Mahidol University
Faculty of Medicine Ramathibodi Hospital
Department of Clinical Epidemiology and Biostatistics

# Reinforcement learning :
# Q-learning

## RADI608: Data Mining and Machine Learning

## RADI602: Data Mining and Knowledge Discovery

**Lect. Anuchate Pattanateepapon, D.Eng.**

Section of Data Science for Healthcare

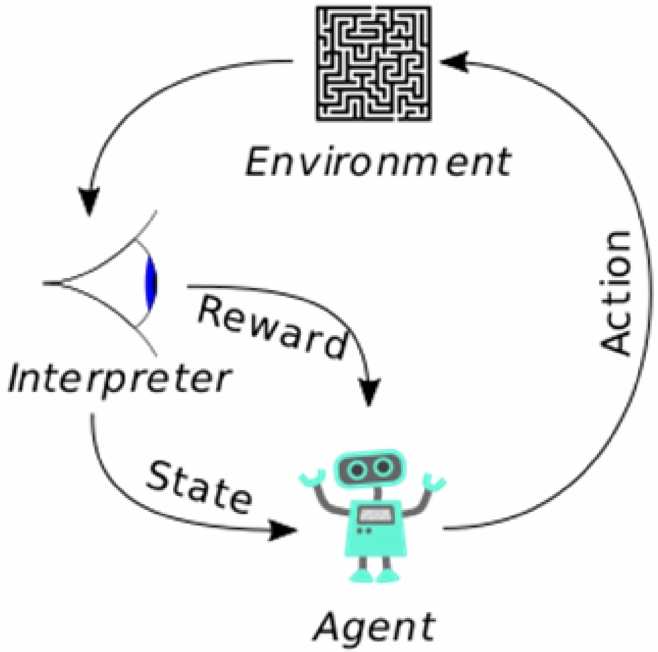
Department of Clinical Epidemiology and Biostatistics

Faculty of Medicine Ramathibodi Hospital, Mahidol University



Wisdom of the Land

# What's a reinforcement learning?

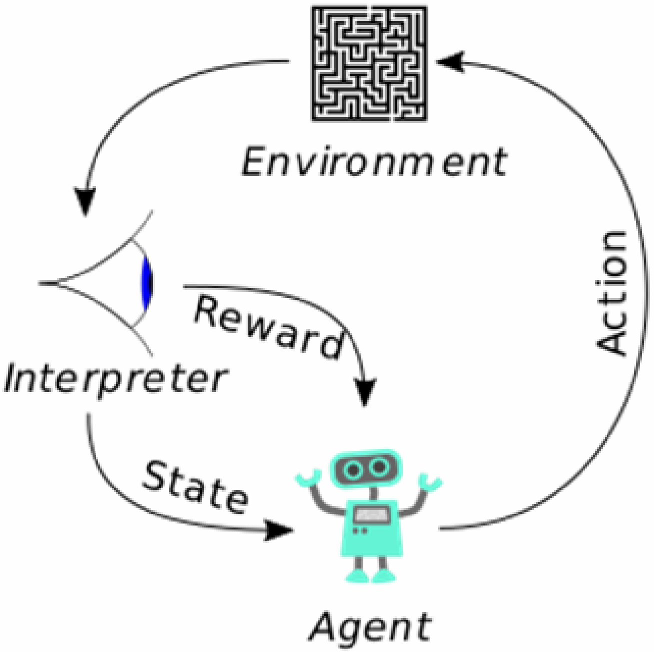



(https://en.wikipedia.org/wiki/Reinforcement_learning

# What's a reinforcement learning?

Reinforcement learning is the science of decision making

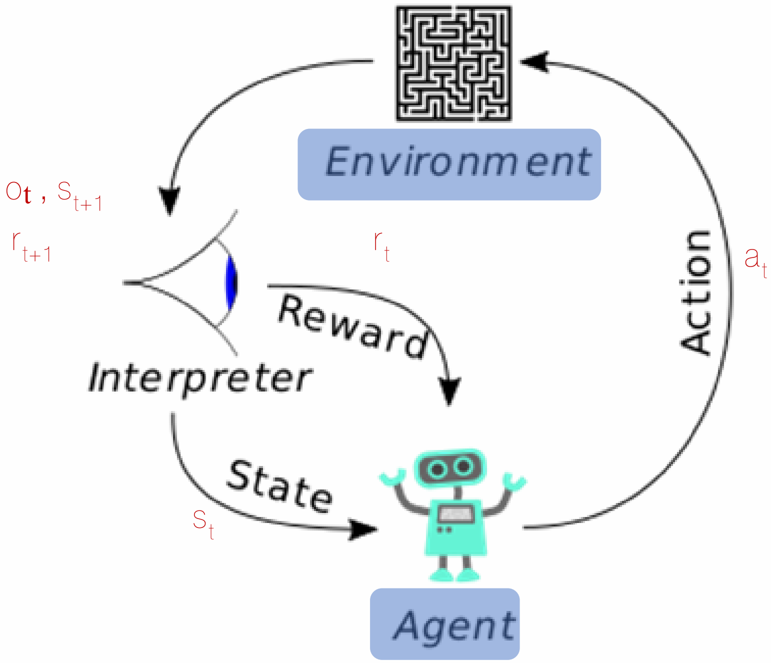
The agent needs to find the "right" actions to take in different situations to achieve its overall goal




Learning: The environment is initially unknown and the agent needs to interact with the environment to improve its policy

Planning: If the environment is known the agent performs computations with its model and then improves its policy

# What's a reinforcement learning?




Agent exploits the environment and makes the decisions on which actions to take at each time step t.

Each action ($a_t$) that agent can do and effect the next data it receives.

Reward ($r_t$) is a scalar feedback signal which indicates how well the agent is doing at step time t.

State ($s_t$) is the info used to determine what happens next

Environment (e) receives the action from the agent and emits a new observation $O_{t+1}$ and scalar reward $r_{t+1}$

Policy ($\pi_\theta$): Agent's behaviour function which is a map from state to action.

Value Function: represents how good is each state and/or action. It is a prediction of future reward

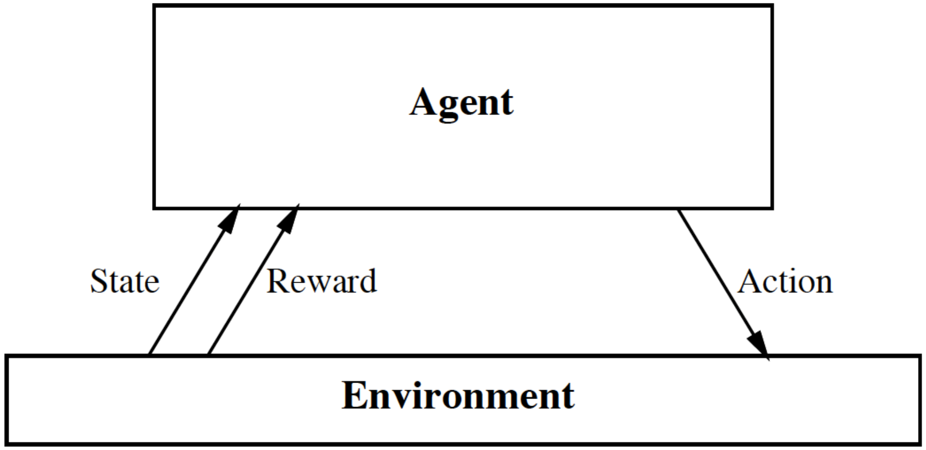
# What's a reinforcement learning?

- An algorithm that designed by concerning software-agent actions in an environment with long-term reward maximization (reinforce good actions).
- The agent learns from the training data through good and bad action that it take and received a scalar feedback (a number called reward).
- RL deals with agents that must sense & act upon their environment.
- This is combines classical AI and machine learning techniques.
- It the most comprehensive problem setting.

# Examples of the reinforcement learning

- A robot cleaning my room and recharging its battery
- Robot-soccer
- How to invest in shares
- Modeling the economy through rational agents
- Learning how to fly a helicopter
- Scheduling planes to their destinations
- and so on

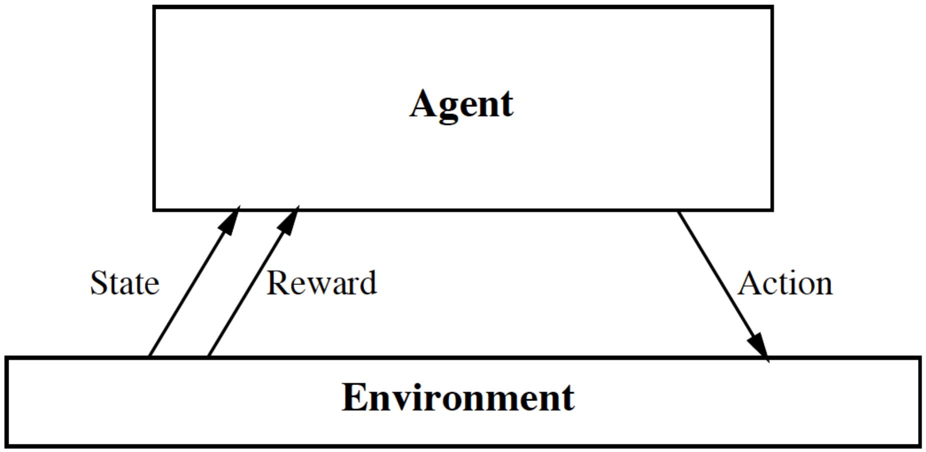
# The Big Picture




$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \cdots$$

Your action influences the state of the world which determines its reward

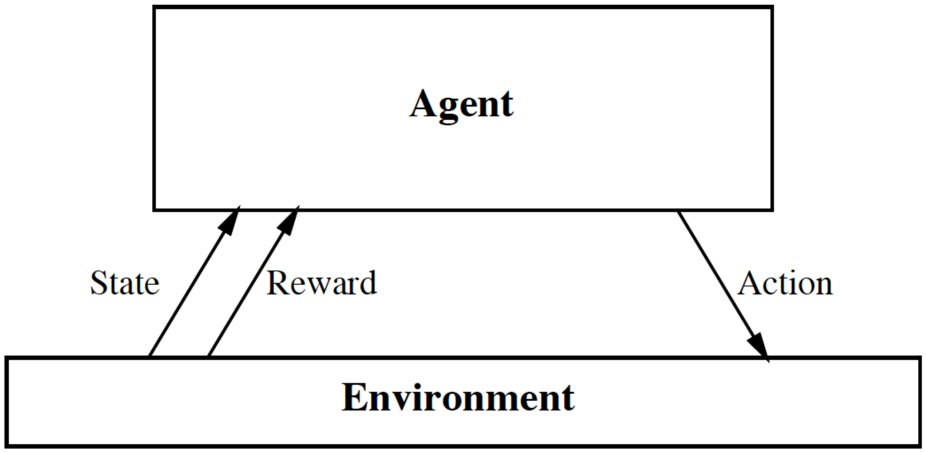
# The Big Picture

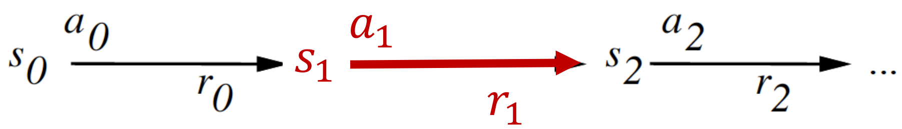



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \dots$$

Your action influences the state of the world which determines its reward

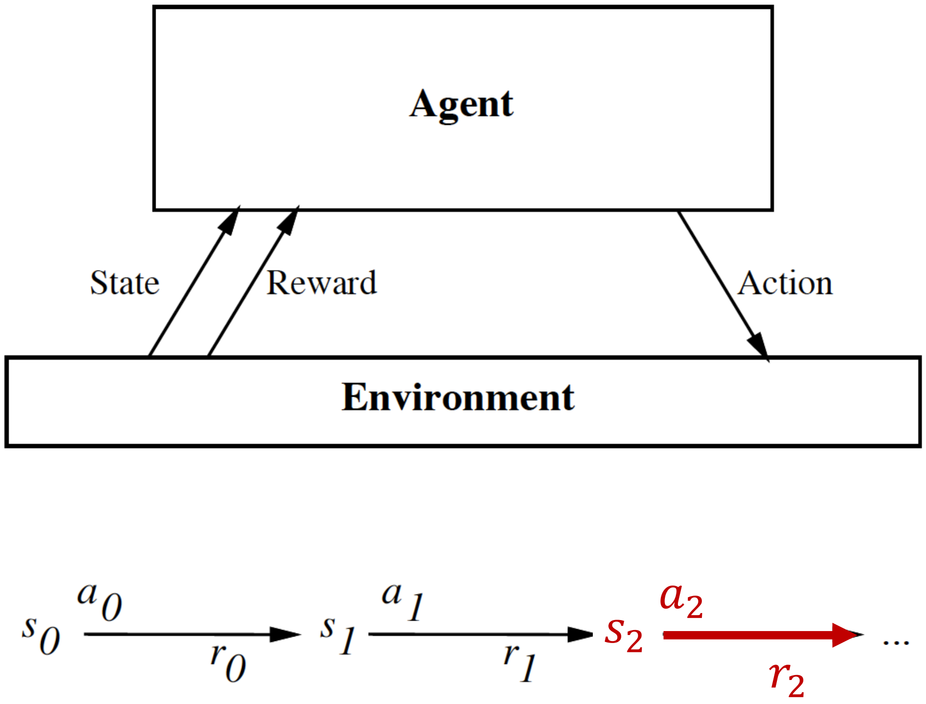
# The Big Picture



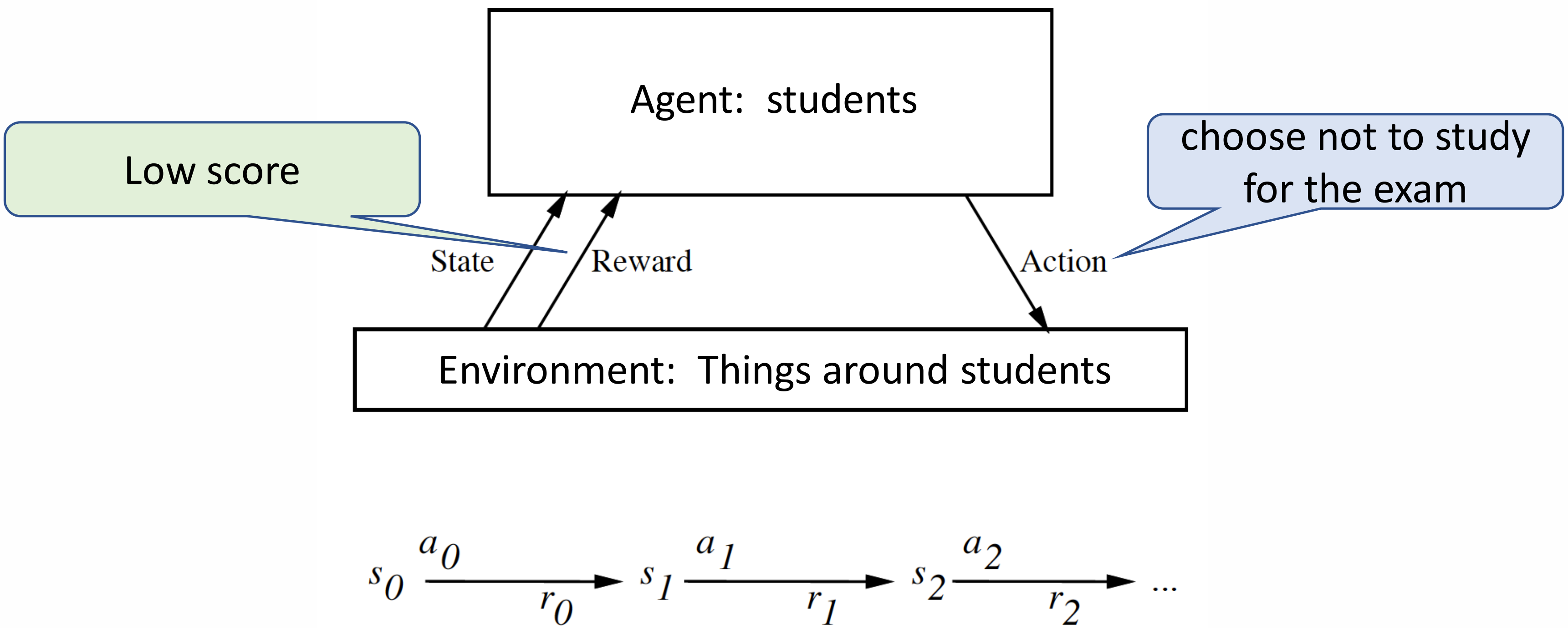

Your action influences the state of the world which determines its reward

# The Big Picture

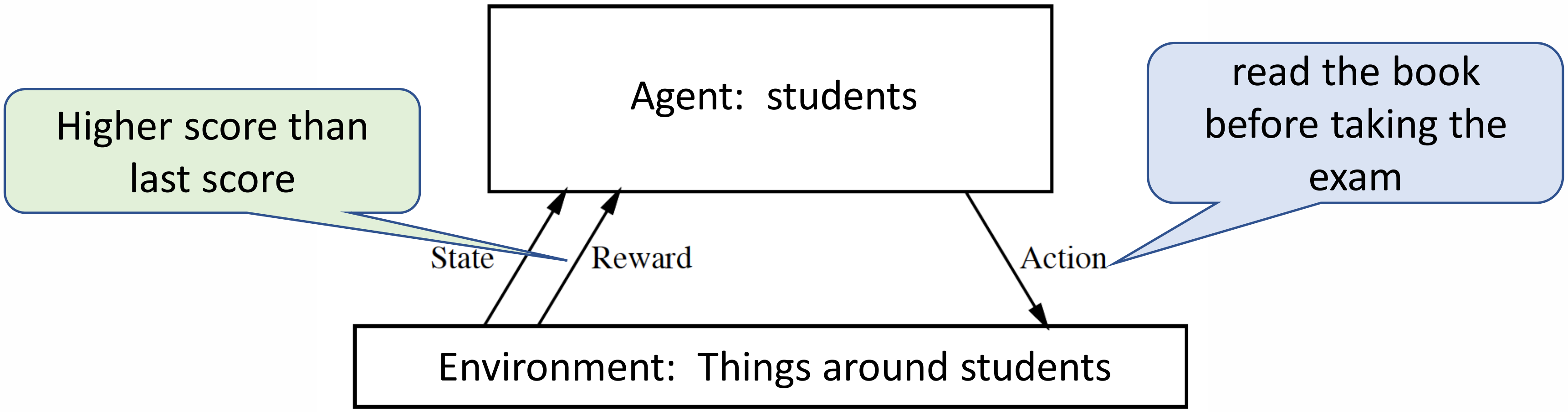



Your action influences the state of the world which determines its reward

# Student who read books to prepare for the exam



# Student who read books to prepare for the exam




$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \ldots$$

# Action and Reward

Sometimes human (Agents) do not always choose to do something (Action) based on the expected reward (Reward) at that moment.

Future outcomes may also be taken into account. By reading a book may make students less stressed and happier than playing games. But, the long-term effects book reading give more benefit than playing a game (good grade, good job opportunity)

Therefore, the sum of present to future rewards (Cumulative Reward ($G_t$)) may be used to calculate the result instead.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots = \sum_{k=0}^{\infty} R_{t+k+1}$$

# Action and Reward

Cumulative Reward : $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots = \sum_{k=0}^{\infty} R_{t+k+1}$

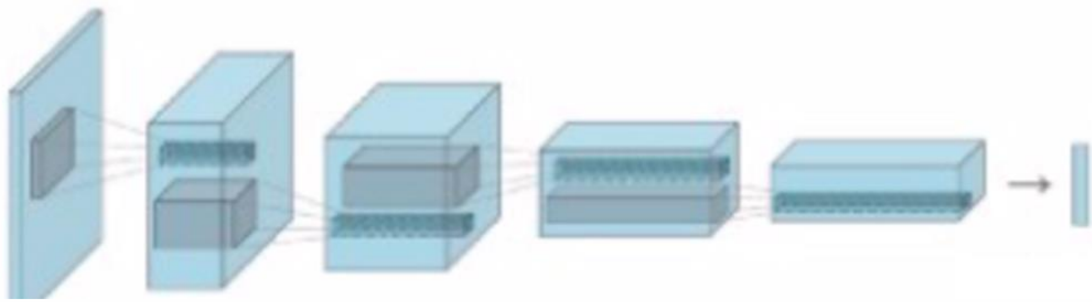
For us to be able to calculate cumulative rewards it is necessary to reduce the importance of distant rewards in the future. It uses something called a Discount Factor ($\gamma$) between 0 and 1, which helps to limit it.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R^{t+3} + \cdots = \sum_{k=0}^{\infty} R_{t+k+1}$$

After the Agent evaluates the current situation through Cumulative Rewards, it is time for the Agent to decide which action to take. That determines what action the Agent should take in the situation (state) according to the policy that they choose to use.

# What is a policy?

The policy is simply a function that maps states to the actions, this policy can be simply approximated using neural networks ( with parameters $\theta$ ) which is also referred to as a functional approximation in traditional RL theory .

State/ observation  Action

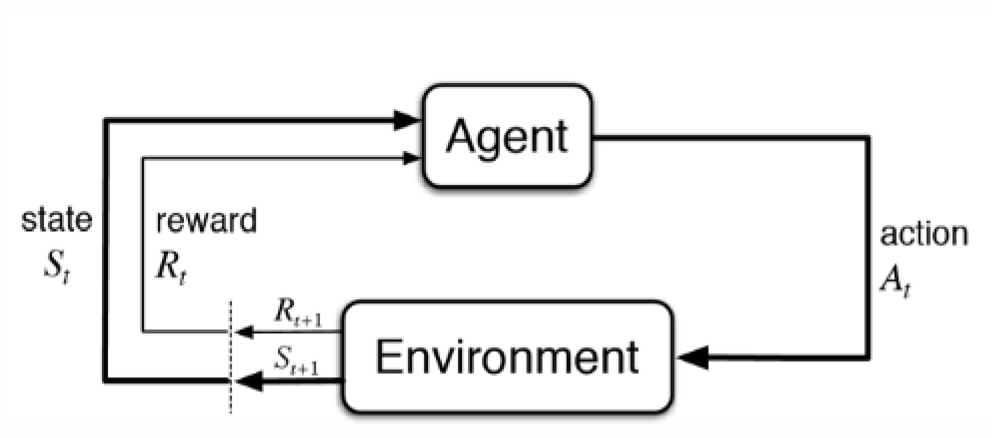
$$\pi_{\theta}(a \mid s)$$

The final goal in a reinforcement learning problem is to learn a policy, which defines a distribution over actions conditioned on states, $\pi(a \mid s)$ or learn the parameters $\theta$ of this functional approximation.

# How Policy is Trained

The process of reinforcement learning involves iteratively collecting data by interacting with the environment. This data is also referred to as experiences in RL theory. It is easy to appreciate why data is called experience if we understand the interaction of an agent with the environment.

Traditionally, the agent observes the state of the environment (s) then takes action (a) based on policy $\pi$(a | s). Then agent gets a reward (r) and next state (s’). So collection of these experiences (<s,a,r,s’>) is the data which agent uses to train the policy ( parameters $\theta$).



# On policy vs Off policy reinforcement learning

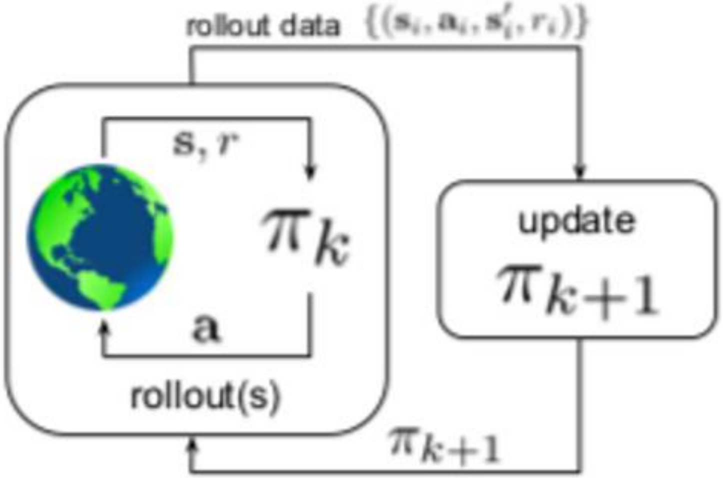
The performance of Reinforcement learning models for hyperparameter optimization is evaluated via on-policy interactions with the target environment. These interactions of An on-policy learner help get insights about the kind of policy that the agent is implementing.

An off-policy, whereas, is independent of the agent's actions. It figures out the optimal policy regardless of the agent's motivation.

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state.

# On policy reinforcement learning

Typically the experiences are collected using the latest learned policy, and then using that experience to improve the policy. This is sort of online interaction. The agent interacts with the environment to collect the samples.




Behavior policy == Policy used for action selection (Target Policy)

# Example of on policy RL

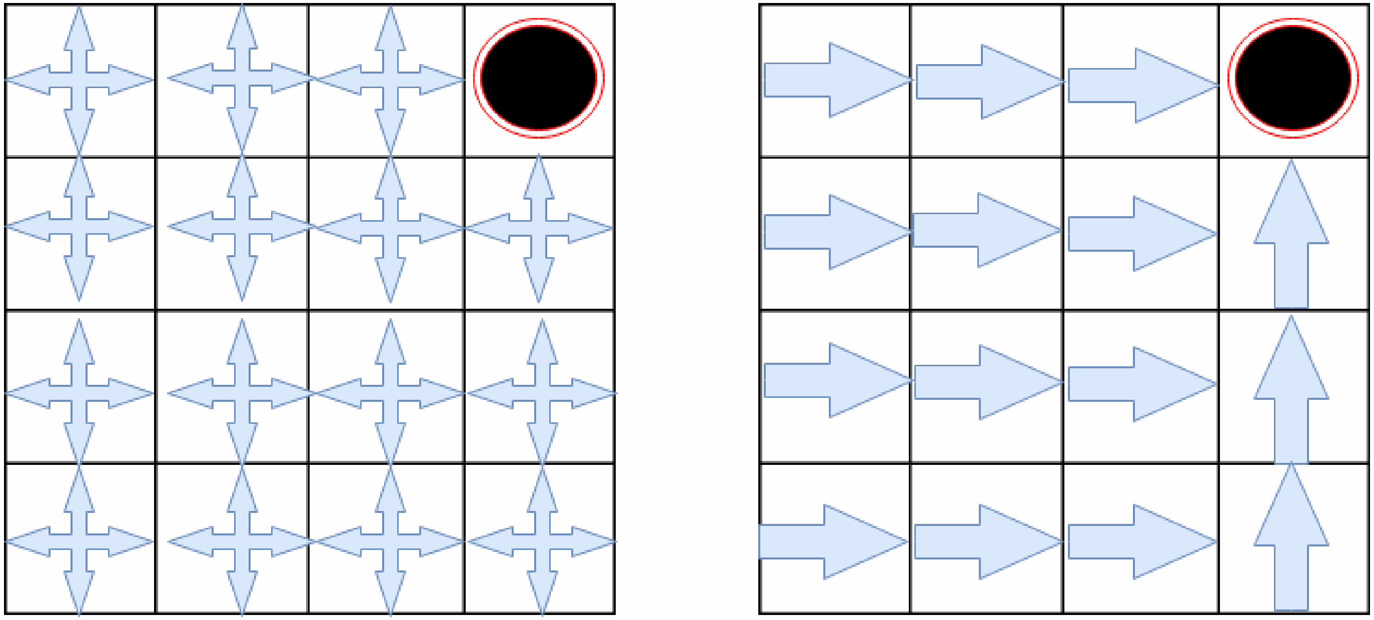
SARSA (state-action-reward-state-action) is an on-policy reinforcement learning algorithm that estimates the value of the policy being followed. In this algorithm, the agent grasps the optimal policy and uses the same to act. The policy that is used for updating and the policy used for acting is the same, unlike in Q-learning. This is an example of on-policy learning.

$\langle S, A, R, S', A' \rangle$

current state S, current action A, reward R, new state S', future action A'.

# Example of on policy RL

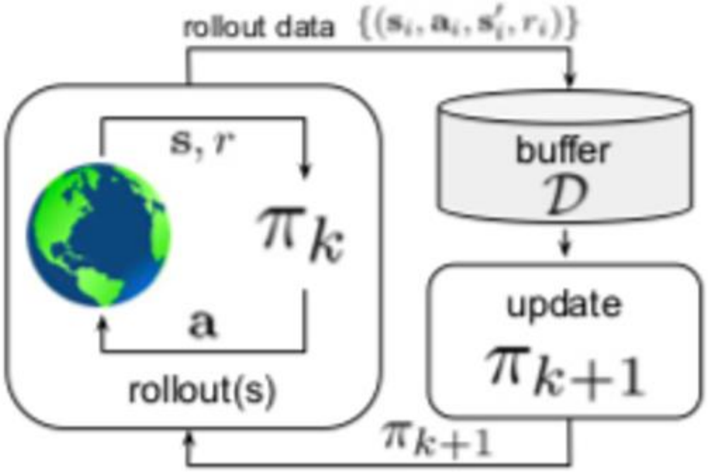
Behavior policy == Policy used for action selection (Target Policy)



**Behavior Policy** **Target Policy**

# Off policy reinforcement learning

In the classic off-policy setting, the agent's experience is appended to a data buffer (also called a replay buffer) D, and each new policy $\pi_k$ collects additional data, such that D is composed of samples from $\pi_0, \pi_1, \ldots, \pi_k$, and all of this data is used to train an updated new policy $\pi_{k+1}$. The agent interacts with the environment to collect the samples.



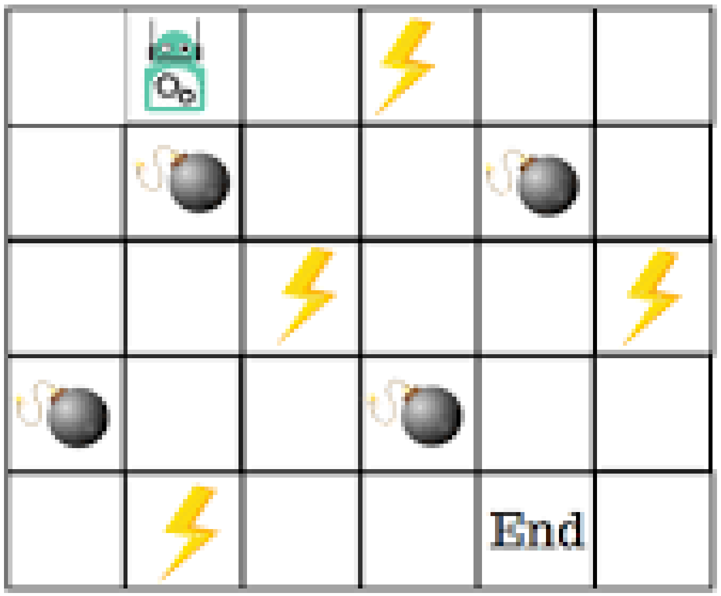

Behavior policy ≠ Policy used for action selection (Target Policy)

# Example of off policy RL

In Q-Learning, the agent learns optimal policy with the help of a greedy policy and behaves using policies of other agents. Q-learning is called off-policy because the updated policy is different from the behavior policy, so Q-Learning is off-policy. In other words, it estimates the reward for future actions and appends a value to the new state without actually following any greedy policy.

# Example of off policy RL

Behavior policy ≠ Policy used for action selection (Target Policy)



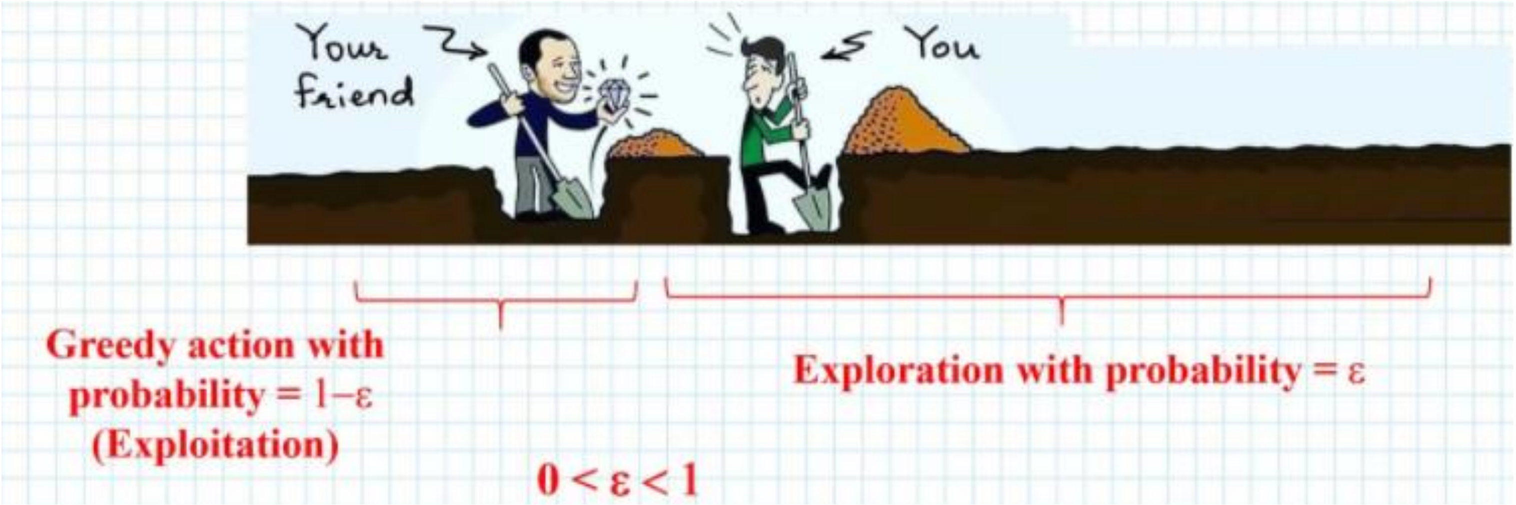

| Actions : | ↑ | → | ↓ | ← |
|---|---|---|---|---|
| Start | 0 | 0 | 0 | 0 |
| Nothing / Blank | 0 | 0 | 0 | 0 |
| Power | 0 | 0 | 0 | 0 |
| Mines | 0 | 0 | 0 | 0 |
| END | 0 | 0 | 0 | 0 |

Q-learning

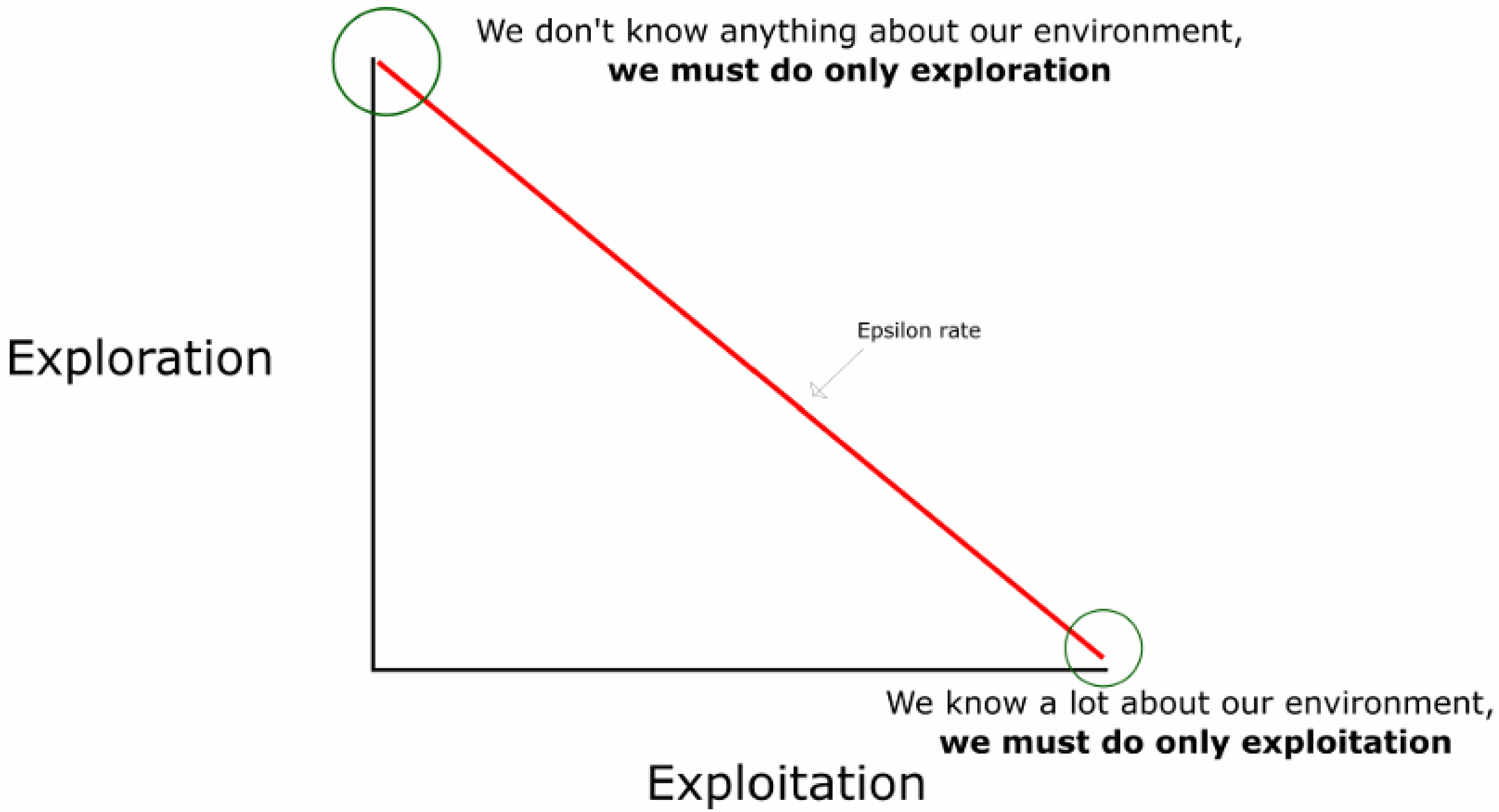
# What is greedy policy?

Greedy policy (exploration with exploitation)

With partial or no knowledge about future rewards, **ε**-greedy policy gives best results as it balances between exploitation of current knowledge and exploration of unknown territory (actions). We can start with **ε** closer to 1 initially (more exploration) and bring it closer to 0 in steps as we learn more and more about the environment.



# Choose an action



# Complications when you use reinforcement learning

- The outcome of your actions may be uncertain
- You may not be able to perfectly sense the state of the world
- The reward may be stochastic.
- Reward is delayed (i.e. finding food in a maze)
- You may have no clue (model) about how the world responds to your actions.
- You may have no clue (model) of how rewards are being paid off.
- The world may change while you try to learn it
- How much time do you need to explore uncharted territory before you exploit what you have learned?

# Reinforcement Learning

## What does it need?

This method assumes that training information is available in the form of a real-valued reward signal given for each state-action transition.

i.e. (s, a, r)

## What problems?

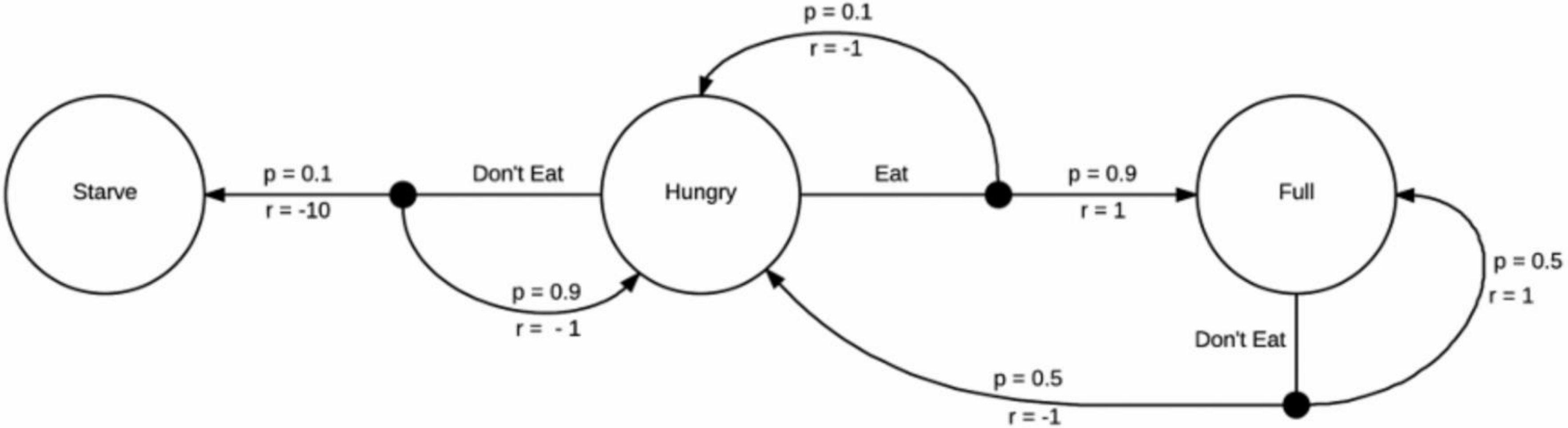
Very often, reinforcement learning fits a problem setting known as a **Markov decision process** (MDP).

# Markov Decision Process (MDP)

MDP is a discrete-time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.

# How to apply MDP in reinforcement learning

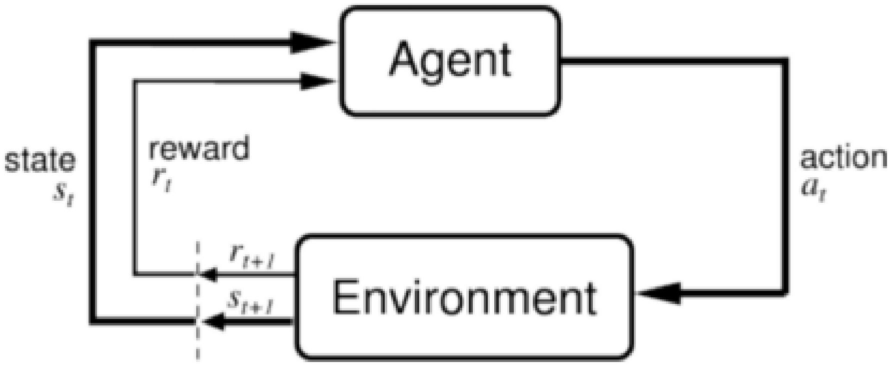
MDP has a graph pointing between the relation of each state, action, and reward where r represents the reward generated by that action and p represents the probability to go to a different state when performing that action.



# A Structure of MDP in reinforcement learning

An MDP is defined by:

- Set of states $S$
- Set of actions $A$
- Transition function $P(s' \mid s, a)$
- Reward function $R(s, a, s')$
- Start state $s_0$
- Discount factor $\gamma$
- Horizon $H$



# The Task

- To learn an optimal *policy* that maps states of the world to actions of the agent.

  I.e., if this patch of room is dirty, I clean it. If my battery is empty, I recharge it.

$$\pi : S \to A$$

What is it that the agent tries to optimize?

Answer: the total future discounted reward:

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

Note: $\gamma$ is a discount rate

immediate reward is worth more than future reward.

# Value Function

- Let's say we have access to the optimal value function that computes the total future discounted reward $V^*(s)$
- What would be the optimal policy $\pi^*(s)$?
- Answer: we choose the action that maximizes:

$$\pi^*(s) = \arg\max_a \sum_{s'} P(s'|s,a)V(s')$$

- We assume that we know what the reward will be if we perform action "a" in state "s": $r(s,a)$

- We also assume we know what the next state of the world will be if we perform action "a" in state "s": $s_{t+1} = \delta(s_t, a)$

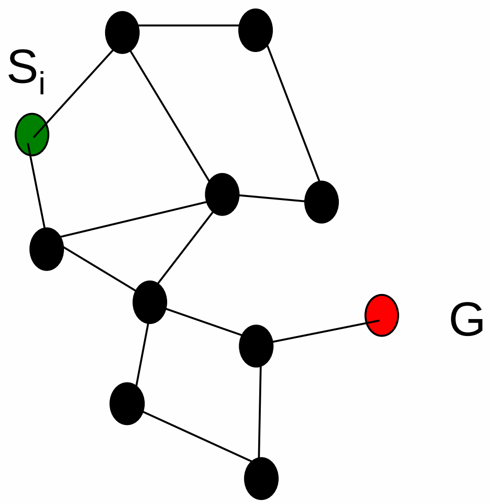
# Example I

- Consider some complicated graph, and we would like to find the shortest path from a node $S_i$ to a goal node G.

- Traversing an edge will cost you "length edge" dollars.

- The value function encodes the total remaining distance to the goal node from any node s, i.e. V(s) = "*1 / distance*" to goal from s.

- If you know V(s), the problem is trivial. You simply choose the node that has highest V(s).



# Simple Reinforcement Learning: Q-learning

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.

Q-learning can identify an optimal action-selection policy for any given finite Markov decision process (FMDP).

Q-learning given infinite exploration time and a partly-random policy.

# Simple Reinforcement Learning: Q-learning

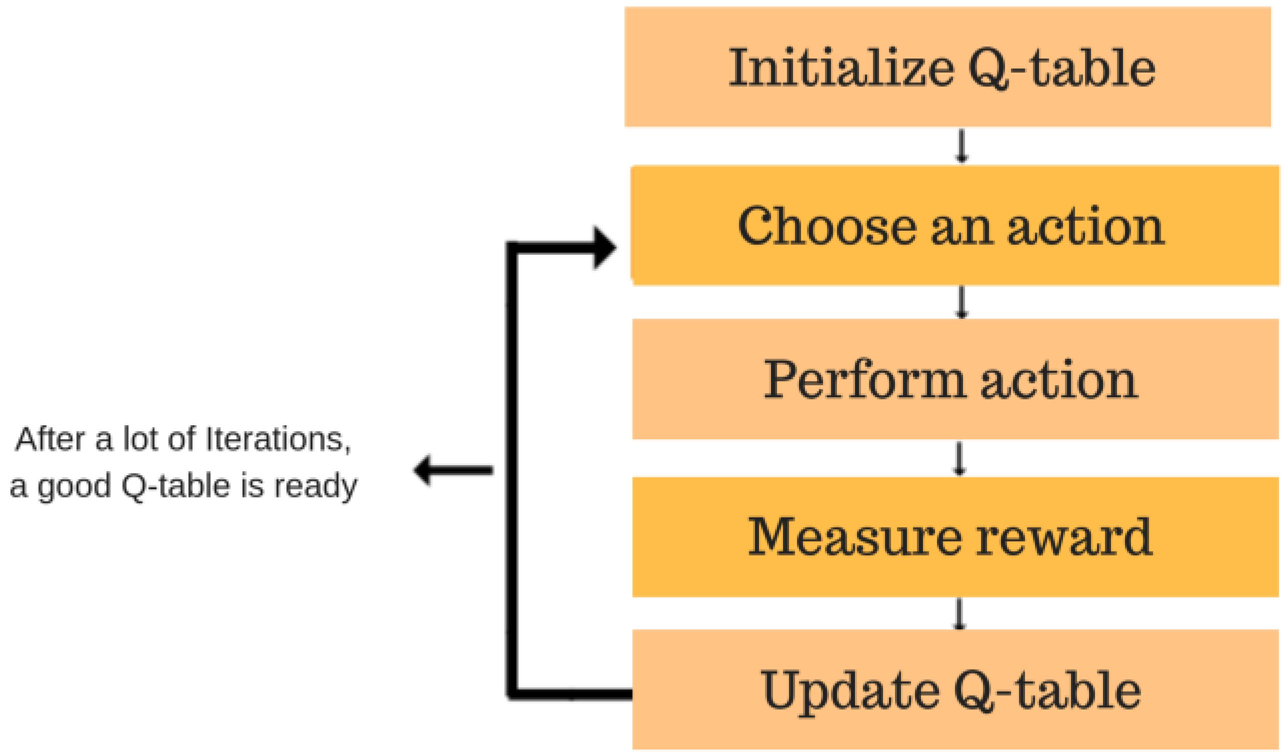
## Advantage:

Converges to an optimal policy in both deterministic and nondeterministic MDPs.

## Disadvantage:

Only practical on a small number of problems.

# Introducing the Q-learning algorithm process



# Q-learning Algorithm

Initialize $Q(s, a)$ arbitrarily

Repeat (for each episode)

Initialize $s$

Repeat (for each step of the episode)

Choose $a$ from $s$ using an exploratory policy

Take action $a$, observe $r$, $s'$ (when $s'$ means future state)

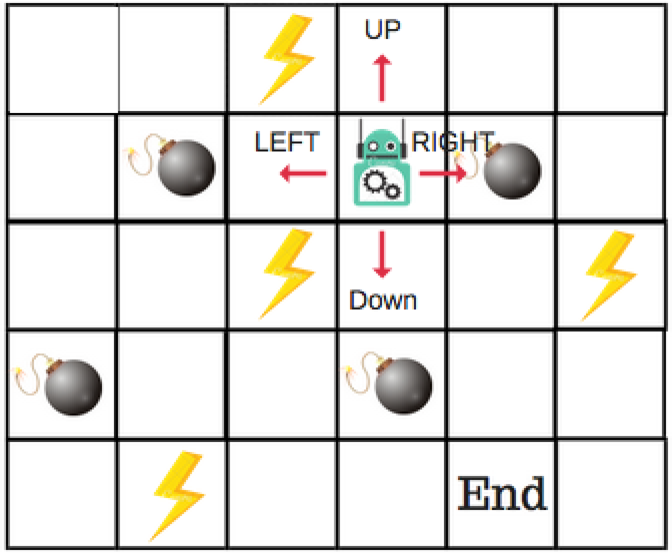
$$Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha[r_{t+1} + \gamma \max_{a'} Q_t(s', a') - Q_t(s, a)]$$

$s \leftarrow s'$

# Introduction to Q-learning Algorithm

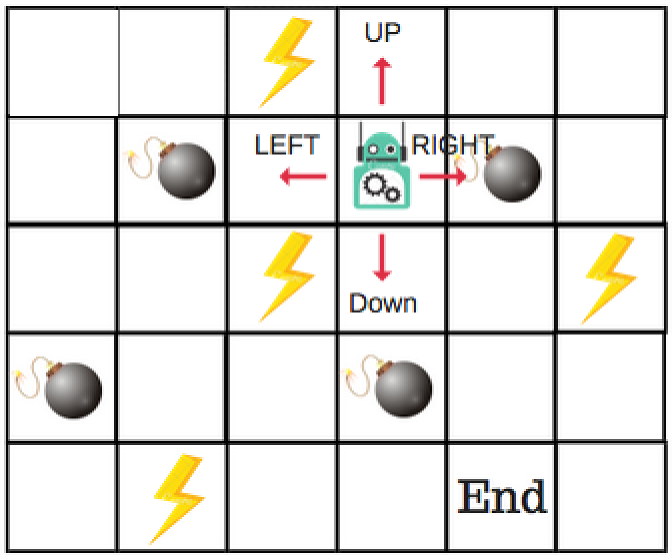
- An episode: $\{ (s_1, a_1, r_1), (s_2, a_2, r_2), \ldots (s_n, a_n, r_n), \}$
- s': $\delta(s, a) \rightarrow s'$ : future state
- Q(s, a) : Q function
- $\gamma, \alpha$ : *discount rate and learning rate*

# Example 2:




There will be four numbers of actions at each non-edge tile. When a robot is at a state it can either move up or down or right or left.

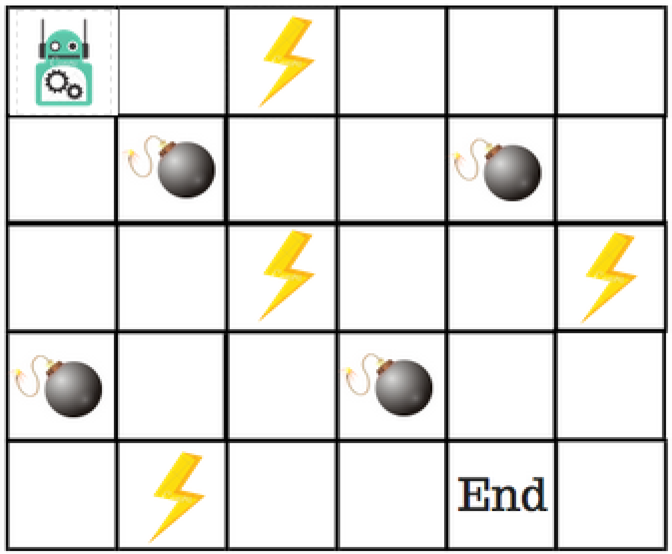
# Example 2:




| Actions : | ↑ | → | ↓ | ← |
|---|---|---|---|---|
| Start | | | | |
| Nothing / Blank | | | | |
| Power | | | | |
| Mines | | | | |
| END | | | | |

There will be four numbers of actions at each non-edge tile. When a robot is at a state it can either move up or down or right or left.

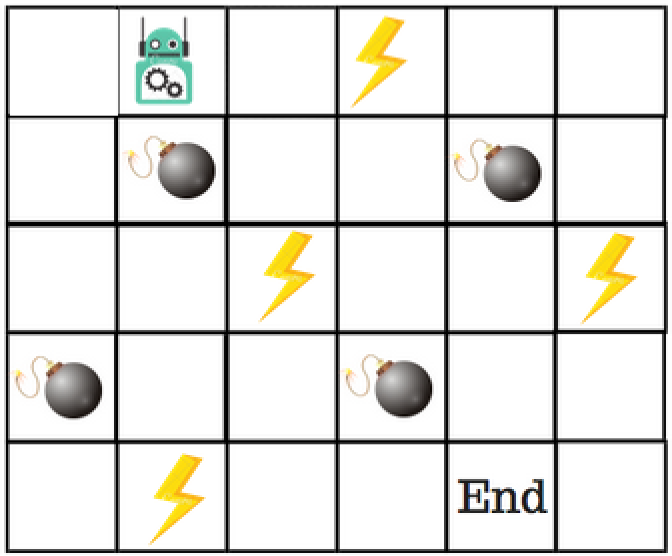
# Step 1: initialize the Q-Table




| Actions : | ↑ | → | ↓ | ← |
|---|---|---|---|---|
| Start | 0 | 0 | 0 | 0 |
| Nothing / Blank | 0 | 0 | 0 | 0 |
| Power | 0 | 0 | 0 | 0 |
| Mines | 0 | 0 | 0 | 0 |
| END | 0 | 0 | 0 | 0 |

We will first build a Q-table. There are n columns, where n= number of actions. There are m rows, where m= number of states. We will initialise the values at 0.

# Steps 2 and 3: choose and perform an action




| Actions : | ↑ | → | ↓ | ← |
|---|---|---|---|---|
| Start | 0 | 0 | 0 | 0 |
| Nothing / Blank | 0 | 0 | 0 | 0 |
| Power | 0 | 0 | 0 | 0 |
| Mines | 0 | 0 | 0 | 0 |
| END | 0 | 0 | 0 | 0 |

For the robot example, there are four actions to choose from: up, down, left, and right. We are starting the training now — our robot knows nothing about the environment. So the robot chooses a random action, say right.

# Steps 4 and 5: evaluate

$$\boxed{\text{New } Q(s,a)} = \boxed{Q(s,a)} + \boxed{\alpha}\,[\boxed{R(s,a)} + \boxed{\gamma}\,\boxed{\max Q'(s',a')} - \boxed{Q(s,a)}]$$

- New Q Value for that state and the action
- Learning Rate
- Reward for taking that action at that state
- Current Q Values
- Maximum expected future reward given the new state (s') and all possible actions at that new state.
- Discount Rate

In the case of the robot game, to reiterate the scoring/reward structure is:

power = +1

mine = -100

end = +100 discount rate = 0.9 learning rate = 0.1

discount rate = 0.9 learning rate = 0.1

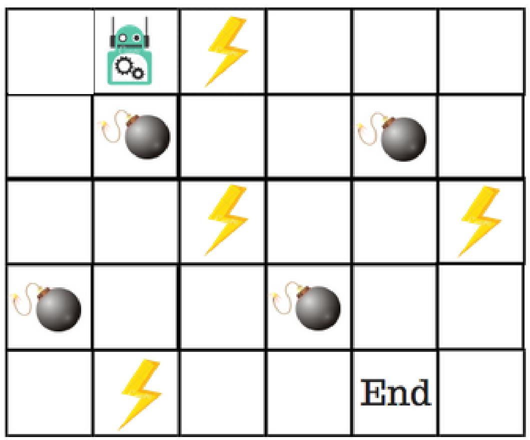
# Steps 4 and 5: evaluate

New Q(start,right) = Q(start,right) + $\alpha$[some ... Delta value ]

Some ... Delta value = R(start,right) + $\gamma$ max( Q`(nothing,down),Q`(nothing,left),Q`(nothing,right)) - Q(start,right)

Some ... Delta value = 0 + 0.9 * 0 - 0 = 0

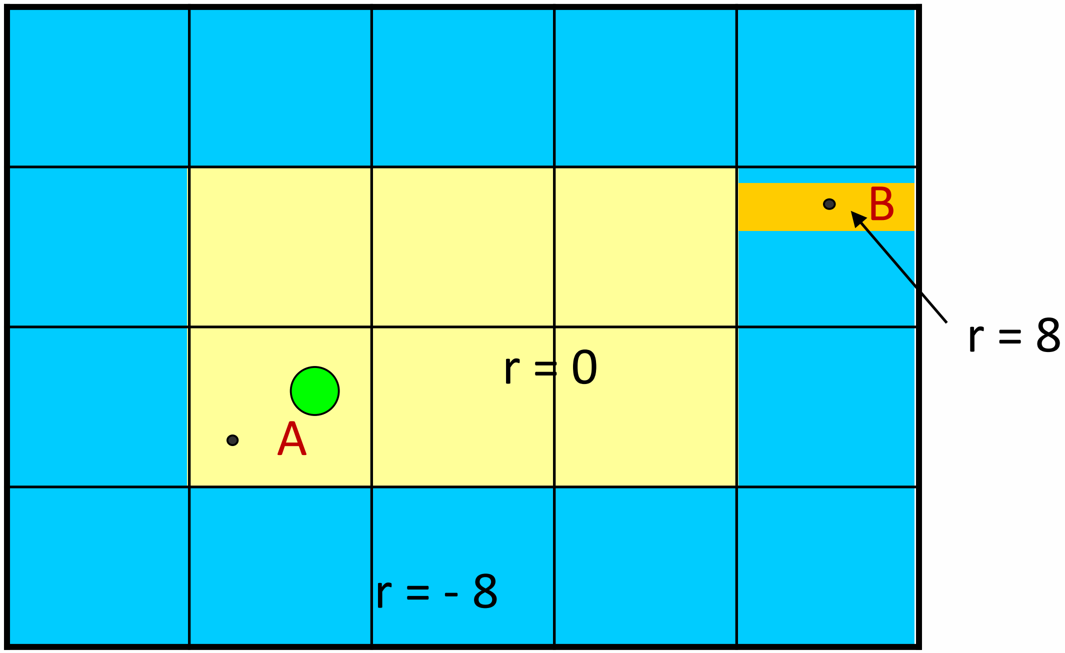
New Q(start,right) = 0 + 0.1* 0 = 0




| Actions : | ↑ | → | ↓ | ← |
|---|---|---|---|---|
| Start | 0 | 0 | 0 | 0 |
| Nothing / Blank | 0 | 0 | 0 | 0 |
| Power | 0 | 0 | 0 | 0 |
| Mines | 0 | 0 | 0 | 0 |
| END | 0 | 0 | 0 | 0 |

We will repeat this again and again until the learning is stopped. In this way the Q-Table will be updated.

# Example 3: A Sample Problem ( A → B)



# States and actions

states:

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |

actions:

N ↑

S ↓

E →

W ←

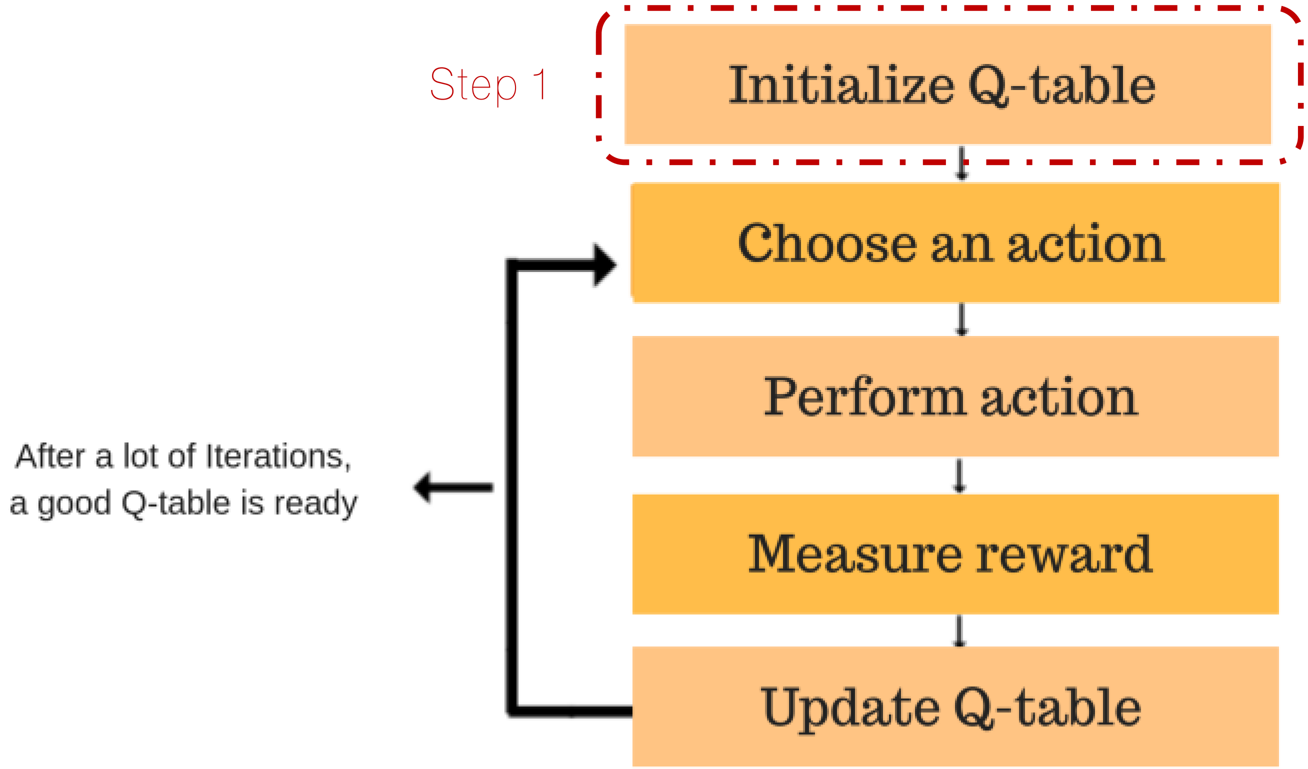
# The Q(s, a) function

states

actions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | | | | | | | | | | | | | | | | | | | | |
| S | | | | | | | | | | | | | | | | | | | | |
| W | | | | | | | | | | | | | | | | | | | | |
| E | | | | | | | | | | | | | | | | | | | | |

# Introducing the Q-learning algorithm process



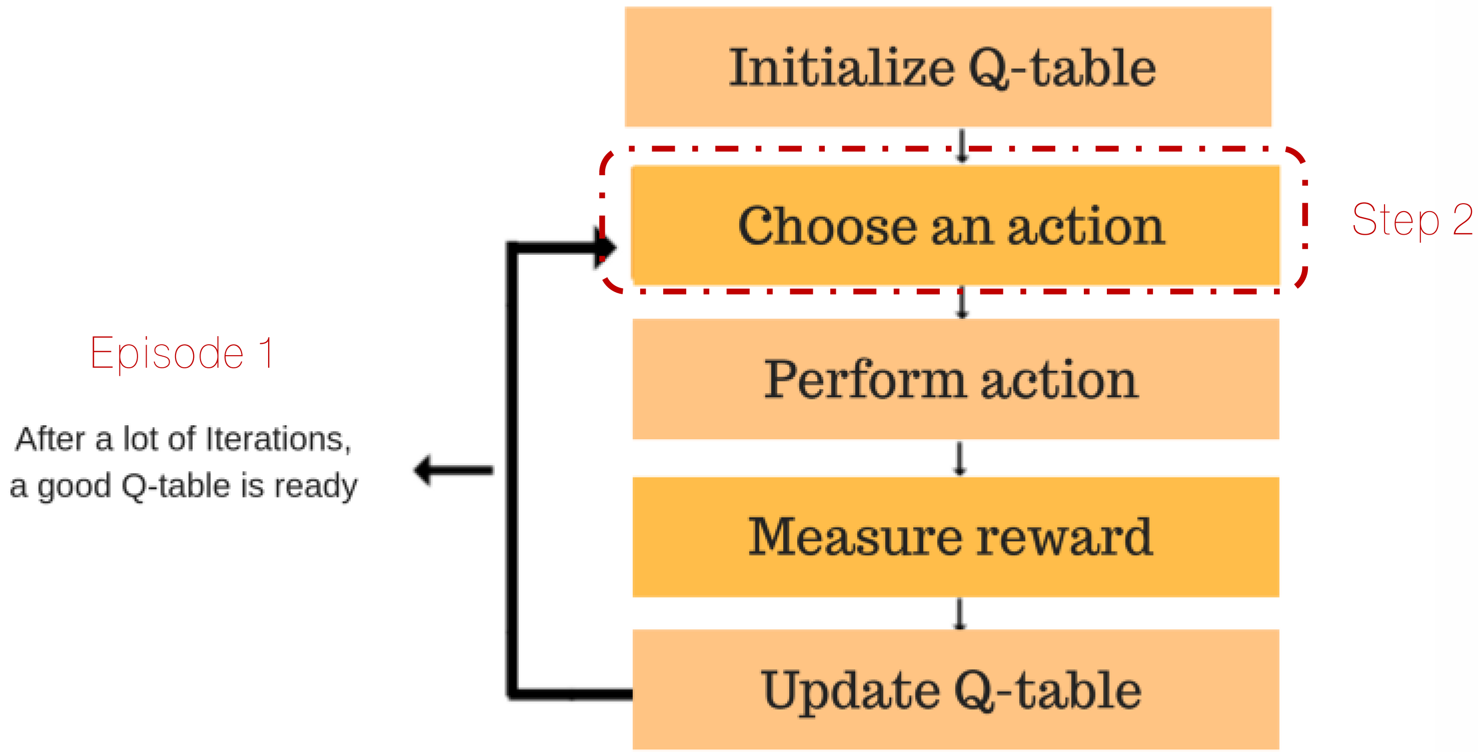
# Initializing the Q(s, a) function

states

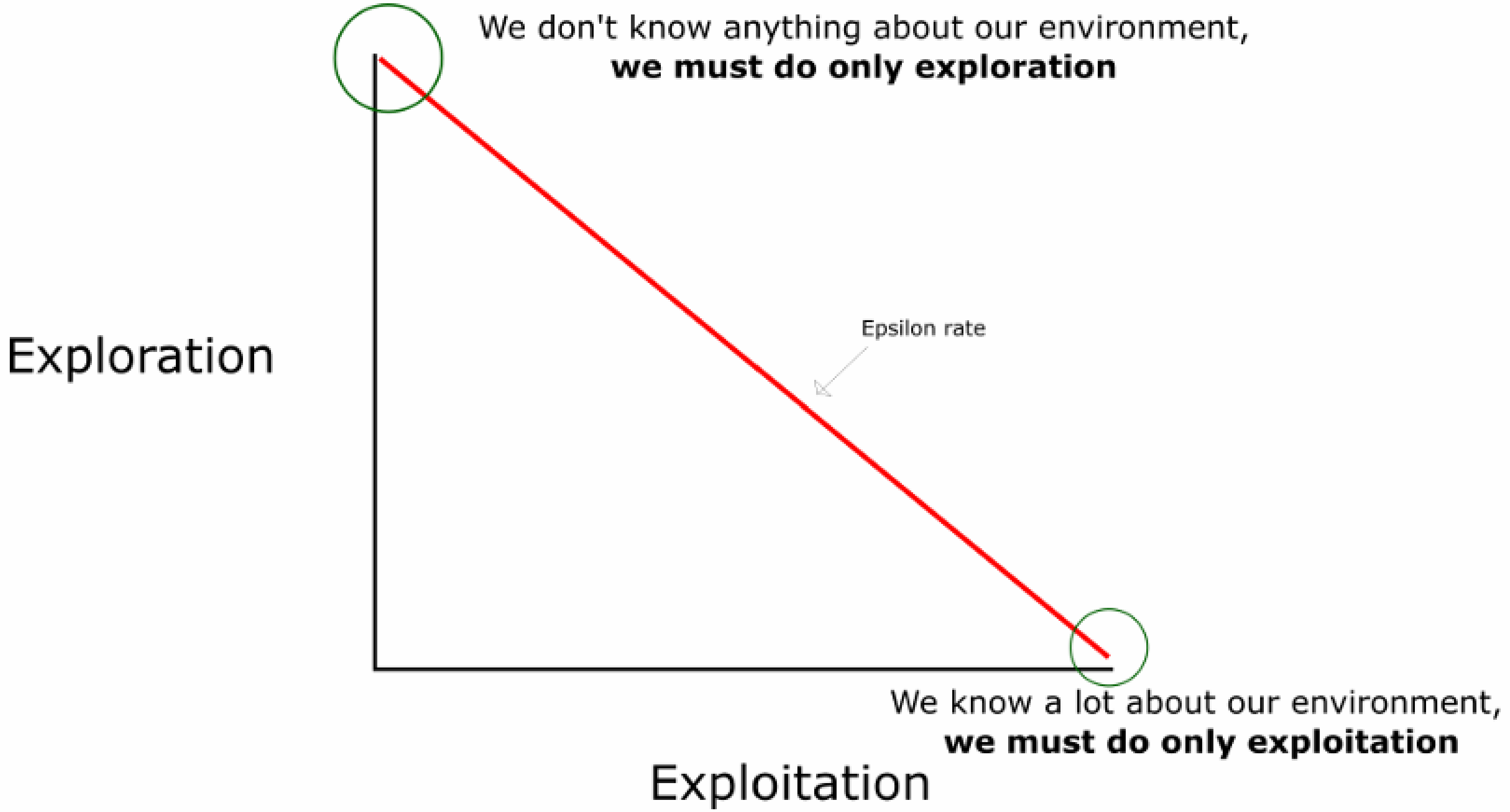
actions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

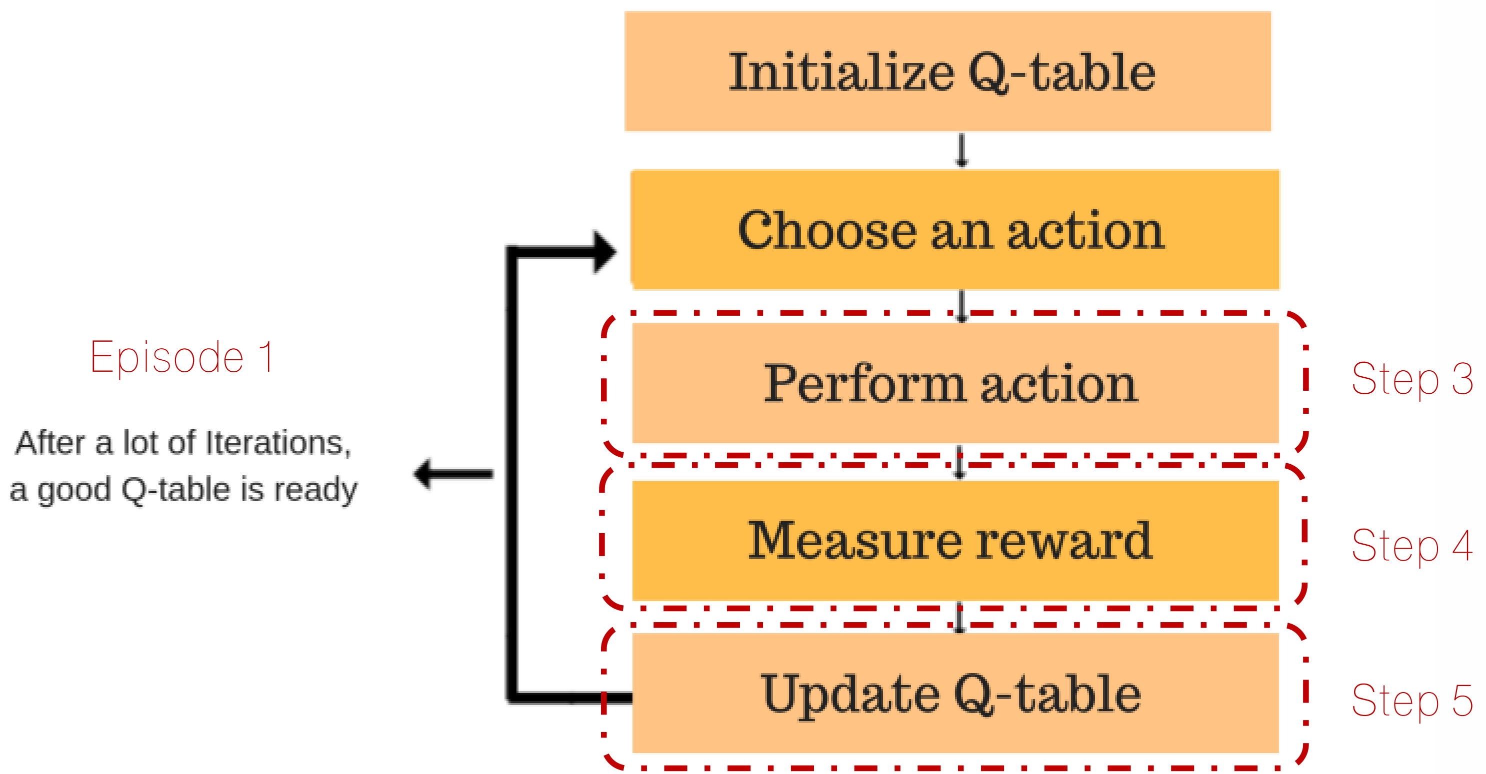
# Introducing the Q-learning algorithm process



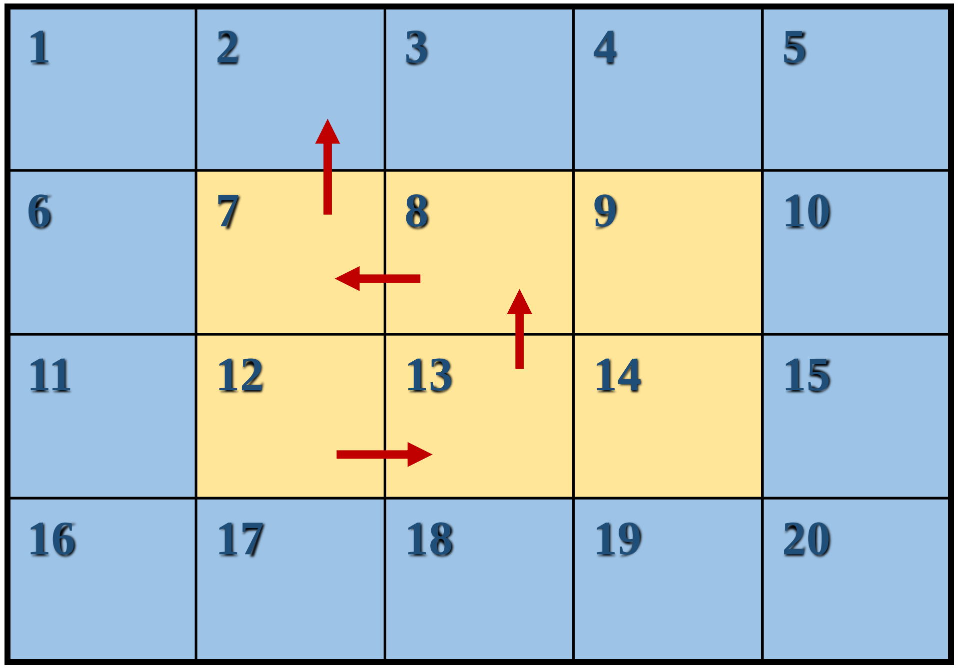
# Choose an action



# Introducing the Q-learning algorithm process



# An episode



# Calculating new Q(s, a) values

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

Where $\alpha$ = 1 and $\gamma$ = 0.5

1st step:

$$Q(s_{12},E) \leftarrow 0 + 1*[0 + 0.5*(0) - 0]$$
$$Q(s_{12},E) \leftarrow 0$$

2nd step:

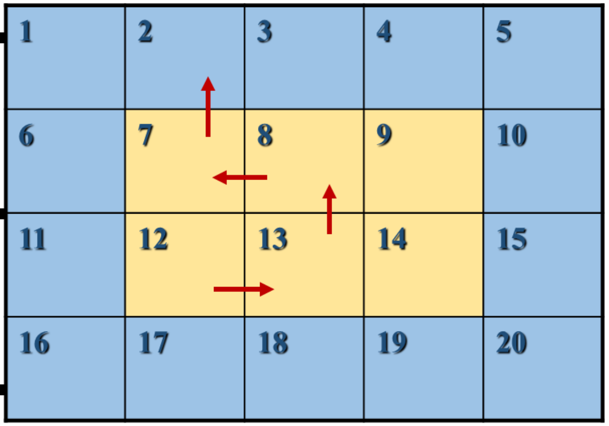
$$Q(s_{13},N) \leftarrow 0 + 1*[0 + 0.5*(0) - 0]$$
$$Q(s_{13},N) \leftarrow 0$$

3rd step:

$$Q(s_{8},W) \leftarrow 0 + 1*[0 + 0.5*(0) - 0]$$
$$Q(s_{8},W) \leftarrow 0$$

4th step:

$$Q(s_{7},N) \leftarrow 0 + 1*[-8 + 0.5*(0) - 0]$$
$$Q(s_{7},N) \leftarrow -8$$



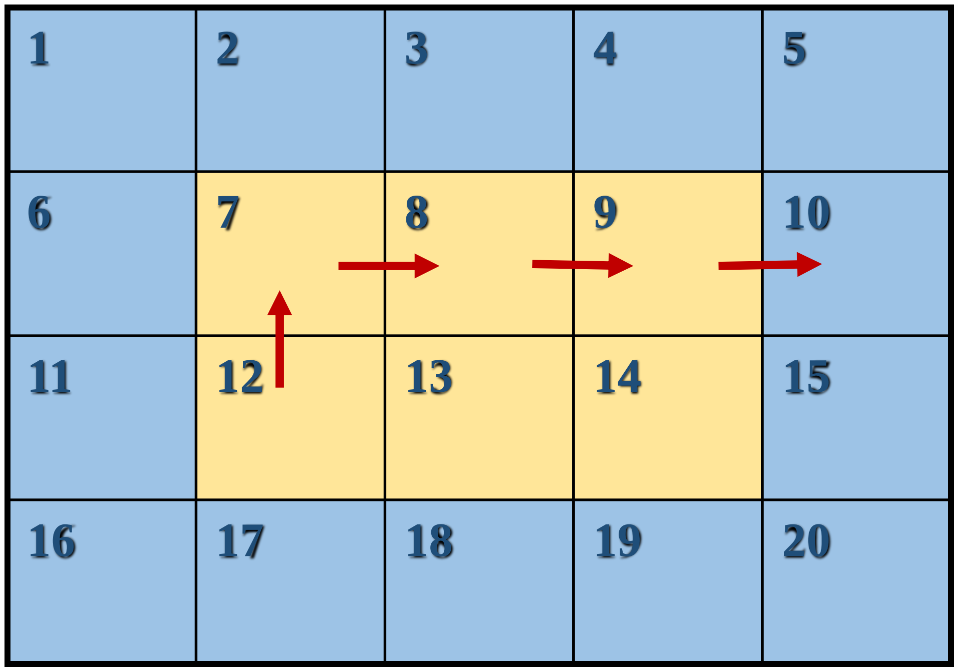
# The Q(s, a) function after the first episode

states

actions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | 0 | 0 | 0 | 0 | 0 | 0 | -8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# A second episode



Q-table of State 7
= -8

# Calculating new Q(s, a) values

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

1st step:

$$Q(s_{12}, N) \leftarrow 0 + 1*[0 + 0.5*\max\{-8,0,0,0\} - 0]$$
$$Q(s_{12}, N) \leftarrow 0$$

2nd step:

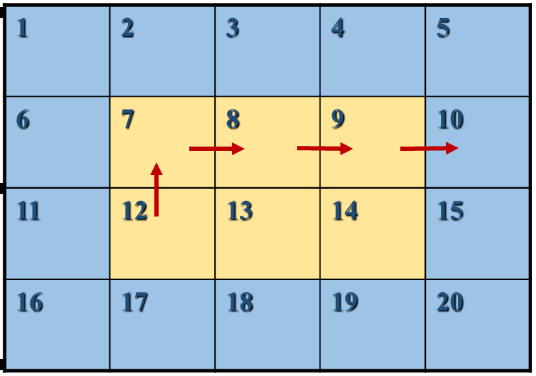
$$Q(s_7, E) \leftarrow 0 + 1*[0 + 0.5*(0) - 0]$$
$$Q(s_7, E) \leftarrow 0$$

3rd step:

$$Q(s_8, E) \leftarrow 0 + 1*[0 + 0.5*(0) - 0]$$
$$Q(s_8, E) \leftarrow 0$$

4th step:

$$Q(s_9, E) \leftarrow 0 + 1*[8 + 0.5*(0) - 0]$$
$$Q(s_9, E) \leftarrow 8$$



# The Q(s, a) function after the second episode

states

actions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | 0 | 0 | 0 | 0 | 0 | 0 | -8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# The Q(s, a) function after a few episodes

states

actions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | 0 | 0 | 0 | 0 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 1 | 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0.5 | 1 | 2 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | -8 | 1 | 2 | 0 | 0 | -8 | 0.5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 8 | 0 | 0 | 1 | 2 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |

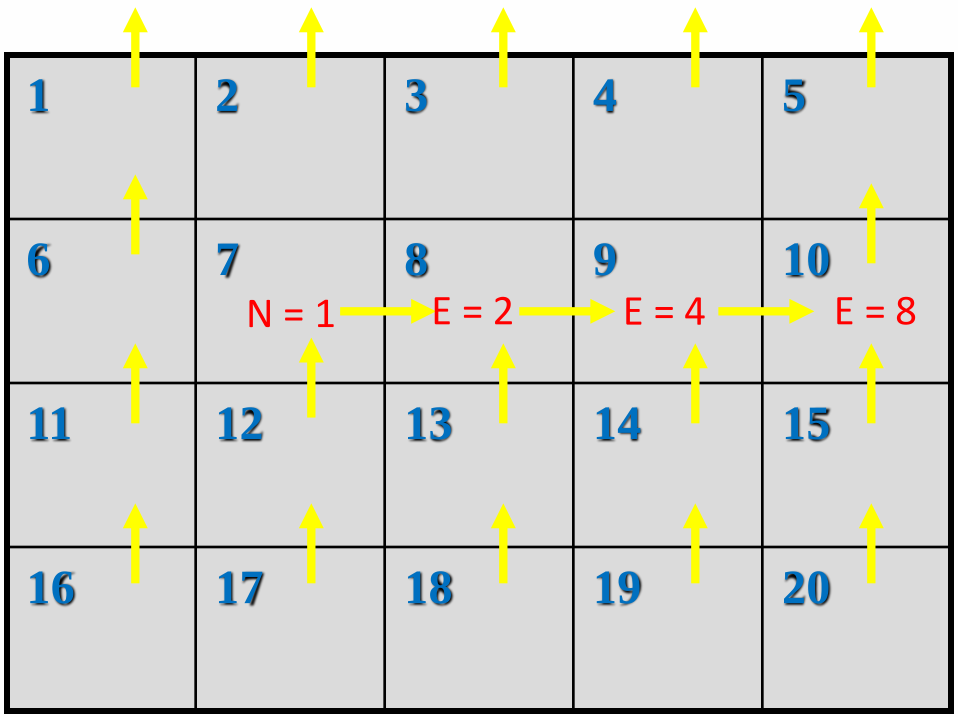
# One of the optimal policies

states

actions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | 0 | 0 | 0 | 0 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 1 | 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0.5 | 1 | 2 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | -8 | 1 | 2 | 0 | 0 | -8 | 0.5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 8 | 0 | 0 | 1 | 2 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |

# An optimal policy graphically




Total Rewards = 15

# Another of the optimal policies

states

actions

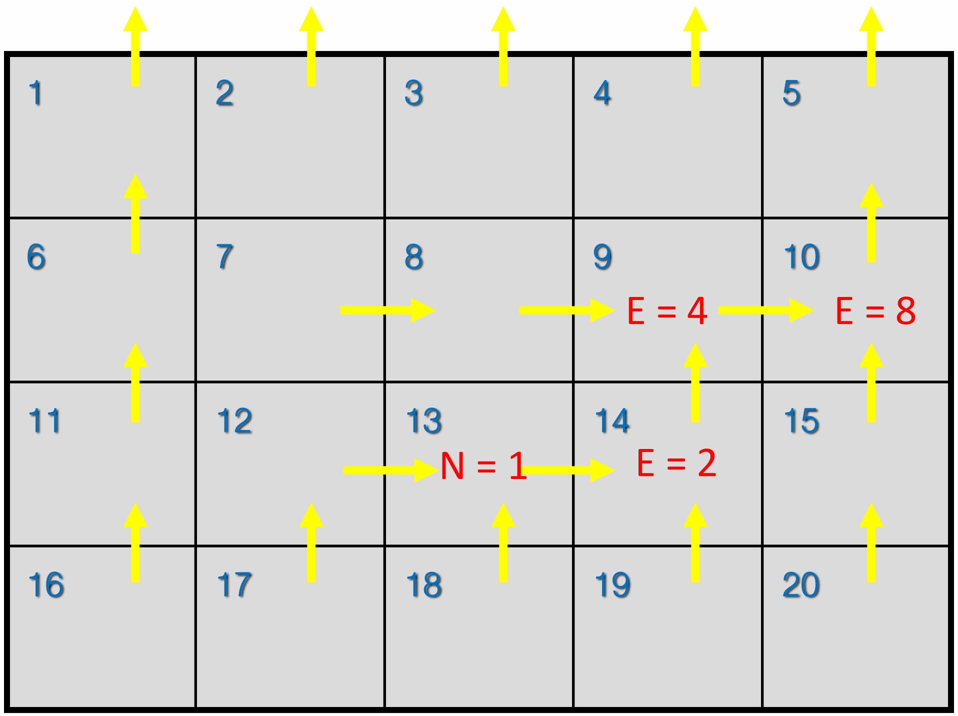
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | 0 | 0 | 0 | 0 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 1 | 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0.5 | 1 | 2 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | -8 | 1 | 2 | 0 | 0 | -8 | 0.5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 8 | 0 | 0 | 1 | 2 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |

states

actions

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | 0 | 0 | 0 | 0 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 1 | 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0.5 | 1 | 2 | 0 | 0 | -8 | -8 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |
| W | 0 | 0 | 0 | 0 | 0 | 0 | -8 | 1 | 2 | 0 | 0 | -8 | 0.5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 8 | 0 | 0 | 1 | 2 | -8 | 0 | 0 | 0 | 0 | 0 | 0 |

# Another optimal policy graphically




Total Rewards = 15

# Q-table with python

# Q-table with python

| Tiles | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| | Hole | | | | | | Beer | | | |

Go for beer!!

The game is simple, there are 10 tiles in a row. All tiles are not equal, some have hole where we do not want to go, whereas some have beer, where we definitely want to go. When the game start, you can spawn on any of tiles, and can either go left or right. The game will go on unless either we have won or its game over, lets call each such iteration an episode.

# Q-table with python

| Tiles | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| | Hole | | | | | | Beer | | | |

Go for beer!!

So, if you spawn on the 0th tile (array-0) or somehow travels to 0th tile, its game over but if we travel to tile 6th (array-6) , we win.

Now suppose I haven't shown you the game map and you only have the option of going left or right, which way will you go? Well you can't say unless you try it out

# Q-table with python

A markov decision process consist of,

1) State (S): It is a set of states. Tiles in our example. So we have 10 states in our game.
2) Action (A): It is a set of actions available form state s. Left & right from our game.
3) Probability of transition P(s‘ I s, a): It is the probability of transition to s' state at time t+1 if we took action a in state s at time t. We were kind of sorted on this front, a left from tile 3 leads to tile 2, no question asked.
4) Reward R(s‘ I s, a): It is the reward we receive if we transition from state s to state s' by taking action a.
5) Discount (Y): It is the discount factor, which represents the difference in future and present rewards.

# Q-table with python

environment_matrix = [[None, 0],
[-100, 0],
[0, 0],
[0, 0],
[0, 0],
[0, 100],
[0, 0],
[100, 0],
[0, 0],
[0, None]]

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Hole | | | | | | Beer | | | |

Also a left from tile 1 leads to hole, so it has a negative reward.

As you can see, taking right from tile 5 and taking left from tile 7 have high reward of 100 as it leads to tile 6.

Tile 0 and 9 have left and right reward as None as there are no -1 or 10th tile.

# Q-table with python

```
q_matrix = [[0, 0],
    [0, 0],
    [0, 0],
    [0, 0],
    [0, 0],
    [0, 0],
    [0, 0],
    [0, 0],
    [0, 0],
    [0, 0]]
```

For starters lets assign all to zero for Q-matrix

# Q-table with python

```
win_loss_states = [0,6]

def getAllPossibleNextAction(cur_pos):

    step_matrix = [x != None for x in environment_matrix[cur_pos]]

    action = []

    if(step_matrix[0]):

        action.append(0)

    if(step_matrix[1]):

        action.append(1)

    return(action)
```

getAllPossibleNextAction pass your current state and it will return all the possible actions. Note for tile 0, only right action is there and same goes for tile 9 with only left action

# Q-table with python

```
def isGoalStateReached(cur_pos):
    return (cur_pos in [6])
def getNextState(cur_pos, action):
    if (action == 0):
        return cur_pos - 1
    else:
        return cur_pos + 1
def isGameOver(cur_pos):
    return cur_pos in win_loss_states
```

isGoalStateReached if the current tile is 6 it will return True

getNextState pass current state and the action, and it will return the next state

isGameOver if the state is 0 or 6, the game is over, this returns True

# Q-table with python

Initial parameters

```
import random
discount = 0.9
learning_rate = 0.1
for _ in range(1000):
    # get starting place
    cur_pos = random.choice([0,1,2,3,4,5,6,7,8,9])
    # while goal state is not reached
    while(not isGameOver(cur_pos)):
        # get all possible next states from cur_step
        possible_actions = getAllPossibleNextAction(cur_pos)
```

# Q-table with python

```
# select any one action randomly
action = random.choice(possible_actions)
# find the next state corresponding to the action selected
next_state = getNextState(cur_pos, action)
# update the q_matrix
q_matrix[cur_pos][action] = q_matrix[cur_pos][action] + learning_rate * (environment_matrix[cur_pos][action] +
    discount * max(q_matrix[next_state]) - q_matrix[cur_pos][action])
# go to next state
cur_pos = next_state
```

# Q-table with python

```
# print status
    print("Episode ", _ , " done")
print(q_matrix)
print("Training done...")
```

# Q-table with python

```
[[0, 0], [-99.99999999730835, 65.60999997057485],
[59.04899994359059, 72.8999999993016], [65.60999999858613,
80.99999999978154],
[72.89999999929572, 89.99999999991468], [80.99999999863587,
99.99999999997391], [0, 0], [99.9999999999985, 80.99999999994624],
[89.99999999999515, 72.89999999997386], [80.99999999999046, 0]]
```

# Q-table with python

| | Action | |
|---|---|---|
| State | Left | Right |
| 0 | 0 | 0 |
| 1 | -100 | 65.61 |
| 2 | 59.049 | 72.9 |
| 3 | 65.61 | 81 |
| 4 | 72.9 | 90 |
| 5 | 81 | 100 |
| 6 | 0 | 0 |
| 7 | 100 | 81 |
| 8 | 90 | 72.9 |
| 9 | 81 | 0 |

# Q-table with python

| | Action | |
|---|---|---|
| State | Left | Right |
| 0 | 0 | 0 |
| 1 | -100 | 65.61 |
| 2 | 59.049 | 72.9 |
| 3 | 65.61 | 81 |
| 4 | 72.9 | 90 |
| 5 | 81 | 100 |
| 6 | 0 | 0 |
| 7 | 100 | 81 |
| 8 | 90 | 72.9 |
| 9 | 81 | 0 |

# Q-table with python

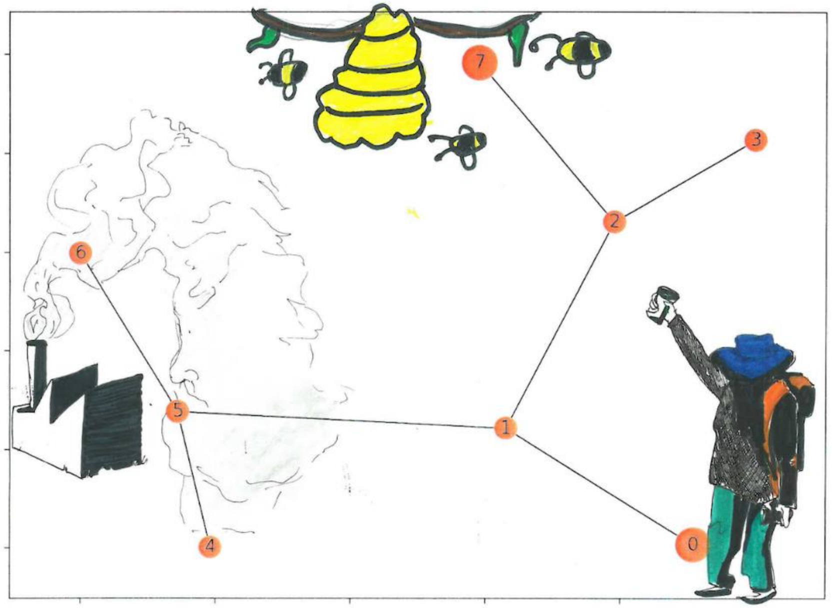
| | Action | |
|---|---|---|
| State | Left | Right |
| 0 | 0 | 0 |
| 1 | -100 | 65.61 |
| 2 | 59.049 | 72.9 |
| 3 | 65.61 | 81 |
| 4 | 72.9 | 90 |
| 5 | 81 | 100 |
| 6 | 0 | 0 |
| 7 | 100 | 81 |
| 8 | 90 | 72.9 |
| 9 | 81 | 0 |

# A Simple Python Example and a Step Closer to AI with Assisted Q-Learning

```
pip install networkx
```

https://amunategui.github.io/reinforcement-learning/index.html

# AI with Assisted Q-Learning




In this walk-through, we'll use Q-learning to find the shortest path between two areas.

# AI with Assisted Q-Learning

```
import numpy as np

import pylab as plt



# map cell to cell, add circular cell to goal point

points_list = [(0,1), (1,5), (5,6), (5,4), (1,2), (2,3), (2,7)]



goal = 7
```

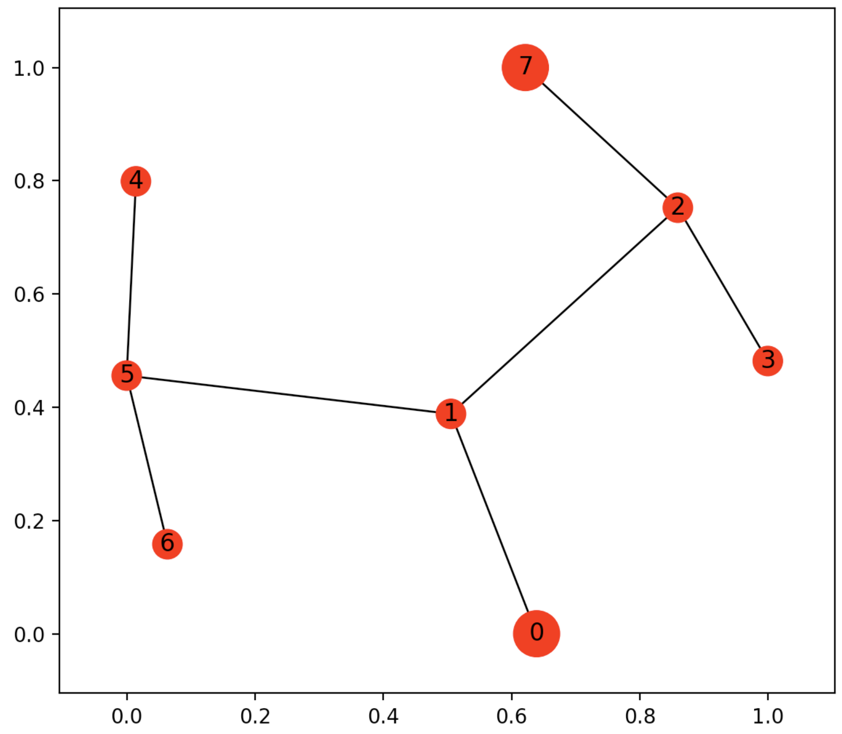
# AI with Assisted Q-Learning

```
import networkx as nx
G=nx.Graph()
G.add_edges_from(points_list)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G,pos)
nx.draw_networkx_edges(G,pos)
nx.draw_networkx_labels(G,pos)
plt.show()
```

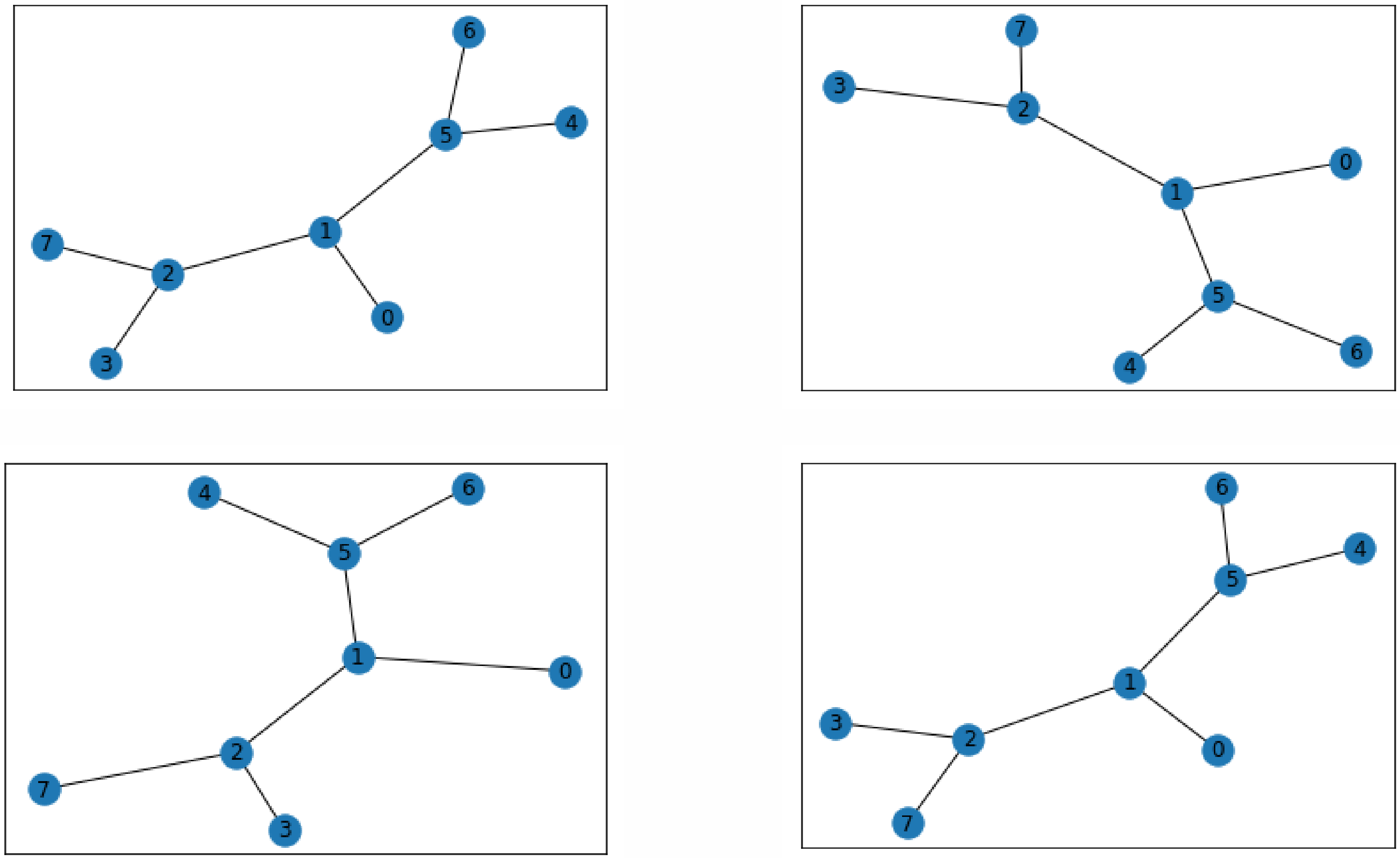
**The edge list** is a simple data structure that you'll use to create the graph. Each row represents a single edge of the graph with some edge attributes.

**The node lists** are the coordinates of the nodes (trail intersections) so that you can plot your graph with the same layout as the trail map

# AI with Assisted Q-Learning



# AI with Assisted Q-Learning



# AI with Assisted Q-Learning

```
# how many points in graph? x points

MATRIX_SIZE = 8


# create matrix x*y

R = np.matrix(np.ones(shape=(MATRIX_SIZE, MATRIX_SIZE)))

R *= -1
```

# AI with Assisted Q-Learning

```
# assign zeros to paths and 100 to goal-reaching point
for point in points_list:
    print(point)
    if point[1] == goal:
        R[point] = 100
    else:
        R[point] = 0
    if point[0] == goal:
        R[point[::-1]] = 100
    else:
        # reverse of point
        R[point[::-1]]= 0

# add goal point round trip
R[goal,goal]= 100
R
```

Create rewards R based on the trial map that you have already created

# AI with Assisted Q-Learning

the y-axis is the state

the x-axis is your possible next actions

```
matrix([[ -1.,   0.,  -1.,  -1.,  -1.,  -1.,  -1.,  -1.],
        [  0.,  -1.,   0.,  -1.,  -1.,   0.,  -1.,  -1.],
        [ -1.,   0.,  -1.,   0.,  -1.,  -1.,  -1., 100.],
        [ -1.,  -1.,   0.,  -1.,  -1.,  -1.,  -1.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,   0.,  -1.,  -1.],
        [ -1.,   0.,  -1.,  -1.,   0.,  -1.,   0.,  -1.],
        [ -1.,  -1.,  -1.,  -1.,  -1.,   0.,  -1.,  -1.],
        [ -1.,  -1.,   0.,  -1.,  -1.,  -1.,  -1., 100.]])
```

# AI with Assisted Q-Learning

```
Q = np.matrix(np.zeros([MATRIX_SIZE,MATRIX_SIZE]))

# learning parameter
gamma = 0.8

initial_state = 1

def available_actions(state):
    current_state_row = R[state,]
    av_act = np.where(current_state_row >= 0)[1]
    return av_act

available_act = available_actions(initial_state)
```

Create method to return available actions by giving a state value

# AI with Assisted Q-Learning

```
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_act,1))
    return next_action

action = sample_next_action(available_act)
```

Create method to return the next action by giving available actions range

# AI with Assisted Q-Learning

```
def update(current_state, action, gamma):
    max_index = np.where(Q[action,] == np.max(Q[action,]))[1]
    if max_index.shape[0] > 1:
        max_index = int(np.random.choice(max_index, size = 1))
    else:
        max_index = int(max_index)
    max_value = Q[action, max_index]
    Q[current_state, action] = R[current_state, action] + gamma * max_value
    print('max_value', R[current_state, action] + gamma * max_value)
    if (np.max(Q) > 0):
        return(np.sum(Q/np.max(Q)*100))
    else:
        return (0)

update(initial_state, action, gamma)
```

Create method to update score in Q-matrix by giving current state, action and gamma

# AI with Assisted Q-Learning

```
# Training
scores = []
for i in range(700):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_act = available_actions(current_state)
    action = sample_next_action(available_act)
    score = update(current_state,action,gamma)
    scores.append(score)
    print ('Score:', str(score))

print("Trained Q matrix:")
print(Q/np.max(Q)*100)
```

Training Q matrix with 700 cases and show score and max value

Print trained Q matrix

# AI with Assisted Q-Learning

the y-axis is the state

the x-axis is your possible next actions

Trained Q matrix:

```
[[ 0.    64.    0.    0.    0.    0.    0.    0.  ]
 [ 51.2   0.   80.    0.    0.   51.2   0.    0.  ]
 [ 0.    64.    0.   64.    0.    0.    0.  100.  ]
 [ 0.     0.   80.    0.    0.    0.    0.    0.  ]
 [ 0.     0.    0.    0.    0.   51.2   0.    0.  ]
 [ 0.    64.    0.    0.   40.96  0.   40.96  0.  ]
 [ 0.     0.    0.    0.    0.   51.2   0.    0.  ]
 [ 0.     0.   80.    0.    0.    0.    0.  100.  ]
```

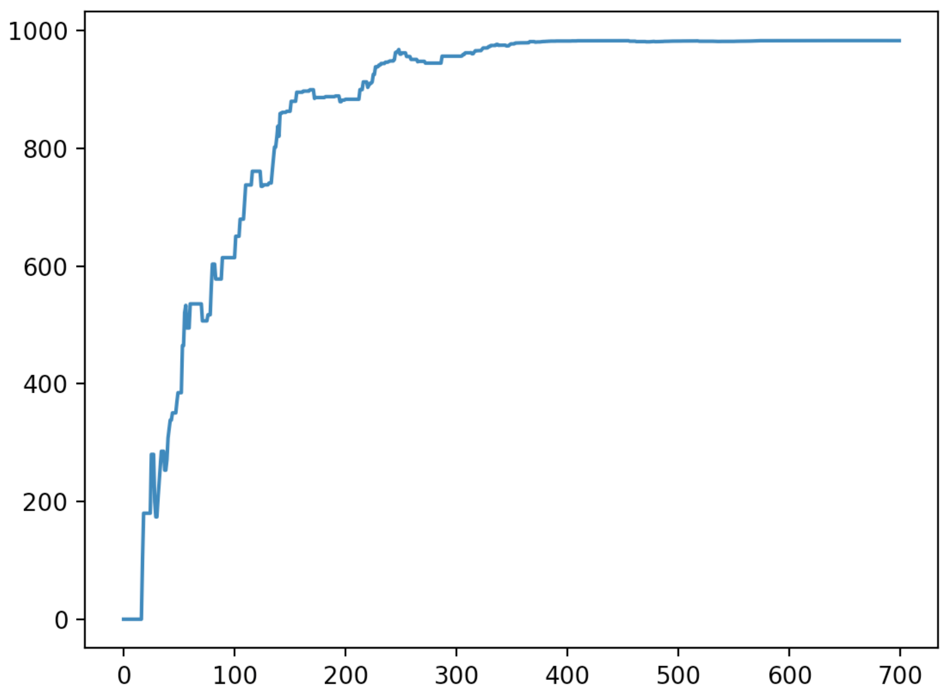
# AI with Assisted Q-Learning

```
# Testing
current_state = 0
steps = [current_state]

while current_state != 7:
    next_step_index = np.where(Q[current_state,] == np.max(Q[current_state,]))[1]

    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)

    steps.append(next_step_index)
    current_state = next_step_index
```

Try to find the most efficient path

# AI with Assisted Q-Learning

```
print("Most efficient path:")
print(steps)

plt.plot(scores)
plt.show()
```



# The problem with tabular Q-learning

## What is the problem?

Only practical in a small number of problems because:

a) Q-learning can require many thousands of training iterations to converge in even modest-sized problems.

b) Very often, the memory resources required by this method become too large.

# Solution

## What can we do about it?

Use generalization.

## What are some examples?

Tile coding, Radial Basis Functions, Fuzzy function approximation, Hashing, Artificial Neural Networks, LSPI, Regression Trees, Kanerva coding, etc.

# **Assignment : due date December 4, 2022**

(20 points)

Find one journal(2021 – present) related to reinforcement learning using in health care/clinic, then describe (the methodologies ) and draw a research framework /block diagram (step-by-step as same as the examples that presented in class)

# References

1. *Ernst D., Geurts P., and Wehenkel L., Tree-Based Batch Mode Reinforcement Learning, Journal of Machine Learning Research 6. Pages 503-556, 2005.*
2. *Liu G., Shulte O., and Li Q., Toward Interpretable Deep Reinforcement Learning with linear Model U-Trees. 2018.*