



# Reinforcement Learning

## Part 1: Introduction





# Learning objectives

Having completed this topic, you should be able to ...

- Explain concepts of reward-based learning & reinforcement learning
- Define terms such as partially/fully observable; stochastic/deterministic; benign/adversarial; discrete/continuous
- Define the Markov property, Markov chains & MDPs
- Describe & implement an algorithm to find optimal policy for MDPs
- Describe & implement reinforcement learning algorithms
- Analyse problems to determine how they can be framed in terms of reinforcement learning
- Discuss some classic applications of reinforcement learning



# Overview of topic

1. Introduction and learning objectives
2. Motivation
3. General principles
4. Markov decision processes
5. The Q-learning algorithm
6. Q-learning worked example

*N.B. this topic is a very brief introduction to the field of RL. A decent textbook such as “Reinforcement Learning: An Introduction” by Sutton and Barto is a good starting point if you would like to learn more. A .pdf copy of the Sutton and Barto book is available for free online.*





# Reinforcement Learning

## Part 2: Motivation



# What is an agent?

“Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators”

(Russell & Norvig, 2009)

```
public class TcpClientSample
{
    public static void Main()
    {
        byte[] data = new byte[1024]; string input, stringData;
        TcpClient server;
        try{
            server = new TcpClient("...", port);
        }catch (SocketException){
            Console.WriteLine("Unable to connect to server");
            return;
        }
        NetworkStream ns = server.GetStream();
        int rcv = ns.Read(data, 0, data.Length);
        stringData = Encoding.ASCII.GetString(data, 0, rcv);
        Console.WriteLine(stringData);
        while(true){
            input = Console.ReadLine();
            if (input == "exit") break;
            newchild.Properties["ou"].Add(
                "Auditing Department");
            newchild.CommitChanges();
            newchild.Close();
        }
    }
}
```

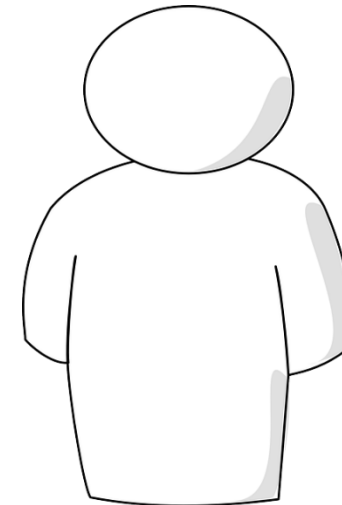
A Computer Program



A Computer



A Robot



A Person



# What is an agent?

- Agents are typically designed to solve decision making problems. These can be “single shot” or sequential – we will focus on sequential problems.
- How does an agent interact with the world?
  - Sensors: to sense the state of the world
  - Actuators: to interact with the world
  - Policy: specifies how an agent should act, based on the perceived state of the world



# What is a state?

- The sensory information available to an agent at a particular timestep
- Agent's state information may be limited (e.g. cannot sense items hidden by a “fog of war” in RTS games)
- Examples of state information:
  - Position (x,y,z coordinates), fuel remaining (robotics, autonomous vehicles)
  - Positions of pieces on a game board (e.g. draughts, backgammon, chess, go)
  - Positions of allies/enemies/powerups (FPS games)
  - Current amounts of resources available (RTS games)
  - Distributions of vehicles queueing at an intersection (traffic signal control)
  - Current demand level/power output (electricity generation scenarios)



# What is an action?

- Actions carried out by an agent change the environmental conditions
- Actions may not always be successful and may not always have the same outcome!
- Examples of actions:
  - Change an agent's/vehicle's/robot's position (e.g. move north/south, up/down)
  - Pick up an object (e.g. ammo, weapons, powerups)
  - Shoot an enemy (FPS games)
  - Build a new unit or structure, research a technology (RTS games)
  - Change the lights at an intersection (traffic signal control)
  - Change the power output of a generator (electricity generation scenarios)





# What is a policy?

- A policy is a mapping from environmental states to actions
- An agent's policy determines how it will act for a given sensory input
- Policies can easily be hand-coded for agents in simple environments (e.g. finite state machines have been widely used in video games)
- More difficult to hand-code policies for complex environments; becomes time consuming. We may not always know the correct action, but we usually know whether a task is being performed well (e.g. high scores in a video game are good).
- Rather than explicitly programming all possible agent behaviour, could instead create agents that can learn how to act (near) optimally
  - **This is the motivation for using reinforcement learning (RL)!**
  - Policies can be generated using many different techniques, e.g. RL, planning, evolution, metaheuristic algorithms, imitation learning etc. We will focus on RL.





# Reinforcement Learning

## Part 3: General principles





# What is Reinforcement Learning (RL)?

- RL agents learn by interacting with their environment
- Simple analogy – training a pet
- Desirable behaviour is rewarded and undesirable behaviour is punished when training a pet – similar principles apply when training an RL agent
- RL is inspired by concepts in the behaviourism literature (reinforcement)

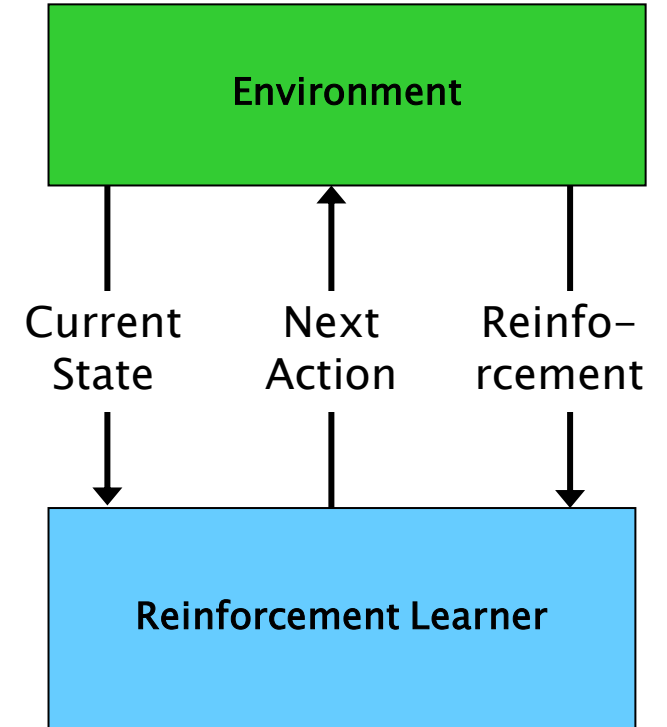


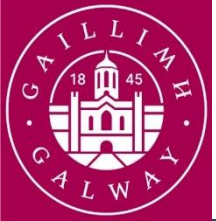




# RL overview (1)

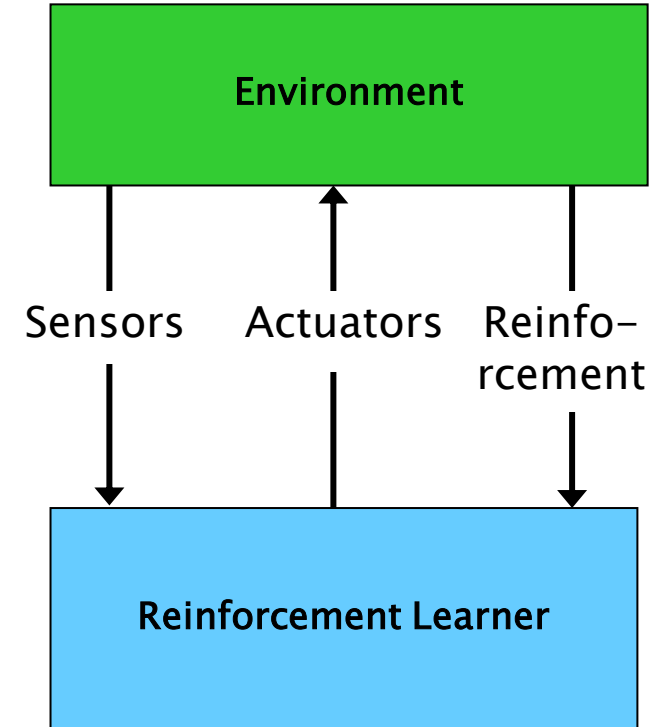
- Learning through trial and error
  - Agent **explores** environment
  - **Rewards** for successful outcomes
  - Punishments for unsuccessful ones
- Perception and action
  - State corresponds to what is perceived
- May Be:
  - Fully or Partially Observable [camera points forward]
  - Discrete or Continuous
  - Benign or Adversarial
  - Deterministic or Stochastic [discussed later]

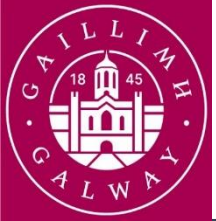




## RL overview (2)

- Sensors and Actuators
- Seeks to maximise long-term reward
  - Acts randomly at first
  - Builds map of states+actions -> rewards
  - Gradually develops optimal strategy
  - Rewards may be **delayed** (not immediate)

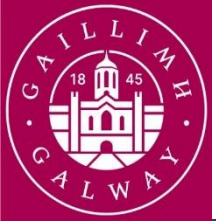




# RL is appropriate when ...

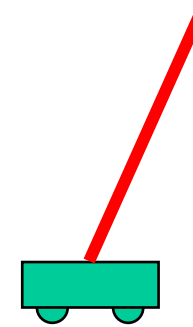
- Tasks are poorly defined
- Domain is not fully known
- Don't know immediate effect of all actions
- Don't know **how** task is done well, just **whether** it is done well





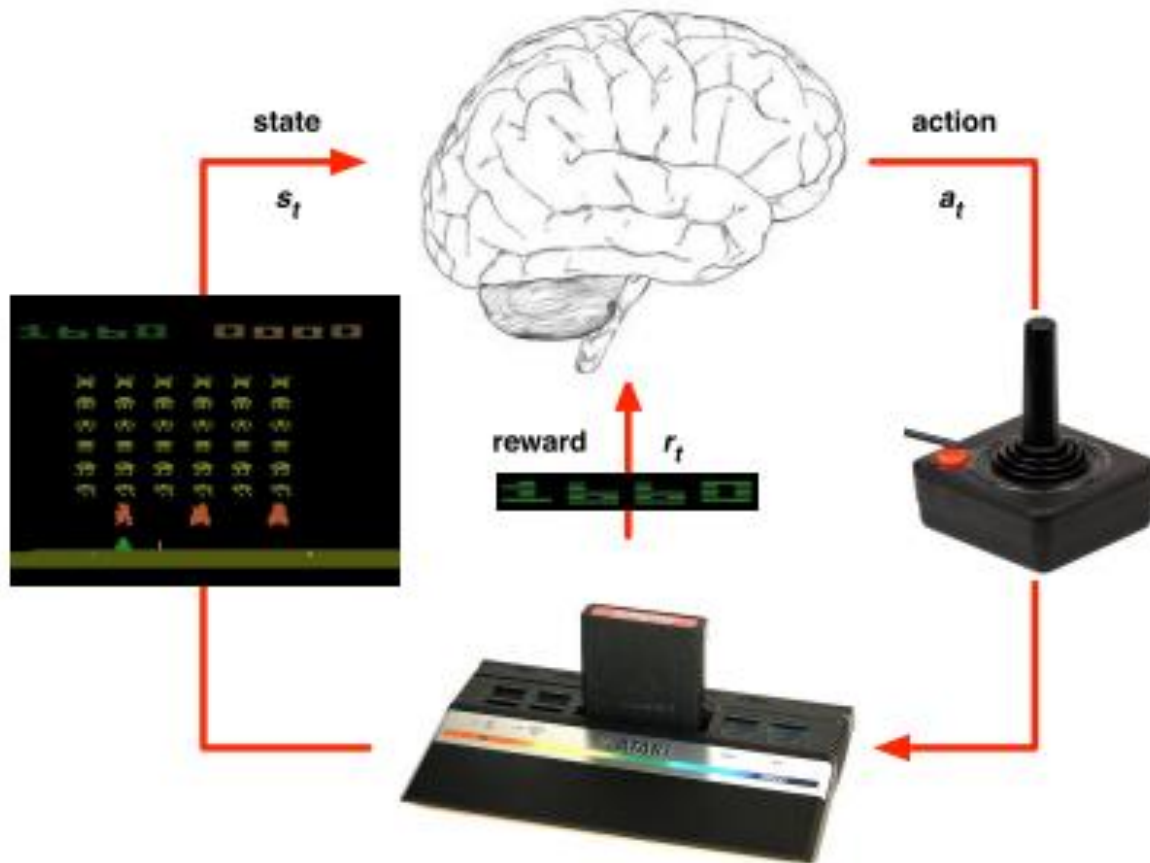
# Classic applications

- Pole-Balancing Robot [Michie, 1968 & others]
  - Inverted pendulum
  - Actions: left, right, pause [fixed vel.]
- TD-Gammon [Tesauro, 1995]
  - Learned to play backgammon from self-play
  - States: Simple & hi-level descriptions of positions
  - Rewards: Win=+100, Lose=-100, Others=0
  - Over 1 million games against itself [small % of state space]
  - Plays at human expert level





# Recent applications of RL (Deepmind)



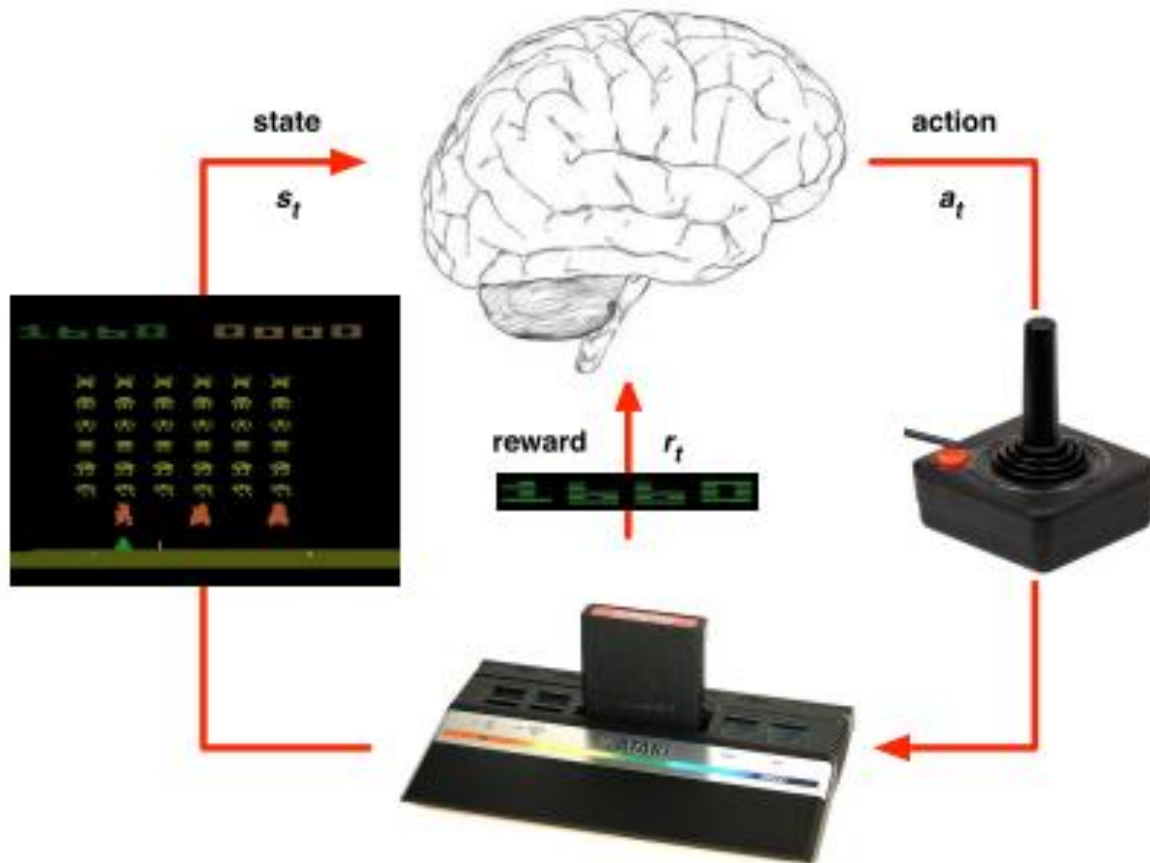
Mnih et al. (2015)



Silver et al. (2017)



# Recent applications of RL (Deepmind)



Mnih et al. (2015)



Silver et al. (2017)





# Reinforcement Learning

## Part 4: Markov decision processes





# Andrey Markov

## Andrey Markov

- 14 June 1856 – 20 July 1922
- Russian mathematician
- Best known for work on theory of stochastic processes
- Subject of his research later became known as Markov chains and Markov decision processes





# Terminology: Deterministic/Stochastic

- Deterministic:
  - Next state of environment is uniquely determined by the current state and the executed action
- Stochastic
  - Inherent level of uncertainty
  - Executed action may result in different states
  - E.g. wheel slip, external forces, opponent, dice
  - Or uncertain due to laziness/ignorance





# Terminology: Markov Property

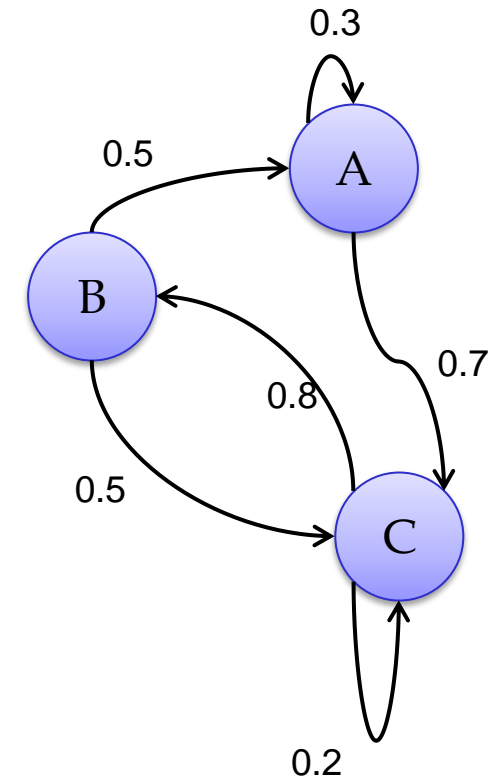
- Stochastic process has the **Markov Property** if its next state depends only upon its current state, not any events that preceded current state
  - "Memoryless"
  - i.e. conditional probability distribution of next state of the process depends only upon the present state
- Draughts example:
  - Current State given by current board layout
  - Next state unaffected by past states
  - $\therefore$  all board states have the Markov property





# Terminology: Markov Chain

- A Markov chain is a stochastic process with the Markov property.
- A Markov chain can undergo transitions from one state to another, where the transition between states is determined by a particular probability distribution.





# Markov Decision Process

- MDP is an extension of Markov chains with 2 additions:  
(1) choosing actions; (2) rewards
- Choice:
  - at every step, agent can choose what action they want to take
  - The result of actions is still stochastic
  - In state  $s$ , any available action  $a$  can be chosen
  - Result is a new random state  $s'$ , with probability depending on  $s$  and  $a$
- Reward:
  - The agent receives a reward based on the old state, action taken, and new state
  - The goal of the MDP is to maximise the expected discounted sum of rewards





# MDP Formal Definition

A Markov decision process is a tuple  $\langle S, A, T, R \rangle$  consisting of:

- A set of states  $S$
- A set of actions  $A$
- A transition function  $T(s, a, s')$ :
  - model of the agent's environment.  $T$  gives the probability of reaching state  $s'$  when selecting action  $a$  in state  $s$
- A reward function  $R(s, a, s')$ :
  - specifies the scalar reward received when action  $a$  is selected in state  $s$ , and the environment transitions to  $s'$
  - Rewards may also be based on reaching a state, e.g.  $R(s)$

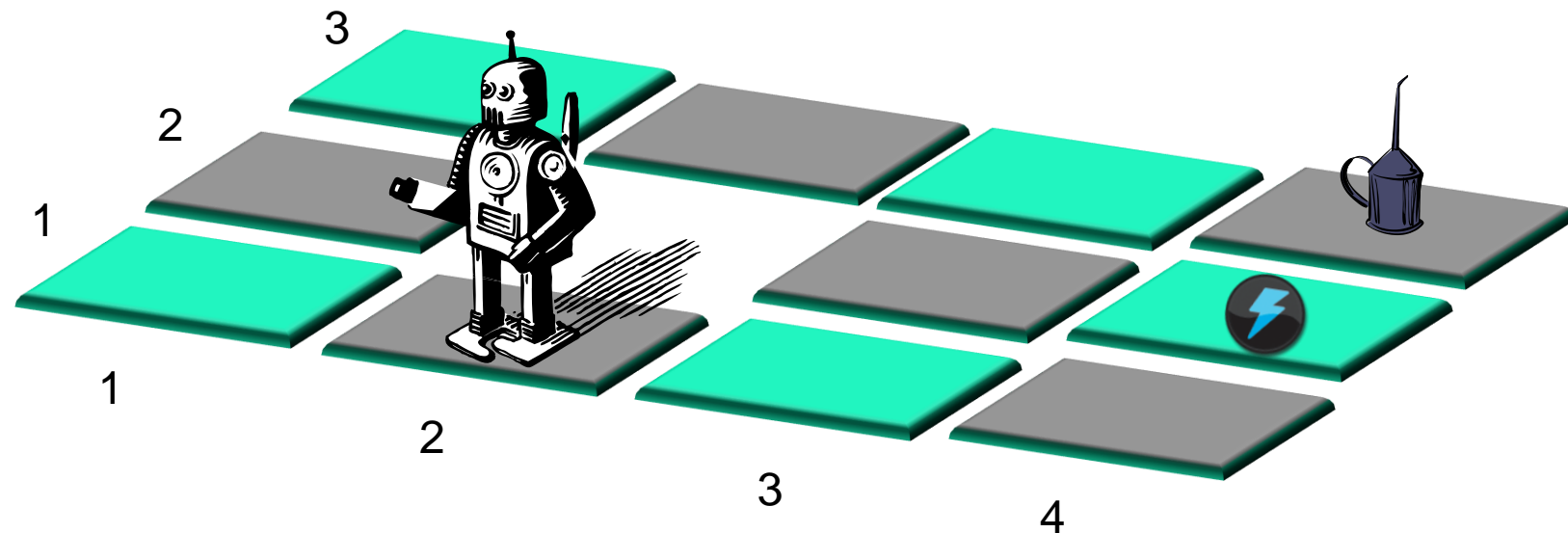
# Simple MDP example

4x3 environment with 11 states [and (2,2) unavailable]  $\rightarrow |S| = 11$

Reward +1 in (4,3); negative reward -1 in (4,2). These are terminal states.

Could include cost for each move e.g. -0.04; encourages agent to solve task in fewer timesteps.

4 actions: North, South, East, West  $\rightarrow |A| = 4$

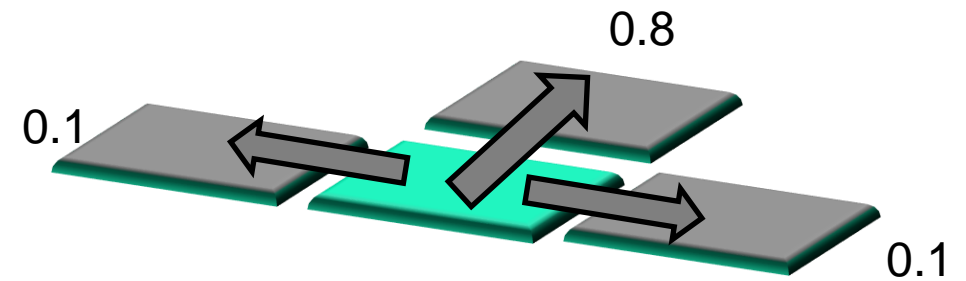


Example based on Russell & Norvig, Ch. 17  
Diagram by Dr Ted Scully.

# Simple MDP example

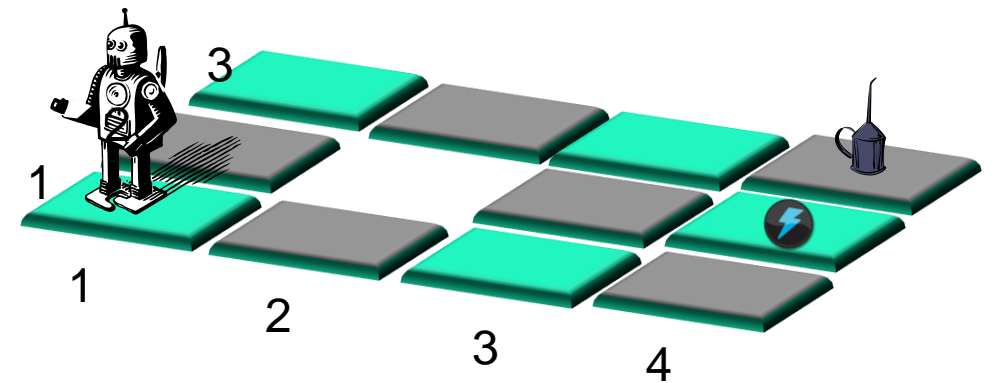
Actions are stochastic:

Take desired action with  $p=0.8$ ; move perpendicular with  $p=0.1$  in each direction (may result in no net movement)



If actions **were not** stochastic, sequence [Up, Up, Right, Right, Right] would lead from (1,1) to (4,3).

Because they **are**, this will work with probability  $(0.8)^5 = 0.32768$ , and probability  $(0.1)^4 \times 0.8$  that these actions end in alternate route to goal  $\Rightarrow$  total is 0.32776.



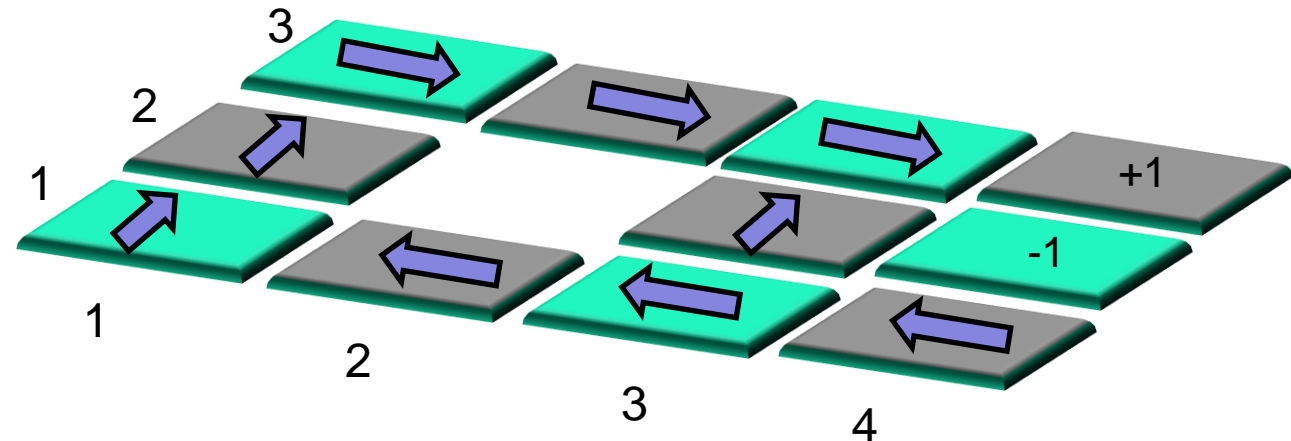




## Policy: Solution to the MDP

- ▶ A policy  $\pi$  represents a solution to an MDP problem; example shown below
- ▶ Defines what action to take in every state
- ▶ Optimal policy  $\pi^*$  is one that yields the highest expected discounted sum of rewards (the highest value to the agent)
- ▶  $V^\pi$   $\rightarrow$  Value of a policy  $\pi$  over a time horizon  $h$  (can have  $h = \infty$ )
  - ▶ Add up all discounted rewards expected during policy execution

$$V^\pi = E \left[ \sum_{t=0}^{t=h} \gamma^t r_t \right]$$





# Bellman Equation

- The value of a state:
  - Immediate reward for that state plus the expected **discounted** value of the next state, assuming that the agent chooses the optimal action
  - Discount factor  $\gamma$  in range 0-1:  
if less than 1, values of future states carry less importance
- Calculated with the Bellman Equation
  - Note that the expected future value is calculated using  $T$ . As well as the value of each possible future state  $s'$ , we also need to consider the likelihood of reaching each  $s'$

$$V(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} T(s, a, s') V(s')$$



# Value Iteration Algorithm

- Iterative algorithm that uses Bellman Equation to calculate the optimal policy for an MDP.

- Simple implementation: see `GridWorld.java`



Sample output:

```
0.8    0.9    0.9    1.0
0.8    0.0    0.7   -1.0
0.7    0.7    0.6    0.4

Best policy:

E     E     E     +
N     #     N     -
N     W     W     W
```

- Initialise reward matrix,  $R$ , for all states
- Initialise  $V'$  for all states to 0
- Repeat until convergence:

$V = V'$

Loop over all states:

- In each state  $s$ :

- $V'(s) := R(s)$  if a terminal state

- $V'(s) :=$  Bellman eqn otherwise (computed using  $V(s)$ , not  $V'(s)$ )





# Limitations of Value Iteration

- Need to know all rewards
- Need to know all transition probabilities
- What if we don't?

## Use Reinforcement Learning!

- Explore an unknown environment
- Try different actions, see what rewards you get, keep going if you get knocked back to the start
- "Playing a new game whose rules you don't know..."



# MDPs in Reinforcement Learning

- Transition model
  - Model of probability of reaching state  $s'$  if you perform action  $a$  in state  $s$
- Markov Decision Process
  - Decision on what action to perform depends on current state only
  - Previous states irrelevant
- In RL context
  - Agent observes state  $s$ , takes action  $a$ :  
Gets from environment reward  $r$ , new state  $s'$
  - MDP  $\Rightarrow$  Transition model and  $r$  are functions of  $s$ ,  $a$  only
- RL tasks are almost always assumed to be **Finite MDP**
  - Finite number of states and actions
  - State description sufficient that the context of previous states is not required





# Defining States, Actions and Rewards

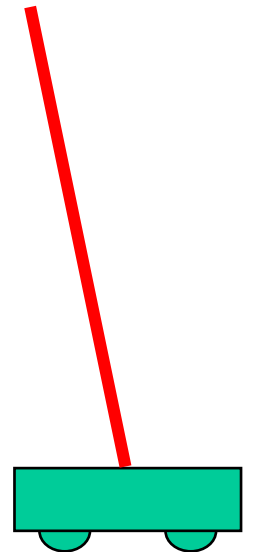
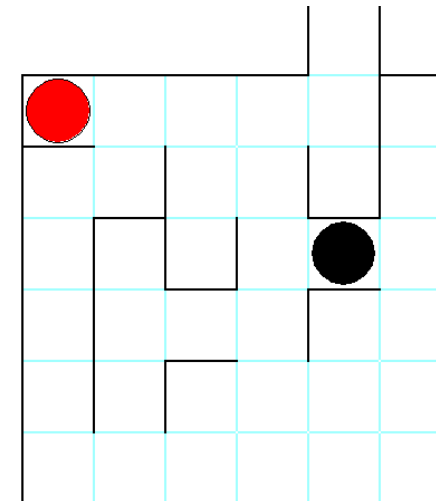
- Defining Actions: usually easy
- Defining Rewards: Must reflect learning goals
  - E.g. shortest path through maze: penalise no. of steps
- Try changing rewards in GridWorld MDP solution:
  - $r = 0$ : take as many steps as possible to avoid -1 pit
  - $r = -0.04$ : minor penalty for each step – prefer shorter paths
  - $r = -0.2$ : life is bad!    $r = -1.7$ : life is horrible!
  - $r = 1$ : life is wonderful!





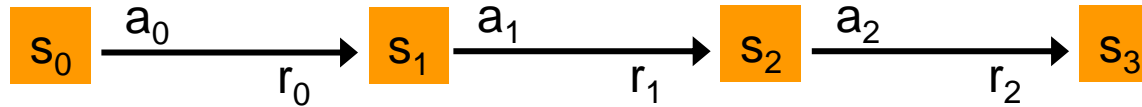
# State Descriptions

- Describing State:
  - Remember MDP assumption
  - Sufficient info so that previous states irrelevant
  - Note: In some applications can only **approximate** MDP
- What state information is needed for these environments to preserve MDP assumption?
  - Simple maze
  - Theseus & Minotaur maze
  - Pole-balancing robot

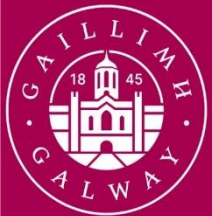




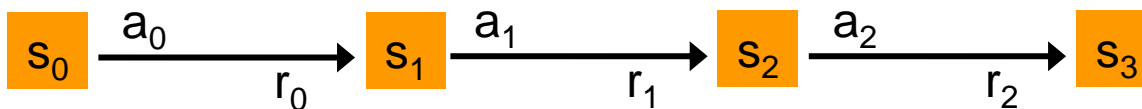
# RL Learning Task (1)



- Basic framework:
  - Agent observes state  $s_t$ , takes action  $a_t$ :  
Gets from environment reward  $r_t$ , new state  $s_{t+1}$
  - Transition function  $T(s_t, a_t) \rightarrow s_{t+1}$  and  
reward function  $R(s_t, a_t, s_{t+1}) \rightarrow r_t$  known by env., not agent
- Need to learn policy function
  - Selects next action given current state:  $p(s_t) \rightarrow a_t$



## RL Learning Task (2)



- Objective:
  - Maximise total **discounted** future reward
  - Total Value of a policy **p** starting from state  **$s_t$** :

$$V^p(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

- $\gamma$  is discount factor:
  - More distant rewards are worth less now
  - Range  $[0,1]$ : should be close to 1 (e.g. 0.9)





# Reinforcement Learning

## Part 5: The Q-learning algorithm





# Q-Learning Algorithm (1)

- Q-Learning: popular RL approach
  - Learns an action-value function:  
 $Q(s,a)$  is value of performing action  $a$  in state  $s$
  - $Q(s,a)$  values estimated from experience
  - Does **not** need to learn transition function or reward fn => “**model-free**” as not attempting to model the environment
- Basic Idea:
  - Construct array, indexed by  $s$  and  $a$ , to hold Q-values
  - In any state, select action with highest Q-value (subject to exploration: later)
  - Tie-breaker: select one at random
  - Q-values initially 0: estimate  $Q(s_t, a_t)$  from immediate reward  $r_t$  and discounted estimated future reward of best action in next state,  $\gamma Q_{\max}(s_{t+1}, a_{t+1})$



## Q-Learning Algorithm (2)

- Q-learning update: (temporal difference update rule)

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

- max: selects action with highest expected value in  $s_{t+1}$
- $\gamma$  = discount factor as before
- $\alpha$  = learning rate: range  $[0,1]$
- Learning Rate:
  - instead of completely discarding old value of  $Q(s_t, a_t)$ , adjust it proportional to  $\alpha$ :
    - Reward/transition might be stochastic
    - $Q(s_{t+1}, a_{t+1})$  is **probably** inaccurate
  - Higher  $\alpha \Rightarrow$  old  $Q(s_t, a_t)$  value less important
  - Often start with high  $\alpha$  and reduce with experience of specific state/action pair:  
 $\alpha = 1/(1 + \text{visits}(s, a))$
  - Deterministic case: can use  $\alpha = 1$





# Q-Learning Algorithm (3)

- Initial stages:
  - All Q-values equal (zero): random actions tried
  - Q-values only updated when first reward (+/-) found
  - Next time adjacent to that state, Q-value affected
- Exploration:
  - Shouldn't always select action with highest Q-value
    - Might have found a sub-optimal solution, giving low reward  $> 0$
  - As with much of AI/ML (and real life!), there is a trade-off between exploiting knowledge, and exploring to find new (possibly better) options
  - Need mechanism to try new actions. One solution:  $\epsilon$ -greedy exploration
    - Choose  $\epsilon$  in range  $[0,1]$ : want low value, e.g. 0.01
    - Each time, pick number  $n$  in range  $[0,1]$
    - If  $n \leq \epsilon$ , choose random action instead of one with highest Q-value



# Q-Learning Algorithm (4)

- The full Q-Learning procedure:
  - For repeated episodes [games] in an environment
    - 1  $\forall s, \forall a: Q(s, a) = 0$
    - 2 Repeat (for each episode):
      - 3 Initialise  $s$  to starting state
      - 4 Repeat (for each step of episode):
        - 5  $a = \arg \max_a Q(s, a)$
        - 6 With probability  $\varepsilon$ :  $a = \text{random action}$
        - 7 Take action  $a$ , observe reward  $r$  and next state  $s'$
        - 8  $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_a Q(s', a) - Q(s, a)]$
        - 9  $s = s'$
      - 10 Until  $s$  is terminal
      - 11 Until stable solution found



# Generalisation in RL

- Q-values table can be very large!
  - Large number of states [e.g. board game] and actions
  - Need to visit all state/actions multiple times to get accurate estimates of Q-values
  - Would be nice to generalise from experience of similar states
- Solution:
  - Replace table with a lookup function implemented using a function approximator
  - Most commonly, feed-forward neural network
  - Used in TD-Gammon and other applications to good effect.
  - Recent Deep RL works use multi-layer ANNs for generalisation



# RL and the Temporal Credit Assignment Problem

- When an agent acts in an environment, and rewards are few and far between, it must take a lot of steps before gaining reward (e.g. at maze exit)
- E.g. just before maze exit, does the final step deserve all the reward?
- RL in general seeks to solve the credit assignment problem by iteratively propagating the influence of delayed reinforcements to all states and actions that led to that reinforcement





# Reinforcement Learning

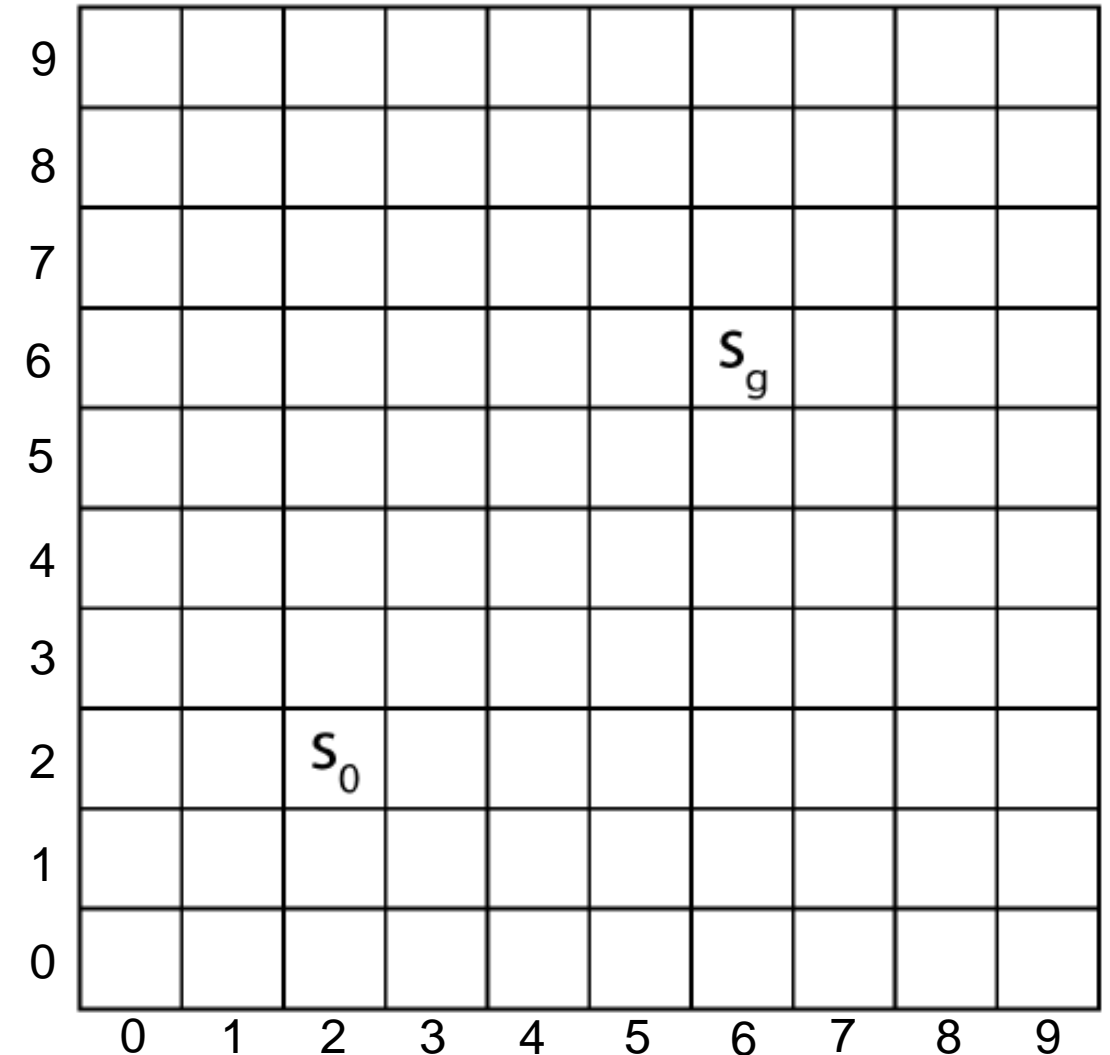
## Part 6: Q-learning worked example

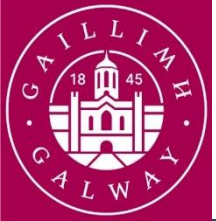




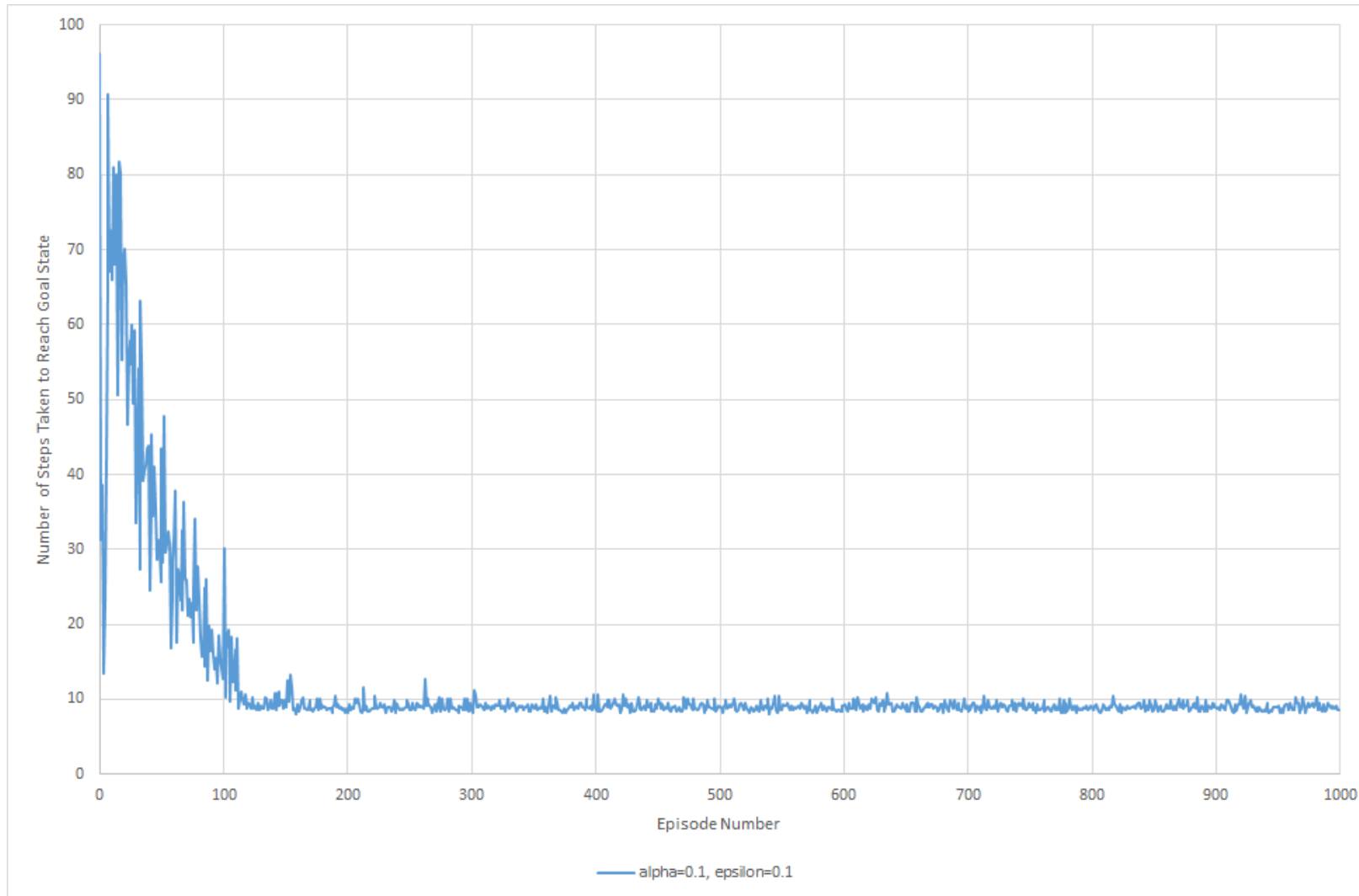
# Worked example: Q-learning in Gridworld

- Gridworld: a single agent starts in an initial position (state)  $s_0$ , and must reach a goal state  $s_g$
- Actions available are to move North, East, South or West
- State-action value estimates (Q values) updated at each timestep
- Episodic – agent learns by trial and error over several episodes
- Sparse reward function – agent is only rewarded when it reaches the goal state
- Rewards: +10 for reaching goal, -1 for all other turns





# Sample learning curve for this problem



Sample code:

[Gridworld\\_Qlearning.zip](#)

Worksheet:

[Gridworld\\_Qlearning.pdf](#)

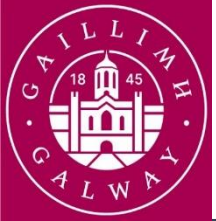
This sample graph shows the steps to goal averaged over 10 runs, with:

$\alpha = 0.1$

$\epsilon = 0.1$

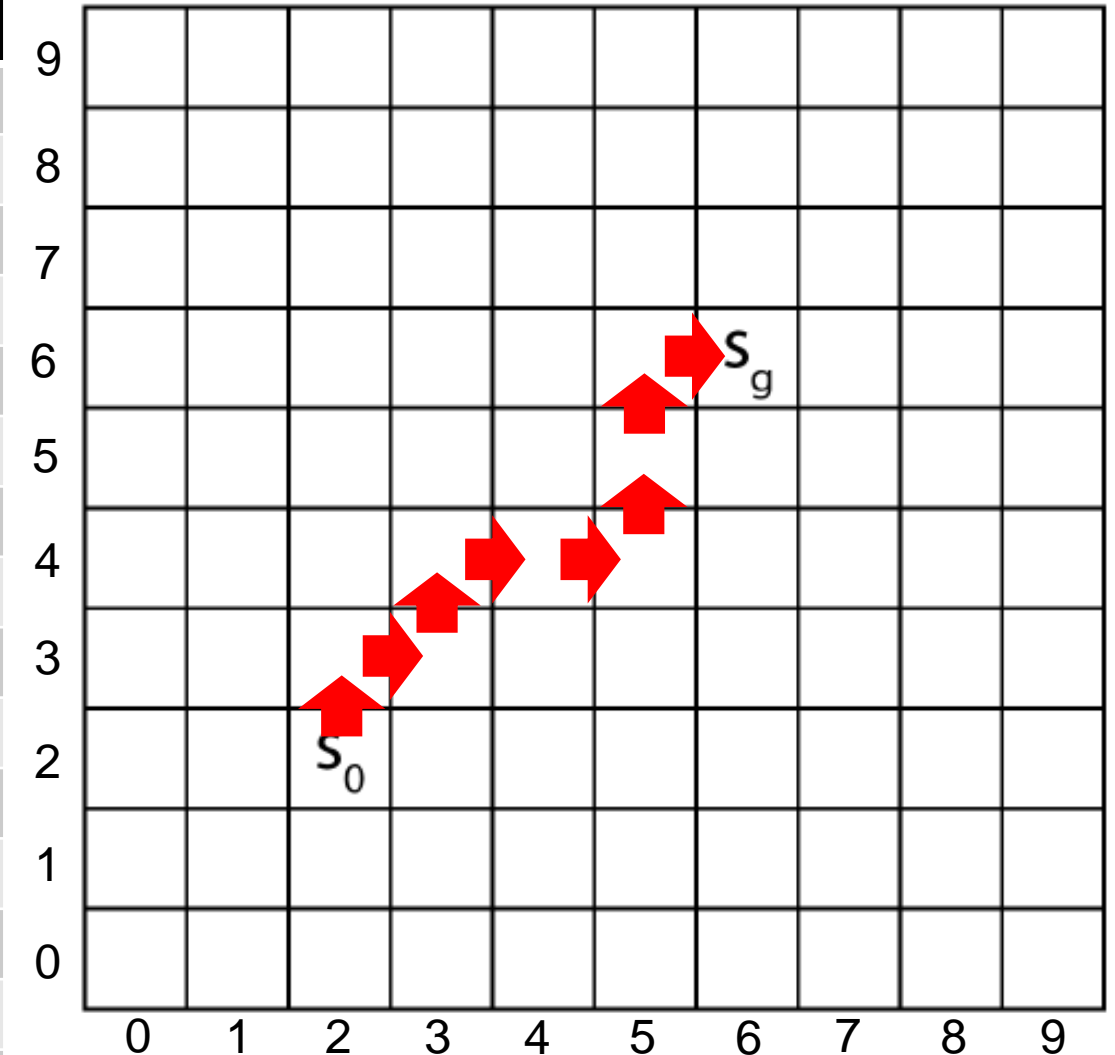
$\gamma = 1.0$

Learning curve flattens out  
=> Convergence!

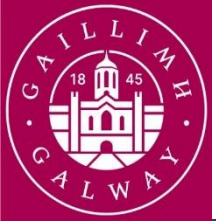


## Q values for a sample optimal path

State (x,y)	North	East	South	West
(0,0)	-2.373	-2.403	-2.397	-2.392
...	...	...	...	...
(2,2)	3.000	2.087	-0.913	-0.600
(2,3)	2.592	4.000	1.524	0.928
...				
(3,3)	5.000	4.323	1.762	2.603
(3,4)	2.701	6.000	3.561	2.851
...				
(4,4)	6.263	7.000	4.506	4.096
...				
(5,4)	8.000	4.219	3.180	5.801
(5,5)	9.000	7.008	6.704	5.840
(5,6)	2.318	10.000	7.511	7.370
...				
(9,9)	-0.200	-0.200	-0.200	-0.200







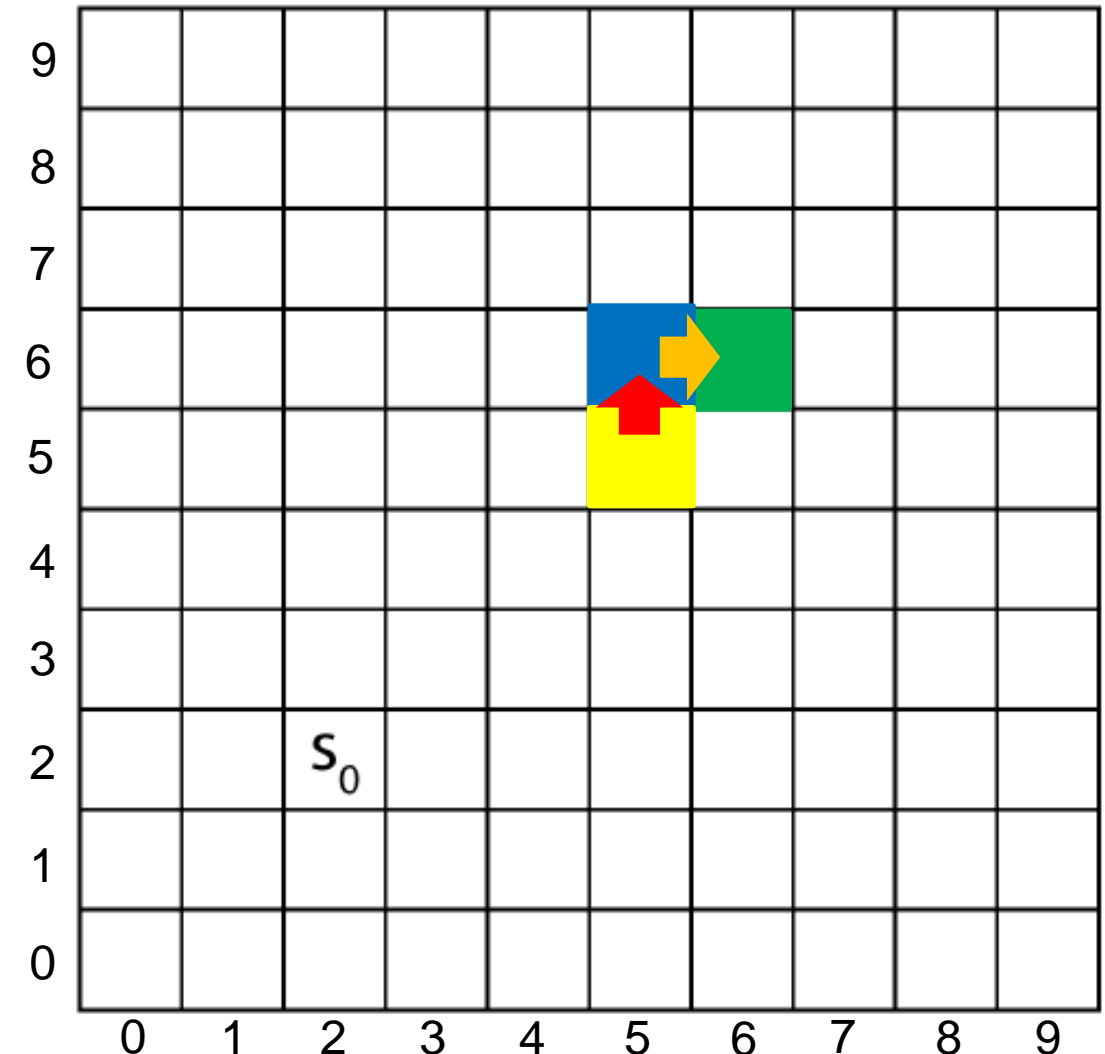




# Example Q-learning process (single timestep)

State (x,y)	North	East	South	West
(5,5)	8.250	7.008	6.704	5.840
(5,6)	2.318	10.000	7.511	7.370

- Initial state  $s$ : (5,5) 
- Max valued action  $a$ : North 
- Next state  $s'$ : (5,6) 
- Reward received  $r$ : -1
- Calculate new Q value  $Q(s,a)$  for  $Q((5,5), \text{North})$  using max valued action in next state  (next slide)



NewQ = 8.325      OldQ = 8.25      Reward Received = -1      Max Q value in next state = 10      OldQ = 8.25

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha [r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Learning rate = 0.1      Discount factor = 1.0

State (x,y)	North	East	South	West
(5,5)	8.250	7.008	6.704	5.840
(5,6)	2.318	10.000	7.511	7.370

Q table at beginning of timestep

State (x,y)	North	East	South	West
(5,5)	8.325	7.008	6.704	5.840
(5,6)	2.318	10.000	7.511	7.370

Q table at end of timestep



## Final remarks

- RL is an area of active research; not as mature as other ML paradigms
- Problems with using RL in the real world:
  - e.g. trial and error learning using expensive equipment/robots can be expensive and dangerous). Could train agent in simulation first, then refine in real hardware
  - Ethical considerations, validating behaviour/compliance with laws and standards
- Current research topics include:
  - Dealing with sparse reward functions, curse of dimensionality, sample complexity
  - Multi-Agent Reinforcement Learning (MARL)
  - Integrating domain knowledge (e.g. advice from a human expert)
  - Safe and explainable decision making