Unsupervised learning

REINFORCEMENT LEARNING

So far ….

- Supervised machine learning: given a set of annotated instances and a set of categories, find a model to automatically categorize unseen instances
- Unsupervided learning: no examples are available
- Three types of unsupervised learning algorithms:
  - **Rule learning –** find regularities in data and learn associations (past lesson)
  - **clustering-** Given a set of instances described by feature vectors, group them in clusters such as intra-cluster similarity is maximized and inter-cluster dissimilarity is maximized (in other courses)
  - **Reinforcement learning** (today)

Reinforcement learning

- **Reinforcement learning:**
  - Agent receives no examples and starts with no model of the environment.
  - Agent gets feedback through rewards, or **reinforcement**.
- Note: it is common to talk about «agents» rather than «learners» since the **output is a sequence of actions**, rather than a classification

Examples of applications

- Control physical systems: walk, drive, swim, ...
- Interact with users: engage customers, personalise channel, optimise user experience, ...
- Solve logistical problems: scheduling, bandwidth allocation,elevator control, power optimisation, ..
- Play games: chess, checkers, Go, Atari games, ...
- Learn sequential algorithms: attention, memory, conditional computation, activations, ...

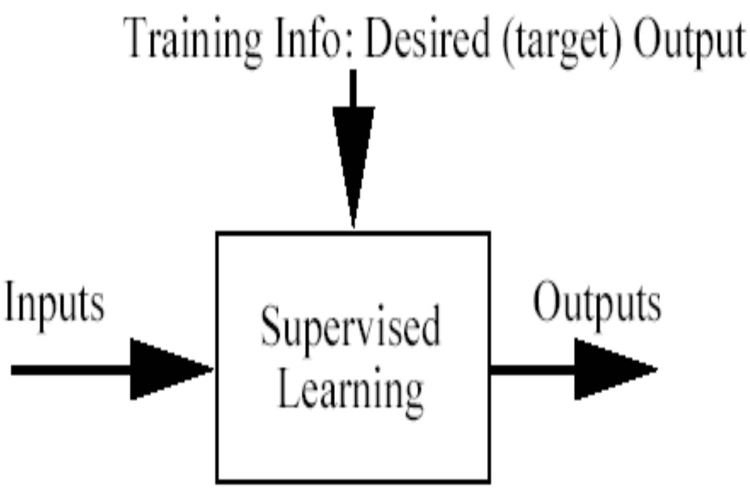
Learning by reinforcement

- Examples:
  - Learning to play Backgammon
  - Robot learning to move in an environment
- Characteristics:
  - No direct training examples – (possibly delayed) rewards instead, or penalty
  - Need for exploration of environment & exploitation
  - The environment might be stochastic and/or unknown
  - The actions of the learner **affect future rewards**

Reinforcement learning

- Unlike most machine learning, focus on a learning agent that acts in environment
- The loop
  - Agent percieves state of environment
  - Agent acts
  - Agent receives reward/punishment, state of environment changes
- The task: Learn to act so as to maximize rewards
  Learn a policy (mapping from states to actions)

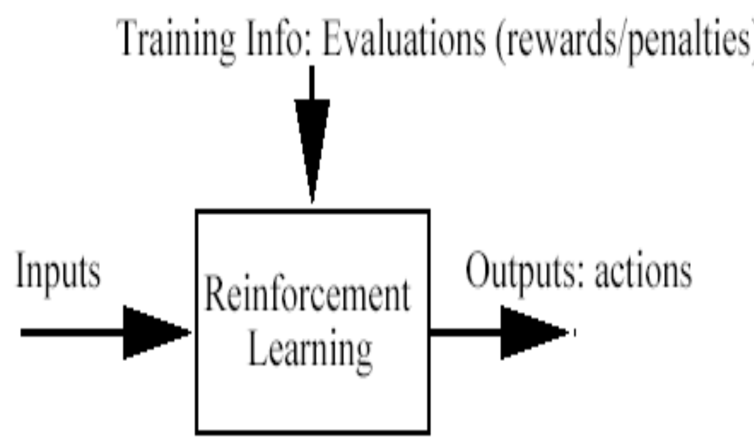
Supervised Learning



Error = (target output - actual output)

Input is an istance, output is a classification of the istance

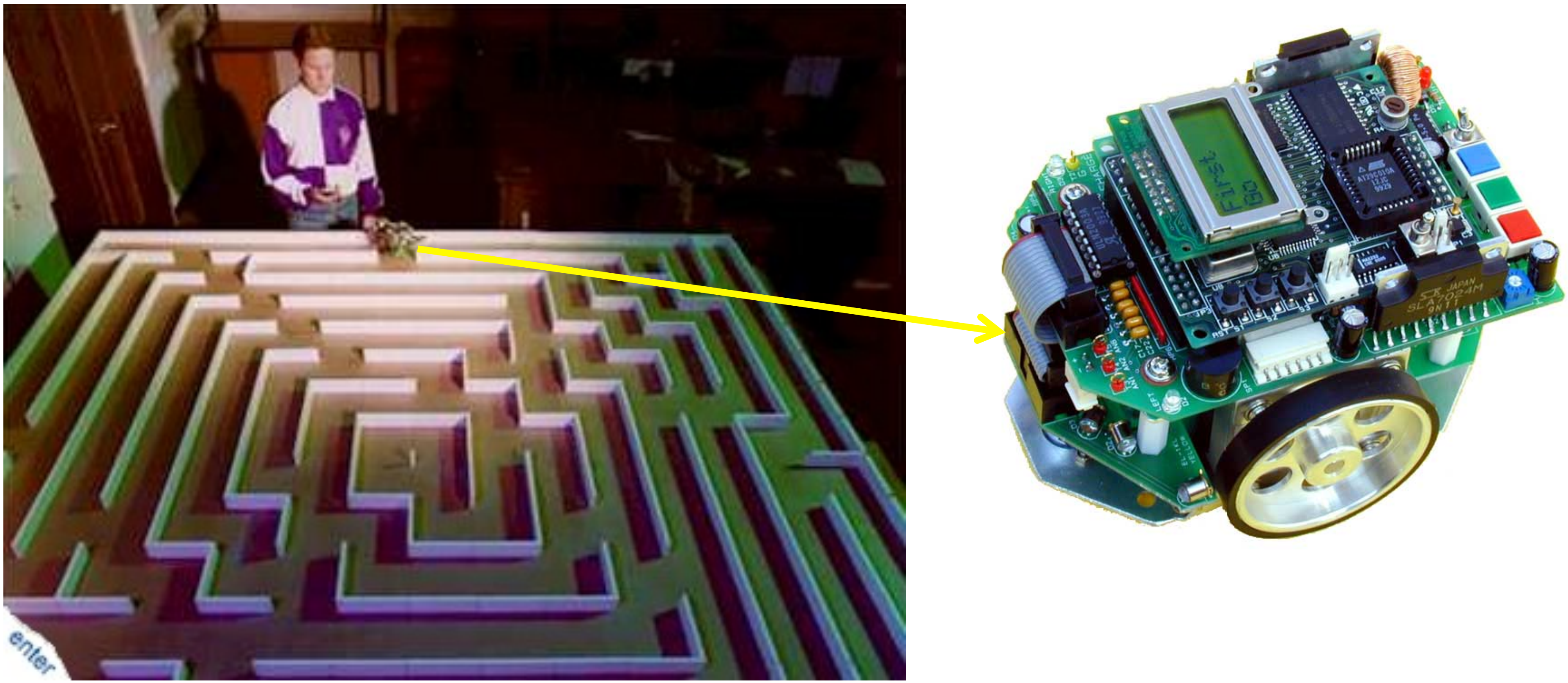
Reinforcement Learning



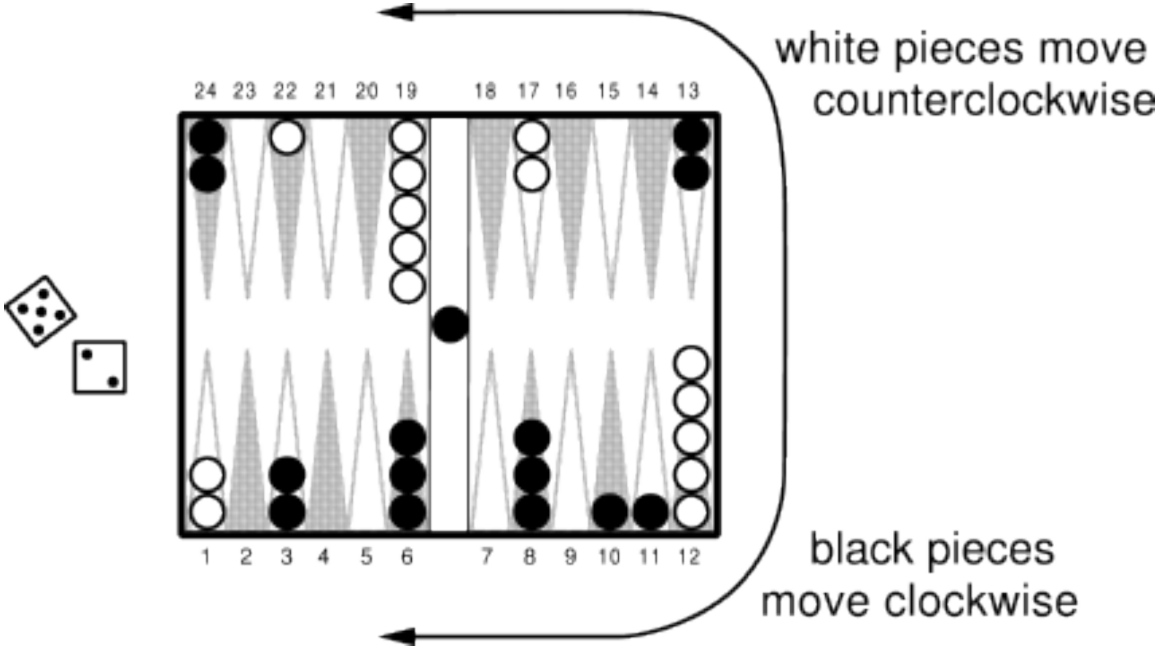
Objective: Get as much reward as possible

Input is some “goal”, output is a sequence of actions to meet the goal

Example: Robot moving in an environment (a maze)

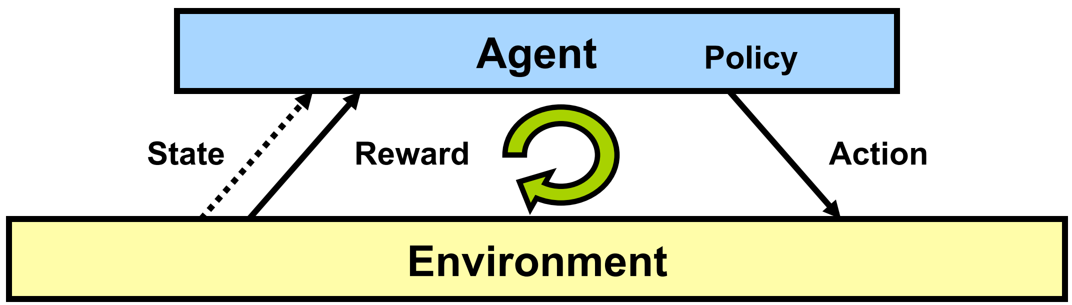


Example: playing a game (e.g. backgammon, chess)



Elements of RL

In RL, the "model" to be learned is a **policy** to meet "at best" some given goal (e.g. win a game)



$$s_0 \xrightarrow{a_0\,:\,r_0} s_1 \xrightarrow{a_1\,:\,r_1} s_2 \xrightarrow{a_2\,:\,r_2} \cdots$$

- **Transition model,** how actions A influence states S
- **Reward R**, immediate value of state-action transition
- **Policy** π S➔A, maps states to actions

Markov Decision Process (MDP)

- MDP is a formal model of the RL problem
- At each discrete time point
  - Agent observes state s_t in **S** and chooses *action* a_t *in **A** (according to some probability distribution)*
  - Receives *reward* r_t from the environment and the *state changes* to s_{t+1}
  - *r: (S,A)→R δ: (S,A)→S r is the reward function and δ the transition matrix*
- Markov assumption:

  $r_t=r(s_t,a_t)$ $s_{t+1}=\delta(s_t,a_t)$ i.e. r_t and s_{t+1} depend **only on the *current* state** and action
  - In general, the functions *r and δ* may not be deterministic (i.e. they are stochastic, described by random variables) and are not necessarily known to the agent

Agent's Learning Task

Execute actions in environment, observe results and

- **Learn action policy** $\pi : S \rightarrow A$ that maximises expected cumulative reward ER

  from any starting state s_0 in S.

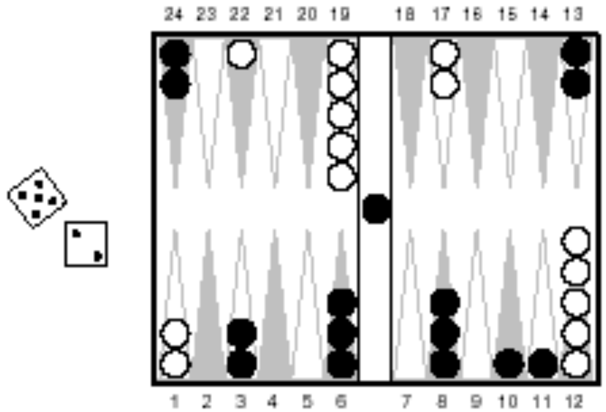
$$ER[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots]$$

  Here $0 \leq \gamma \leq 1$ is the **discount factor** for future rewards, r_k is the reward at time k.

- Note:
  - Target function is $\pi : S \rightarrow A$
  - There are **no training examples** of the form (s,a) but only of the form ((s,a),r), i.e. for –some- state-action pair we know the reward/ penalty

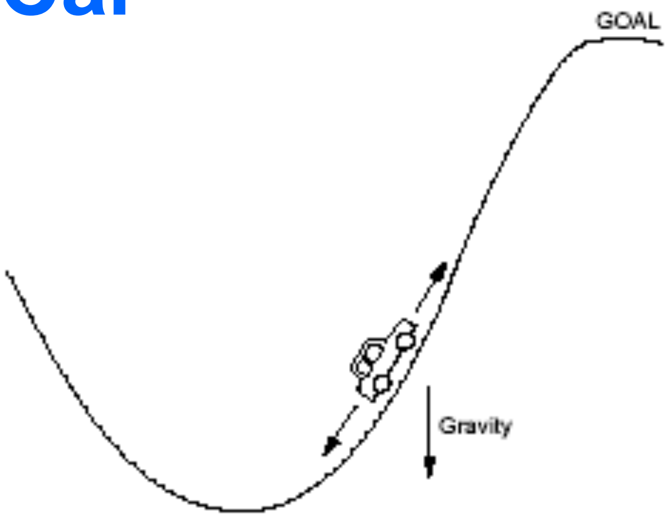
Example: TD-Gammon

- Immediate reward:
  - +100 if win
  - -100 if lose
  - 0 for all other states



- Trained by playing 1.5 million games
- Now approximately equal to the best human player

Example: Mountain-Car



- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- Rewards
  - Reward=-1for every step, until the car reaches the top
  - Reward=1 at the top, 0 otherwise
- The possible reward will be maximised by minimising the number of steps to the top of the hill

Value function

We will consider a **deterministic world** first

In a deterministic environment, rewards are known and the environment is known

- Given a policy π (adopted by the agent), define an **evaluation function** over states:

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

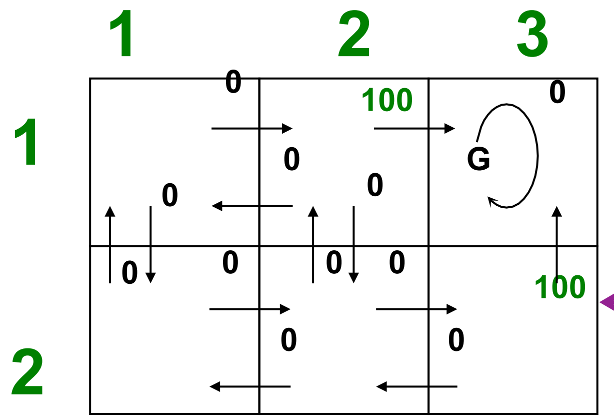
$$V^{\pi}(s_t) = r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \ldots) = r_t + \gamma V^{\pi}(s_{t+1})$$

$V^{\pi}(s_t)$ is the value of being in state s_t according to policy π

Remember: a policy $\pi : S \rightarrow A$
is a mapping from states to actions,
target is finding optimal policy

Example: robot in a grid environment

Grid world environment
Six possible states
Arrows represent possible actions
G: **goal state**
Actions: UP, DOWN, LEFT, RIGHT



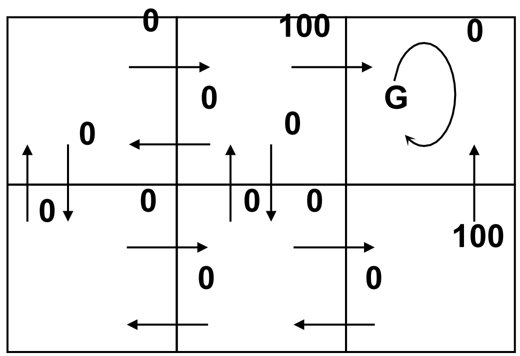
r((2,3), UP)=100

***r*(*state, action*)**
immediate reward values

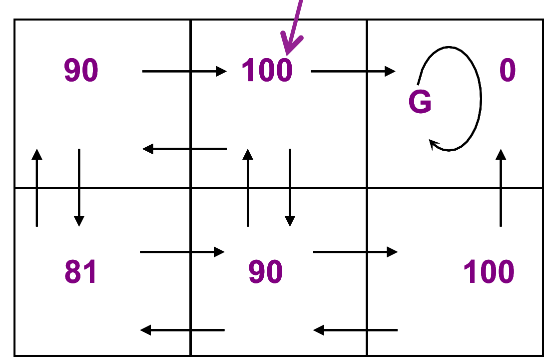
Known environment: grid with cells
Deterministic: possible moves in any cell are known, rewards are known

Example: robot in a grid environment

The "value" of being in (1,2) according to some known π (e.g. π= move to 1,3) is 100



***r*(*state, action*) immediate reward values**

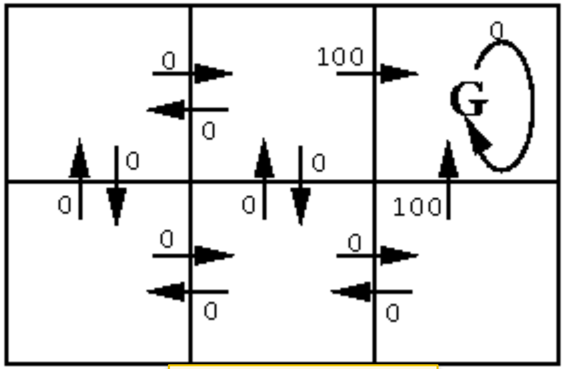


V*(*state*) values**

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- **Value function**: maps states to state values, according to some policy π. **How is V estimated?**

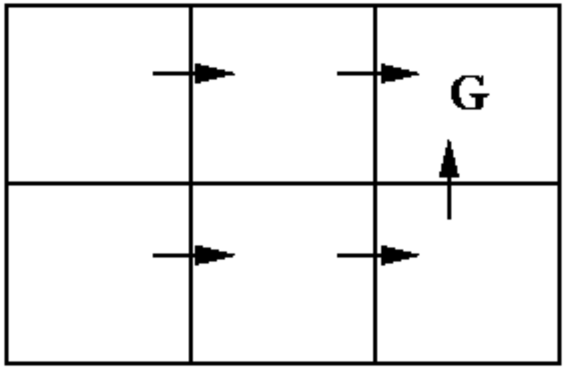
Computing V(s) (if policy π is given)



rewards

Suppose the following is an

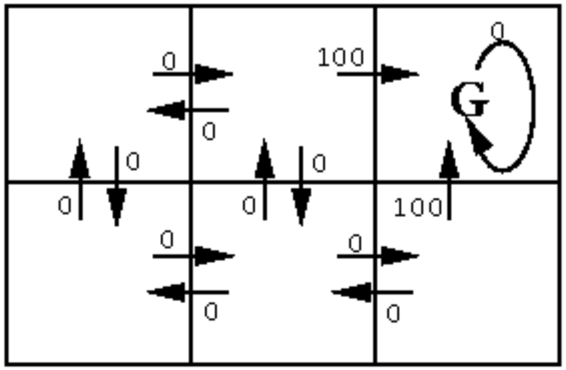
Optimal policy – denoted with π^* :



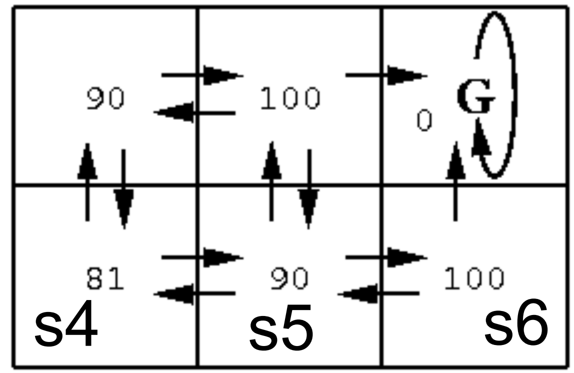
π^* tells us what is the best thing to do when in each state.

According to π^* , we can compute the values of the states for this policy – denoted $V\pi^*$, or V^*

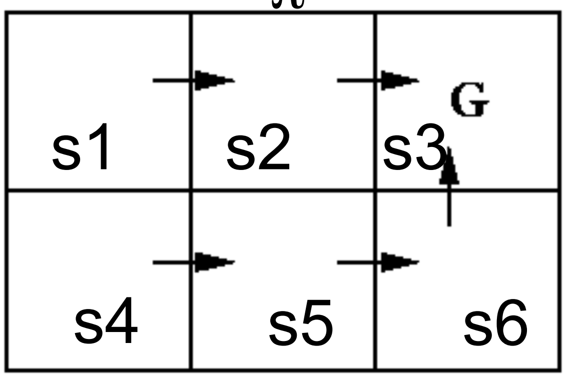
r(s.a) (immediate reward) values:



$V^*(s)$ values, with **γ=0.9**



π^*



$$V^*(s_t) = r_t + \gamma V^*(s_{t+1})$$

$V^*(s_6) = 100 + 0.9*0 = 100$

$V^*(s_5) = 0 + 0.9*100 = 90$

$V^*(s_4) = 0 + 0.9*90 = 81$

Etc.

However, π^* is usually unknown and **the task** is to **learn the optimal policy**

$$\pi^* = \arg\max_{\pi} V^{\pi}(s), (\forall s)$$

How do we learn optimal policy π^* ?

- Target function is π : *state* → *action*
- However…
  - We have **no training examples** of the form <*state*, *action*>
  - We only know: <<*state*, *action*>, *reward*>
- *Reward might not be known for all states!*

Utility-based agents

- To learn V^{π^*} (abbreviated V^*) perform look ahead search to choose best action from any state s

$$\pi^*(s) \equiv \arg\max_a \left[r(s, a) + V^*(\delta(s, a)) \right]$$

- Works well if agent knows
  - δ : *state* $\times$ *action* $\rightarrow$ *state*
  - r : *state* $\times$ *action* $\rightarrow \mathbb{R}$
- When agent doesn't know δ and r, cannot choose actions this way
- *Need a greedy method*

Q-learning: an utility-based learner in deterministic environments

- Q-values
  - Define a new function *Q* very similar to *V**
  - If agent learns *Q*, it can choose optimal action even without knowing δ or r
- Using *Q:* $Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) = r(s,a) + \gamma V^*(s')$

$$\pi^*(s) \equiv \arg\max_a \left[r(s,a) + V^*(\delta(s,a)) \right]$$

$$\pi^*(s) \equiv \arg\max_a \; Q(s,a)$$

What's new?? Apparently again depends on (unknown) δ and r

Learning the Q-value

- Note: *Q* and *V** closely related

$$V^*(s_t) = r_t + \gamma V^*(s_{t+1}) \qquad\qquad V^*(s) \equiv \max_{a'} Q(s, a')$$

- Allows us to write *Q* recursively as (**Bellman equation**):

$$\begin{aligned} Q(s(t), a(t)) &= r(s(t), a(t)) + \gamma V^*(\delta(s(t), a(t))) \\ &= r(s(t), a(t)) + \gamma \max_{a'} Q(s(t+1), a') \end{aligned}$$

- In other words, maximum future reward for a given state **s** and action **a**, is the immediate reward plus maximum future reward for the next state. This policy is also called **Temporal Difference learning**
- In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. The algorithm is shown in the next slide.

Simple Q pseudo-code

```
initialize Q[numstates,numactions] arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    Q[s,a] = Q[s,a] + α(r + γmaxa' Q[s',a'] - Q[s,a])
    s = s'
until terminated
```

α in the algorithm is a **learning rate** that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. In particular, when α=1, then the update is exactly the same as Bellman equation.

It has been shown that Algorithm converges for “sufficient” number of iterations

Algorithm to utilize the Q table

Input: **Q** matrix, initial state

1. Set current state = initial state , randomly select a possible action and compute next state
2. From next state, find action that produce maximum Q value, update Q
3. Set current state = next state
4. Go to 2 until current state = goal state

The algorithm above will return sequence of current state from initial state until goal state.

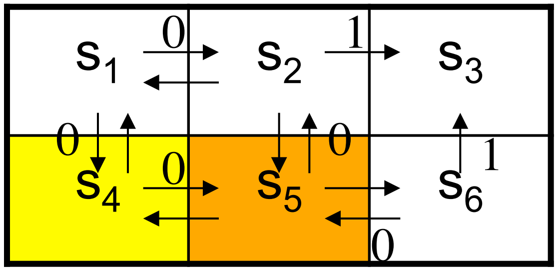
Comments: Parameter γ has range value of 0 to 1($0 \le \gamma < 1$).

If γ is closer to zero, the agent will tend to consider only

immediate reward. If γ is closer to one, the agent will consider future reward with greater weight, willing to delay the

reward.

Q-Learning: Example (γ=0.9) Episode 1



s=s_4 possible moves: North, East
Select (at random) to move to s5

- Set current state = initial state , randomly select a possible action and compute next state
- From current state, find action that produce maximum Q value
- Update

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

actions

	N	S	W	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0
s_5	0	0	0	0
s_6	0	0	0	0

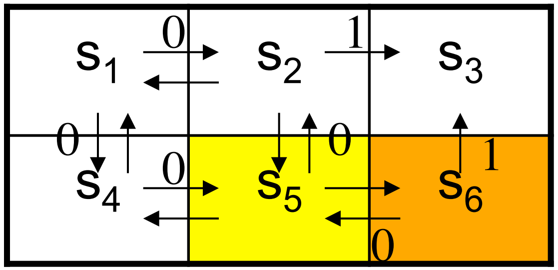
Q-values table

But in s5: $\forall a,\ \hat{Q}(s5,a) = 0$ therefore:

$$\hat{Q}(s4,E) = r(s4,E) + \gamma \max_{a'} \hat{Q}(s5,a') = 0 + 0{,}9 \times 0 = 0$$

The “max Q” function search for the “most promising action” from s5

Q-Learning: Example (γ=0.9)



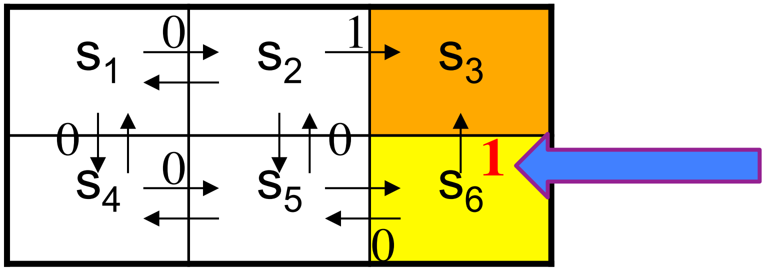
$S_4 \rightarrow S_5$ randomly select s_6 as next move

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

	N	S	O	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	**0**
s_5	0	0	0	**0**
s_6	0	0	0	0

From s_5, moving to s_6 again does not allow rewards (all Q(s6,a)=0)
Update Q(s5,E)←0+argmax(Q(s6,a'))=0+ +0,9xQ(s6,N))=0+0,9x0

Q-Learning: Example (γ=0.9)



$s_4 \rightarrow s_5 \rightarrow s6$ randomly select s_3 as next move

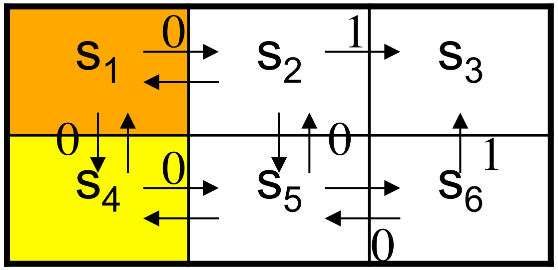
In s3 r=1 so Q is updated to 1

	N	S	O	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	**0**
s_5	0	0	0	**0**
s_6	**1+0.9*0 =1**	0	0	0

$$\hat{Q}(s_6, N) \leftarrow 1 + 0{,}9 \max_{a'} \hat{Q}(s_3, a') = 1 + 0{,}9 \times 0 = 1$$

- **GOAL STATE REACHED: END OF FIRST EPISODE**

Q-Learning: Example (γ=0.9) Episode 2

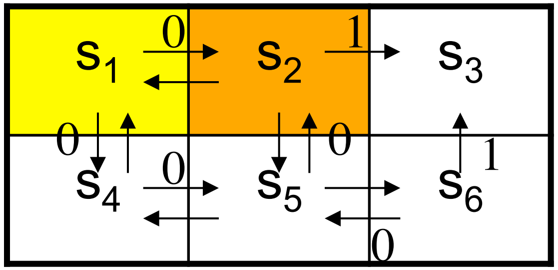


$s = s_4$, $s' = s_1$

	N	S	O	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0	0	0	**0**
s_6	**1**	0	0	0

$$\hat{Q}(s_4, N) \leftarrow 0 + 0,9 \times \max_{a'} \hat{Q}(s_1, a') = 0$$

Q-Learning: Example (γ=0.9)

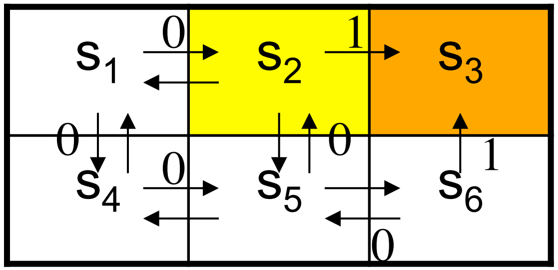


$s_4 \rightarrow s_1$, choose $s' = s_2$

	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	0
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0	0	0	**0**
s_6	**1**	0	0	0

$$\hat{Q}(s_1, E) \leftarrow 0 + 0{,}9 \times \max_{a'} \hat{Q}(s_2, a') = 0$$

Q-Learning: Example (γ=0.9)



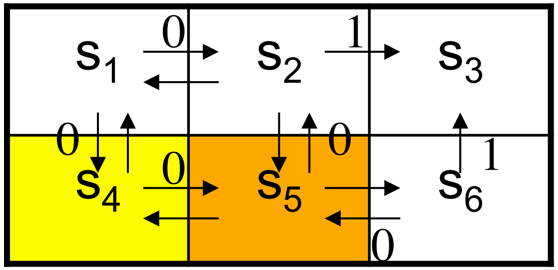
s_4 ➔ s1 ➔ s_2, choose s'=s_3

	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	1
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0	0	0	**0**
s_6	**1**	0	0	0

$$\hat{Q}(s_2, E) \leftarrow 1 + 0{,}9 \max_{a'} \hat{Q}(s_3, a') = 1$$

- **GOAL STATE REACHED: END OF 2nd EPISODE**

Q-Learning: Example (γ=0.9) Episode 3

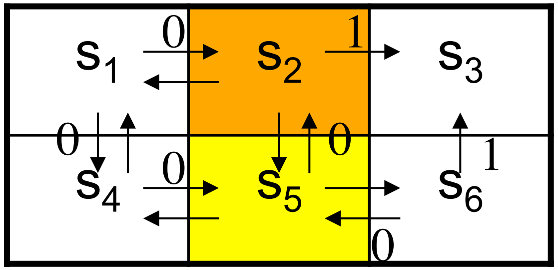


s=s_4, s'=s_5

	N	S	O	E
s_1	0	0	0	0
s_2	0	0	0	1
s_3	0	0	0	0
s_4	0	0	0	0+0
s_5	0	0	0	0
s_6	1	0	0	0

$$\hat{Q}(s_4, E) \leftarrow 0 + 0{,}9 \max_{a'} \hat{Q}(s_5, a') = 0$$

Q-Learning: Example (γ=0.9)

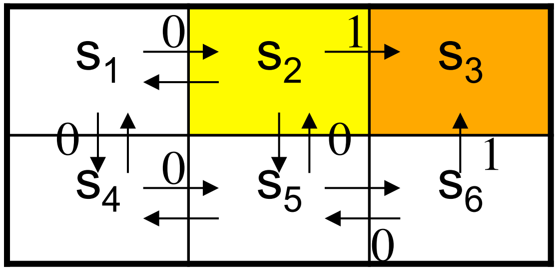


	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0,9	0	0	**0**
s_6	**1**	0	0	0

s=s_5, s'=s_2

$$\hat{Q}(s_5, N) \leftarrow 0 + 0{,}9 \max_{a'} \hat{Q}(s2, a') = 0 + 0{,}9 \times 1 = 0{,}9$$

Q-Learning: Example (γ=0.9)



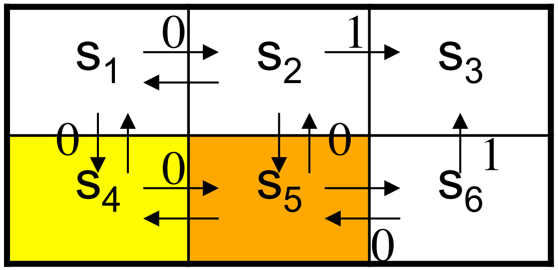
$s{=}s_2,\ s'{=}s_3$

	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1+0**
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

$$\hat{Q}(s_2, E) \leftarrow 1 + 0{,}9 \max_{a'} \hat{Q}(s_3, a') = 1$$

- **GOAL STATE REACHED: END OF 3rd EPISODE**

Q-Learning: Example (γ=0.9) Episode 4

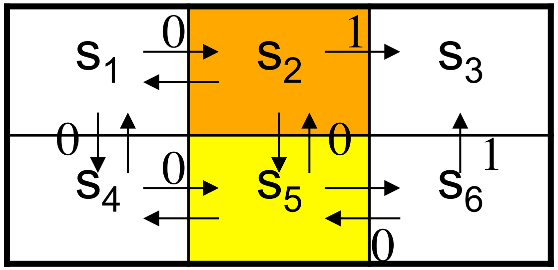


$s=s_4$, $s'=s_5$

	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

$$\hat{Q}(s_4, E) \leftarrow 0 + 0{,}9 \max_{a'} \hat{Q}(s_5, a') = 0{,}9 \times 0{,}9 = 0{,}81$$

Q-Learning: Example (γ=0.9)

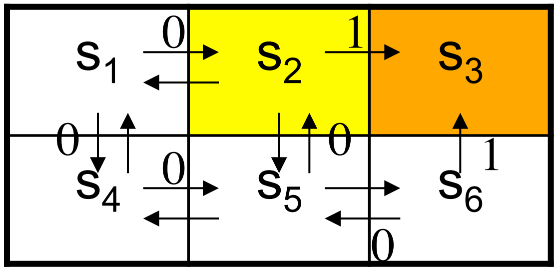


$s=s_5$, $s'=s_2$

	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0,9**	0	0	**0**
s_6	**1**	0	0	0

$$\hat{Q}(s5, N) \leftarrow 0 + 0{,}9 \max_{a'} \hat{Q}(s_2, a') = 0{,}9 \times 1 = 0{,}9$$

Q-Learning: Esempio (γ=0.9)

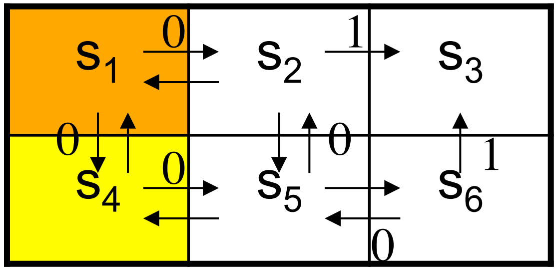


$s=s_2$, $s'=s_3$

	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1+0**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

- **GOAL REACHED: END OF 4th EPISODE**

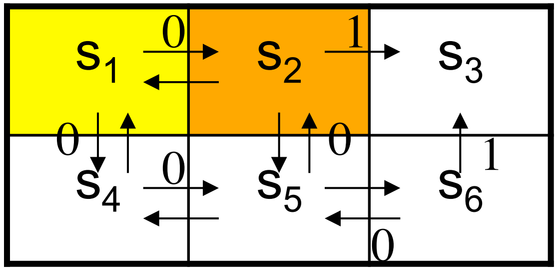
Q-Learning: Example (γ=0.9)



$s = s_4,\ s' = s_1$

	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0+0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

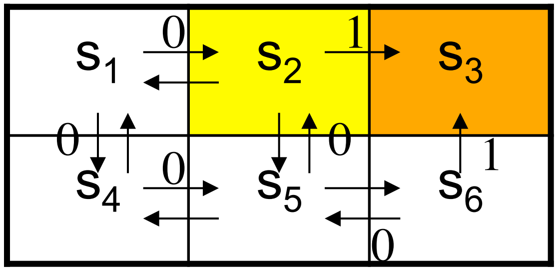
Q-Learning: Example (γ=0.9)



$s=s_1, s'=s_2$

	N	S	O	E
s_1	0	0	0	**0.9**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

Q-Learning: Esempio (γ=0.9)



	N	S	O	E
s_1	0	0	0	**0.9**
s_2	0	0	0	**1+0**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

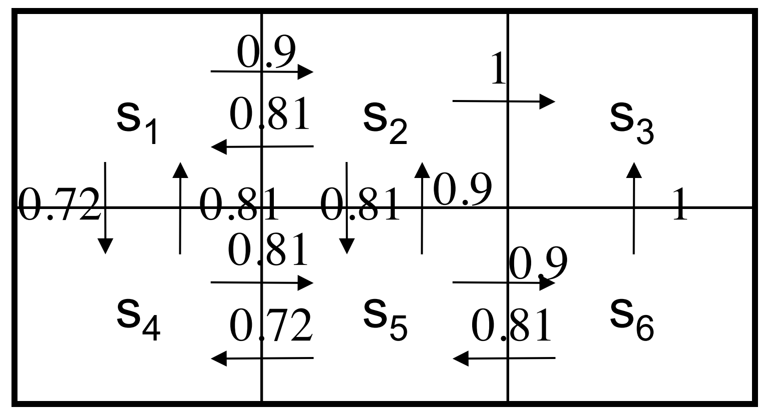
$s=s_2,\ s'=s_3$

- **GOAL REACHED: END OF 5th EPISODE**

Q-Learning: Example(γ=0.9)

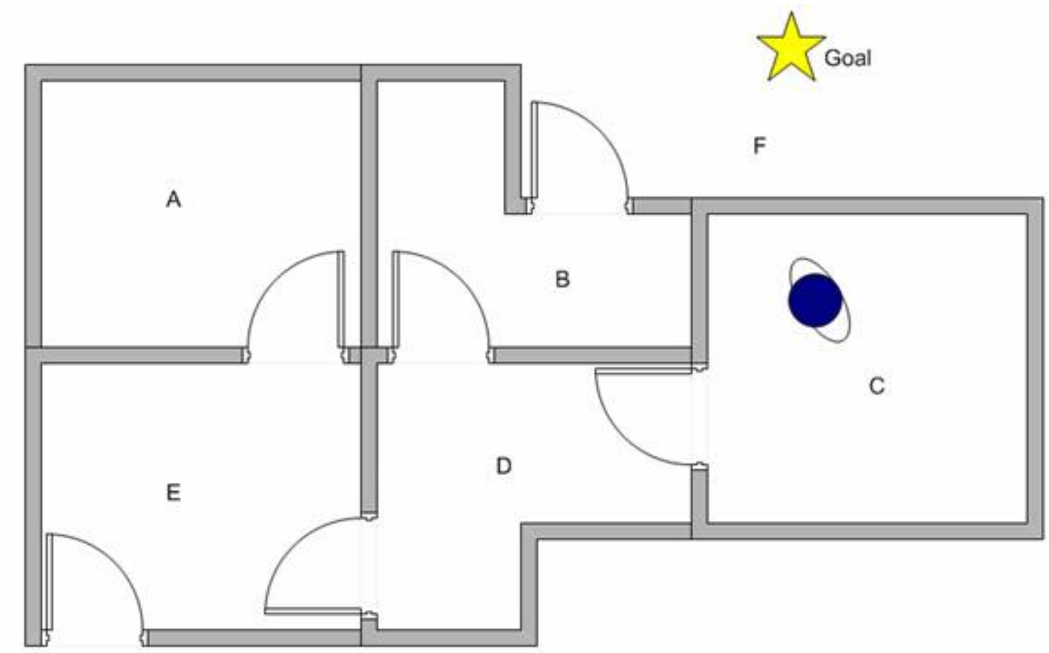
After several iterations, the algorithm converges to the following table:

	N	S	O	E
s_1	0	**0.72**	0	**0.9**
s_2	0	**0.81**	**0.81**	**1**
s_3	0	0	0	0
s_4	**0.81**	0	0	**0.81**
s_5	**0.9**	0	**0.72**	**0.9**
s_6	**1**	0	**0.81**	0



Yet another example

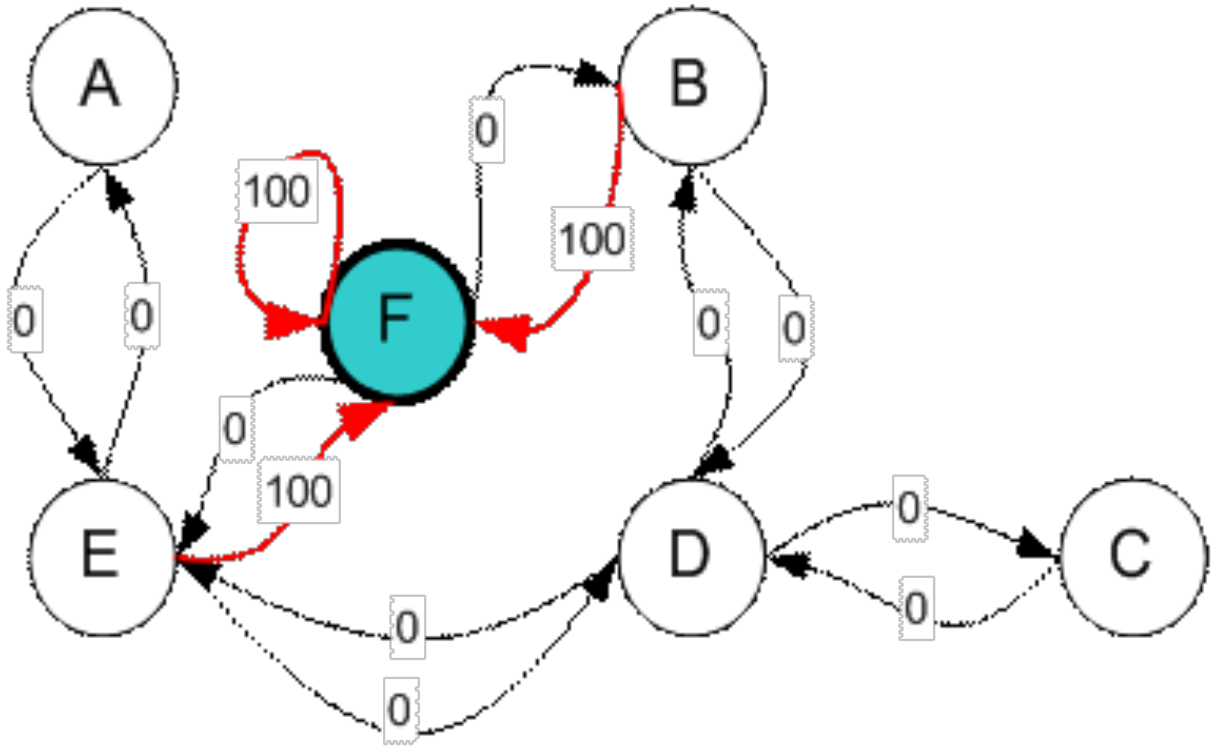
- Environment



A to E: rooms, F: outside building (target).

The aim is that an agent learn to get out of building from any of rooms in an optimal way.

- Modeling of the environment



State, Action, Reward and Q-value

- Reward matrix

$$
\mathbf{R} = \begin{array}{c} state \setminus action \\ A \\ B \\ C \\ D \\ E \\ F \end{array}
\begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{bmatrix}
- & - & - & - & 0 & - \\
- & - & - & 0 & - & 100 \\
- & - & - & 0 & - & - \\
- & 0 & 0 & - & 0 & - \\
0 & - & - & 0 & - & 100 \\
- & 0 & - & - & 0 & 100
\end{bmatrix} \end{array}
$$

- Q-table and the update rule

$$\mathbf{Q} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Q table update rule:

$$\mathbf{Q}(state, action) = \mathbf{R}(state, action) + \gamma \cdot Max[\mathbf{Q}(next\ state, all\ actions)]$$

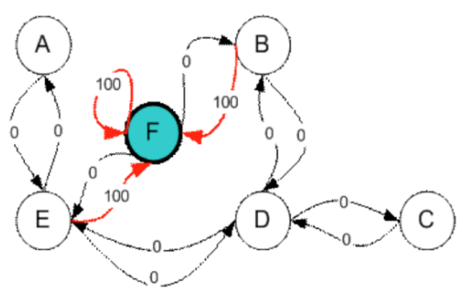
$0 \le \gamma < 1$: learning parameter

Numerical Example

Let us set the value of learning parameter 0.8 and initial state as room B.

$$
\mathbf{Q} = \begin{array}{c|cccccc}
 & A & B & C & D & E & F \\
\hline
A & 0 & 0 & 0 & 0 & 0 & 0 \\
B & 0 & 0 & 0 & 0 & 0 & 0 \\
C & 0 & 0 & 0 & 0 & 0 & 0 \\
D & 0 & 0 & 0 & 0 & 0 & 0 \\
E & 0 & 0 & 0 & 0 & 0 & 0 \\
F & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

$$
\mathbf{R} = \begin{array}{c|cccccc}
\text{state} \backslash \text{action} & A & B & C & D & E & F \\
\hline
A & - & - & - & - & 0 & - \\
B & - & - & - & 0 & - & 100 \\
C & - & - & - & 0 & - & - \\
D & - & 0 & 0 & - & 0 & - \\
E & 0 & - & - & 0 & - & 100 \\
F & - & 0 & - & - & 0 & 100
\end{array}
$$



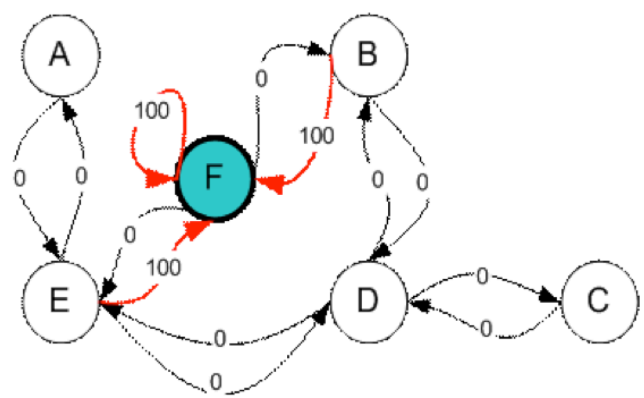
Episode 1: start from B

Look at the second row (state B) of matrix **R**. There are two possible actions for the current state B, that is to go to state D, or go to state F. By random selection, we select to go to F as our action.

$$\mathbf{Q}(state, action) = \mathbf{R}(state, action) + \gamma \cdot Max\left[\mathbf{Q}(next\ state, all\ actions)\right]$$

$$\mathbf{Q}(B,F) = \mathbf{R}(B,F) + 0.8 \cdot Max\{\mathbf{Q}(F,B), \mathbf{Q}(F,E), \mathbf{Q}(F,F)\} = 100 + 0.8 \cdot 0 = 100$$

$$\mathbf{Q} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \boxed{100} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



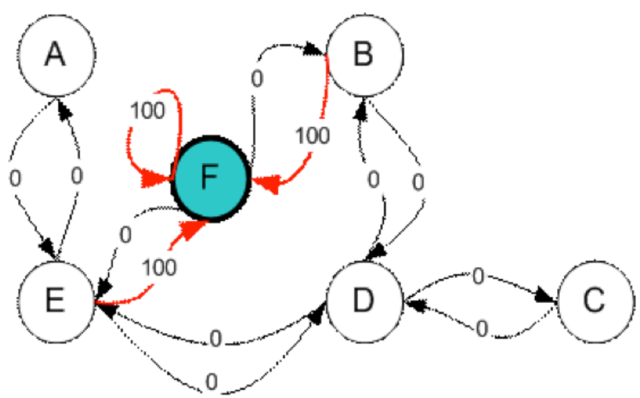
Episode 2: start from D

This time for instance we randomly have state D as our initial state. From **R**; it has 3 possible actions, B, C and E. We randomly select to go to state B as our action.

$$\mathbf{Q}(state, action) = \mathbf{R}(state, action) + \gamma \cdot Max\left[\mathbf{Q}(next\ state, all\ actions)\right]$$

$$\mathbf{Q}(D,B) = \mathbf{R}(D,B) + 0.8 \cdot Max\{\mathbf{Q}(B,D), \mathbf{Q}(B,F)\} = 0 + 0.8 \cdot Max\{0, 100\} = 80$$

$$\mathbf{Q} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \boxed{80} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



Episode 2 (cont'd)

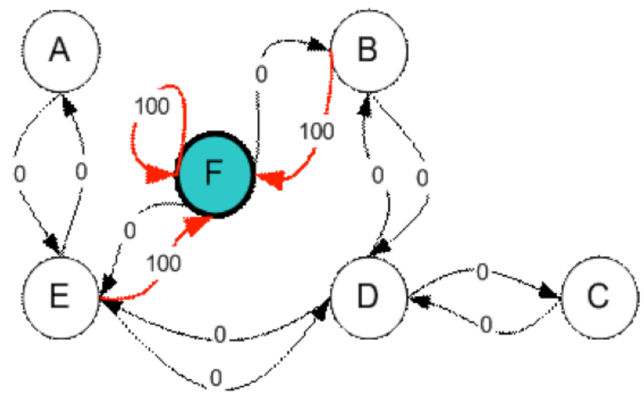
The next state is B, now become the current state. We repeat the inner loop in Q learning algorithm because state B is not the goal state. There are two possible actions from the current state B, that is to go to state D, or go to state F. By lucky draw, our action selected is state F.

$$\mathbf{Q}(state, action) = \mathbf{R}(state, action) + \gamma \cdot Max\left[\mathbf{Q}(next\ state, all\ actions)\right]$$

$$\mathbf{Q}(B,F) = \mathbf{R}(B,F) + 0.8 \cdot Max\{\mathbf{Q}(F,B), \mathbf{Q}(F,E), \mathbf{Q}(F,F)\}$$
$$= 100 + 0.8 \cdot Max\{0,0,0\} = 100$$

$$\mathbf{Q} = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

No change



After Many Episodes

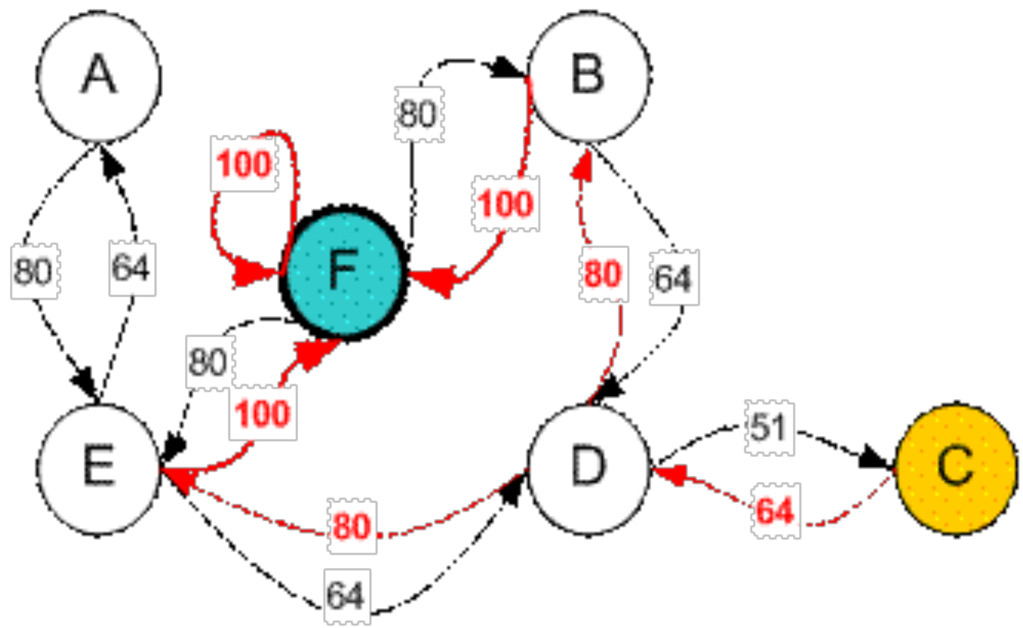
If our agent learns more and more experience through many episodes, it will finally reach convergence values of Q matrix as

$$\mathbf{Q} = \begin{array}{cc} \mathit{state} \backslash \mathit{action} & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} & \begin{bmatrix} - & - & - & - & 400 & - \\ - & - & - & 320 & - & 500 \\ - & - & - & 320 & - & - \\ - & 400 & 256 & - & 400 & - \\ 320 & - & - & 320 & - & 500 \\ - & 400 & - & - & 400 & 500 \end{bmatrix} \end{array}$$

Normalized to percentage

$$\hat{\mathbf{Q}} = \begin{array}{cc} \mathit{state} \backslash \mathit{action} & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} & \begin{bmatrix} - & - & - & - & 80 & - \\ - & - & - & 64 & - & 100 \\ - & - & - & 64 & - & - \\ - & 80 & 51 & - & 80 & - \\ 64 & - & - & 64 & - & 100 \\ - & 80 & - & - & 80 & 100 \end{bmatrix} \end{array}$$

Once the Q matrix reaches almost the convergence value, our agent can reach the goal in an optimum way. To trace the sequence of states, it can easily compute by finding action that makes maximum Q for this state.



For example from initial State C, using the Q matrix, we can have the sequences C – D – B – F or C-D-E-F

The non-deterministic case

- What if the reward and the state transition are non-deterministic? – e.g. in Backgammon learning and other games moves depends on rolls of a dice!
- Then V and Q needs redefined by taking **expected values:**

$$V^{\pi}(s) \equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...] = E[\sum_{i=0}^{\infty} \gamma^i r_{t+i}]$$

$$Q(s,a) \equiv E[r(s,a) + \gamma V^*(\delta(s,a))]$$

- Mean values can be estimated observing several sequences of moves (episodes).

- Similar reasoning and convergent update iteration will apply

Non-deterministic case

$$
\begin{aligned}
V^{\pi}(s) &\equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots] \\
&\equiv E[\sum_{i=0}^{\infty} \gamma^i r_{t+i}] \\
Q(s,a) &\equiv E[r(s,a) + \gamma V^*(\delta(s,a))] \\
&\equiv E[r(s,a)] + \gamma E[V^*(\delta(s,a))] \\
&\equiv E[r(s,a)] + \gamma \sum_{s'} \boxed{P(s'|s,a)}\boxed{V^*(s')} \\
&\equiv E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \boxed{\max_{a'} Q(s',a')}
\end{aligned}
$$

Where *P(s'|s,a)* is the conditional probability of landing in *s'* when the system is in *s* and performs action *a*

Non-deterministic case

How is the Q updating rule modified for the non-deterministic case?

- Alter training rule to

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \max_{a'} \hat{Q}_{n-1}(s',a')]$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

With probability $(1-\alpha_n)$ system stays in current state and gets no reward, with probability α_n it makes a move

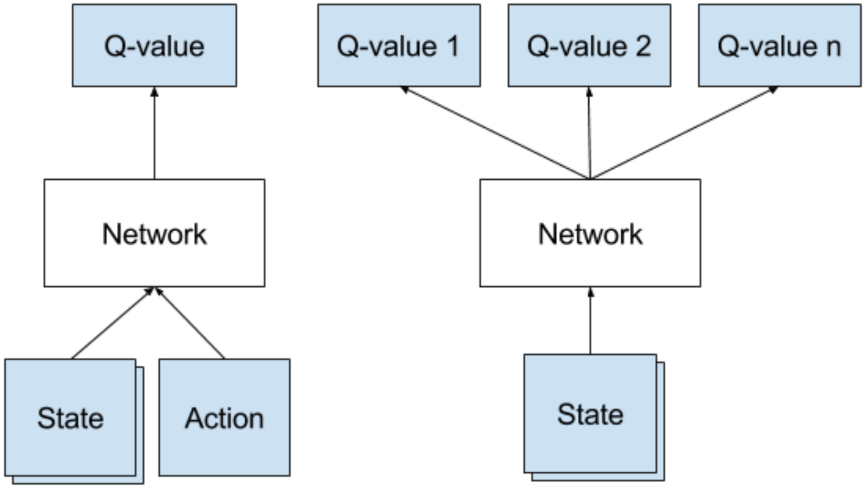
- It can be proven that $\hat{Q}$ converges to Q [Watkins and Dayan, 1992]

Deep Q learning (1)

- If many practical applications, e.g., games, the dimension of a Q table is very large
- For example, consider Atari games: The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the existence of each individual brick.
- If apply a pixel-level processing– e.g., take four last screen images, resize them to 84×84 and convert to grayscale with 256 gray levels – we would have 25684×84×4≈1067970 possible game states. This means 1067970 rows in our imaginary Q-table – that is more than the number of atoms in the known universe!

Deep Q (2)

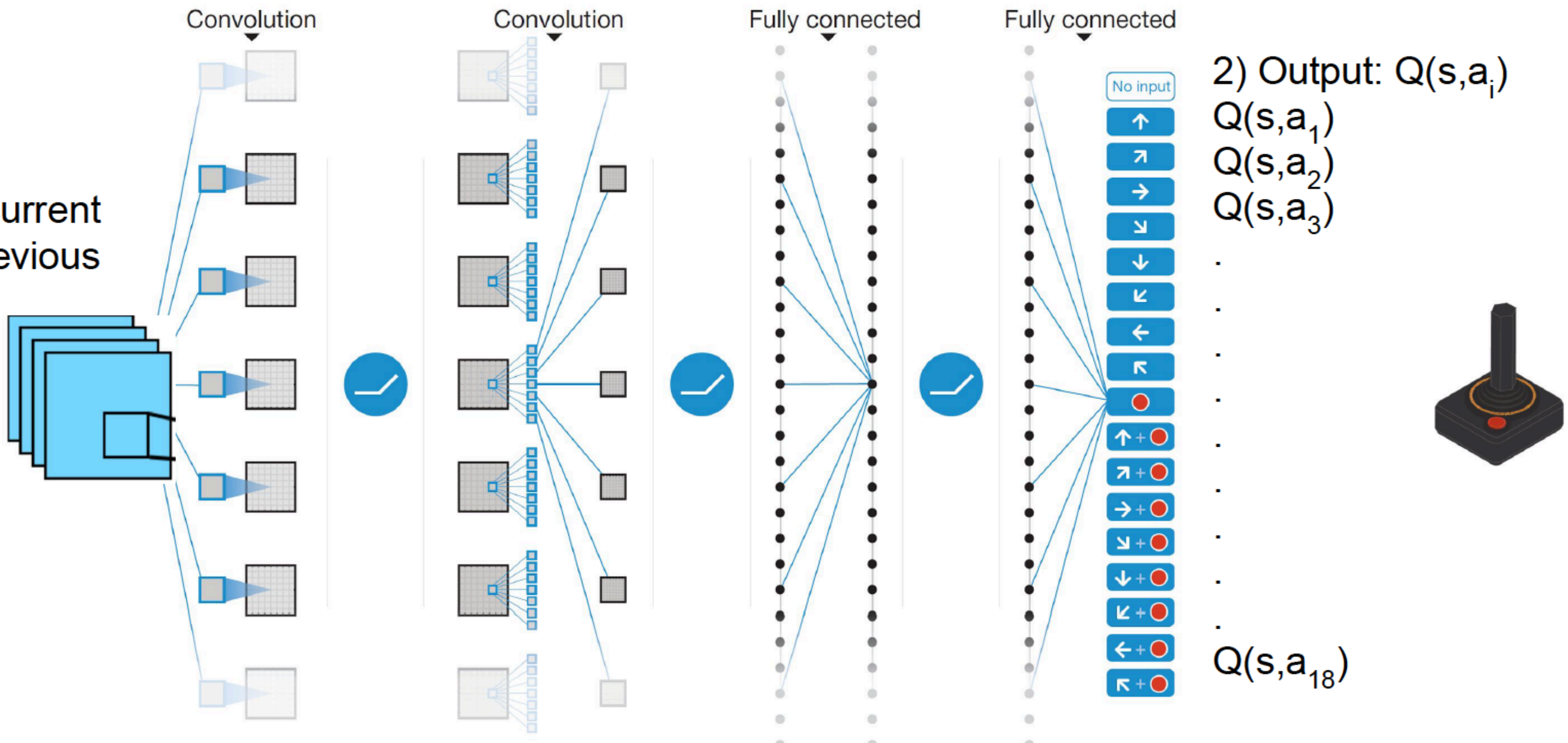
- The intuition is that, in order to learn Q values, we can use neural networks. We train for some state and action, and we can then use the trained network to compute Q for any state and action



- In the right-hand side formulation, input is a (possibly multi-dimensional) representation of a state s and action a, output is the Q value (s,a)
- In the left formulation, input is a state s, output are the Q for all possible actions a1, a2…

Example: Deep Mind network for Atari games (Mnih 2013)

1) Input:
4 images = current frame + 3 previous



Input to the network are four 84×84 grayscale game screens. Outputs of the network are Q-values for each possible action (18 in Atari).

Example: Deep Mind network for Atari games (Mnih 2013)

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Notice that there are **no pooling layers**!
Pooling layers allow for translation invariance – the network becomes insensitive to the location of an object in the image.
That makes perfectly sense for an image classification task, but for games the location of objects (e.g., the ball) is crucial in determining the potential reward and we wouldn't want to discard this information!

How does it works?

Given a transition <s,a,r,s'>, the Q-table update rule in the “classic” algorithm must be replaced with the following:

1. Do a feedforward pass on the Deep Network for current state **s** and get predictions for Q(s,a), for any possible action *a*;
2. Do another feedforward pass for the next state s' and calculate maximum over all network outputs $max_{a'}Q(s',a')$.
3. Set Q-value “target” (ground truth) for action *a* to: $[r+\gamma \times max_{a'}Q(s',a')]$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. To compute the error on Q(s,a), use the standard loss (error) function:

$$L = \frac{1}{2}[\underbrace{r + \gamma max_{a'} Q(s', a')}_{\text{target}} - \underbrace{Q(s,a)}_{\text{prediction}}]^2$$

5. Use gradient descent with back-propagation to update network weights

More issues

- We have shown how to estimate the future reward in each state using Q-learning and approximate the Q-function using a convolutional neural network.
- But it turns out that approximation of Q-values using non-linear functions (such as NNs) is not very stable and very slow, even with conv-nets.
- Several “tricks” can be used to speed convergence:
- Most important is **experience replay**. During gameplay all the experiences <*s*,*a*,*r*,*s*'> are stored in a “replay” memory. When training the network, random samples from the replay memory are used instead of the most recent transition (in other words, we don’t follow the sequence of moves of a player).
- This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum.

Summary

- Reinforcement learning is suitable for learning in *uncertain* environments where rewards may be *delayed* and subject to chance
- The goal of a reinforcement learning program is to maximise the *eventual* reward
- Q-learning is a form of reinforcement learning that doesn't require that the learner has prior knowledge of how its actions affect the environment