Universidade do Minho
Escola de Engenharia
Departamento de Informática

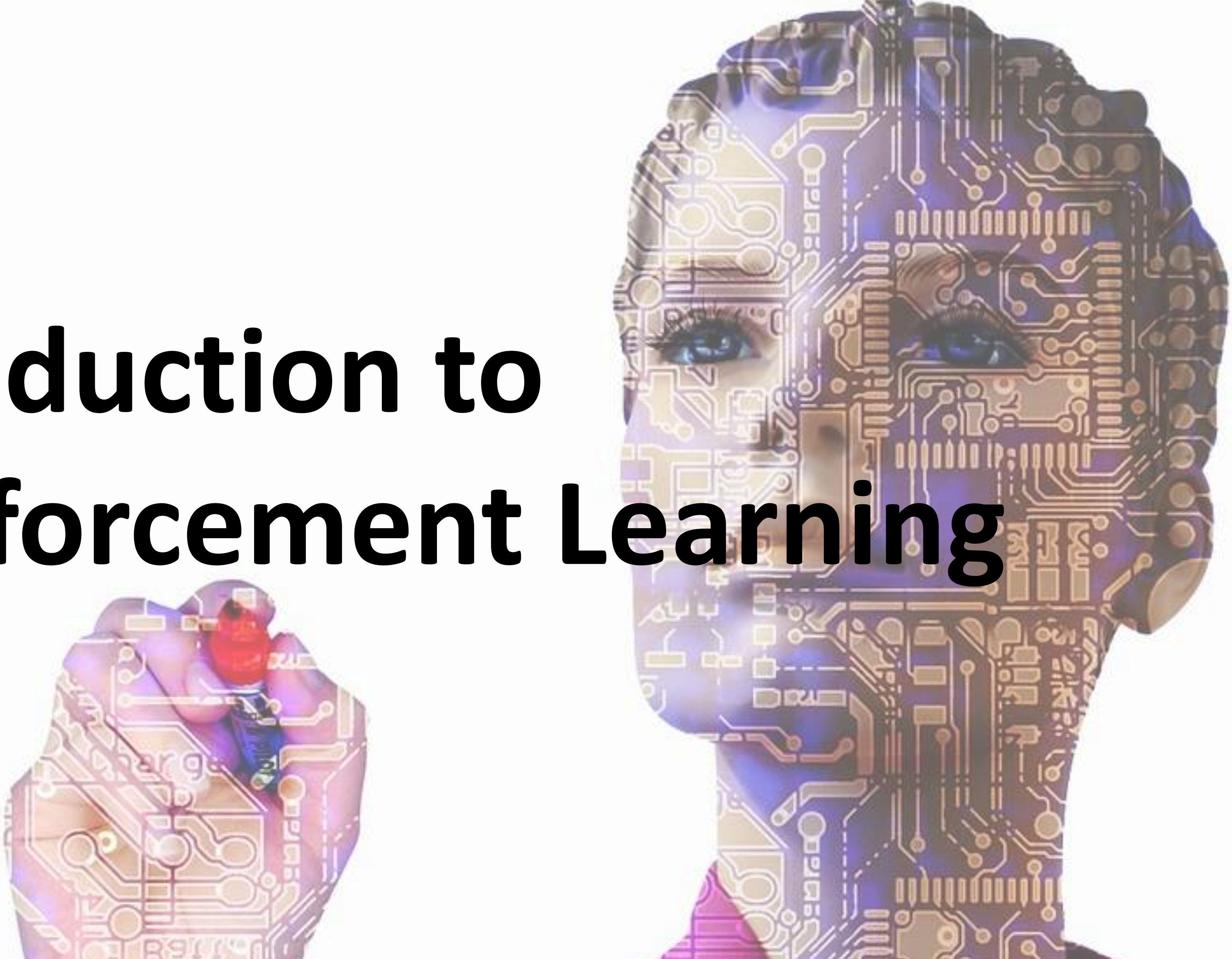
Mestrado Integrado em Engenharia Informática
Computação Natural
2018/2019

Paulo Novais, Cesar Analide, Filipe Gonçalves

- Paulo Novais – pjon@di.uminho.pt
- Cesar Analide – cesar.analide@di.uminho.pt
- Filipe Gonçalves – fgoncalves@algoritmi.uminho.pt

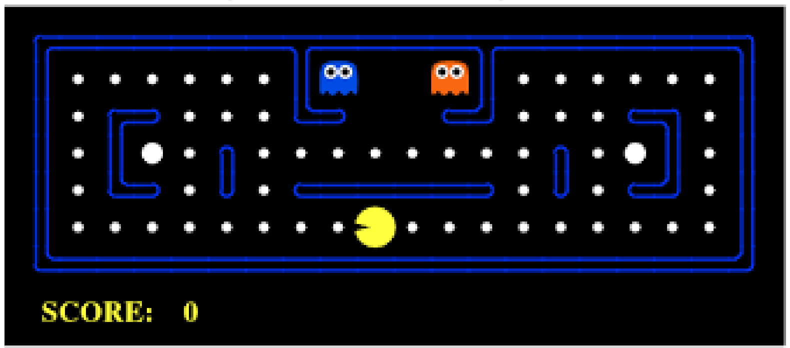
- Departamento de Informática
  Escola de Engenharia
  Universidade do Minho
- Grupo ISLab – (Synthetic Intelligence Lab)
- Centro ALGORITMI
  Universidade do Minho

Introduction to Reinforcement Learning



- Reinforcement Learning supports automation by learning from the environment it is present in
  - so does Machine Learning and Deep Learning, using different strategies, with automation support
- Deep Learning and Machine Learning are learning processes, but which are most focused on finding patterns in the existing data
- Reinforcement Learning, on the other hand, learns by trial and error method, and eventually, gets to the right actions or the global optimum
  - Pros: it is not required to provide the whole training data as in Supervised Learning. Instead, a few chunks would suffice

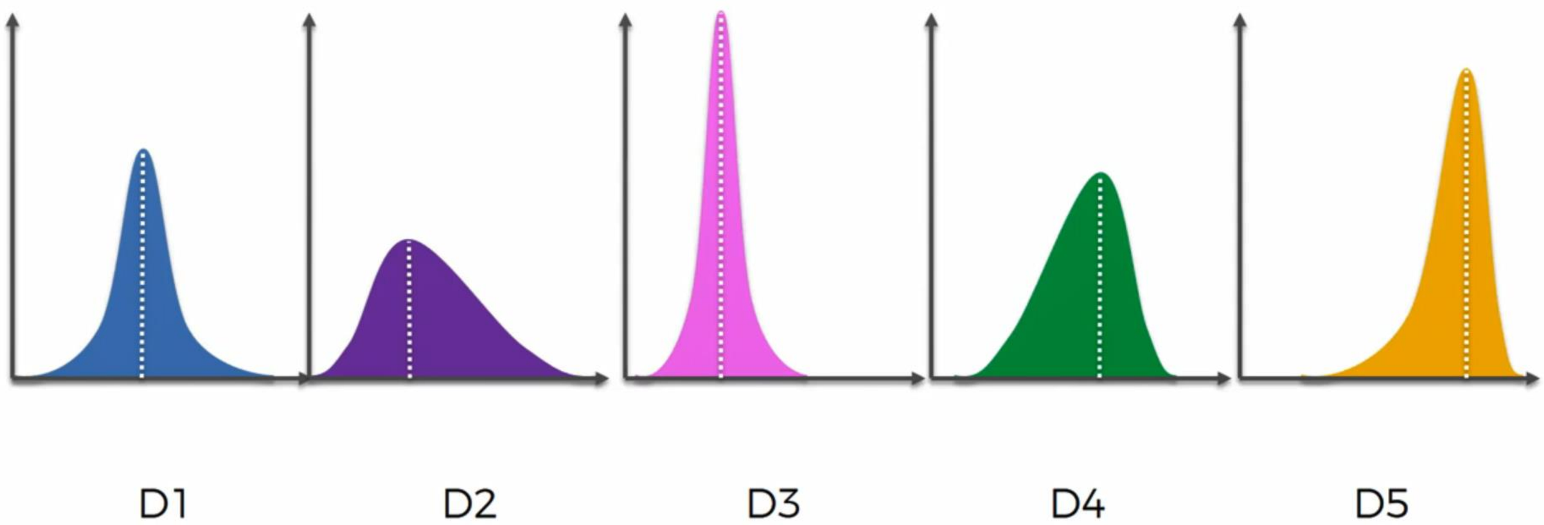
- You have some sort of agent that “explores” some space
- As it goes, it learns the value of diferente state changes in diferente conditions
- Those values inform subsequente behaviour of the agent
- Examples:
  - Pac-Man
  - Cat & Mouse Game
  - Multi-armed Bandit problem
- Yields fast on-line performance once the space has been explored



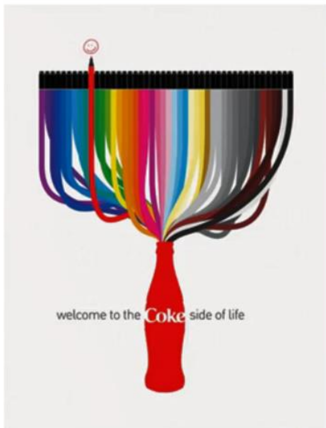
- Applications where reinforcement systems are applied:
  - Self Driving Cars
  - Gaming
  - Robotics
  - Recommendation Systems
  - Advertising and Marketing

- The Multi-armed Bandit Problem (Exploration vs Exploitation Problem)

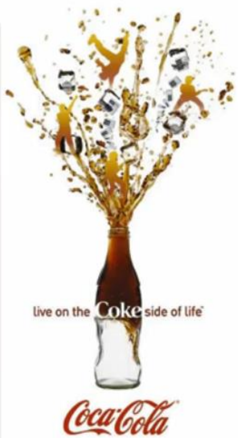
- The Multi-armed Bandit Problem (Exploration vs Exploitation Problem)



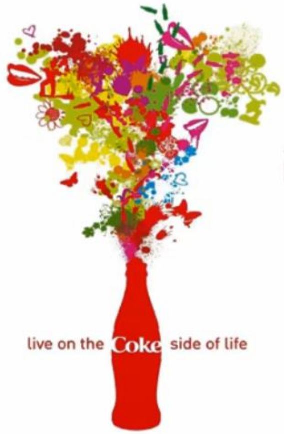
- Marketing - Ads Selection (Exploration vs Exploitation Problem)



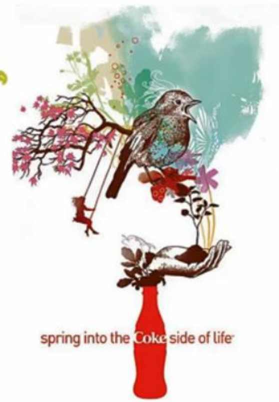
D1



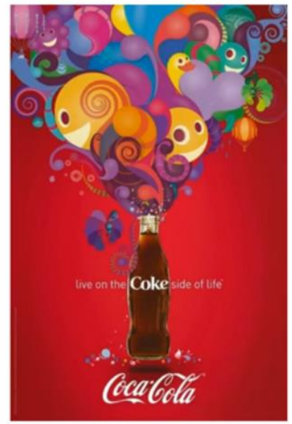
D2



D3



D4



D5

- Marketing - Ads Selection (Exploration vs Exploitation Problem)
  - We have d arms. For example, arms are ads that we display to users each time they connect to a web page
  - Each time a user connects to this web page, that makes a round
  - At each round n, we choose one ad to display to the user
  - At each round n, ad i gives reward:

    $r_i(n) \in \{0, 1\}$: $r_i(n) = 1$ if the user clicked on the ad i, 0 if the user didn't
  - Goal: maximize the total reward we get over many rounds

- Marketing - Ads Selection
  - Index: Person
  - Column: Ads
    - 1: Person clicked on Ad
    - 0: Person ignored Ad

Index	Ad 1	Ad 2	Ad 3	Ad 4	Ad 5	Ad 6	Ad 7	Ad 8	Ad 9	Ad 10
0	1	0	0	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	0
7	1	1	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	1	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0
12	0	0	0	1	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	1	0
14	0	0	0	0	0	0	0	1	0	0
15	0	0	0	0	1	0	0	1	0	0
16	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	1	0	0
19	0	0	0	0	0	0	0	0	1	0
20	0	1	0	0	0	0	0	1	0	0
21	0	0	0	0	1	0	0	0	0	1

- Marketing - Ads Selection
  - Random Selection

```python
# Random Selection

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing Random Selection
import random
N = 10000
d = 10
ads_selected = []
total_reward = 0
for n in range(0, N):
    ad = random.randrange(d)
    ads_selected.append(ad)
    reward = dataset.values[n, ad]
    total_reward = total_reward + reward

# Visualising the results - Histogram
plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
```

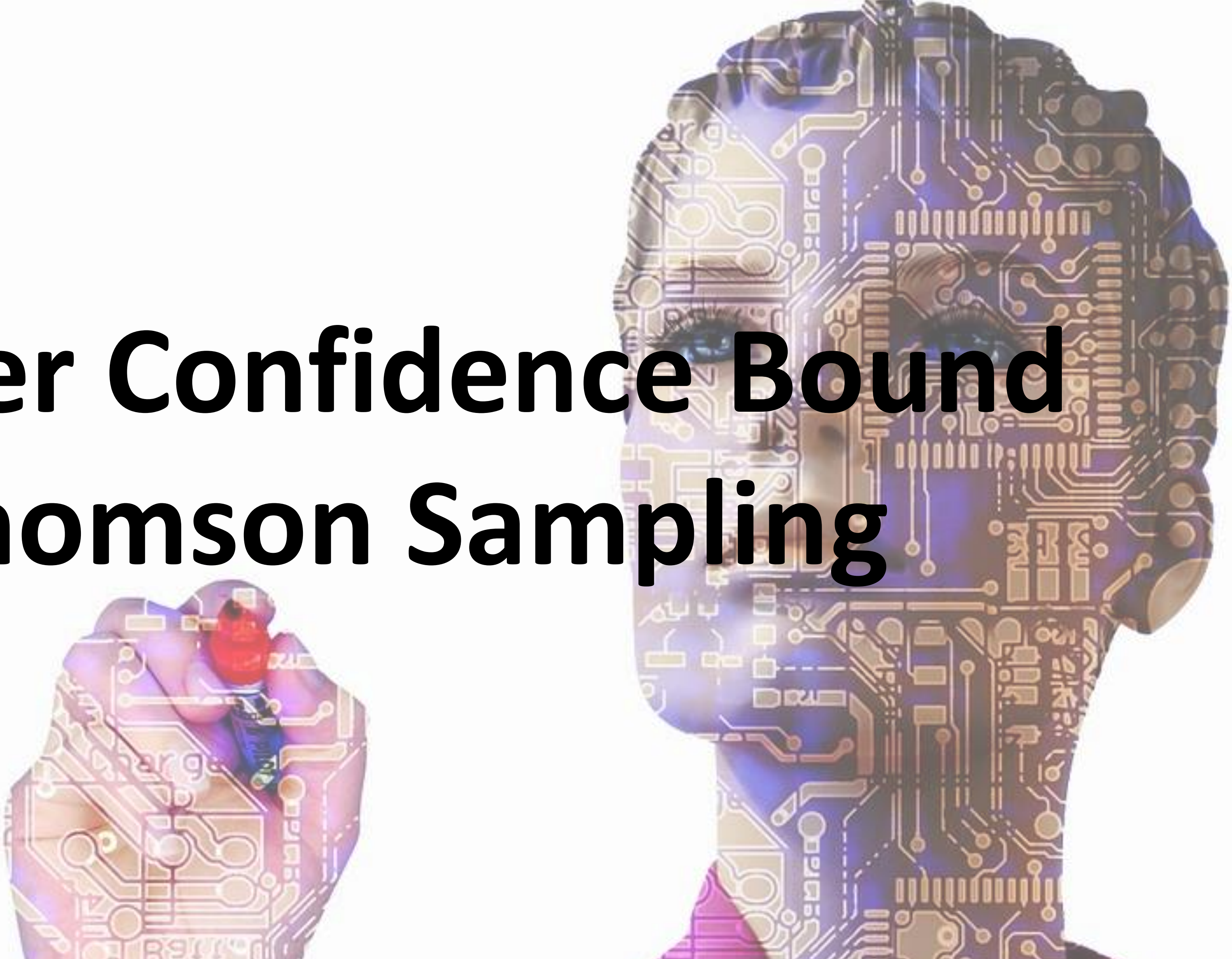
Computação Natural@ 2018/2019

- Marketing - Ads Selection
    - Random Selection

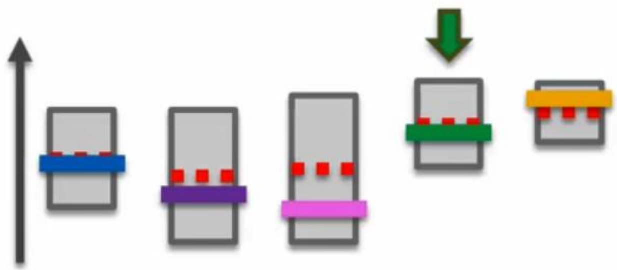
i	Type	Size	Value
0	int	1	4
1	int	1	6
2	int	1	1
3	int	1	0
4	int	1	4
5	int	1	5
6	int	1	0
7	int	1	8
8	int	1	5
9	int	1	3
10	int	1	8
11	int	1	5
12	int	1	8
13	int	1	4
14	int	1	0

total_reward	int64	1	1193

Upper Confidence Bound vs Thomson Sampling

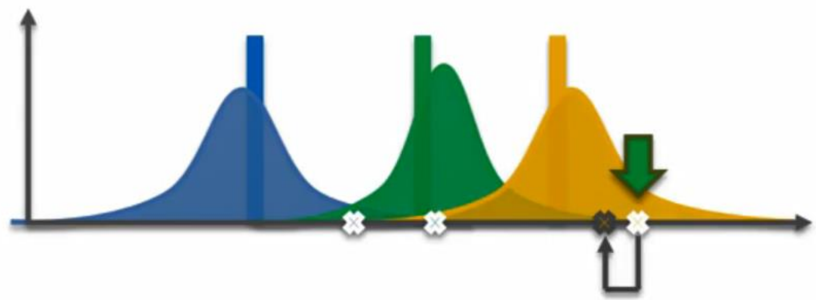


UCB



- Deterministic
- Requires update at every round

Thompson Sampling



- Probabilistic
- Can accommodate delayed feedback
- Better empirical evidence

- Upper Confidence Bound Algorithm

Step 1. At each round n, we consider two numbers for each ad i:

- $N_i(n)$ - the number of times the ad i was selected up to round n,
- $R_i(n)$ - the sum of rewards of the ad i up to round n.

Step 2. From these two numbers we compute:

- the average reward of ad i up to round n

$$\bar{r}_i(n) = \frac{R_i(n)}{N_i(n)}$$

- the confidence interval $[\bar{r}_i(n) - \Delta_i(n), \bar{r}_i(n) + \Delta_i(n)]$ at round n with

$$\Delta_i(n) = \sqrt{\frac{3}{2}\frac{\log(n)}{N_i(n)}}$$

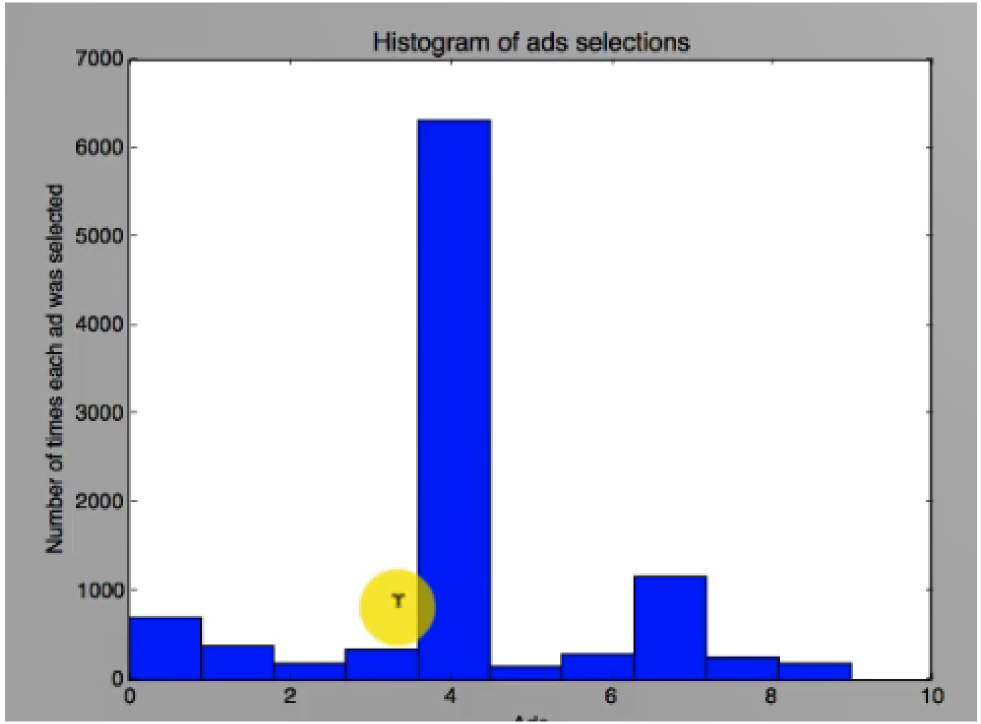
Step 3. We select the ad i that has the maximum UCB $\bar{r}_i(n) + \Delta_i(n)$.

- Marketing - Ads Selection
  - Upper Confidence Bound

```python
# Upper Confidence Bound

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing UCB
import math
N = 10000
d = 10
ads_selected = []
numbers_of_selections = [0] * d
sums_of_rewards = [0] * d
total_reward = 0
for n in range(0, N):
    ad = 0
    max_upper_bound = 0
    for i in range(0, d):
        if (numbers_of_selections[i] > 0):
            average_reward = sums_of_rewards[i] / numbers_of_selections[i]
            delta_i = math.sqrt(3/2 * math.log(n + 1) / numbers_of_selections[i])
            upper_bound = average_reward + delta_i
        else:
            upper_bound = 1e400
        if upper_bound > max_upper_bound:
            max_upper_bound = upper_bound
            ad = i
    ads_selected.append(ad)
    numbers_of_selections[ad] = numbers_of_selections[ad] + 1
    reward = dataset.values[n, ad]
    sums_of_rewards[ad] = sums_of_rewards[ad] + reward
    total_reward = total_reward + reward
```

- Marketing - Ads Selection
    - Upper Confidence Bound

1	Type	Size	Value
9985	int	1	4
9986	int	1	4
9987	int	1	4
9988	int	1	4
9989	int	1	4
9990	int	1	4
9991	int	1	4
9992	int	1	4
9993	int	1	4
9994	int	1	4
9995	int	1	4
9996	int	1	4
9997	int	1	4
9998	int	1	4
9999	int	1	4



total_reward	int64	1	2178
upper_bound	float64	1	0.3101723664789182

- Thompson Sampling Algorithm

Step 1. At each round n, we consider two numbers for each ad i:

- $N_i^1(n)$ - the number of times the ad i got reward 1 up to round n,
- $N_i^0(n)$ - the number of times the ad i got reward 0 up to round n.

Step 2. For each ad i, we take a random draw from the distribution below:

$$\theta_i(n) = \beta(N_i^1(n) + 1, N_i^0(n) + 1)$$

Step 3. We select the ad that has the highest $\theta_i(n)$.

- Thompson Sampling Algorithm
  - Ad i gets rewards $\boldsymbol{y}$ from Bernoulli distribution $p(\boldsymbol{y}|\theta_i) \sim \mathcal{B}(\theta_i)$.
  - θ_i is unknown but we set its uncertainty by assuming it has a uniform distribution $p(\theta_i) \sim \mathcal{U}([0,1])$, which is the prior distribution.
  - Bayes Rule: we approach θ_i by the posterior distribution

$$\underbrace{p(\theta_i|\mathbf{y})}_{\text{posterior distribution}} = \frac{p(\boldsymbol{y}|\theta_i)p(\theta_i)}{\int p(\boldsymbol{y}|\theta_i)p(\theta_i)d\theta_i} \propto \underbrace{p(\boldsymbol{y}|\theta_i)}_{\text{likelihood function}} \times \underbrace{p(\theta_i)}_{\text{prior distribution}}$$

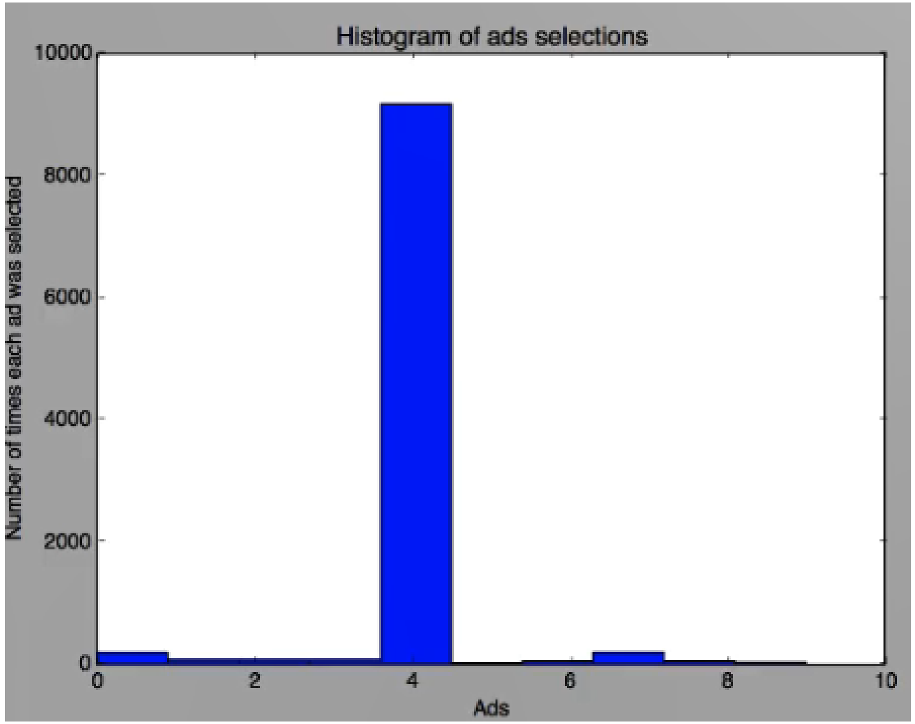
  - We get $p(\theta_i|\mathbf{y}) \sim \beta(\text{number of successes} + 1, \text{number of failures} + 1)$
  - At each round n we take a random draw $\theta_i(n)$ from this posterior distribution $p(\theta_i|\boldsymbol{y})$, for each ad i.
  - At each round n we select the ad i that has the highest $\theta_i(n)$.

- Marketing - Ads Selection
  - Thompson Sampling

```python
# Thompson Sampling

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing Thompson Sampling
import random
N = 10000
d = 10
ads_selected = []
numbers_of_rewards_1 = [0] * d
numbers_of_rewards_0 = [0] * d
total_reward = 0
for n in range(0, N):
    ad = 0
    max_random = 0
    for i in range(0, d):
        random_beta = random.betavariate(numbers_of_rewards_1[i] + 1, numbers_of_rewards_0[i] + 1)
        if random_beta > max_random:
            max_random = random_beta
            ad = i
    ads_selected.append(ad)
    reward = dataset.values[n, ad]
    if reward == 1:
        numbers_of_rewards_1[ad] = numbers_of_rewards_1[ad] + 1
    else:
        numbers_of_rewards_0[ad] = numbers_of_rewards_0[ad] + 1
    total_reward = total_reward + reward
```

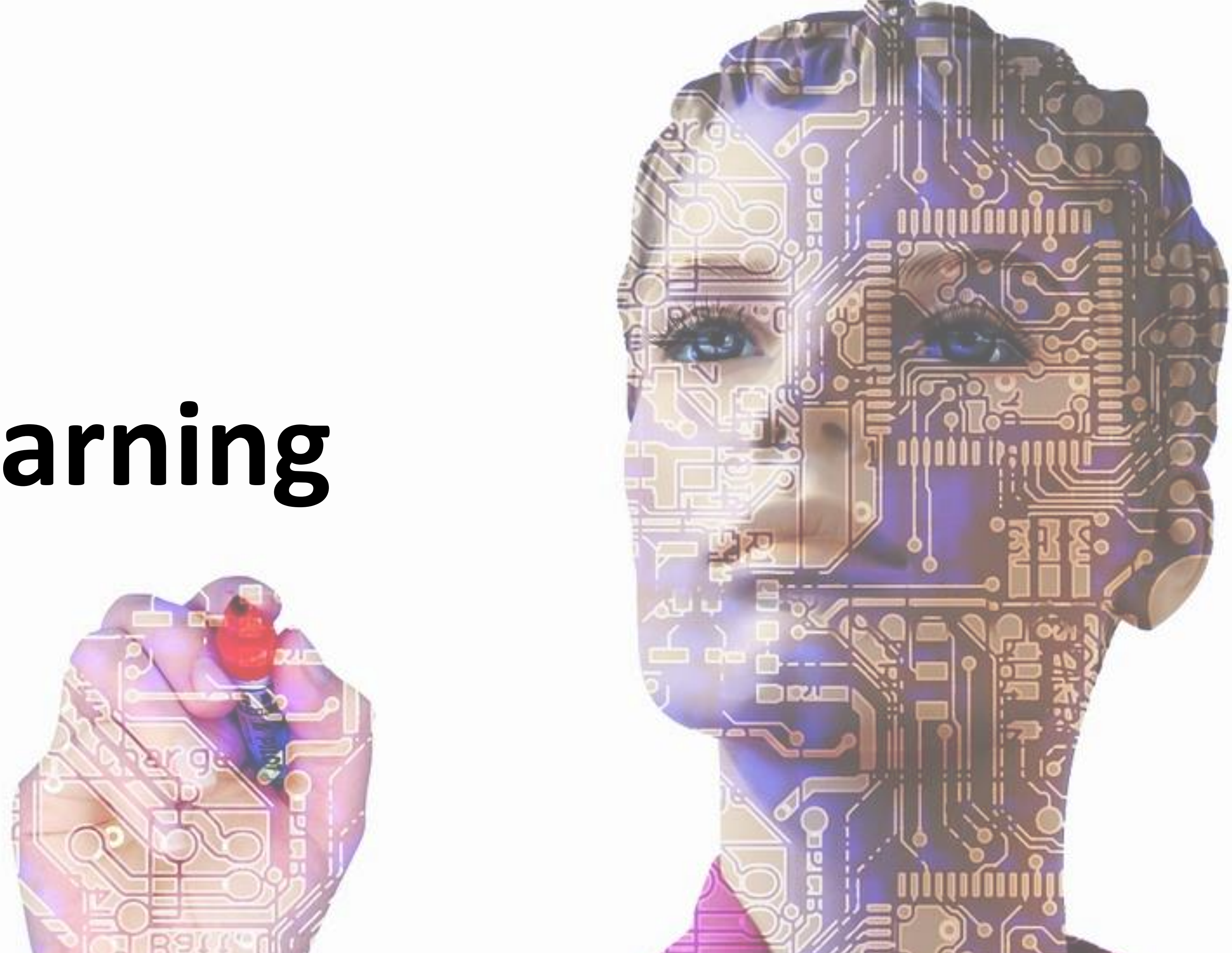
- Marketing - Ads Selection
  - Thompson Sampling

i	Type	Size	Value
9985	int	1	4
9986	int	1	4
9987	int	1	4
9988	int	1	4
9989	int	1	4
9990	int	1	4
9991	int	1	4
9992	int	1	4
9993	int	1	4
9994	int	1	4
9995	int	1	4
9996	int	1	4
9997	int	1	4
9998	int	1	4
9999	int	1	4

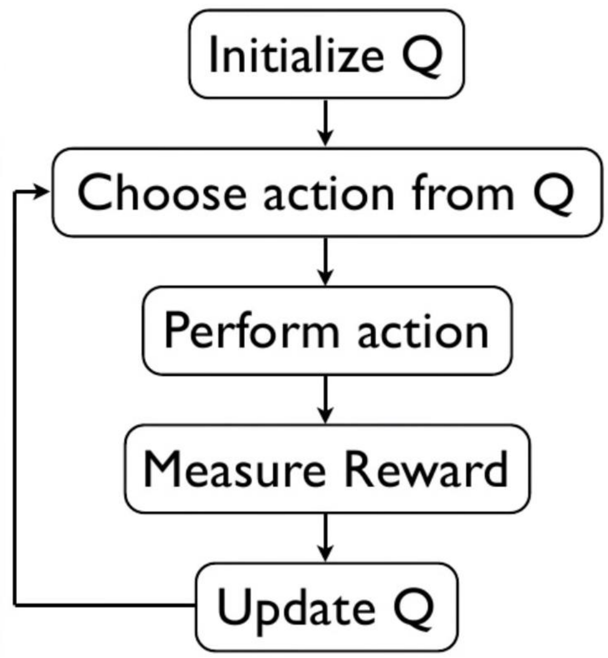


total_reward	int64	1	2613

Q-Learning



- A specific implementation of reinforcement learning:
- Determined by:
    - Set of environmental states S (called State)
    - Set of possible actions in those states A (called Actions)
    - Value of each state/action Q (called Q-value or Action-value)
- Start off with Q values of 0 / random-values
- Explore the space
- As bad things happen after a given state/actions, reduce its Q-value
- As rewards happen after a given state/action, increase its Q-value

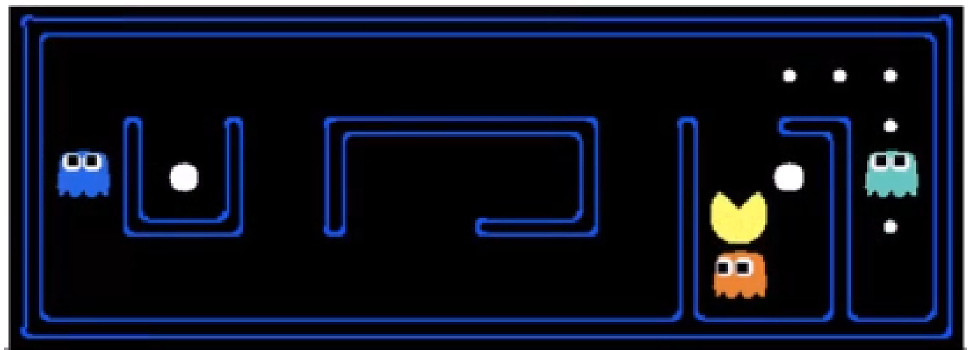


- In RL we want to obtain a function Q(S,A) that predicts the best action A in state S in order to maximize a cumulative reward
- This function can be estimated using Q-learning, which iteratively updates Q(S,A) using the Bellman Equation

$$Q(s,a) = \underbrace{r}_{\text{immediate reward}} + \underbrace{\overset{\text{discount factor = 0.90}}{\gamma} max_{a'} Q(s', a')}_{\text{future reward}}$$

Pac-man game exemple (analyse the figure below):

- What are some state/actions here?
    - Pac-man has a wall to the West
    - Pac-man dies if he moves one step South
    - Pac-man just continues to live if going North or East
- You can "look ahead" more than one step by using a discount factor when computing Q (where S is previous state, S' is current state)



- Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker
    - MDP's are just a way to describe what was mentioned using mathematical notation
- States are still described as S and S'
- State transition functions are described as: $P_a(s, s')$
- "Q" values are described as a reward function: $R_a(s, s')$

- The Markov Decision Process can be designed as:
    - Set of states, S
    - Set of actions, A
    - Reward function, R
    - Policy, π
    - Value, V
- We take an action (A) to transition from our start state to our end state (*S*) -> getting rewards (R) for each action we take
- Our actions can lead to a positive reward or negative reward
- The set of actions we took define our policy (π) and the rewards we get in return defines our value (V)

- Take into account the following steps:
  - Initialize Q Matrix (defines reward matrix, where lines define States and columns define Actions to transport to other States)
  - Choose action from Q
  - Perform action
  - Measure Reward and Update Q
  - Repeat

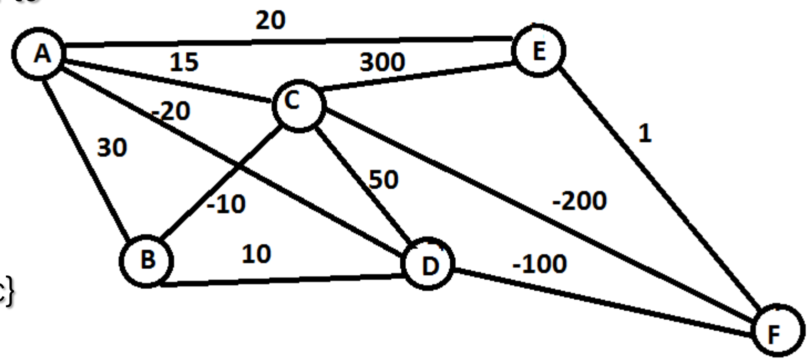
$$\text{States}\;\left\{\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\right.\overbrace{\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}}^{\text{Actions}}$$

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 100 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 80 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Q-values Matrix

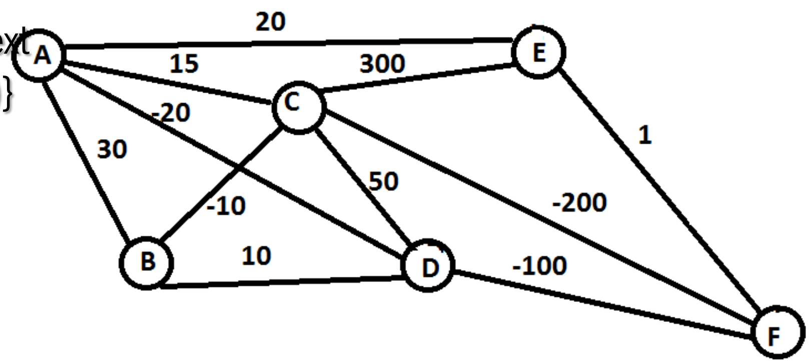
Example: Shortest path problem

- Task: Go from A to F, with as low cost as possible
- Numbers at each edge between two places represent the cost taken to traverse the distance
- Negative cost are actually some earnings on the way
- Value is the total cumulative reward when you do a policy:
  - The set of states are the nodes: {A, B, C, D, E, F}
  - The action to take is to go from one place to other: {A -> B, C -> D, etc}
  - The reward function is the value represented by edge, i.e. cost
  - The policy is the "way" to complete the task: {A -> C -> F}

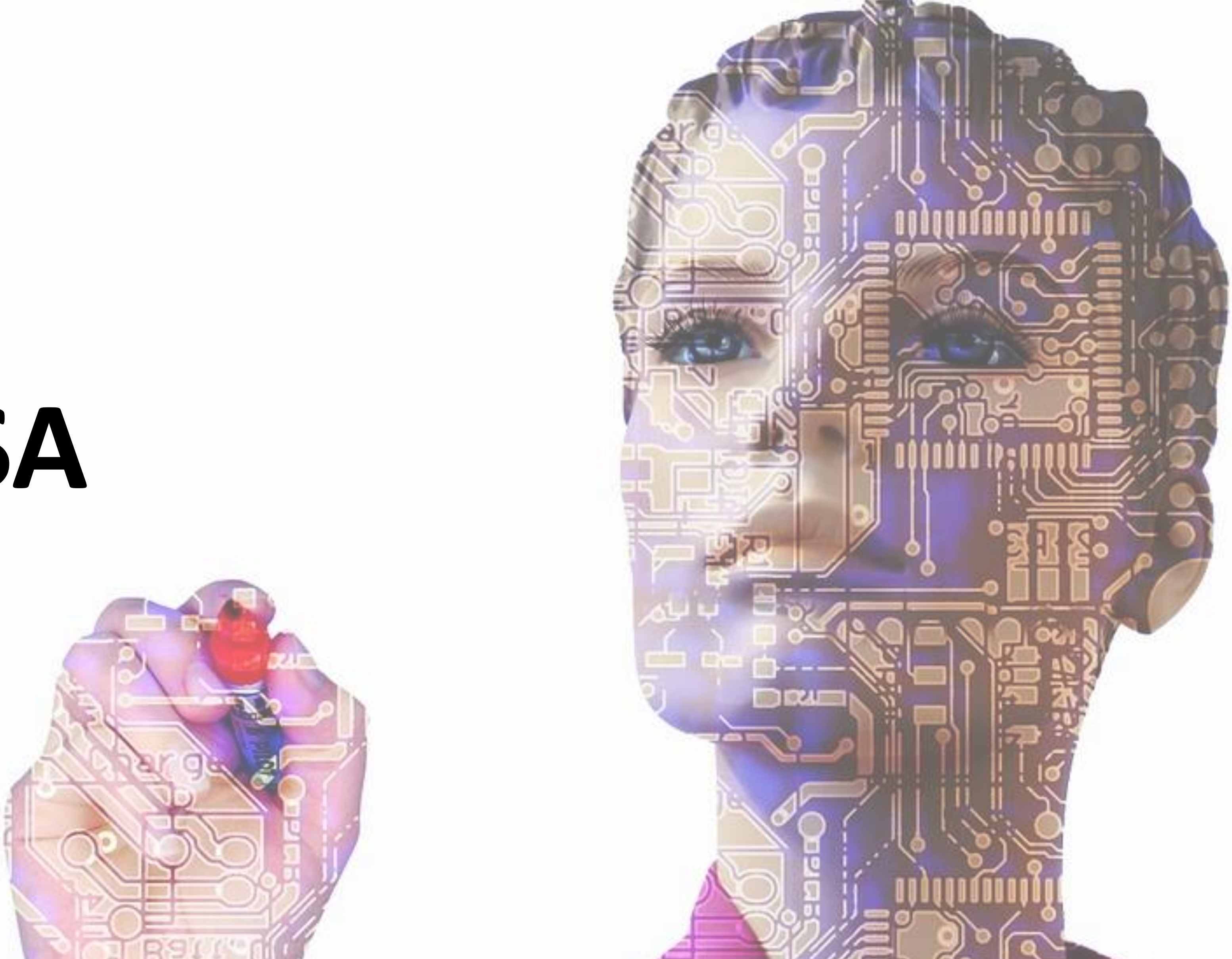


Example: Shortest path problem

- At place A, the only visible path is your next destination (a.k.a observable space)
- Algorithm can take a greedy approach and take the best possible next step, which is going from {A -> D} from a subset of {A -> (B, C, D, E)}
- At place D, it should go to place F, since it can choose from {D -> (B, C, F)}. We see that {D -> F} has the lowest cost and hence we take that path
- Our policy was to take {A -> D -> F} and our Value is -120
- The implemented algorithm is known as **epsilon greedy**
    - **Limitation: it does not explore other alternatives (A->C->F)**
    - **Solution: add a probability of random exploration**



SARSA



- State-Action-Reward-State-Action (SARSA) very much resembles Q-learning
- Key difference: SARSA learns the Q-value based on the action performed by the current policy instead of the greedy policy

- SARSA $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
  - The action a_{t+1} is the action performed in the next state s_{t+1} under current policy

- Q-learning $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$

  - Also written as: $Q(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * \max_a Q(s_{t+1}, a))$, where γ is the discount factor and r_t is the reward received from the environment at timestep t

- Q-Learning: $Q(s_{t+1},a_{t+1}) = \max_a Q(s_{t+1},a)$, i.e., under the ε-greedy policy, Q-Learning computes the difference between Q(s,a) and the maximum action value

- SARSA: $Q(s_{t+1},a_{t+1}) = \varepsilon \cdot \text{mean}_a Q(s_{t+1},a) + (1-\varepsilon) \cdot \max_a Q(s_{t+1},a)$, i.e., SARSA computes the difference between Q(s,a) and the weighted sum of the average action value and the maximum

SARSA(λ): Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
Sates $\mathcal{X} = \{1, \dots, n_x\}$
Actions $\mathcal{A} = \{1, \dots, n_a\}$, $\quad A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
$\lambda \in [0, 1]$: Trade-off between TD and MC

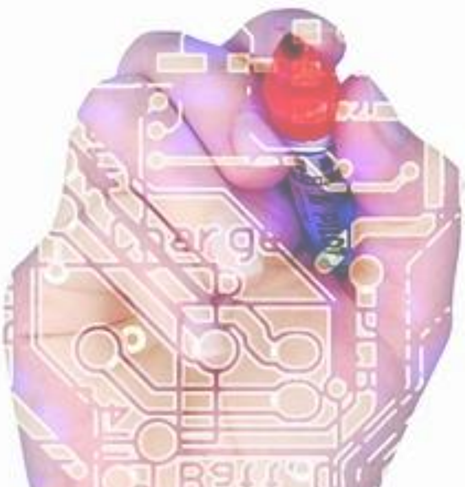
```
procedure QLearning(𝒳, A, R, T, α, γ, λ)
    Initialize Q : 𝒳 × 𝒜 → ℝ arbitrarily
    Initialize e : 𝒳 × 𝒜 → ℝ with 0.                ▷ eligibility trace
    while Q is not converged do
        Select (s, a) ∈ 𝒳 × 𝒜 arbitrarily
        while s is not terminal do
            r ← R(s, a)
            s′ ← T(s, a)                                ▷ Receive the new state
            Calculate π based on Q (e.g. epsilon-greedy)
            a′ ← π(s′)
            e(s, a) ← e(s, a) + 1
            δ ← r + γ · Q(s′, a′) − Q(s, a)
            for (s̃, ã) ∈ 𝒳 × 𝒜 do
                Q(s̃, ã) ← Q(s̃, ã) + α · δ · e(s̃, ã)
                e(s̃, ã) ← γ · λ · e(s̃, ã)
            s ← s′
            a ← a′
    return Q
```

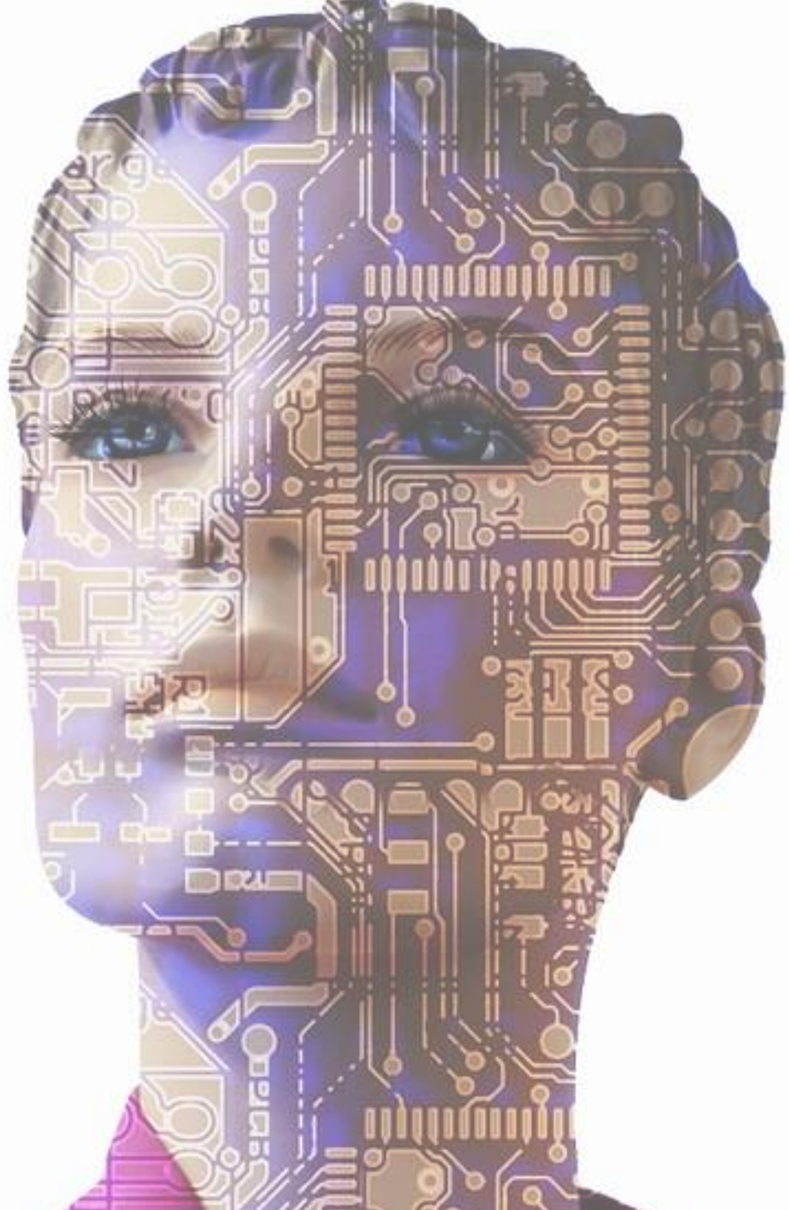
Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
Sates $\mathcal{X} = \{1, \dots, n_x\}$
Actions $\mathcal{A} = \{1, \dots, n_a\}$, $\quad A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$

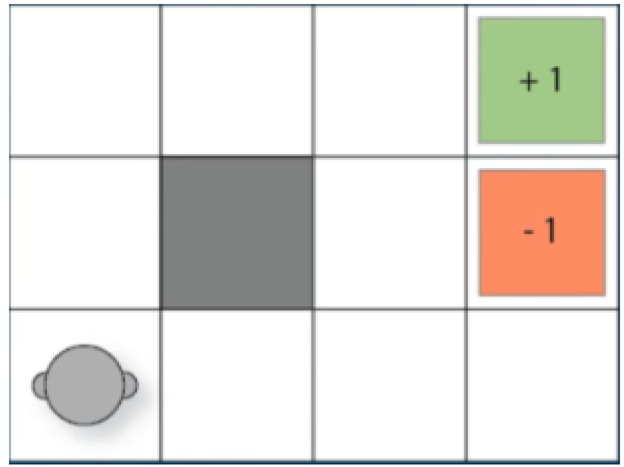
```
procedure QLearning(𝒳, A, R, T, α, γ)
    Initialize Q : 𝒳 × 𝒜 → ℝ arbitrarily
    while Q is not converged do
        Start in state s ∈ 𝒳
        while s is not terminal do
            Calculate π according to Q and exploration strategy (e.g. π(x) ←
arg max_a Q(x, a))
            a ← π(s)
            r ← R(s, a)                                  ▷ Receive the reward
            s′ ← T(s, a)                                 ▷ Receive the new state
            Q(s′, a) ← (1 − α) · Q(s, a) + α · (r + γ · max_a′ Q(s′, a′))
            s ← s′
    return Q
```

Exercise





- Game to be solved:
    - Possible actions: Up, Down, Left, Right
    - (1,1) -> Wall, can't go here
    - (2,0) -> start position
    - (0,3) -> terminal (+1 reward)
    - (1,3) -> terminal (-1 reward)
- Formulation:
    - 12 positions
    - 11 states (where the robot is)
    - 4 actions
    - Small game!
    - But many concepts to be learned
- Compare the performance of the SARSA and Q-Learning algorithms



Gridworld

Implementing Reinforcement Learning:

- Python Markov Decision Process Toolbox:
  - http://pymdptoolbox.readthedocs.org/en/latest/api/mdp.html
- Cat & Mouse Example:
  - https://github.com/studywolf/blog/tree/master/RL/Cat%20vs%20Mouse%20exploration
- Pac-Man Example:
  - https://inst.eecs.berkeley.edu/~cs188/sp12/projects/reinforcement/reinforcement.html

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mestrado Integrado em Engenharia Informática
Computação Natural
2018/2019

Paulo Novais, Cesar Analide, Filipe Gonçalves