Introduction to Reinforcement Learning

Learning the Basics

by Vantec & RASTEC

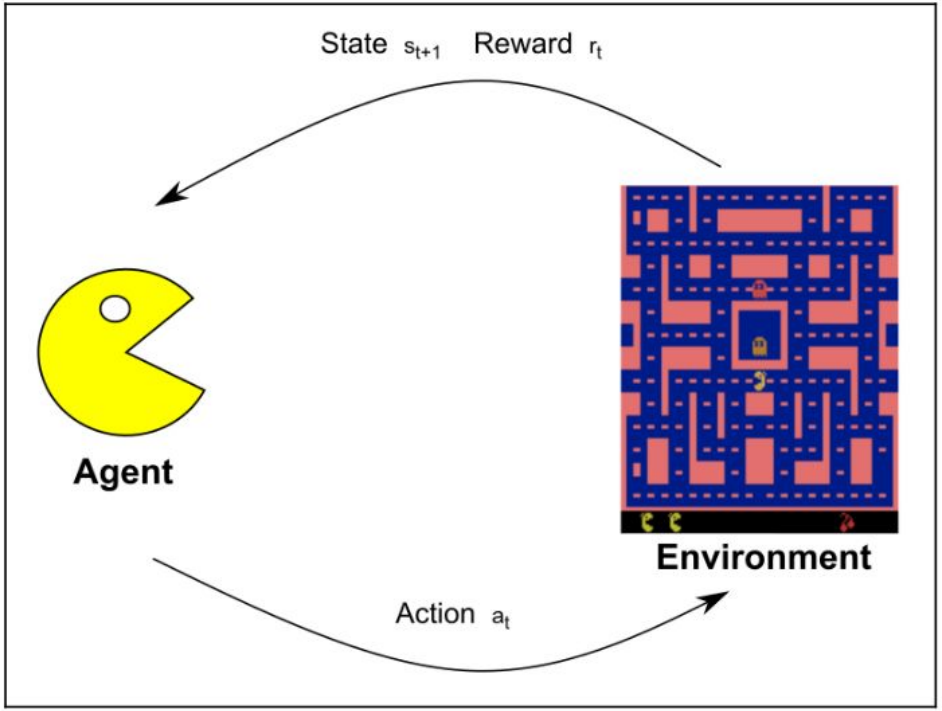
GitHub Day 1

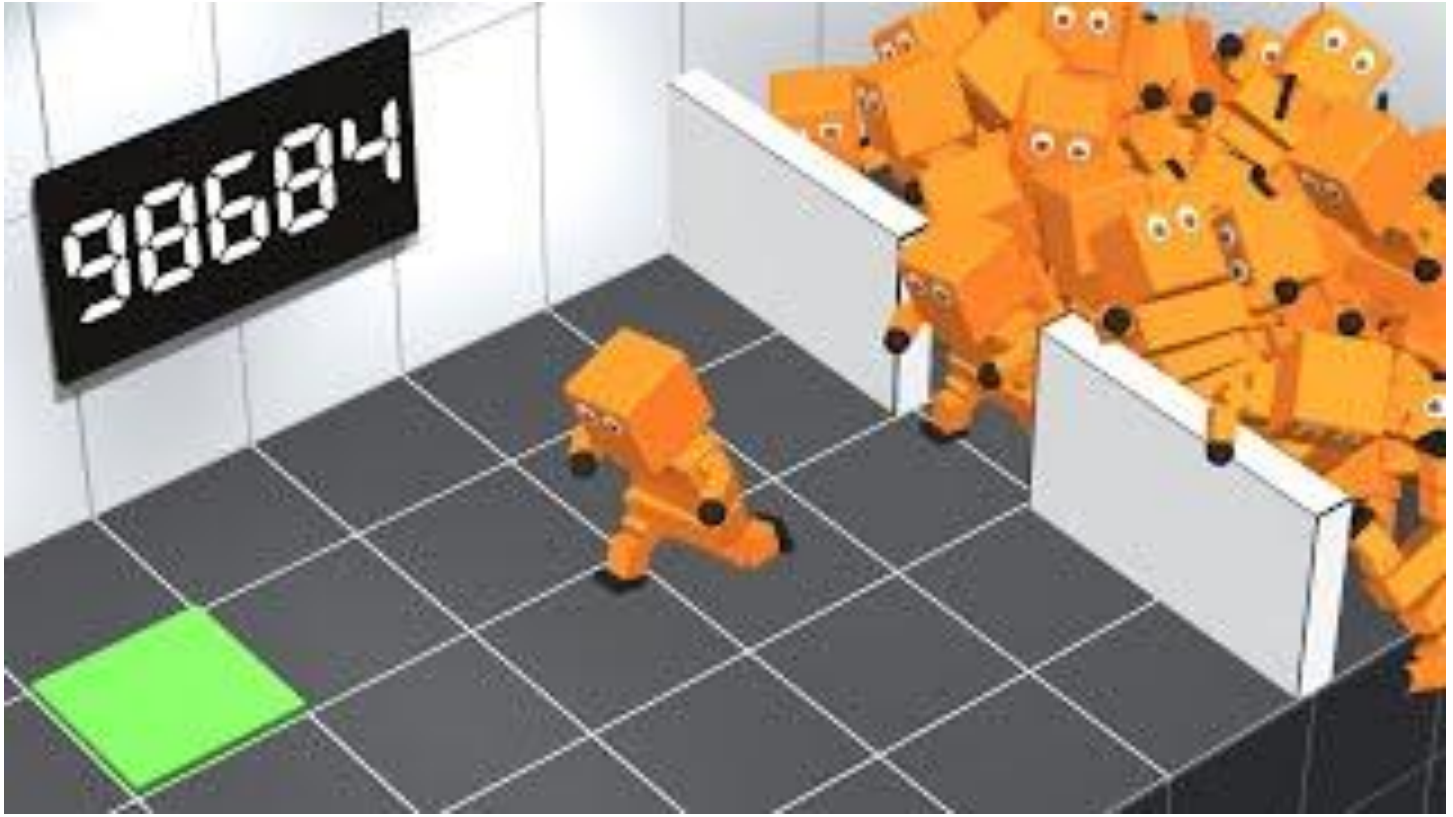
https://github.com/EMendezJ/LARS-RL-TUTORIAL

What is Reinforcement Learning?

Reinforcement Learning (RL) is a computational approach where an **agent** learns to make decisions by **interacting** with an **environment**, receiving feedback in the form of **rewards**.

What is Reinforcement Learning?

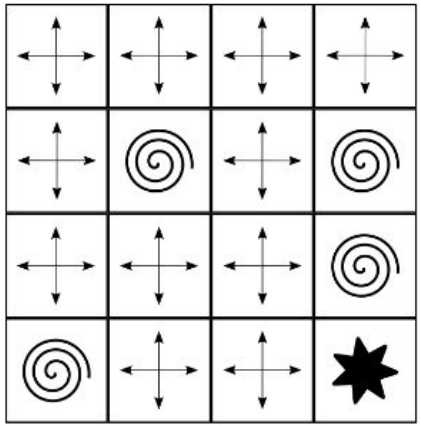



98684

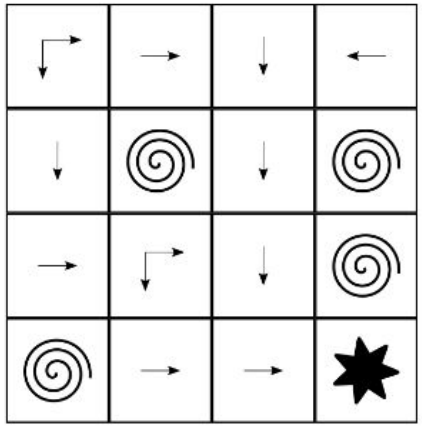
Elements of RL

Policy

A **policy** defines how the **agent** chooses its **actions** based on the **current state**



Random Initial Policy



Final Optimal Policy

*The policy chooses the action that maximizes the cumulative reward from that state.

- Mappings from states to actions.
- Says what actions to perform in each state.

On-Policy

The policy used to make decisions is the same one being improved.

The agent explores and learns from its own behavior.

Off-Policy

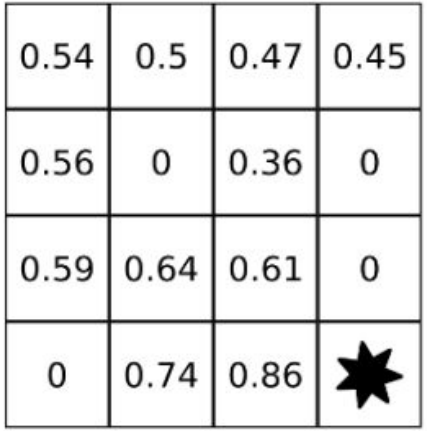
The policy being learned (target policy) is different from the one used to collect experience (behavior policy).

Usually, the behavior policy explores a lot, while the target policy is the optimal one.

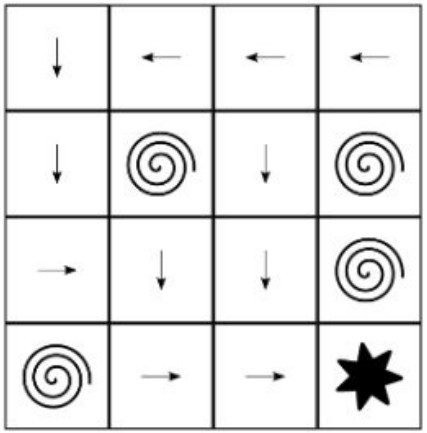
The Value Function

The **value function** tells us how good it is for the agent to be in a certain state (or to take a certain action in that state), in terms of **expected long-term reward**.

A high reward doesn't imply a high-value function and a low reward doesn't imply a low-value function.



Final States Values

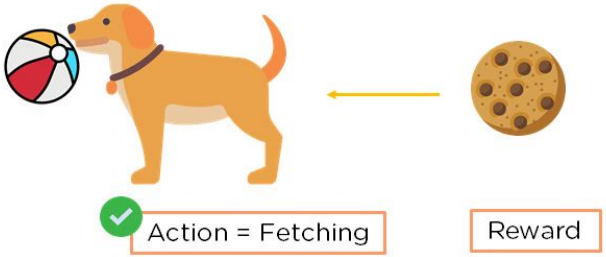


Corresponding Optimal Policy

Reward

A **reward** is a **numerical feedback signal** that tells the agent **how good or bad** its last action was.

- In a **game**, the reward could be the score gained or lost after each move.
- In **robotics**, it might be based on how close the robot gets to a target.
- In **finance**, it could be the profit or loss after a trade.

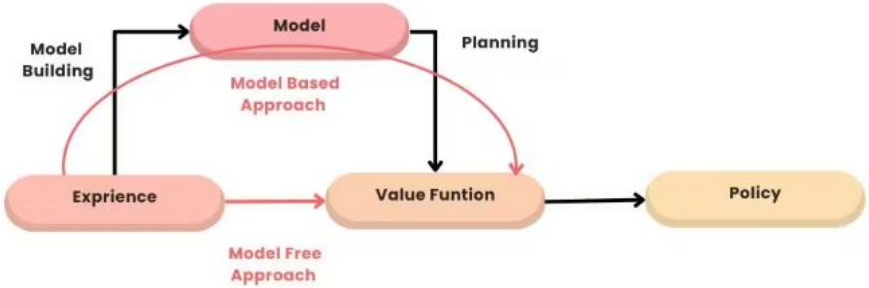


Model

The **model** details how the **environment behaves**, **predicting** the **next state** and the **reward**, given a state and an action.

- In a **chess game**, the agent can simulate possible future moves and outcomes.

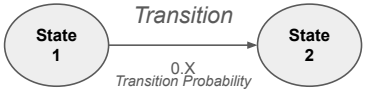
*It's **very difficult** (and often impossible) to have an accurate model of the environment in real-world scenarios.



CODE TIME!

The Markov Decision Process

We can abstract certain situations into *Markov decision chains.*



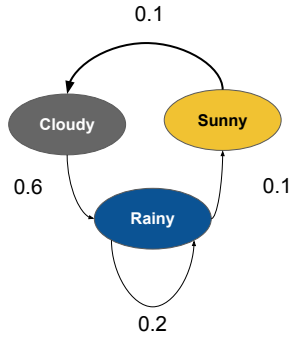
Markov Decision Chain

"It will rain in an hour because it's currently cloudy"

Interpreted as

Current State	Next State	Transition Probability
Cloudy	Rainy	0.6
Rainy	Rainy	0.2
Sunny	Cloudy	0.1
Rainy	Sunny	0.1

State Table



State Diagram

Elements of the Markov Decision Processes

MDP is an extension of the Markov Chain

Almost all RL problems can be represented as MDPs

Elements of the MDP

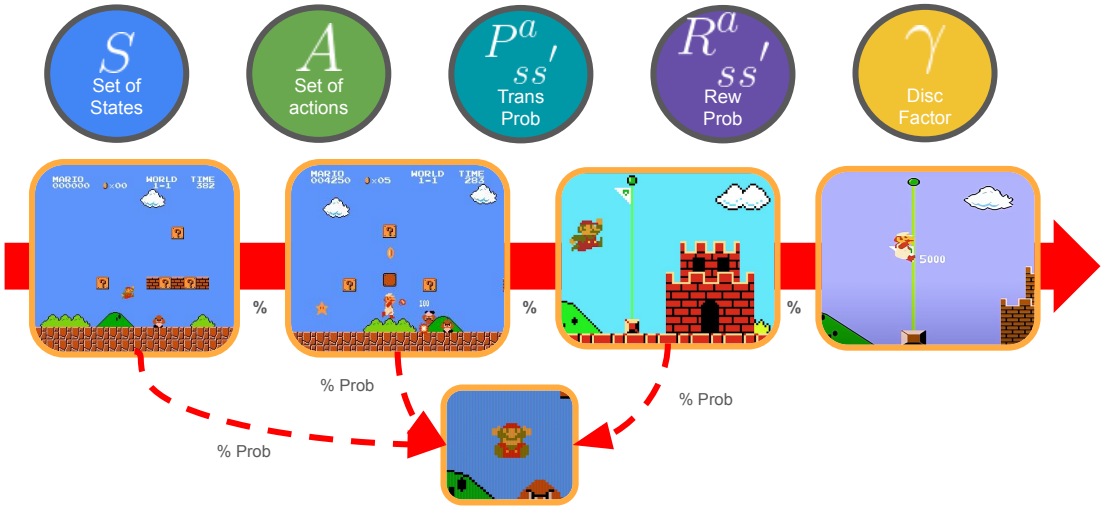
1. Set of states (S) for the agent to be in

2. Set of Actions (A) the agent can perform to go from state to state.

3. Transition Probability ($P^a_{ss'}$) of moving between states via some action.

4. Transition Probability ($R^a_{ss'}$) of moving between states via some action.

5. Discount Factor (γ), controls the importance of immediate and future rewards.



Example of the Markov Decision Process present in Super Mario Bros

Understanding Rewards and Returns

An agent interacts with its environment by performing an action. It changes its state and receives a reward.

An agent tries to maximize the total amount of rewards it receives from the environment instead of immediate rewards.

The total amount of rewards the agent receives from the environment are called returns.

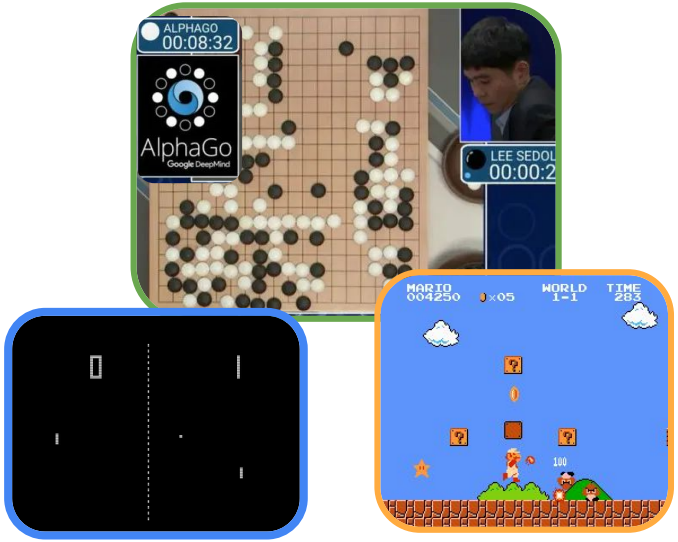
Good action +1

Bad action -1

$$R_t = r_{t+1} + r_{t+2} + ... + r_T$$

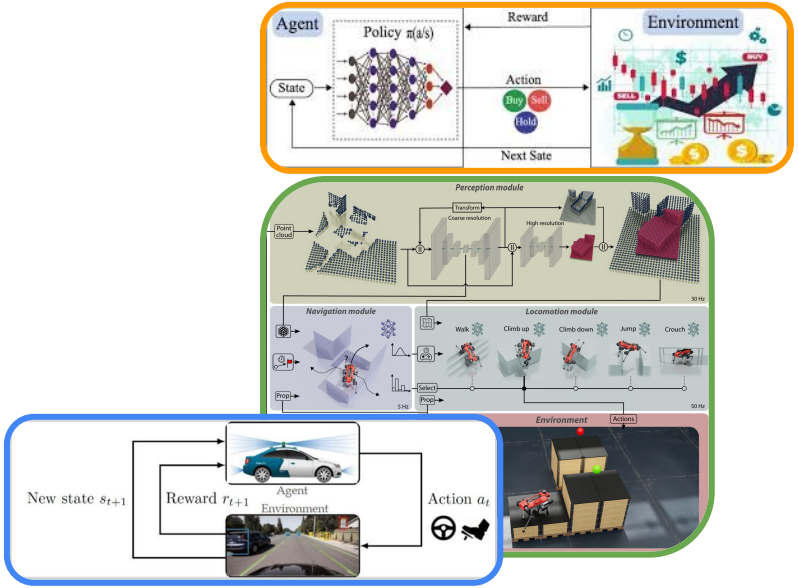
Returns Based on Time

Episodic & Continuous Tasks



Episodic

They have a terminal State



Continuous

Goes on Indefinitely

The Discount Factor (γ)

Agent goal = Maximize Return

$R_t = r_{t+1} + r_{t+2} + ... + r_T$ **Episodic**

$R_t = r_{t+1} + r_{t+2} + ...$ **Continuous**

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ...$$

General Form

Importance is given to immediate and future rewards depending on the use case.

0 to 1

0 = Immediate rewards

1 = Future Rewards

Markov Decision Processes (MDP) + Monte Carlo (maze solver)

Policy Function

$$\pi(s) : S \rightarrow A$$

Indicates mappings from states to actions. Says what actions to perform in each state. Ultimate goal lies in finding the optimal policy which specifies the correct action to perform in each state. Maximizing the reward.

The State Value Function

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t | s_t = s]$$

$$V^{\pi}(s) = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s]$$

Specifies the expected return starting from state s according to policy π

We follow the value of being in a state. The better the value, the better the state!

Markov Decision Processes (MDP) + Monte Carlo (maze solver)

State-action Value Function (Q function)

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t | s_t = s, a_t = a]$$

Value of taking an action in a state following a policy π

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a]$$

Difference between value function and Q function is that the value function specifies the goodness of a state, while the Q function specifies the goodness of an action in a state.

State-action Value Function (Q function)

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} | s_t = s, a_t = a]$$

Q functions can be viewed in a value table. Also called a Q table, let's say we have 2 states and 2 actions. Our Q table looks like the following:

The Q table shows the value of all possible state-action pairs. By looking at the table we can conclude that performing action 1 in state 1 and action 2 in state 2 is the better option as it has high value.

State	Action	Value
State 1	Action 1	0.03
State 2	Action 2	0.02
State 2	Action 1	0.5
State 2	Action 2	0.9

Whenever we say value function V(S) or Q function Q(S, a), it actually means the value table and Q table.

Markov Decision Processes (MDP) + Monte Carlo (maze solver)

Bellman Equation

- Helps us find the optimal policies and value functions (Solve the MDP).
- It is omnipresent in RL
- There can be many different value functions according to different policies.
- Optimal value function V*(s) yields the maximum value according to all other value functions.

$$V^*(s) = max_\pi V^\pi(s)$$

The optimal policy is the one that results in an optimal value functions.

The optimal policy is the one that results in an optimal value functions.

Optimal value function can be computed by taking the maximum of the Q function: $V^*(s) = max_a Q^*(s, a)$

Markov Decision Processes (MDP) + Monte Carlo (maze solver)

Bellman Equation for the Value function can be represented as:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s') \right]$$

It indicates the recursive relation between a value of a state and its successor state and the average over all possibilities.

Bellman Equation for the Q function can be represented as:

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} [\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} \pi(a' \mid s') Q^{\pi}(s',a')]$$

$$V^{*}(s) = \max_{a} \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{*}(s') \right]$$

Bellman Optimality Equation

Markov Decision Processes (MDP) + Monte Carlo (maze solver)

Solving the Bellman Equation

We use **Dynamic Programming**

Break the problem into subproblems, for each subproblem we compute and store the solution. If the same subproblem occurs, we do not recompute the solution. We use the previously computed one. It helps save a lot of computational time.

We have 2 powerful algorithms:

- Value iteration
- Policy iteration

Value Iteration

We start with a random value function. We start looking for an optimal one in iterative fashion until we find the optimal value function.

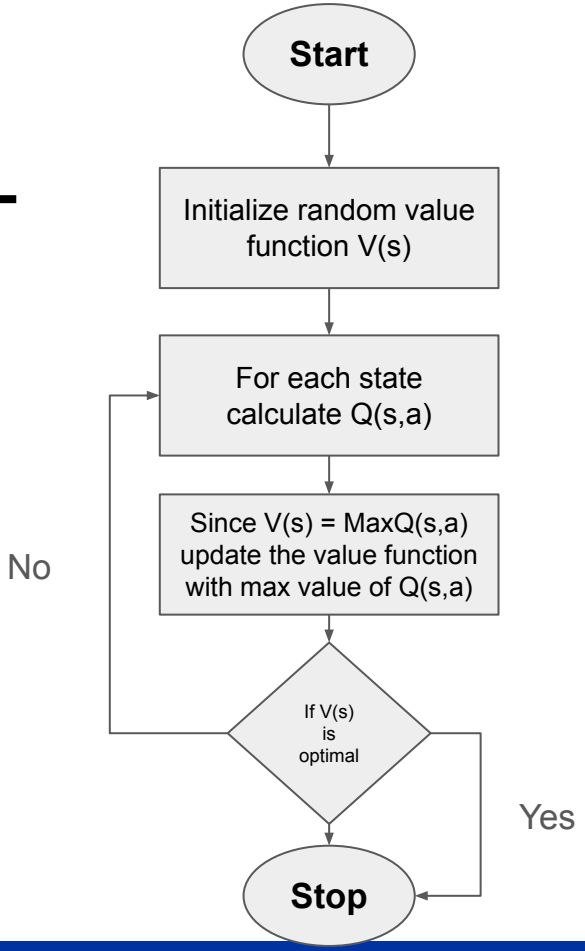
Policy Iteration

We start with a random policy, then we find its value function. If it is not optimal then we find the new improved policy.

Value Iteration

We start with a random value function. We start looking for an optimal one in iterative fashion until we find the optimal value function.

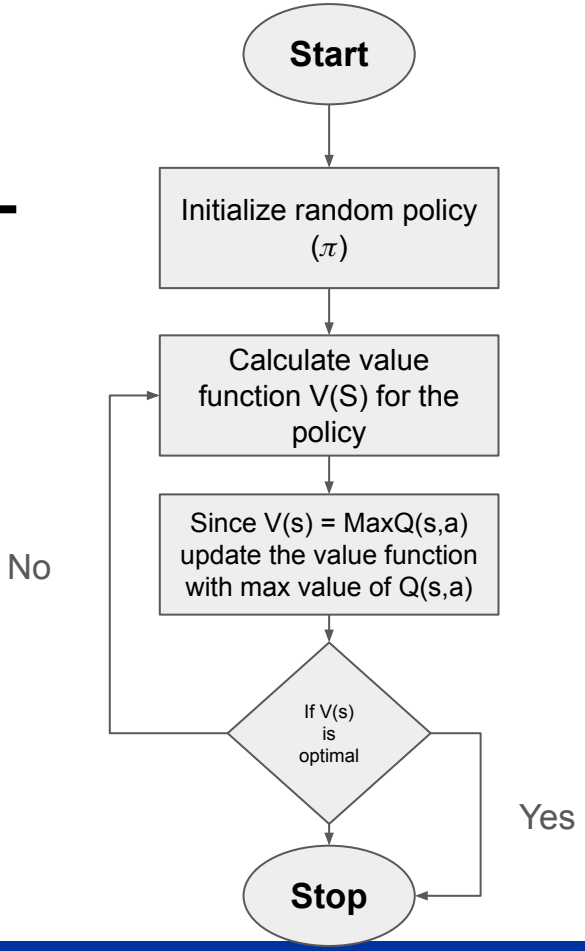
1. **Initialize random value function for each state.**
2. **Compute the Q function for all state action pairs Q(s,a)**
3. **Update value function with max value from Q(s,a)**
4. **We repeat these steps until the value change is really small.**



Policy Iteration

We start with a random policy, then we find its value function. If it is not optimal then we find the new improved policy.

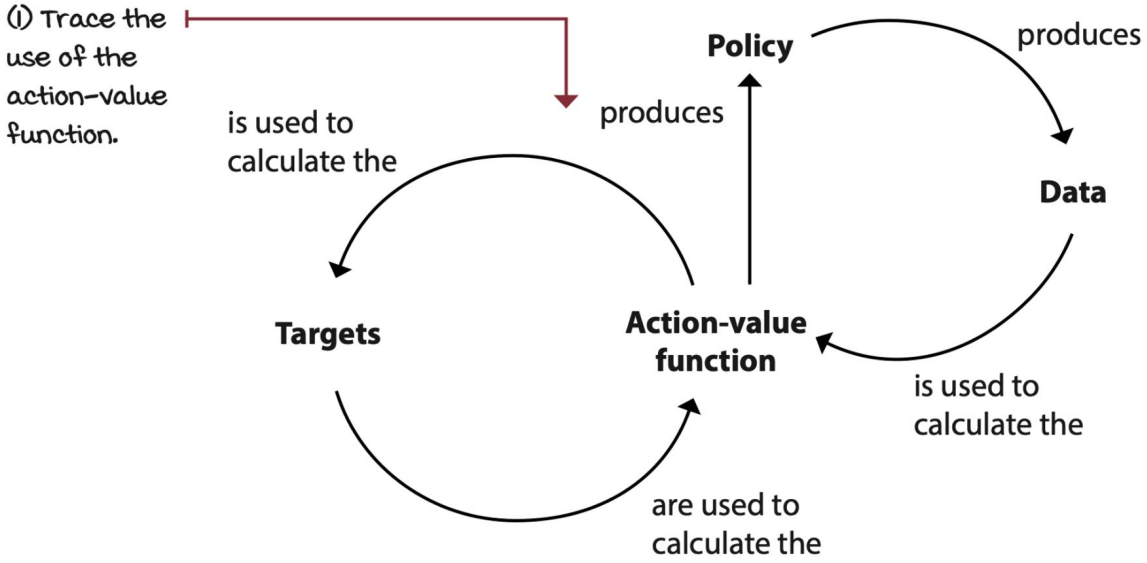
1. **Initialize a random policy**
2. **We find the value function for that random policy and evaluate to check if it's optimal (policy evaluation).**
3. **If not optimal we find a new improved policy, called policy improvement.**
4. **We repeat these steps until we find the optimal policy**



Solving the Frozen Lake Problem

TO THE NOTEBOOK!!

Action-Value functions



An action-value function (or **Q-function**) represents the expected return (future cumulative reward) of taking a particular action in a given state and then following a specific policy thereafter.

Action-Value functions (Bellman Equation)

$$V_\pi(s) = \mathbf{E}_\pi[R_{t+1} + \gamma * V_\pi(S_{t+1}) | S_t = s]$$

Value of state s

Expected value of immediate reward

And uses the policy to choose its actions for all time steps

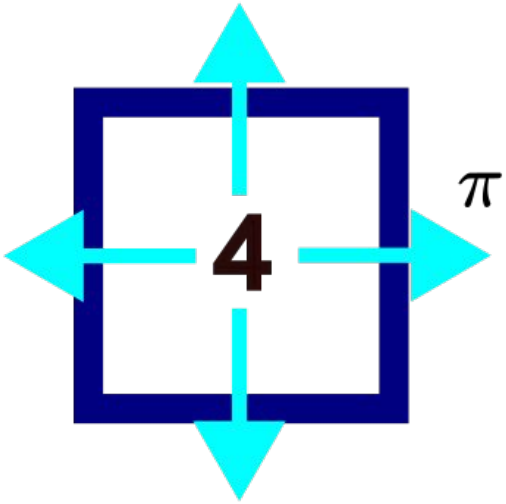
+ the discounted value of next_state

If the agent starts at state s

Action-Value functions

$$\pi(UP|4) = 0.2$$

$$\pi(RIGHT|4) = 0.2$$



$$\pi(LEFT|4) = 0.3$$

$$\pi(DOWN|4) = 0.3$$

Temporal Difference Learning

- Doesn’t require model dynamics
- Doesn’t wait until the end of the episode to estimate the VF.
- Approximates current estimate based on previously learned estimate (bootstrapping).

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

New value of state t

Former estimation of value of state t

Learning Rate

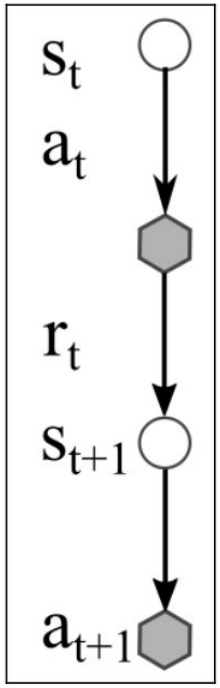
Reward

Discounted value of next state

TD Target

SARSA and Q-Learning

SARSA

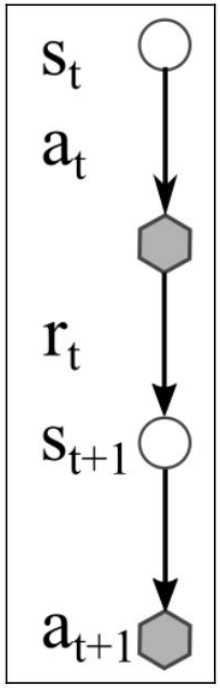


The name **SARSA** comes from the update that is based on the state, s_t; the action, a_t; the reward, r_t; the next state, s_{t+1}; and finally, the next action, a_{t+1}. Putting everything together, it forms $s,\ a,\ r,\ s,\ a$.

- On-policy algorithm (learns from actions taken by the current policy)

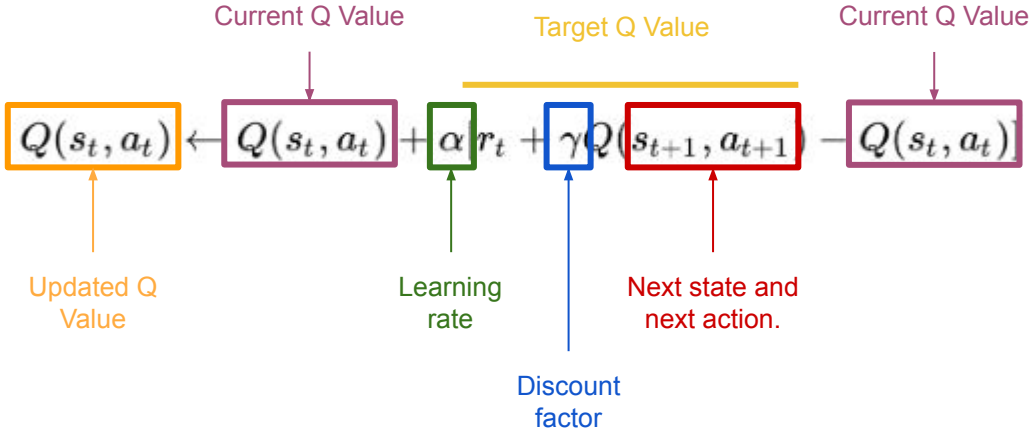
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

SARSA



The name **SARSA** comes from the update that is based on the state, s_t; the action, a_t; the reward, r_t; the next state, s_{t+1}; and finally, the next action, a_{t+1}. Putting everything together, it forms $s,\ a,\ r,\ s,\ a$.

- On-policy algorithm (learns from actions taken by the current policy)



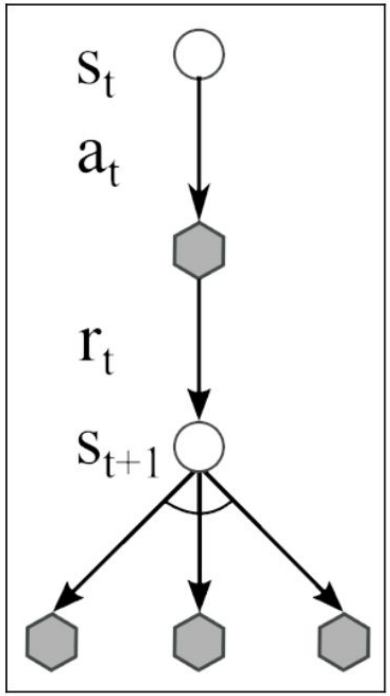
Pseudocode

Initialize $Q(s,a)$ for every state-action pair
$\alpha \in (0,1], \gamma \in (0,1]$

for N episodes:
    $s_t = env\_start()$
    $a_t = \epsilon greedy(Q, s_t)$

    while s_t is not a final state:
        $r_t, s_{t+1} = env(a_t)$ # env() take a step in the environment
        $a_{t+1} = \epsilon greedy(Q, s_{t+1})$
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
        $s_t = s_{t+1}$
        $a_t = a_{t+1}$

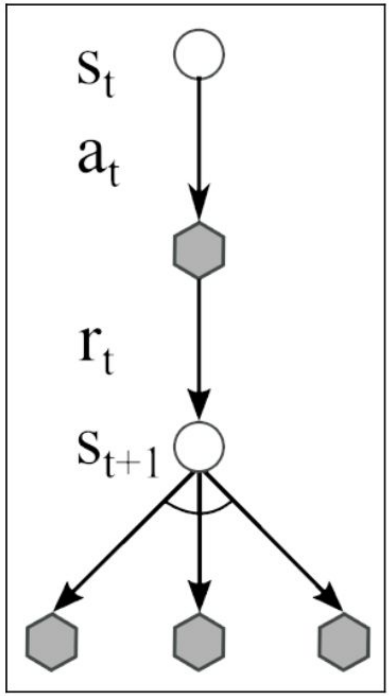
Q-Learning



Q-learning is an **off-policy algorithm**, meaning that it **learns** about the **best possible actions** to take, regardless of which actions are actually taken during training (It approximates the Q-function by using the current optimal action value).

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

Q-Learning



Q-learning is an **off-policy algorithm**, meaning that it **learns** about the **best possible actions** to take, regardless of which actions are actually taken during training (It approximates the Q-function by using the current optimal action value).

Current Q Value

Target Q Value

Current Q Value

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

Updated Q Value

Best possible action

Pseudocode

Initialize $Q(s,a)$ for every state-action pair
$\alpha \in (0,1], \gamma \in (0,1]$

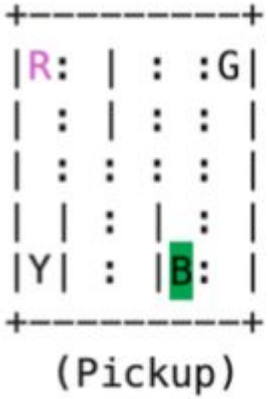
for N episodes:
    $s_t = env\_start()$
    while s_t is not a final state:
        $a_t = \epsilon greedy(Q, s_t)$
        $r_t, s_{t+1} = env(a_t)$ # env() take a step in the environment
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
        $s_t = s_{t+1}$

CODE TIME!

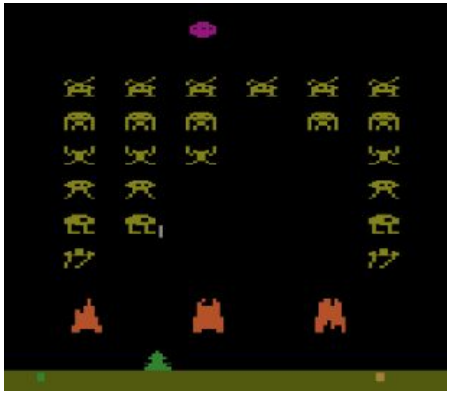
⚠️

These methods do not scale because, for a large number of states and actions, the table's dimensions increase exponentially and can easily exceed the available memory.

Frozen Lake
(16 states)

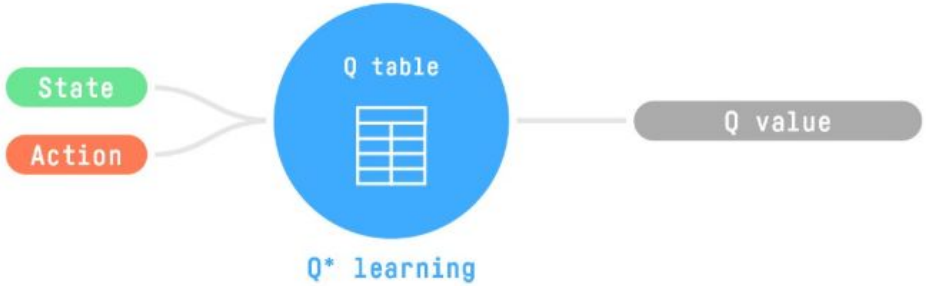


Taxi V3
(500 states)

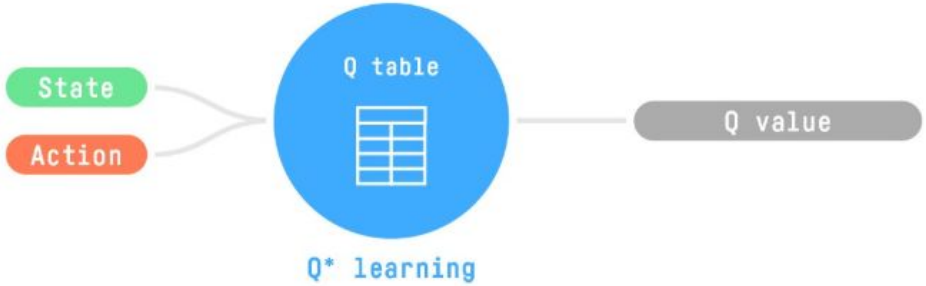


Atari
(256^100800 states)

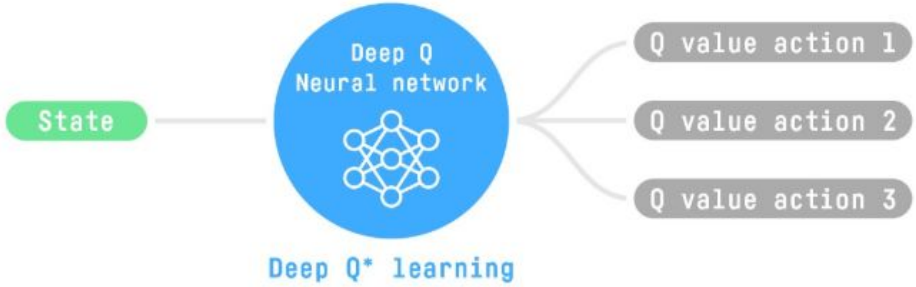
Deep Q-Learning (DQN)



We use a **traditional algorithm to create a Q table** that helps us find what **action to take for each state.**

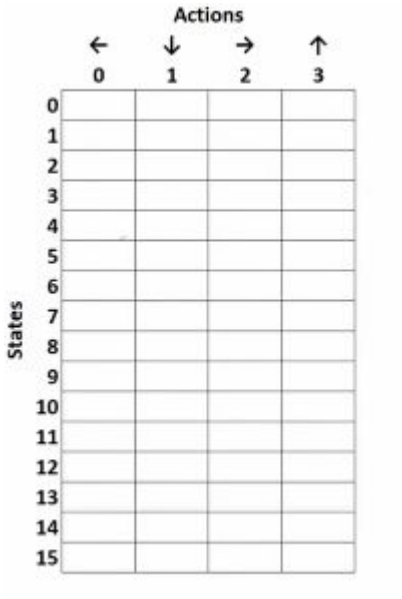


We use a **traditional algorithm to create a Q table** that helps us find what **action to take for each state.**



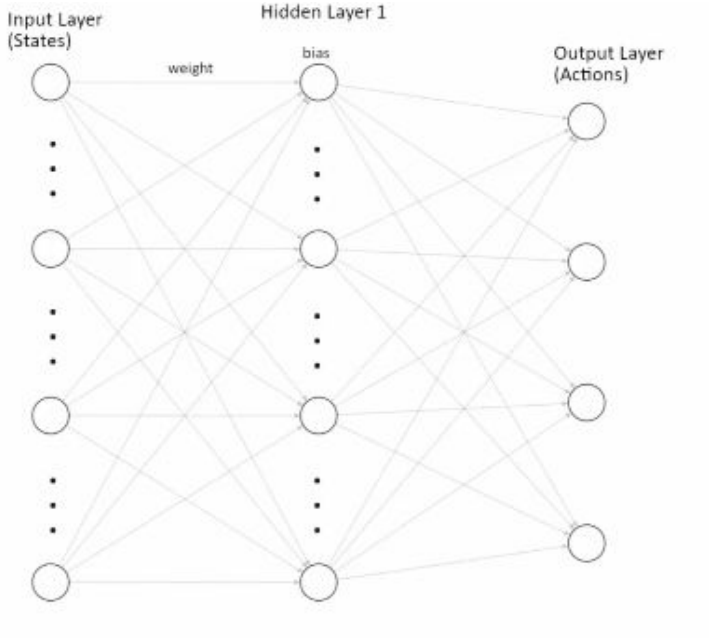
We will use a **Neural Network** (to approximate the **Q value**).

Q-Learning



Frozen Lake

Deep Q-Learning



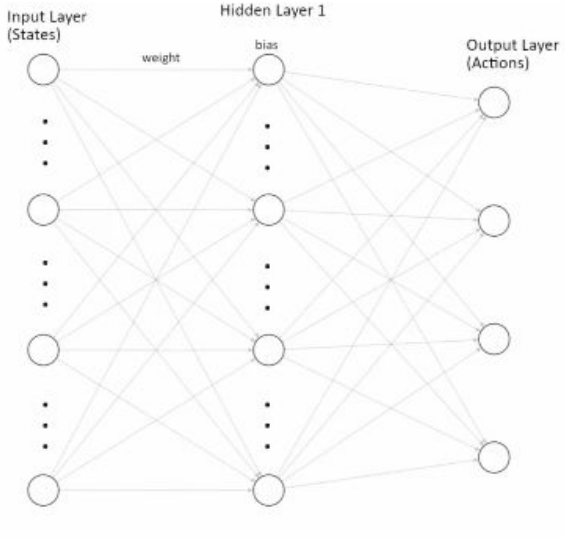
Deep Q-Learning (DQN)

$$\theta \leftarrow \theta - \alpha[r + \gamma max_{a'} Q_\theta(s', a') - Q_\theta(s, a)]\nabla_\theta Q_\theta(s, a)$$

Describes how the weights are updated.

Temporal-difference error

Gradient of the Q-network



Weights: A Deep Neural Network learns a set of weights to approximate the Q-value function.

Loss Function

$$L(\theta) = E_{(s,a,r,s')}[(y_i - Q_\theta(s_i, a_i))^2]$$

Mean Squared Error Loss
between the target Q-value and the predicted Q-value.

Target Q-Value

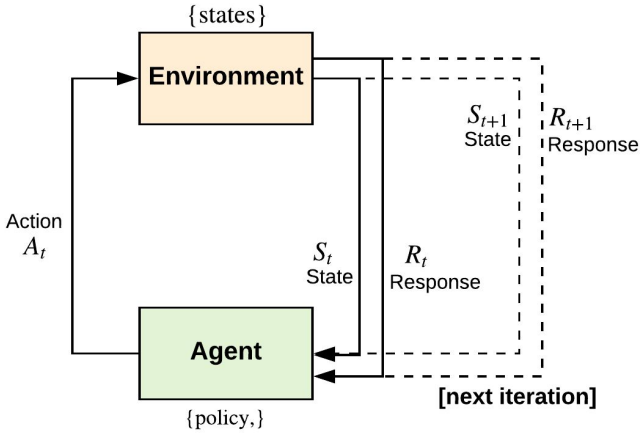
$$y_i = r_i + \gamma max_{a'_i} Q_\theta(s'_i, a'_i)$$

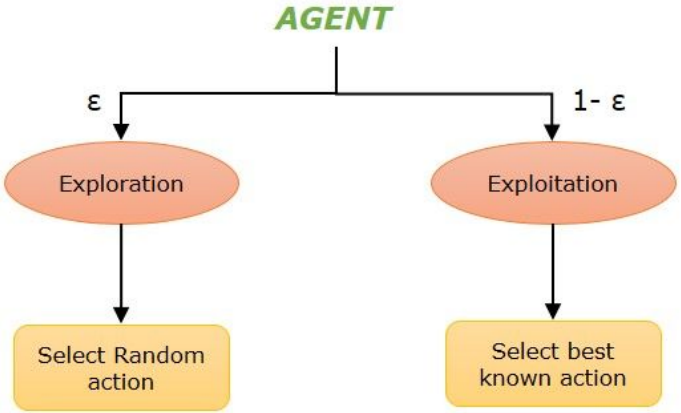
Target Q-Value
The reward plus the best possible estimated value from the next sate

CODE TIME!

Exploration-Exploitation Trade-off

When you go to a restaurant you've visited several times, do you order the same food or try something new?

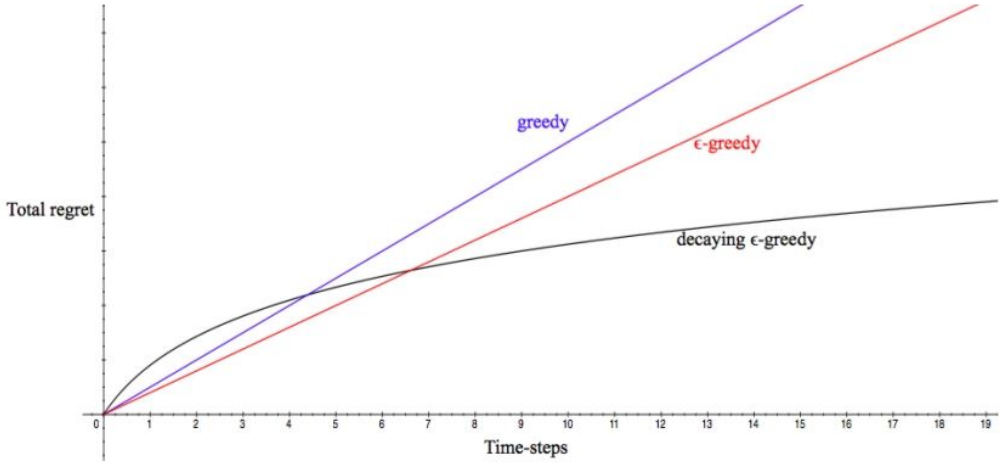


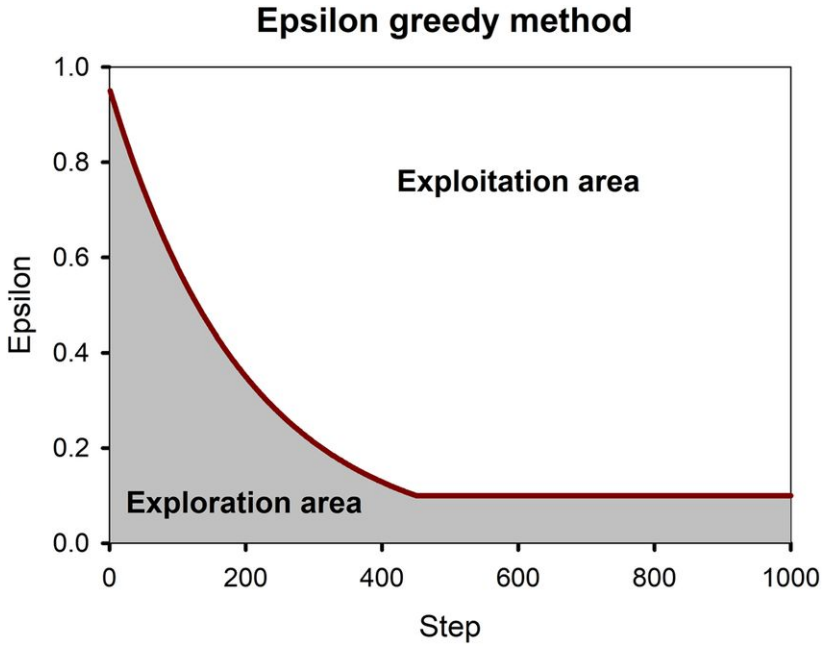


Exploration-Exploitation Trade-off

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step of episode):
        Choose a from s using policy derived from Q (e.g., ε-greedy)
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$;
    until s is terminal

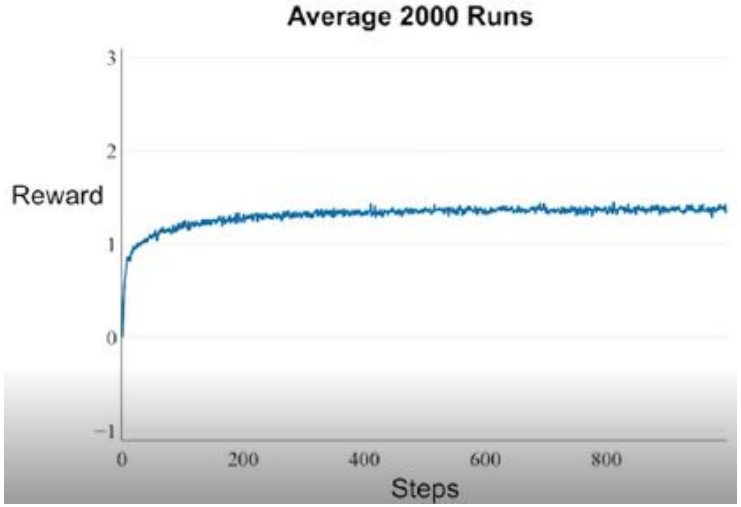
Exploration-Exploitation Trade-off





Exploration-Exploitation Trade-off

arriba=0.89 abajo=0.32 izquierda=0.13 derecha=0.75	arriba=0.60 abajo=0.34 izquierda=0.16 derecha=0.64	arriba=0.06 abajo=0.31 izquierda=0.43 derecha=0.99	arriba=0.69 abajo=0.79 izquierda=0.18 derecha=0.93
arriba=0.20 abajo=0.25 izquierda=0.97 derecha=0.58	arriba=0.94 abajo=0.14 izquierda=0.81 derecha=0.31	arriba=0.81 abajo=0.57 izquierda=0.48 derecha=0.96	arriba=0.68 abajo=0.95 izquierda=0.76 derecha=0.14
arriba=0.96 abajo=0.20 izquierda=0.82 derecha=0.43	arriba=0.11 abajo=0.27 izquierda=0.32 derecha=0.78	arriba=0.1 abajo=0.21 izquierda=0.36 derecha=0.88	arriba=0.02 abajo=0.19 izquierda=0.31 derecha=0.66
arriba=0.43 abajo=0.74 izquierda=0.39 derecha=0.91	arriba=0.48 abajo=0.42 izquierda=0.22 derecha=0.42	arriba=0.07 abajo=0.11 izquierda=0.67 derecha=6.98	arriba=0.78 abajo=0.75 izquierda=0.22 derecha=0.41

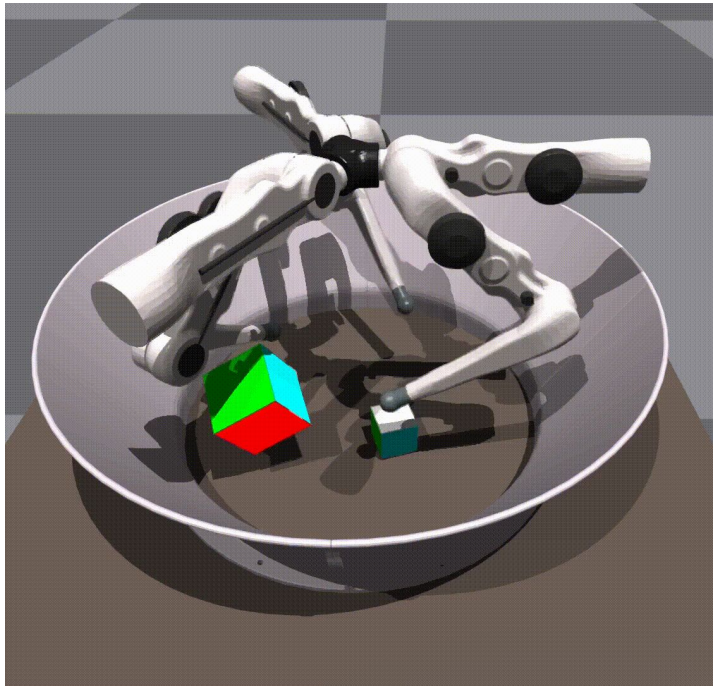


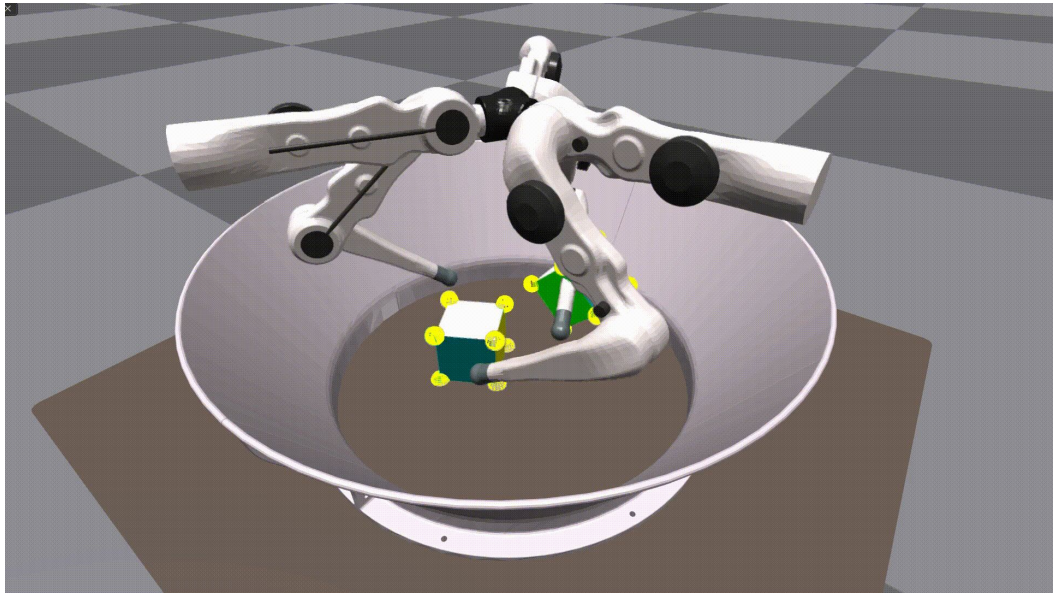
Greedy action

$$A_t \leftarrow \begin{cases} \underset{a}{\text{argmax}}\ Q_t(a) & \textbf{with probability } 1-\epsilon \\ a \sim Uniform(\{a_1 \dots a_k\}) & \textbf{with probability } \epsilon \end{cases}$$

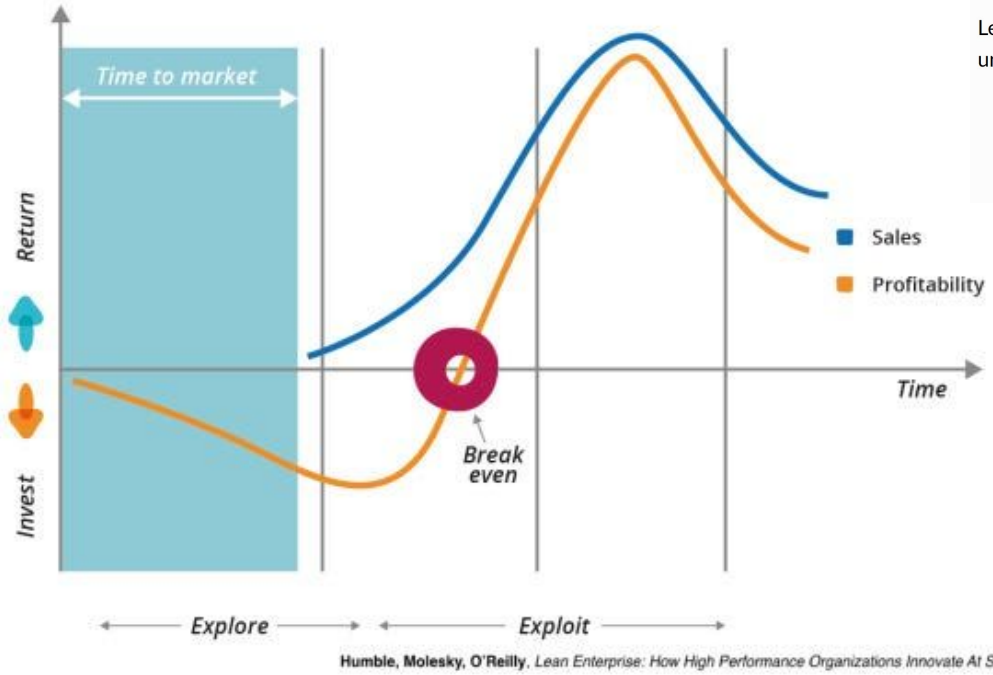
Random action: with the same probability, select one of the other actions

Exploration-Exploitation Trade-off





Exploration-Exploitation Trade-off

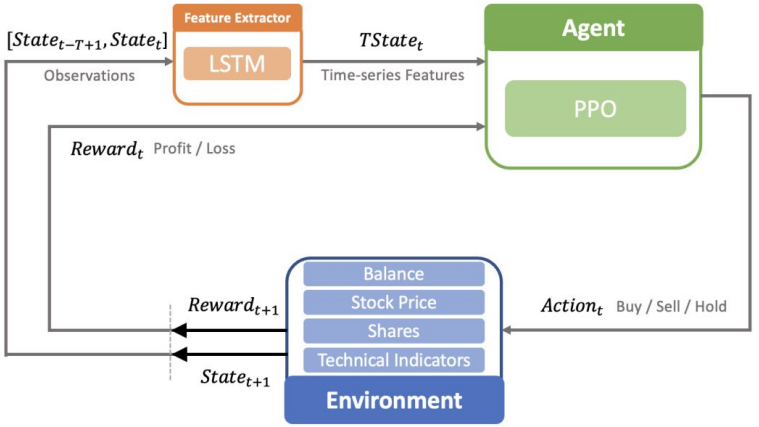


Humble, Molesky, O'Reilly, *Lean Enterprise: How High Performance Organizations Innovate At Scale*

Key Equations

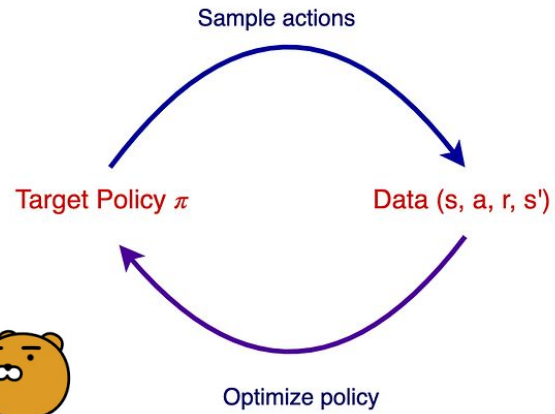
Let π_θ denote a policy with parameters θ, and $J(\pi_\theta)$ denote the expected finite-horizon undiscounted return of the policy. The gradient of $J(\pi_\theta)$ is

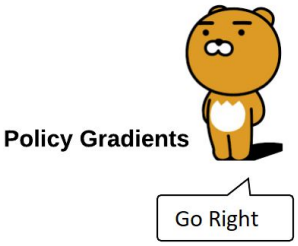
$$\nabla_\theta J(\pi_\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right],$$



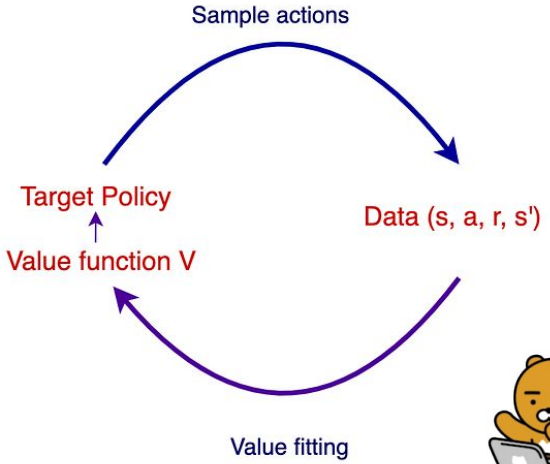
Exploration-Exploitation Trade-off

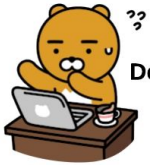
On-policy





Policy Gradient methods

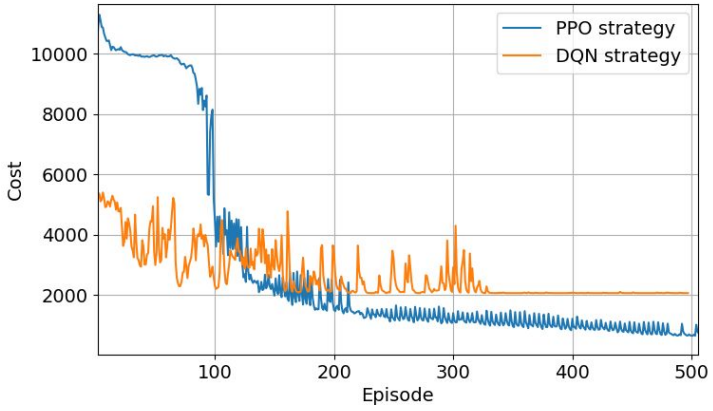


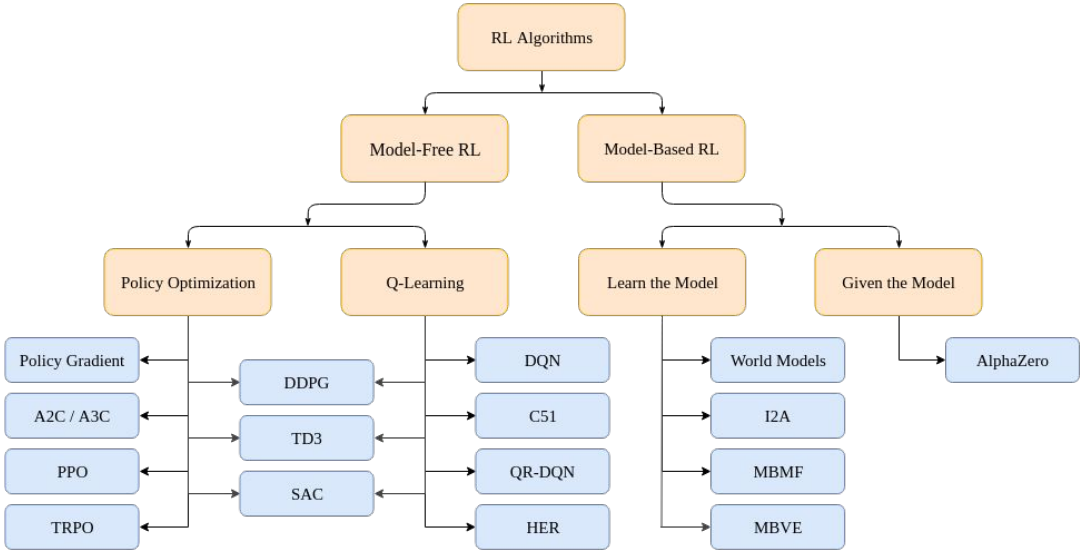


Value learning

Please wait, I am still calculating Q value, only 41891 actions left...

Exploration-Exploitation Trade-off





CODE TIME!

Applications

Robotics!!!

Robotics!!!

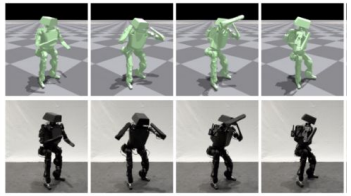

Unitree H1 EVOLUTION V4.0
The World's First Full-size Motor Drive
Humanoid Robot Flips on Ground

Robotics!!!

More applications!!!

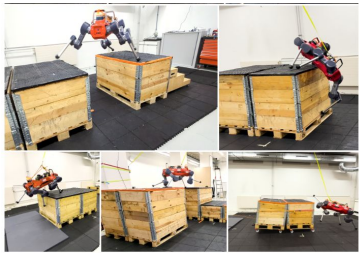
https://www.linkedin.com/feed/update/urn:li:activity:7380127828962480128/

Papers about RL!



Serifi, A., Grandia, R., Knoop, E., Gross, M., & Bächer, M. (2024). *Robot Motion Diffusion Model: Motion Generation for Robotic Characters.* In *SIGGRAPH Asia Conference Papers.* https://youtu.be/eRXS98c_Suc

Grandia, R., Knoop, E., Hopkins, M., Wiedebach, G., Bishop, J., Pickles, S., Müller, D., & Bächer, M. (2024, July). *Design and Control of a Bipedal Robotic Character.* In *Robotics: Science and Systems XX.* Robotics: Science and Systems Foundation. https://youtu.be/7_LW7u-nk6Q



Hoeller, D., Rudin, N., Sako, D., & Hutter, M. (2023). *ANYmal Parkour: Learning Agile Navigation for Quadrupedal Robots.* arXiv preprint arXiv:2306.14874. https://youtu.be/PjWvf90l4cg

Q&A

Bibliography

Lonza, A. (2019). *Reinforcement Learning Algorithms with Python: Learn, understand, and develop smart algorithms for addressing AI challenges.* Packt Publishing. ISBN 9781789139709.