

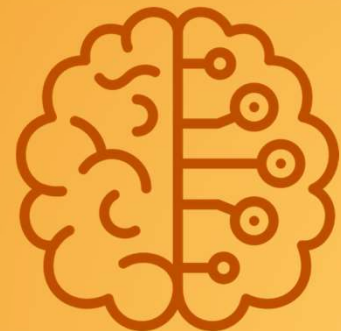
---

# CS 4375 – Introduction to Machine Learning

---

**Reinforcement Learning**

**Erick Parolin**



[Based on the slides of Nicholas Ruozzi and Ronald J. Williams]

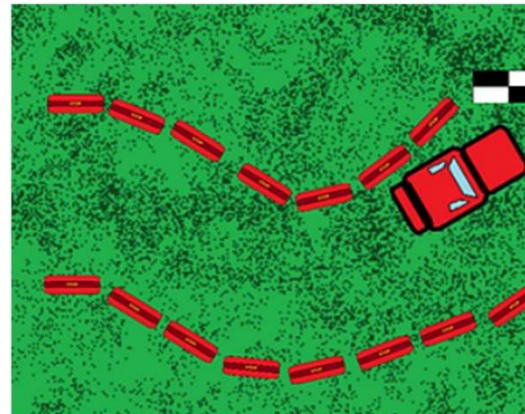
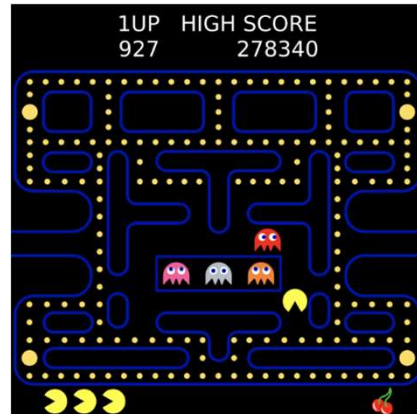
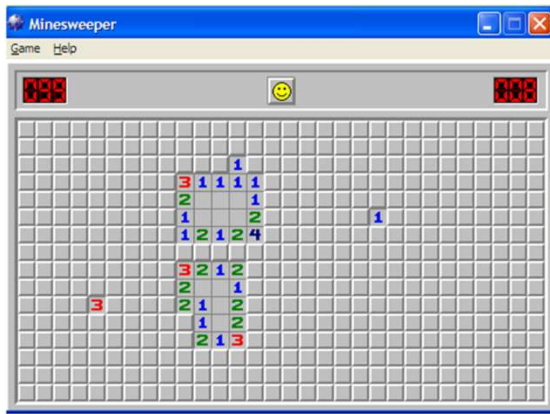
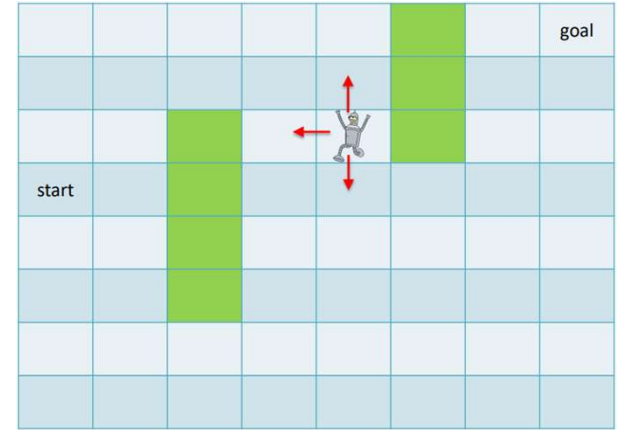
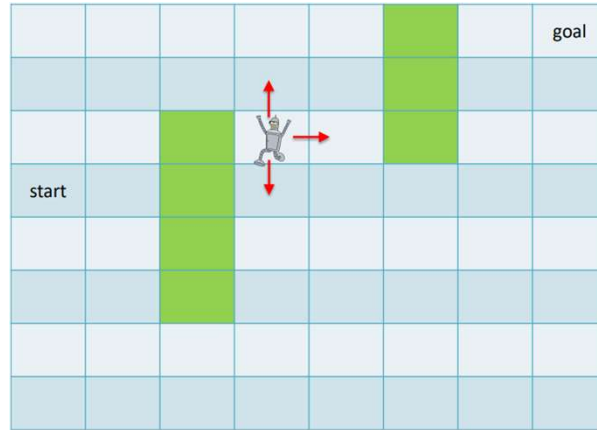
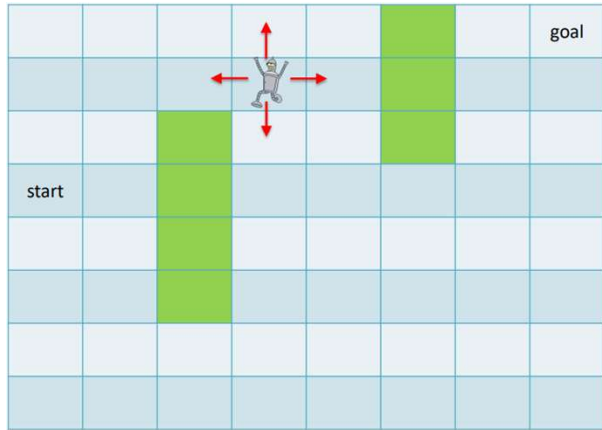
# Reinforcement Learning

- Autonomous ***agent*** that interacts with an environment through a series of actions
  - E.g., a robot trying to find its way through a maze
  - Actions include turning and moving through the maze
  - The agent earns rewards from the environment under certain (perhaps unknown) conditions
- The agent's goal is to maximize the reward
  - We say that the agent learns if, over time, it improves its performance

# Reinforcement Learning

- Often formalized (mathematically) as **Markov Decision Processes (MDPs)** or Partially Observable Markov Decision Processes (POMDPs)
- **MDPs** are described by series of states (state of the environment) and a collection of actions corresponding to each state (allowable actions that change the state of the environment)
  - The next state depends (perhaps probabilistically) on only the current state and the chosen action
  - Each state/action pair has an associated **reward** (possibly probabilistic)
- Markov chains are a simple form of MDP with only one action and no rewards.

# Reinforcement Learning – Examples



# Reinforcement Learning

- Rewards can be positive or negative
  - E.g., the robot might receive a small penalty each time it takes a step that does not reach the goal
- Objective of the learning process is to **develop a policy** (a way to choose actions given the current state) **to maximize the reward**
  - Could be difficult to do as rewards may be delayed
    - E.g., the robot receives a reward for reaching the end of the maze, but only penalties in-between

# Reinforcement Learning

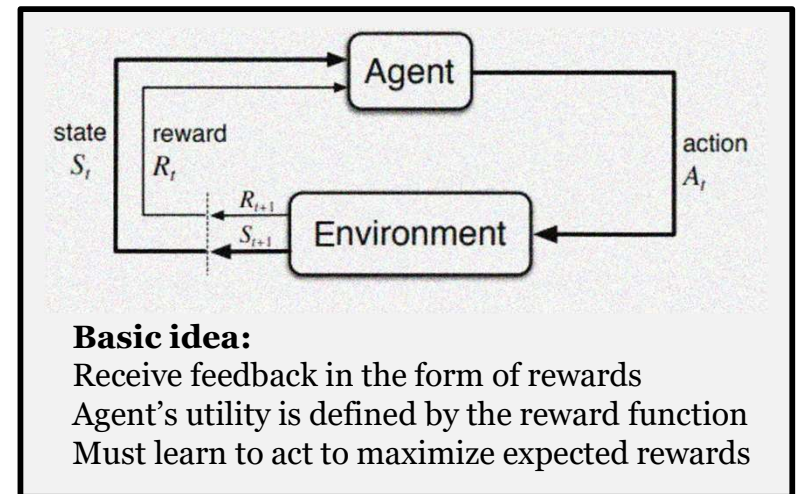
Agent at step  $t$ :

Observes the state of the system

Selects an action to perform

Receives some reward

This process is repeated indefinitely



# Reinforcement Learning

## Policies

- A policy is the prescription by which the agent selects an action to perform
  - **Deterministic:** the agent observes the state of the system and chooses an action
  - **Stochastic:** the agent observes the state of the system and then selects an action, at random, from some probability distribution over possible actions

# Reinforcement Learning

## Applications

- Robot pathfinding
- Planning
- Elevator scheduling
- Manufacturing processes
- Network routing
- Game playing

# Reinforcement Learning

## Formal Definition

- A deterministic **MDP** consists of the following
  - A finite set of states  $\mathcal{S}$
  - A set of allowable actions  $\mathcal{A}_s$  for each  $s \in \mathcal{S}$
  - A transition function  $T: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$
  - A reward function  $R: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$
- We want to learn a policy,  $\pi: \mathcal{S} \rightarrow \mathcal{A}$  that maximizes the sum of rewards we see over lifetime

# MDP – Policies

## Example

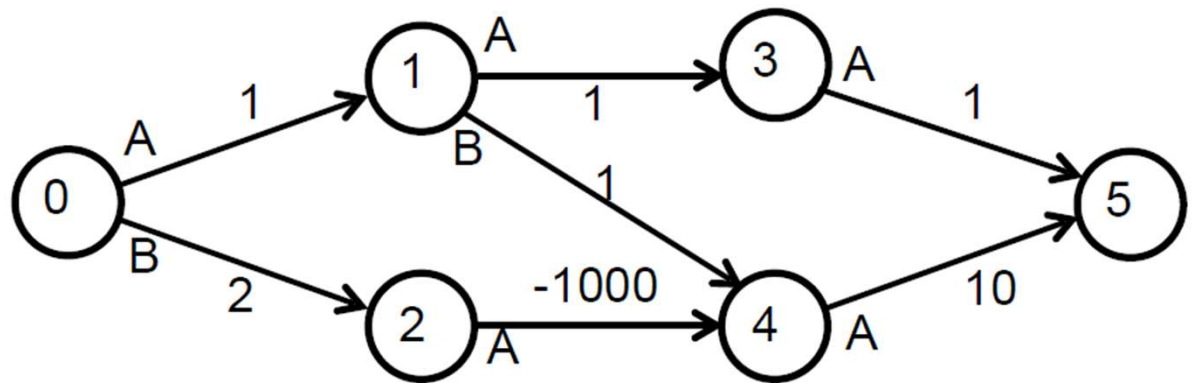
**A policy  $\pi(s)$  returns what action to take in states**

**There are 3 policies for this MDP**

**Policy 1:**  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$

**Policy 2:**  $0 \rightarrow 1 \rightarrow 4 \rightarrow 5$

**Policy 3:**  $0 \rightarrow 2 \rightarrow 4 \rightarrow 5$



# MDP – Comparing Policies

## Value Functions

How can we evaluate the quality of policy  $\pi$ ?

- A value function  $V: \mathcal{S} \rightarrow \mathbb{R}$  assigns a real number to each state
- A particular value function of interest will be the reward function

$$V^\pi(\mathcal{S}) = \sum_{t=0}^{\infty} r(t)$$

where the state at time  $t$  is generated from the state at time  $t - 1$  by applying the action dictated by the policy,  $\pi(S_{t-1})$ , and  $r(t)$  denotes the reward at time  $t$ .

# Value Functions

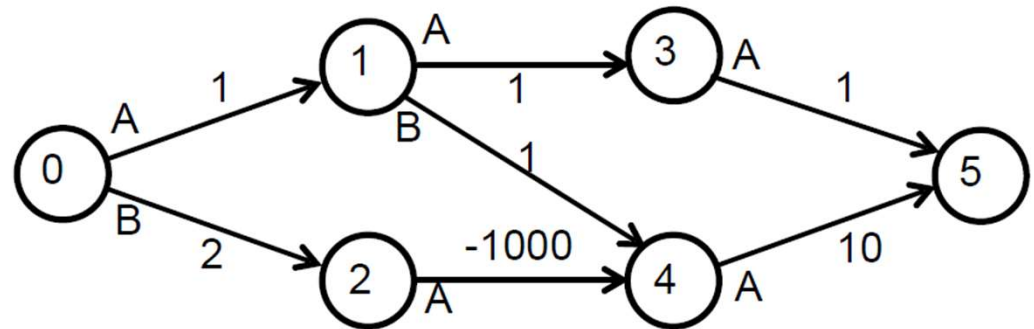
## Back to Example

Which policy is best?

**Policy 1:**  $0 \rightarrow 1 \rightarrow 3 \rightarrow 5 = 1 + 1 + 1 = 3$

**Policy 2:**  $0 \rightarrow 1 \rightarrow 4 \rightarrow 5 = 1 + 1 + 10 = 12$

**Policy 3:**  $0 \rightarrow 2 \rightarrow 4 \rightarrow 5 = 1 + 1000 + 10 = -988$



# Value Functions

**Value Function** estimates the expected total reward an agent can obtain starting from a given state and following a specific policy thereafter.

- Helps assess how “*good*” or valuable a state is in terms of *future rewards*.
- Guides the agent in improving its policy by identifying which states yield higher expected rewards, helping the agent make better decisions over time.

Value Function is defined as the recursion:

$$V^{\pi}(S) = R(S, \pi(S)) + V^{\pi}(S')$$

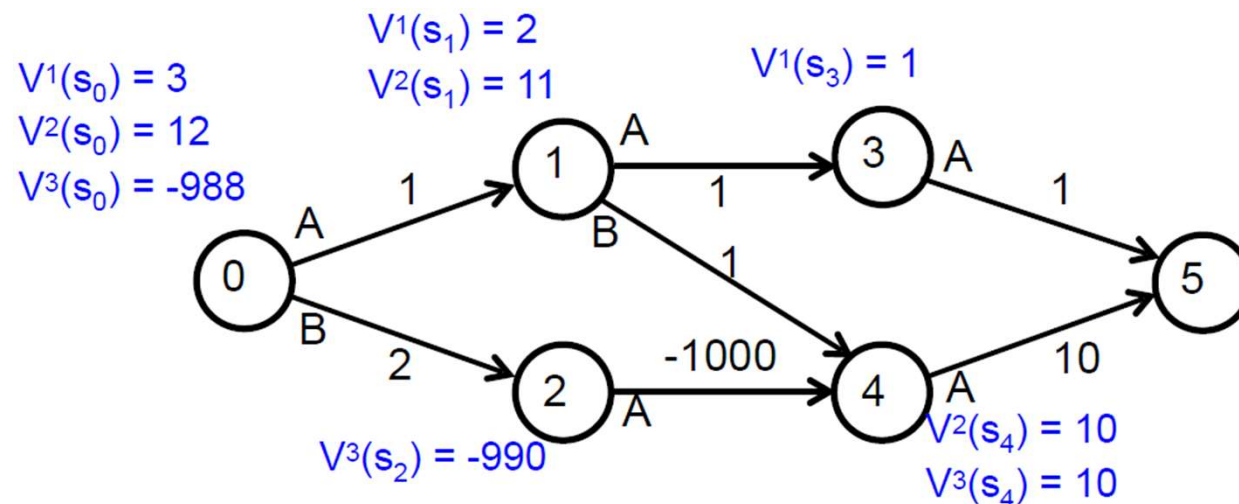
where  $S'$  is the next state determined by the action in from policy  $\pi(S)$ , or  $S' = T(S, \pi(S))$

# Value Functions

## Back to Example

For a fixed policy  $\pi$ , we associate a **value** with each state  $V^\pi(S)$

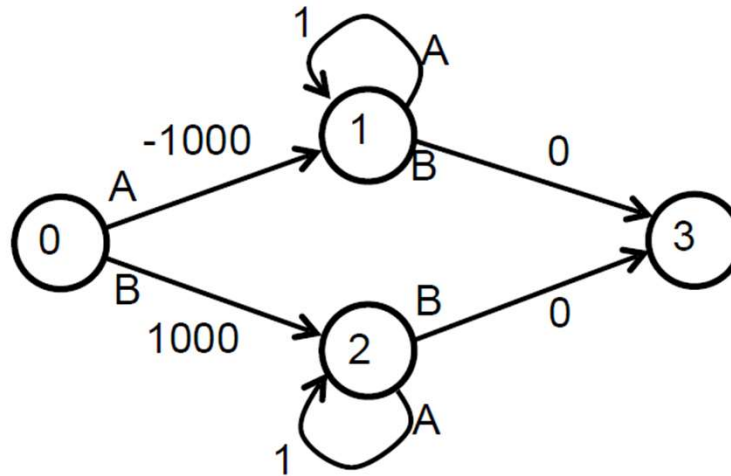
- Tells us how good it is to run policy  $\pi$  from that state  $S$



# Value Functions – Problems!

**What if we have this MDP...**

- Number of steps is now unlimited because of loops
  - Value of states 1 and 2 is infinite for some policies
  - Bad: All policies with a nonzero reward cycle have infinite value



# Value Functions – Problems!

## Analogy

- Let's say a data scientist makes *USD 150K/year*.
- How much this scientist will make in his entire life?
  - $150K + 150K + \dots = \infty$
- **Horizon Problem:** there is no limit on the “future”, so, this sum can be infinite.
- This definition is useless unless we consider a finite time horizon.
  - But, we usually don't have a good way to define such a time horizon.

# Value Functions – Problems!

## Solving the Problem: Intuition

“A reward (payment) in the future is not worth quite as much as a reward now.”

- E.g., because of inflation
- Assuming payment  $n$  years in future is worth only  $(0.9)^n$  of payment now, what is the scientist's **discounted sum of future rewards**?
- The discounted sum of future rewards using discount factor  $\gamma$  is  
 $(\text{reward now}) + \gamma (\text{reward in 1 time step}) + \gamma^2 (\text{reward in 2 time steps}) + \gamma^3 (\text{reward in 3 time steps}) + \dots$

# Value Functions – Discount Factor

## Solution: Discount Factor $\gamma$

- Determines the importance of future rewards
- **Balances Present vs. Future Rewards**
  - Controls how much the agent values immediate rewards versus future rewards.
  - Higher  $\gamma$  (close to 1) makes the agent prioritize long-term rewards,
  - Lower  $\gamma$  (close to 0) makes it focus on short-term gains.
- The agent is trying to optimize total future discounted reward:

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

Note: immediate reward is worth more than future reward.

# Value Functions – Discount Factor

## Solution: Discount Factor $\gamma$

- Determines the importance of future rewards
- **Balances Present vs. Future Rewards**
  - Controls how much the agent values immediate rewards versus future rewards.
  - Higher  $\gamma$  (close to 1) makes the agent prioritize long-term rewards,
  - Lower  $\gamma$  (close to 0) makes it focus on short-term gains.
- Rewriting the recursion for value function :

$$V^\pi(S) = R(S, \pi(S)) + \gamma V^\pi(S')$$

# Reinforcement Learning – Objective

## Objective

Find a policy  $\pi^*: S \rightarrow A$  such that

$$V^{\pi^*}(s) \geq V^{\pi}(s)$$

for all  $s \in S$  and all policies  $\pi$

Any policy that satisfies this condition is called an optimal policy  
(may not be unique)

There always exists an optimal policy

How do we find it?

# Optimal Policies

- Can find an optimal policy via **dynamic programming** approach
  - Compute the optimal value,  $V^{\pi^*}(S)$ , for each state
  - Greedily select the action that maximizes reward
- We can describe the optimal value via a recurrence relation

$$V^{\pi^*}(S) = \max_{a \in A_s} \left( R(s, a) + \gamma V^{\pi^*}(T(s, a)) \right)$$

- This is one of the so-called **Bellman equations**
- Justifies the greedy strategy (all optimal strategies are “greedy” in this sense)

# Greedy Strategy

- Given a value function  $V: \mathcal{S} \rightarrow \mathbb{R}$ , we say that  $\pi$  is greedy for  $V$  if

$$\pi(s) \in \underset{a}{\operatorname{argmax}} (R(s, a) + \gamma V(T(s, a)))$$

- If  $\pi$  is not an optimal policy, then  $\pi'$  which is greedy for  $V^\pi$  must satisfy  $V^\pi(s) \leq V^{\pi'}(s)$  for all  $s \in \mathcal{S}$
- This suggests that we can, starting from any policy, obtain a better policy

# Value Iteration

- Choose an initial value function  $V_0$  (could be anything)
- Repeat until converge:

For each  $s$ :

$$V_{t+1}(S) = \max_{a \in A_s} (R(s, a) + \gamma V_{t+1}(T(s, a)))$$

- This process always converges to the optimal value,  $V_*$ , as long as  $\gamma \in (0,1)$

# Policy Iteration

- Choose an initial policy  $\pi_0$
- Repeat until converge:
  - For each  $s$ :
    - Compute  $V^{\pi_t}$
    - Choose  $\pi_{t+1}$  to be a greedy policy with respect to  $V^{\pi_t}$
- This process always converges to an optimal policy

# Q-Values

- For learning, it will be useful to express value functions in terms of **Q-value** functions
- Q-value represents the **expected total reward** an agent can get by taking a **specific action in a given state** and then following the optimal policy afterward.
  - Helps the agent evaluate which actions are better by comparing their expected rewards
- For a policy  $\pi$ ,  $Q^\pi: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  is defined to be the value of the policy  $\pi$  starting from state  $s$  where the first action is taken to be  $a$

$$Q^\pi(s, a) = R(s, a) + \gamma V^\pi(T(s, a))$$

For any **optimal policy**  $\pi^*$ ,  $V^{\pi^*}(s) = \max_a Q^{\pi^*}(s, a)$

A policy  $\pi$  is said to be **greedy** with respect to  $Q$  if  $\pi(s) = \arg \max_a Q(s, a)$

# Reinforcement Learning

## Setting for Reinforcement Learning

- The agent is the learner whose task is to maximize its respective rewards
- The agent starts with no knowledge about the environment (*rewards* and *transitions*) and learns by interacting with it (trial and error), updating its Q-values based on the rewards it receives after taking actions in different states.
- Approaches to RL:
  - Learn the MDP (T and R functions) first, then compute optimal value/policy from it
  - Learn only the values (don't learn the MDP or explicitly model it)
    - Can be advantageous in practice as MDPs can require a significant amount of storage to specify completely

# Q-Learning Algorithm

Choose an initial state-value function  $Q(s, a)$

Let  $s$  be the initial state of the environment

Repeat until convergence:

    Choose an action  $a$  for the current state  $s$  based on  $Q$

    Take action  $a$  and observe the reward  $r$  and the new state  $s'$

    Update the table entry for  $Q(s, a)$  using **Q-learning rule**:

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

$$s \leftarrow s'$$

# Q-Learning – Picking Actions

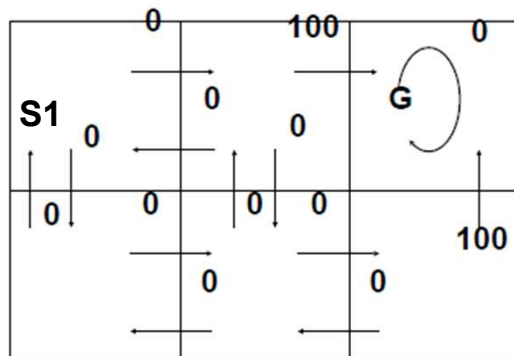
How should we pick an action to take based on  $Q$ ?

- Shouldn't always be greedy (we won't explore much of the state space this way)
- Shouldn't always be random (will take a long time to generate a good  $Q$ )
- **$\epsilon$ -greedy strategy**: with some small probability choose a random action, otherwise select the greedy action.

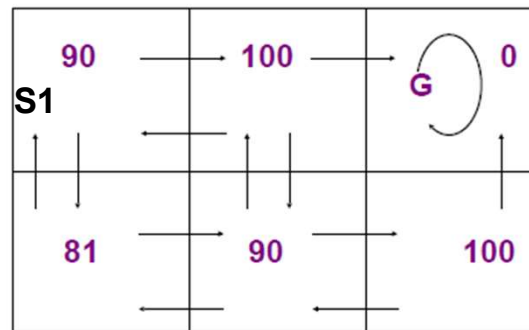
# Q-Learning – Example

Q-learning, learns the expected utility of taking a particular action  $\mathbf{a}$  in state  $\mathbf{s}$

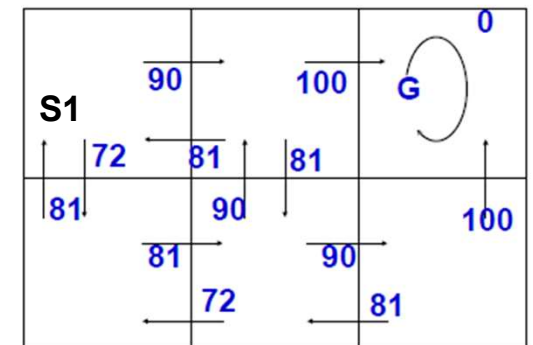
- Example: Find your way to the goal (assuming  $\gamma = 0.9$ )



$R(s, a)$ : reward values



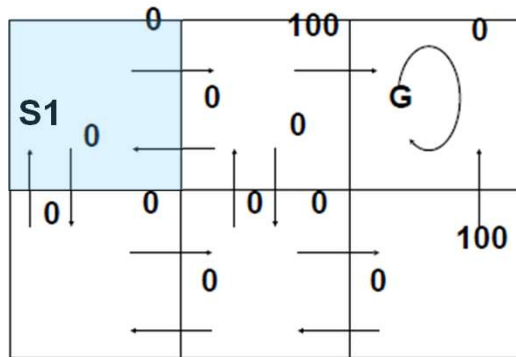
$V^*(s)$  values



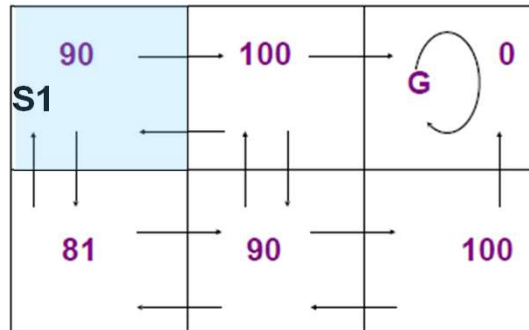
$Q(\text{state}, \text{action})$  values

# Q-Learning – Example

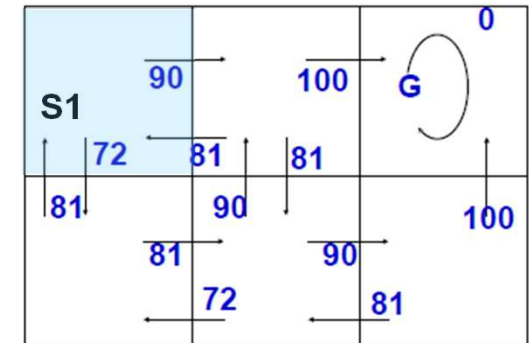
Executing rightward move from state **S1**:



$R(s, a)$ : reward values



$V^*(s)$  values



$Q(\text{state}, \text{action})$  values

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

$$Q(s1, a_{\text{right}}) \leftarrow 0 + 0.9 \max_{a'} \{81, 81, 100\}$$

$$Q(s1, a_{\text{right}}) \leftarrow 90$$

# Deep Q-Learning

- If the state space is large, these techniques are intractable
- If the goal is to learn  $Q(s, a)$ , we could use techniques from **supervised learning**
- If the  $Q$  function is approximated by a neural network, the correctness guarantees for  $Q$ -**learning no longer apply** (Learning might converge poorly or not at all)
- Experience replay:
  - Every time a *state-action* pair is explored by the  $Q$ -learner, that pair is added to a **replay set** with its corresponding reward and transition
  - At each iteration, the replay set is sampled and the samples are used to update the weights of the neural network

# Deep Q-Learning

Choose an initial  $\theta$  for  $Q(*, * | \theta)$ , an initial state  $s$ , and an empty replay set  $R$

Repeat until convergence:

Choose an action  $a$  for the current state  $s$  based on  $Q(s, * | \theta)$

Take action  $a$  and observe the reward  $r$  and the new state  $s'$ , add  $(s, a, s', r)$  to  $R$

Sample  $S \subset R$ :

For each element in  $S$ , set  $y_{(s,a,s',r)} = r + \gamma \max_{a'} Q(s', a' | \theta)$

Perform one step of gradient descent starting at  $\theta$  on

$$\sum_{(s, a, s', r) \in S} \left( Q(s, a | \theta) - y_{(s,a,s',r)} \right)^2$$

Update  $\theta$  based on the gradients

$s = s'$

# Readings

- **Pattern Recognition and Machine Learning, by Bishop – Chapter 13**