((3,5))
```

But we have

```
Val((3, 6)) = 0,  Rew((3,6), ↓) = -1, Val((3,5)) = 0
```

Update rule

Idea from *supervised learning*:

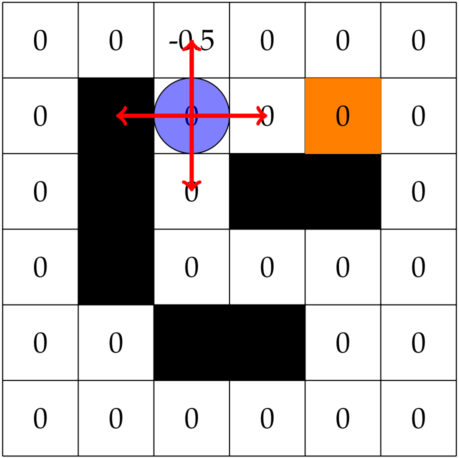
```
Val((3,6)) <- Val((3,6)) +
              δ * (Rew((3,6), ↓) + Val((3,5)) - Val((3,6)))
```

Therefore (say `δ = 0.5`)

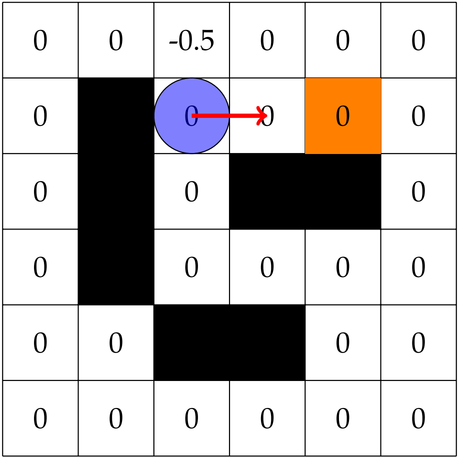
```
Val(3,6) <- 0 + 0.5 * (-1 + 0 - 0) = -0.5
```

The new situation

0	0	-0.5	0	0	0
0		0	0	0	0
0		0			0
0		0	0	0	0
0	0			0	0
0	0	0	0	0	0


0 0 -0.5 0 0 0
0 0 0 0 0
0 0 0
0 0 0 0 0
0 0 0 0
0 0 0 0 0 0

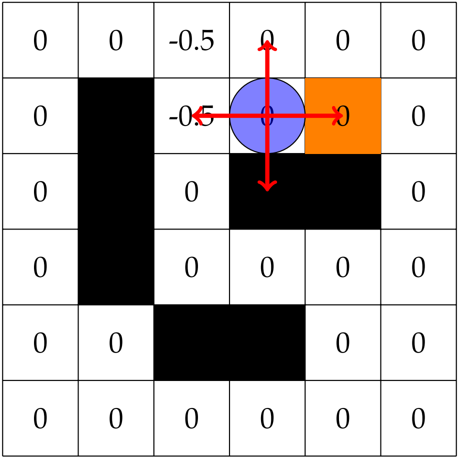
Pick a new direction



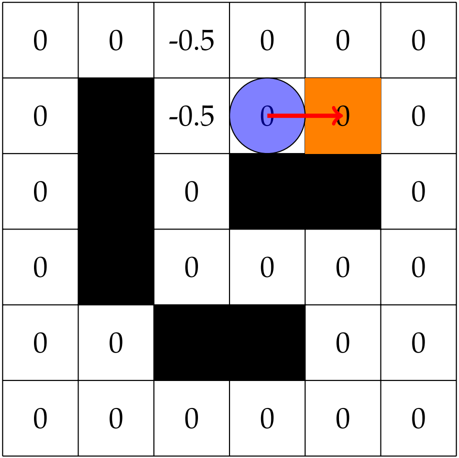
Move and update

0	0	-0.5	0	0	0
0		-0.5	0	0	0
0		0			0
0		0	0	0	0
0	0			0	0
0	0	0	0	0	0

And so on…



A good choice



A good choice

0	0	-0.5	0	0	0
0		-0.5	0.5	0	0
0		0			0
0		0	0	0	0
0	0			0	0
0	0	0	0	0	0

The game goes on

0	0	-0.5	0	0	0
0		-0.5	0.5	0	0
0		0			0
0		0	0	0	0
0	0			0	0
0	0	0	0	0	0

Discount factors

In this setting, we maximise the sum of rewards. Problem: if the number of steps is large, the values become difficult to estimate (huge absolute value). Moreover, in many situations we need to operate with *infinite horizons*.

To deal with this problem, we introduce a *discount factor*: `0 < β < 1`

Instead of maximising

```
Rew(s₀, pol(s₀)) + Rew(s₁, pol(s₁)) + Rew(s₂, pol(s₂)) + ...
```

we maximise

```
Rew(s₀, pol(s₀)) + β*Rew(s₁, pol(s₁)) + β²*Rew(s₂, pol(s₂)) + ...
```

where s_{t+1} = `sys(sₜ, pol(sₜ))`

Value function

If we have an infinite horizon, the value function is *stationary*:

$V_i(s) = \max_{pol} \Sigma_i^{\infty} rew(s_t, pol(s_t))$ where

$s_i = s$

$s_{t+1} = sys(s_t, pol(s_t))$

This is independent of `i`!

Bellman's equation

We choose `pol(s)` that maximises

```
Rew(s, a) + β * Val(sys(s, a))
```

Therefore

```
Val(s) = maxₐ (Rew(s, a) + β * Val(sys(s, a)))
```

Value iteration and function approximation

```
Val(s) = maxₐ (Rew(s, a) + β * Val(sys(s, a)))
```

Bellman's equation allows us to iteratively approximate the optimal value function.

```
Val(s) <- Val(s) +
          δ * (maxₐ (Rew(s, a) + β * Val(sys(s, a)))) - Val(s))
```

The optimal value function `Val` is unique (but there might be many policy functions that realise it).

Value iteration

```
Val(s) <- Val(s) +
          δ * (maxₐ (Rew(s, a) + β * Val(sys(s, a)))) - Val(s))
```

Problem: we require knowledge of the reward function.

We could use supervised learning to approximate the reward function.

But now we have two function approximations and an optimisation at every step!

Q-learning

Watkins' idea

The reward function only enters the picture when combined with `Val`, e.g.:

```
Val(s) = maxₐ (Rew(s, a) + β * Val(sys(s, a)))
```

Chris Watkins (1989): maybe it's simpler to learn the combination of `Rew` and `Val`!

```
Q(s, a) = Rew(s, a) + β * Val(sys(s, a))
```

Q-learning and `Val`

```
Q(s, a) = Rew(s, a) + β * Val(sys(s, a))
```

We have `Val(s) = maxₐ Q(s, a)` therefore, knowing `Q` is sufficient for determining `Val`.

Q-learning and the optimal policy

```
Q(s, a) = Rew(s, a) + β * Val(sys(s, a))
```

The optimal policy is the one that maximises

```
Rew(s, a) + β * Val(sys(s, a))
```

Therefore, the optimal action in a state `s` is the one that maximises the `Q` function:

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

Recursive equation for Q-learning

```
Q(s, a) = Rew(s, a) + β * Val(sys(s, a))
```

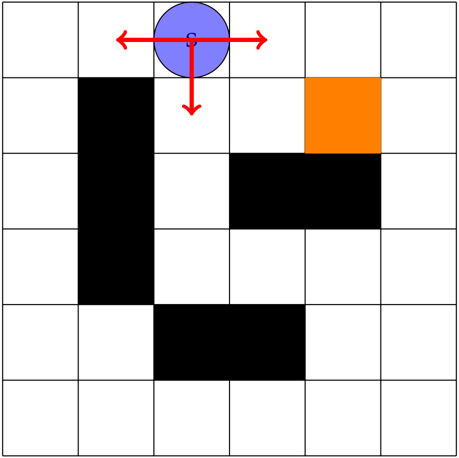
```
Val(s) = maxₐ Q(s, a)
```

Therefore

```
Q(s, a) = maxₐ (Rew(s, a) + β * maxₓ Q(sys(s, a), x))
```

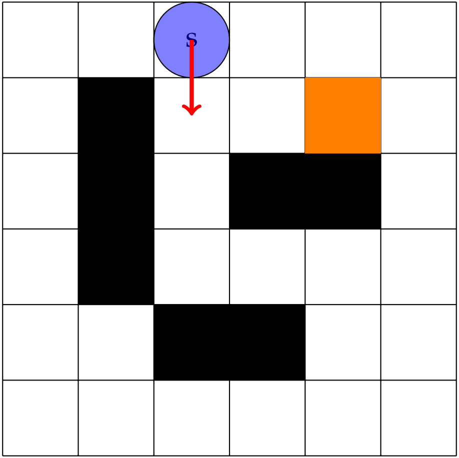
We have a recursive equation for the `Q` function.

Q-learning in action



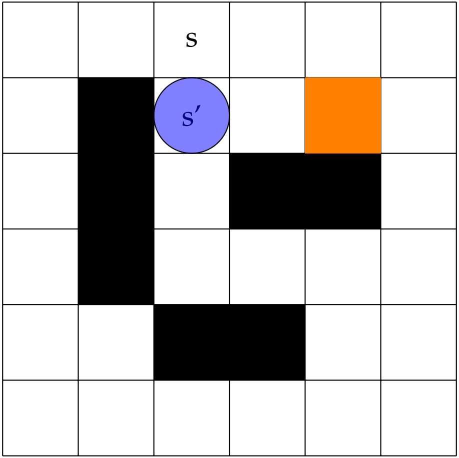
$$Q(s, \leftarrow) = -10$$
$$Q(s, \downarrow) = -8$$
$$Q(s, \rightarrow) = -9$$

Q-learning in action



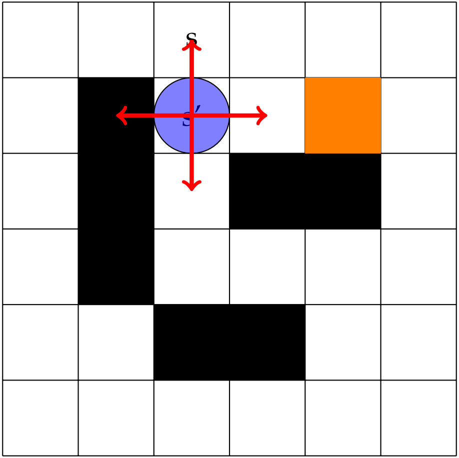
$$Q(s, \downarrow) = -8$$

Q-learning in action



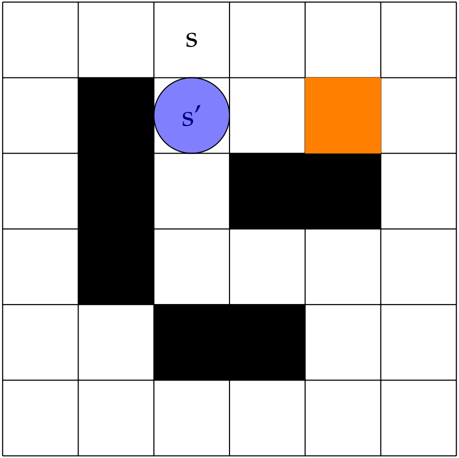
$$Q(s, \downarrow) = -8$$
$$Rew(s, \downarrow) = -1$$

Q-learning in action



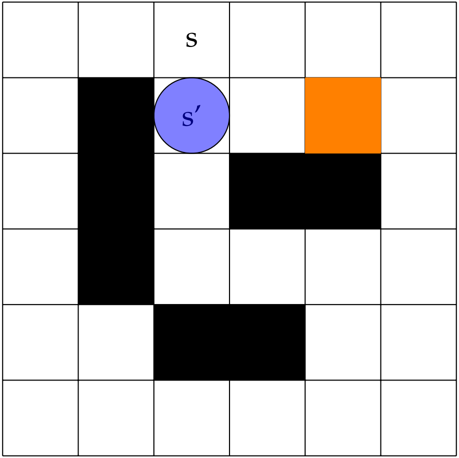
$$Q(s, \downarrow) = -8$$
$$Rew(s, \downarrow) = -1$$
$$Q(s', \leftarrow) = -\infty$$
$$Q(s', \downarrow) = -8$$
$$Q(s', \rightarrow) = -5$$
$$Q(s', \uparrow) = -10$$

Q-learning in action



$Q(s, \downarrow) = -8$
$Rew(s, \downarrow) = -7$
$Val(s') =$
$max(-\infty, -8, -5, -10)$
$Rew(s, \downarrow) + \beta Val(s') =$
-6

Q-learning in action



$Q(s, \downarrow) = -8$
$Rew(s, \downarrow) + \beta Val(s') =$
-6
$Q(s, \downarrow) \leftarrow$
$-8 + \delta(-6 - (-8)) = -6$

Stochastic systems

We have assumed a deterministic system, but in most cases this is unrealistic. E.g., the maze environment could take us (with a small probability) in a different direction than we chose. In that case, we need to maximise the *expected value* of the sum of rewards. Q-learning can be applied to these cases almost without any change!

Conclusions

Reinforcement learning applications: games, RoboCup Soccer, self-driving cars, etc. See also discussion in Week 1 (robot-catching arm, self-driving car, barriers on emissions…) Reinforcement learning seems to make the most out of very little: is it the road to Artificial General Intelligence? Chris Watkins (http://www.cs.rhul.ac.uk/~chrisw/):

> *I have long felt the standard model of RL is deceptively attractive, but limited.*