

AI - BlackJack

LIBERT Adrien, AURIEDE Nino, Jorge, Adam

Artificial Intelligent - 2023-2024



Introduction

Originally, Blackjack was a game played in the 18th century in French casinos. At that time, it was known as "Vingt et un" (Twenty-One).

Favored by Madame du Barry and later adopted by Napoleon during his exile in Saint Helena, Blackjack made its way across the Atlantic after the Revolution.

In America, it entered gambling circles where bonuses were invented to retain players. If your first two cards were a jack of spades and an ace, you would receive an extra payout, hence the name Jack, for the jack card, and black, representing the color, to name the game. These bonuses no longer exist, but the principle remains the same.

The **goal** is to beat the Bank, represented by the Dealer, without exceeding **21**; otherwise, you lose your bet.

Rules

The goal is to beat the Bank without going over 21.

Card Values:

Number cards (2 through 10) are worth their face value.

Face cards (Jacks, Queens, Kings) are worth 10.

Aces can be worth 1 or 11, depending on which value benefits the player's hand more.

The hand called "Blackjack" consists of an Ace and a card worth 10, making a total of 21, received

The Deal:

Each player starts with two cards, as does the dealer.

One of the dealer's cards is face down (the 'hole card') and the other is face up.

Player's Turn:

If the player has a blackjack, he waits for the results to be announced.

If the player does not have a blackjack, several choices are possible:

Request one or more additional cards to get closer to 21 without exceeding it.

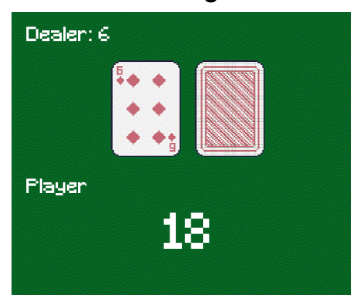
The player then says "Hit!" or taps on the table.

Stop or "stand," and therefore keep their cards. The player hovers their hand over the cards to signal this intention to the dealer, or says "I stay" or "I stop".

Double their bet ("Double down"), on the condition that they only receive one more card after this. The player doubles their bet and receives a third and final card.

Special Cases

If the player has two cards of the same value, they can "split," meaning they separate their two cards to play with two hands, and an additional bet equal to their initial bet.



Once the hands are separated, the dealer adds a card to each hand (so that there are two cards per hand), and the player plays each hand in the same way as a single hand.

If the dealer's first card is an Ace, the player has the option to take insurance against a possible blackjack by the dealer.

For this, the player pays half of their initial bet.

If the dealer gets blackjack, they receive double their insurance (and lose their bet unless the player also has blackjack, in which case they get their bet back);

If the dealer does not get blackjack, the player loses their insurance, and the game continues normally.

Winning and Losing:

If you go over 21, you 'bust' and lose, regardless of the dealer's hand.

If you don't bust and your hand is higher than the dealer's, you win.

If both you and the dealer have the same total, it's a 'push', and your bet is returned.

A 'Blackjack' is when your first two cards are an Ace and a 10-value card.

Development

Before talking about Monte-Car and Reinforcement learning, there are three principal strategies, such as basic strategy, simple strategy, and card counting.

Basic Strategy is the most widely accepted and used method for playing Blackjack.

It's a set of rules that outlines the best mathematical action a player can take in any given situation, based on their hand and the dealer's upcard.

Whether to hit, stand, double down, split, or surrender, depends on your hand and the dealer's visible card.

For example, if you have a total of 16 and the dealer has a 10, the basic strategy suggests you should hit.

The goal of Basic Strategy is to minimize the house edge and improve your chances of winning. While it doesn't guarantee a win every time, it statistically improves your odds over the long run.

Simple Strategy is a less detailed version of Basic Strategy.

It's easier to remember but slightly less effective.

This strategy simplifies the decision-making process by focusing on general rules rather than specific hand combinations.

For example, it might suggest always hitting on a certain range of numbers, regardless of the dealer's card. It's designed for players who want a good balance between optimal play and ease of use. While it gives up a bit of the edge gained by Basic Strategy, it's still much better than playing with no strategy at all.

Card Counting is a more advanced strategy that involves keeping track of the high and low cards dealt from the deck.

Players assign values to each card (like +1, 0, -1) and keep a running count as cards are dealt.

This count helps the player gauge the composition of the remaining cards in the deck.

A high positive count suggests a higher number of high-value cards (like 10s and Aces) are left in the deck, which is advantageous to the player.

By knowing the composition of the remaining cards, players can make more informed decisions about betting and playing hands.

This strategy can significantly reduce the house edge and even give the player an edge in some cases.

We are going to combine Q-learning and the Monte Carlo method in reinforcement learning, each bringing its unique advantages.

Q-learning is particularly effective for real-time updates of value estimates and for learning optimal policies without the need for a detailed model of the environment.

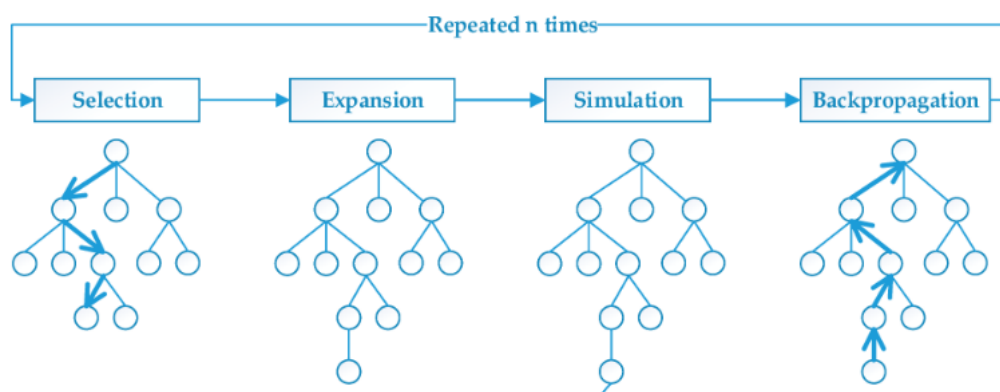
This method is ideal for situations where quick and adaptive decisions are necessary.

On the other hand, the Monte Carlo method excels in direct learning from the complete returns of episodes, which is extremely useful in environments with clear terminal states, where the final outcome of the episode is crucial for learning.

Monte Carlo Tree Search with Partially Observable Markov Decision Processes (MCTS POMDP) for Blackjack:

Our approach employs a MCTS POMDP framework to simulate Blackjack scenarios, model decision-making in an environment with incomplete information.

The algorithm begins by constructing a decision tree, where each node represents a probabilistic state of the game, factoring in both the player's and dealer's hands.



When expanding the tree, the algorithm mathematically calculates probabilities and expected values for each potential action from every game state, thus simulating various possible game trajectories.

Player strategies are implemented through probability distributions and statistical models, ranging from Basic Strategy, which minimizes the house edge, to complex systems like Card Counting, quantified through conditional probabilities and Bayesian inference.

For each simulated action, the algorithm plays out the hand according to Blackjack rules. This includes dealing cards to the player and the dealer, with the dealer's actions typically following a set rule (e.g., hitting until reaching 17).

The outcomes of these actions, including the probability of hitting a specific hand or busting, are used to generate new nodes, each representing a distinct probability distribution of potential game states.

The algorithm records what gains or losses the player had in this particular game. This includes whether the player won, lost, or tied, and by what margin. The process of expanding the tree and simulating games is repeated many times, with each iteration representing a different possible path through a game of Blackjack. By running numerous simulations, MCTS averages out the results, allowing for an assessment of the effectiveness of different strategies in various game situations. This entire process is repeated thousands of times. Each iteration represents a separate game of Blackjack. The algorithm can average out the results to understand the effectiveness of the chosen strategy. After extensive simulations, the algorithm identifies the strategy with the highest average gains. This involves calculating the expected value of each strategy and selecting the one with the maximum expected payoff. The optimal strategy is determined based on maximizing the expected utility, a fundamental concept in decision theory, indicating the strategy that should yield the highest return over the long run.

```

function SELECTACTION( $b, d$ )
     $h \leftarrow \emptyset$ 
    while Convergence Condition not satisfied do
         $s \sim \text{SAMPLEBELIEF}(b)$ 
        POMDPSIMULATE( $s, h, d$ )
    end while
    return  $\text{argmax}_a Q(h, a)$ 
end function

function SIMULATE( $s, h, d$ )
:   if  $d = 0 \vee \text{deckIsOutOfCards}(s)$  then return 0
:   end if
:   if  $h \notin T$  then
:       for all  $a \in A(s)$  do
:            $(N(h, a), Q(h, a)) \leftarrow (N_0(h, a), Q_0(h, a))$ 
:       end for
:        $T = T \cup \{h\}$ 
:       return ROLLOUT( $s, d, \pi_0$ )
:   end if
:    $a \leftarrow \text{argmax}_a Q(h, a) + c \sqrt{\frac{\log N(h)}{N(h, a)}}$ 
:    $(s', o, r) \sim G(s, a)$ 
:    $q \leftarrow r + \gamma \text{SIMULATE}(s', hao, d - 1)$ 
:    $N(h, a) \leftarrow N(h, a) + 1$ 
:    $Q(h, a) \leftarrow Q(h, a) + \frac{q - Q(h, a)}{N(h, a)}$ 
:   return  $q$ 
: end function

```

Pseudo-code MCTS POMDP

Objective: Choose the best action in an environment where the complete state is not fully observable.

Process:

Initialization: Creates an empty history h and initializes the state s based on the current belief b .

Simulation: Repeats simulations (calling POMDPSIMULATE) for different possible states sampled from the belief b . Each simulation evaluates possible actions and their outcomes.

Action Selection: After several simulations, determines the action that has, on average, the best outcome (the highest Q-value) and selects it.

Objective: Simulate the outcome of an action in the environment and update the Q and N values (which represent the estimated quality of an action and the number of times the action has been tested, respectively).

Process:

Check Stop Conditions: If the simulation depth d is exhausted or if the deck is out of cards, the function stops.

Exploration and Exploitation: If the history h is not in the set T , initializes the Q and N values. Otherwise, choose the action that seems best based on the current Q values, while still allowing for some exploration.

Update: Simulates the chosen action, generates a new state s_0 , and recursively updates the Q and N values based on the results of this simulation.

To **implement** our Monte Carlo simulation for Blackjack, we need to create several distinct functions, each serving a specific purpose in the simulation:

At the end of the document there is a github link to see our first code (it's just the beginning).

First we've created a class `BlackjackNode` that will be helpful to do à MTCS.

```
class BlackjackNode:
    def __init__(self, player_hand, dealer_hand, deck, parent=None,
is_terminal=False):
        self.player_hand = player_hand
        self.dealer_hand = dealer_hand
        self.deck = deck
        self.parent = parent
        self.is_terminal = is_terminal
        self.visits = 0
        self.wins = 0
        self.children = []
```

mcts(node, iterations):

Role: Implements the Monte Carlo Tree Search algorithm for Blackjack.

iterations: Number of iterations for the algorithm.

Returns: The best action according to the MCTS algorithm.

select(node):

Role: Selects the best child node using the Upper Confidence Bound for Trees (UCT).

Returns: The selected node.

expand(node):

Role: Expands the node by adding new child nodes.

Returns: The newly added node.

simulate(node):

Role: Simulates the game from the given node until a terminal condition is reached.

Returns: The result of the simulation (1 for win, 0 for loss).

backpropagate(node, result):

Role: Updates the parent nodes' statistics after a simulation.

result: Result of the simulation.

best_uct(node):

Role: Selects the best child node using UCT.

Returns: The best node.

best_child(node):

Role: Selects the best child node based on the number of wins.

Returns: The best node.

draw_card(deck):

Role: Draws a random card from the deck.

Returns: The drawn card.

play_game(player_hand, dealer_hand, deck):

Role: Simulates a round of Blackjack between the player and the dealer.

Returns: The result of the round (win, loss, or draw).

calculate_score(hand):

Role: Calculates the score of a hand (sum of card values).

Returns: The score of the hand.

simulate_games(number_of_games, strategy):

Role: Simulates a specified number of games using the specified strategy.

Parameters:

number_of_games: Number of games to simulate.

strategy: The strategy to use (in this case, "mcts" or another strategy).

Returns: The results of the simulations (number of wins, losses, draws).

Random Deck Generator (generate_deck):

This function is crucial for simulating the randomness of card games.

It generates a standard, shuffled deck of 52 cards.

Its signature is `def generate_deck() -> list:`, and it returns a shuffled list of card objects, each representing a unique card in the deck.

Card Distribution (deal_hand):

Essential for the game's initial setup, this function deals two cards to either a player or the dealer from the deck.

The signature is `def deal_hand(deck: list) -> list:`, and it takes the deck as input, returning a hand consisting of two cards.

Strategy Implementation (apply_strategy):

This function simulates the player's decision-making process based on their current hand, the dealer's visible card, and a predefined strategy like Basic Strategy, Simple Strategy, or Card Counting.

Its signature, `def apply_strategy(player_hand: list, dealer_hand: list, strategy: str) -> list:`, reflects these inputs and outputs the player's hand after applying the chosen strategy.

Gameplay Simulation (`play_game`):

Here, a single hand of Blackjack is played out following the game rules and strategies of both the player and dealer.

The function's signature, `def play_game(player_hand: list, dealer_hand: list, deck: list) -> dict:`, indicates that it takes the hands and the deck as inputs and returns a dictionary with the game's outcome.

Simulation and Result Recording (`simulate_games`):

To understand the effectiveness of different strategies, this function repeats the gameplay thousands of times and records the results. Its signature is `def`

`simulate_games(number_of_games: int, strategy: str) -> dict:`, denoting that it takes the number of games to simulate and the chosen strategy as inputs and returns a dictionary summarizing the outcomes.

Reinforcement learning to find the best strategy (Jorge,Adam,and Adrien)

Reinforcement learning (RL) methods have gained increasing popularity for solving complex decision-making problems, including games such as Blackjack.


The foundation of this RL approach is the Q-learning method.

Q-learning is a reinforcement learning technique that operates without requiring a model of the environment.

The 'Q' in Q-learning stands for the quality of a particular action taken in a given state.

This function learns to identify the best decisions to make in different situations.

$$Q : S \times A \rightarrow \mathbb{R} \quad Q(s_t, a_t)$$



State Action

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a)$$

If you take an action, you get a reward. Gamma is the parameter, to prioritize the reward. Firstly, creates a **Q-value table**, initializing all state-action pairs with a default value (usually set to zero).

This table will be updated as the agent learns from its experiences: the agent will run through a set of iterations (called scenarios) and will choose different actions to get the most possibly accurate final Q-table

Decision-making in this RL approach is governed by the epsilon policy (defines the probability with which the agent will choose a random action rather than the best-known action, a number between 0 and 1).

This policy strikes a balance between exploring new actions and exploiting known actions that yield high rewards.

With a certain probability (epsilon), the agent chooses a random action, promoting exploration.

Otherwise, with 1 - epsilon probability, it chooses the action with the highest Q-value in the current state, focusing on exploitation.

Initially, when the agent has little knowledge about the environment, a higher epsilon is useful for encouraging exploration.

Over time, as the agent learns more about the environment, epsilon is often reduced to favor the exploitation of acquired knowledge.

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

As the agent interacts with the environment, the Q-values are updated to reflect new information.

The update function adjusts the Q-values based on the reward received after taking an action and the estimated rewards of the next state.

This update is guided by the learning rate (alpha) and the discount factor (gamma), which balance immediate and future rewards.

These RL functions are integrated into a simulation framework.

Before starting the simulations, Q-values are initialized. During each game round, the epsilon-greedy policy is used to decide actions.

After each action, Q-values are updated based on the outcomes.

This process allows the agent to learn an effective strategy for Blackjack through repeated gameplay and analysis.

This RL method, when applied to Blackjack, enables an agent to learn from each game round and gradually improve its strategy. The combination of Q-value updates and the epsilon-greedy policy allows the agent to navigate the balance between risk (exploration) and reward (exploitation).

We can define two actions at the beginning :

Stand with 0 as identification

Hit with 1 as identification

Use OpenAI Gym for the test environment:

make(): To initialize or create a new simulation environment.

Example: `env = gym.make("Blackjack-v1", natural=True, sab=False)` creates a basic environment for Blackjack.

reset(): To reset the environment to its starting state and obtain an initial observation.

step(action): To execute an action in the environment and observe the outcome in terms of a new state, reward, and end-of-episode indicator (whether it's terminated or truncated).

render(): To display the current state of the environment, useful for debugging and visualization.

close(): To close the environment and release resources at the end of the program's execution.

First observation then action at the beginning our agent will execute action randomly, however, as the agent executes more and more scenarios, they will play closer and closer to the optimal decisions.

The problem, however, is that the **State Space** of blackjack is quite large, so the agent will require **a lot of scenarios** compared to applying RL to other games.

To **implement** our Q-learning functions with OpenAI gym for Blackjack, we need to create some part

Imports and setup the environment:

To create an environment for Gymnasium, we need to import the necessary libraries. These typically include Gymnasium itself for the environment, NumPy for numerical operations, and Matplotlib for visualization.

The environment is set up by calling the make function from Gymnasium with the specific environment name, in this case, 'Blackjack-v1'.

`env = gym.make('Blackjack-v1', natural=True, sab=False)` basic environment

Once the environment is created, we can start.

Executing an action:

the agent must decide between two possible actions based on its current observation: to "hit" and receive another card, or to "stand" and end their turn.

The observation consists of the player's current hand value, the dealer's visible card, and whether the player has a usable ace.

Based on this information, the agent uses a policy to decide the next action.

The policy can be random or based on learned Q-values which estimate the expected rewards for taking certain actions given the current state.

The agent:

The BlackJack Q-learning agent is built with few key parameters.

The `learning_rate` determines how quickly the agent incorporates new information.

The `epsilon` values manage the exploration-exploitation trade-off: initially, the agent explores the environment more, but over time, it starts to exploit known strategies that yield higher rewards.

The `discount_factor` is critical in calculating the present value of future rewards, influencing how far-sighted the agent's strategy will be.

The agent maintains a record of action values (Q-values) which are updated as it learns from experiences.

Update:

After each action taken during an episode, the agent observes the outcome and updates the Q-values based on the reward received and the estimated future rewards.

The difference between the expected Q-value and the received reward, adjusted by the learning rate and discounted future rewards, is used to update the Q-value for the state-action pair.

This process allows the agent to learn over time which actions are most beneficial.

Training involves running the agent through many episodes, during which it continually refines its policy by updating Q-values.

The agent must play a large number of episodes to adequately explore the environment and converge towards an optimal policy.

During training, the exploration rate (`epsilon`) is gradually reduced, ensuring that the agent relies more on its learned experiences as it becomes more proficient at the game.

Add visualisation:

To enhance our understanding of the agent's learning progress and strategy, integrating visualizations using Matplotlib is highly beneficial.

The agent's training performance over time, highlighting trends in rewards and decision-making patterns.

Naturally, the agent will lose a lot in the beginning, since the game of blackjack is already designed to be hard for even the highest-level humans to play optimally, so the agent with no knowledge will make a lot of mistakes that could seem like obvious blunders for a human.

Conclusions

Integrate these functions MCTS and Q-learning simulation create a robust decision making for the environment.

The Q-learning can be used to evaluate and MCTS guides exploration.

After each action, use the `update_function` to update the Q-values based on the reward received and the next state.

We can conclude that combining Q-learning and the Monte Carlo method is one of the best solutions we can work with while developing a Black Jack program.

Thanks to it being efficient , and running a simulation various times in order to get the optimum solution and the best strategy to work with.

We could use it in different programs if multiple runs and constante changes are necessary.

Some functions are available here : <https://github.com/AdrienLibert/BlackJackMCTS.git>

Thank you for this first part of the project.

This exchange between Spanish and French has allowed us to open our minds to teamwork.

As a group of students working on the project, we've successfully combined our diverse perspectives and skills.

Annex

Bibliography

How to Count Cards in Blackjack

▶ Comment compter les cartes au blackjack

Topic: Card Counting Strategies in Blackjack

Provides insights into the strategic aspect of Blackjack, focusing on card counting methods.

[AI Learns to Play BlackJack!!](#)

Topic: Application of Reinforcement Learning and MCTS in Blackjack

Explores how artificial intelligence can be trained to play Blackjack using advanced algorithms like Reinforcement Learning and MCTS.

[Monte Carlo Tree Search](#)

Topic: Detailed study of MCTS

Academic report providing a comprehensive analysis of MCTS, its mechanisms, and applications.

[Simulating Blackjack using Monte Carlo Method](#)

Topic: Combining Blackjack Strategy with the Monte Carlo Simulation

Discusses how the Monte Carlo simulation method can be applied to simulate and analyze Blackjack strategies.

[Reinforcement Learning Implementation](#)

Topic: Reinforcement Learning in Blackjack

A GitHub repository showcasing an implementation of a Blackjack game solver using Reinforcement Learning techniques.

[MCTS in Julia Programming Language](#)

Topic: Implementation of MCTS in Julia

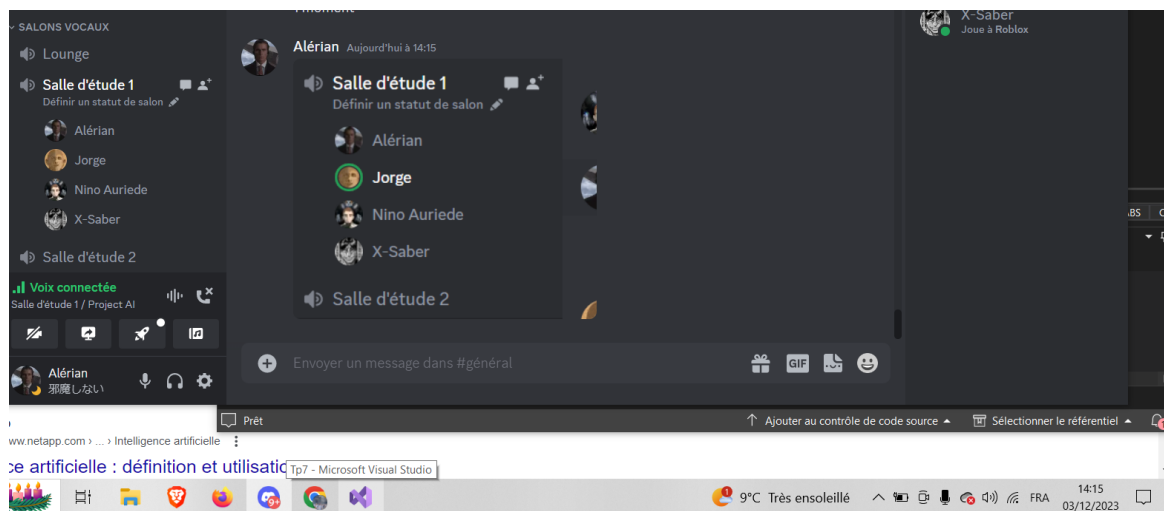
Offers a practical example of MCTS algorithm implementation in the Julia programming language, useful for understanding how MCTS works in a coding context.

Python AI - Blackjack Tutorial on YouTube

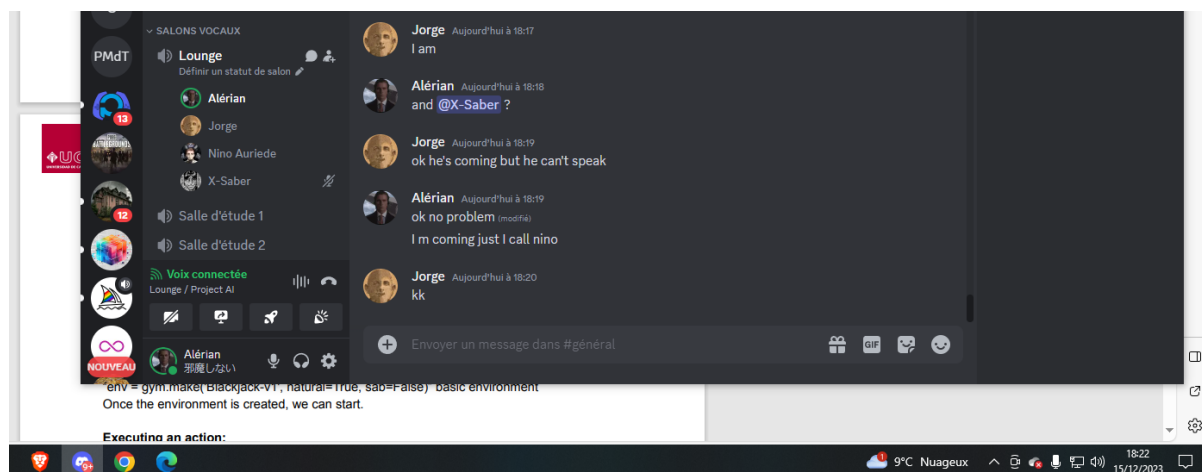
[AI Tutorial for Blackjack](#)

Topic: Python AI in Blackjack

A video tutorial demonstrating how AI can be programmed in Python to play Blackjack, suitable for those who prefer visual and practical learning.



Proof 1



Proof 2

Contributions :

Introduction = Nino and Adrien

MCTS = Nino and Adrien

Reinforcement Learning = Jorge,Adam and Adrien

Conclusion = Jorge and Adam