# Q-Learning Blackjack Agent

This repository contains the code for a reinforcement learning agent designed to play Blackjack using the Q-Learning algorithm.

## Disclaimer & Context (Skippable)

This project was developed with the assistance of AI tools and various online resources for both theoretical understanding and code snippets. I do not claim this code to be 100% original, as its creation was a learning process heavily influenced by external knowledge. The primary goal of this repository is education—to explore and demonstrate the basics of Q-Learning and general Artificial Intelligence concepts.
Since the initial development was done in Spanish, my native language, some variable names or comments in the source files (blackjack_gim.py) may be poorly translated or contextually confusing. This document aims to fully explain the logic and structure to compensate for any language inconsistencies in the code itself.
If you have any questions or require clarification on the code, please feel free to contact me at: mcelada04@gmail.com

## File Structure Overview (Skippable)

- blackjack_gim.py: This is the core file containing the game environment, the Q-Learning implementation, and the training loop. All the theory discussed below pertains to the variables and logic within this file.
- trys.py: This file is a copy of the main code used specifically for running tests and generating the statistical data presented in the conclusion.

## Theory and Code Explanation

### The Game Environment (Table Class)

The Table class serves as the core of the game environment (often called a "Gym" or "Simulator).” Its responsibilities include managing the game state, applying rules, and determining rewards.
The two most critical components for any reinforcement learning environment are:

1. Resets: Reinforcement Learning requires massive amounts of repetition. The reset() function is essential as it ensures the game returns to the exact initial state after every episode.
2. State Representation: The state is the raw, structured information the AI receives from the environment. It contains all the necessary data that the agent needs to make an informed decision. Any information not included in the state is completely ignored by the AI. This process is managed by a dedicated get_state() function, which consistently produces a list (or similar structure) used to index the Q.

## The Agent's Intelligence (The Q-Table)

The Q (called Quality table) is the actual AI, the "thinking" component. It is typically implemented as a dictionary, where the keys are a combination of a (state, action) pair, and the value is the estimated Quality (Q) of taking that specific action while in that specific state.
Q(state, action) = estimation

When the AI needs to choose its next move, it follows a strategy known as Epsilon-Greedy:

- Epsilon-Greedy Exploration: At any given step, there is a probability “epsilon” that the AI will choose a completely random action instead of the optimal one. This randomness is crucial because it encourages exploration, preventing the AI from getting stuck in local maximums, always repeating the same action that can be good but not the optimal one. Randomness implies the AI will try and maybe discover unexpected paths. After many repetitions, the value of epsilon is progressively reduced so the AI can transition from random exploration to exploitation using what it has already learned.
- Exploitation: If the random exploration is not triggered, the function "getQ()" wilñ return the estimation for exactly that state and action, so the code cycles to all possible actions and  then filters the optimal. The last line is to prevent collapses in case the AI finds two equally optimal actions.

## The Learning Process (The Bellman Equation)

When an action is completed, the Q-Table updates itself to "learn.". The simple explanation is that the AI will update the estimation for that state and action; in a deeper level this update the Qand is calculated using the Bellman Equation for reinforcement learning.
The Q-Table update is defined as:
Q(s,a) = Q(s,a) + alpha * (reward + gamma * max_a' Q(s',a') - Q(s,a))

Or in my code:
Q_table[(state, action)] = current_Q + alpha * (reward + gamma * max_Q_new - current_Q)

Breaking down the variables used in the code:

- current_Q : The current Q-value for the state-action pair being updated.
- alpha: This determines the weight of the new information relative to the old. A higher alpha means the AI will quickly discard (or heavily reduce the weight of) old information in favor of the new reward, leading to faster but potentially less stable learning. This is a trade on stability and adaptability, being lower numbers a stable learning and higher a faster learning.
- reward: The immediate reward obtained after taking a concrete action in a concrete state. Given by the gim.
- max_Q_new  (Max Future Reward): This is the maximum expected reward the AI thinks it can obtain from the new state for all possible actions. It serves as a method for the AI to self-evaluate the quality of the action it just took. I don't know why it's named “Max Future Reward”.

- gamma: This factor decides how much importance the AI gives to the calculated maximum future reward max_Q_new. A high gamma means the AI prioritizes long-term future rewards, while a low gamma means it focuses more on the immediate reward (higher means more importance of the theoretical best action, and lower means more importance to the actual reward).

# Discoveries I made (usefull)

## Defining the State

The core in the creation of an AI is defining the state. What's going to be the core of the project? What's the actual thing the AI will learn? A really high number of variables will make the learning really slow and inefficient, while a really low number of variables will make the learning really fast but with poor decision-making. What's the indispensable information? Also this is reinforced AI, so it needs to be able to "abstract" at some point, so the state can't be too specific. In this case I sacrificed a lot of precision for the sake of generalization, and I cut variables so it can actually learn on a limited number of games. If the state need to be more numerous, it will be needed another type of AI (like neural networks, WORK IN PROGRESS)

## The Reward System

The reward system is fragile. Small changes in reward values can lead the AI to learn vastly different, sometimes even bad habits and even not learning from some states, as they are being overshadowed by higher numbers. I recommend creating a balanced system where rewards and penalties are kept relatively small and in a consistent range (between 10 and -10). The key is to assign the lowest practical integer numbers (like 1, -1, 2, -2, etc.) to maintain stability and prevent the AI from over-prioritizing one specific outcome.
Also important is thinking about "0" rewards, as it means it is a neutral action, neither good nor bad, in some cases leaning to not give information and will just get "skipped".

## Persistence (Saving the Q-Table)

AI training often requires extensive iterations, making it vulnerable to interruptions (power cuts, crashes, etc.). It is vital to design a good saving system. It’s only needed to save the Q dictionary.
Saving at important checkpoints (every 10,000 games) is also useful for testing. Since AI learning can be random even with the same parameters, having different "snapshots" of the Q allows for the creation of different patterns or strategies for the same code.

# Conclusion: Statistical Validation

To validate the agent's performance, I performed a simple statistical analysis.
I ran 1,000 games with the trained Q-Learning AI and 1,000 games with a purely random agent (as that's what AI is in the first steps, in the basics):

We used the Z-test for two independent samples to compare the mean performance, setting up the hypothesis as:

- H0 = mean_ia == mean_random
- H1 == mean_ia > mean_random

By using a unilateral test with a significance level of alpha = 0.05 (95% confidence), the results allowed us to reject H0 and say the AI plays at least 75% better than random. This statistically validates the claim that the AI plays Blackjack. (Refer to the bottom of trys.py for the raw statistical data.)