European Roulette: Full Game Guide

The first form of roulette was devised in 18th-century France. Many historians believe **Blaise Pascal** introduced a primitive form of roulette in the 17th century.

To determine the winning number, a croupier spins a wheel in one direction, then spins a ball in the opposite direction around a tilted circular track running around the outer edge of the wheel. The ball eventually loses momentum, passes through an area of deflectors, and falls onto the wheel and into one of the colored and numbered pockets on the wheel. The winnings are then paid to anyone who has placed a successful bet.

1. Overview of European Roulette

European Roulette is a popular casino game played on a wheel with 37 numbered pockets (1 to 36 and a single 0). This version has a house edge of 2.70%, making it more favorable to players compared to American Roulette (which has an extra 00 pocket).

2. Table Layout and Bet Types

Roulette Table Layout

Below are common bet types in roulette:

A. Inside Bets (Higher Payouts, Higher Risk)

- **Straight Up:** Bet on a single number (e.g., 17). Payout: 35:1
- **Split:** Bet on two adjacent numbers (e.g., 14 and 17). Payout: 17:1
- **Street:** Bet on a row of 3 numbers (e.g., 1, 2, 3). Payout: 11:1
- **Corner:** Bet on a block of four numbers (e.g., 17, 18, 20, 21). Payout: 8:1
- **Six Line:** Bet on two adjacent rows (e.g., 28–33). Payout: 5:1
- **Basket:** Bet on 0, 1, 2 or 0, 2, 3. Payout: 11:1

B. Outside Bets (Lower Payouts, Higher Chance)

- Dozens: 1–12, 13–24, 25–36. Payout: 2:1
- Columns: First/Second/Third vertical columns. Payout: 2:1
- Red/Black: All red or all black numbers. Payout: 1:1
- Odd/Even: All odd or all even numbers. Payout: 1:1
- Low/High: 1–18 or 19–36. Payout: 1:1

- 

3. What is the House Edge?

House edge is the casino's advantage over the player.

Analogy:

Imagine you're flipping a coin where:

- Heads: you win £1
- Tails: you lose £1.10

Even though the coin is fair, over time, you'll lose more than you win. That extra £0.10 loss is similar to the house edge.

In European Roulette, the house edge arises because:

- There are 37 numbers, but you're paid as if there were only 36.
- E.g., betting on a single number pays 35:1, but real odds are 1 in 37.

4. Applications in Academic Research & AI

A. Designing Winning Strategies

Winning strategies in roulette aim to optimize returns or minimize losses based on statistical behavior. Common systems include:

- Martingale: Double the bet after every loss until a win occurs.

A.

	1st bet	2nd bet	3rd bet	4th bet	5th bet	6th bet	7th bet	8th bet	9th bet	10th bet	11th bet	12th bet	13th bet	14th bet	15th bet	16th bet
Amount	$1	$2	$4	$8	$16	$32	$64	$128	$256	$512	$1,024	$2,048	$4,096	$8,192	$16,384	$32,768
Success	$1	$1	$1	$1	$1	$1	$1	$1	$1	$1	$1	$1	$1	$1	$1	$1
Risk	$1	$3	$7	$15	$31	$63	$127	$255	$511	$1,023	$2,047	$4,095	$8,191	$16,383	$32,767	$65,535

B.

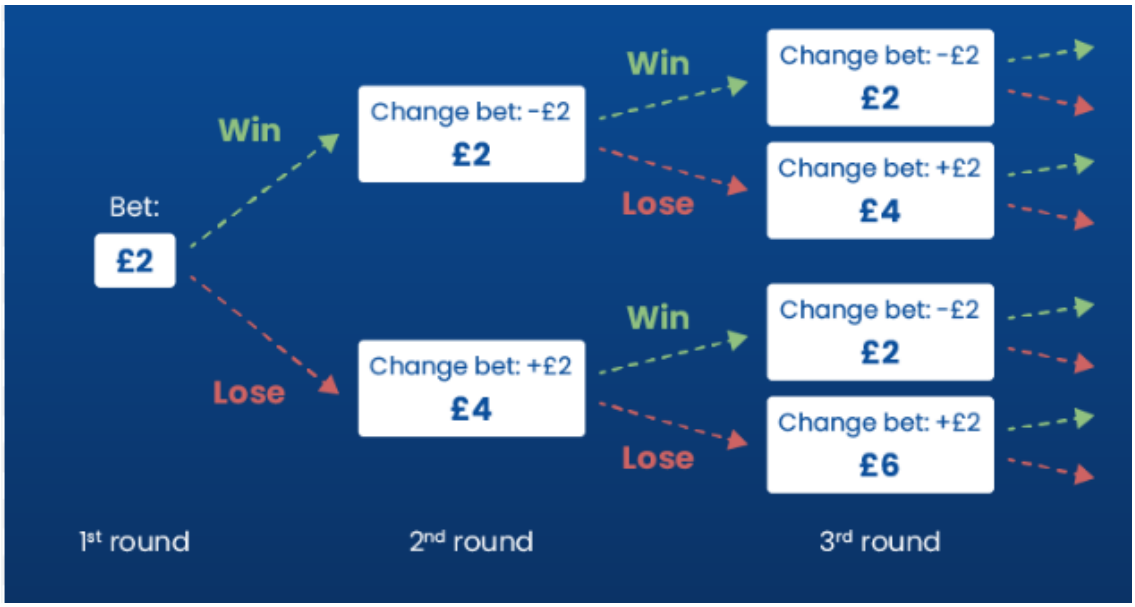
Winning Spin	Stake	Total Loss From Previous Spins	Profit On Winning Spin	Overall Profit
1	£1	£0	£1	£1
2	£2	£1	£2	£2 - £1 = £1
3	£4	£3	£4	£4 - £3 = £1
4	£8	£7	£8	£8 - £7 = £1
5	£16	£15	£16	£16 - £15 = £1
6	£32	£31	£32	£32 - £31 = £1
7	£64	£63	£64	£64 - £63 = £1
8	£128	£127	£128	£128 - £127 = £1
9	£256	£255	£256	£256 - £255 = £1
10	£512	£511	£512	£512 - £511 = £1
11	£1024	£1023	£1024	£1024 - £1023 = £1
12	£2048	£2047	£2048	£2048 - £2047 = £1

• Example: Start with £1.

◦ Lose: Bet £2 → Lose again: Bet £4 → Win: Get £8 total → Net profit = £1

◦ Risk: After 6 losses in a row, the bet becomes £64 — high bankroll needed

• D’Alembert: Increase bet by one unit after a loss, decrease by one after a win



Example: Start with £5.

- Lose: Bet £6 → Win: Bet £5 → Lose: Bet £6 → Win: Bet £5 (less risky)
- Net gain is slower, but risk is better managed than Martingale

Keefer Progression: A Fibonacci-like sequence (e.g., 5, 10, 15, 30, 60...) used across multiple dozens 1, 2, 3, 6, 12, 24, 48, 96 Total = 192

- Example: Bet 5–10–15 on three dozens. If any dozen wins, reset to 5
- If loss continues, progression goes to 30, 60, etc. Reset on win
- Combines breadth (dozen coverage) and depth (increasing bet) for balanced play
- Players can set a stop-loss (maximum allowed loss) and take-profit (exit point on profit) to manage their session outcomes

C. Simulating Outcomes (Machine Learning / Reinforcement Learning)

To model the decision-making process of a betting agent:

- Use Reinforcement Learning (RL) models like Q-learning or Deep Q Networks (DQN)
- Define components:
    - States: Previous outcomes, balance, bet history, hot numbers
    - Actions: Choice of bets (e.g., red, black, dozen, number)
    - Rewards: Profit or loss based on bet result
    - The RL agent updates its policy to maximize expected long-term return
    - Train over thousands of spins for convergence

D. Studying Probability Distributions and Expected Value

Expected Value (EV) is the average result of a bet over time. The formula:

EV = (Win Probability × Win Amount) + (Loss Probability × Loss Amount)

Example – Single Number Bet:

- Win probability = 1/37
- Win amount = 35
- Loss probability = 36/37
- Loss amount = -1

EV = (1/37 × 35) + (36/37 × -1) = -0.027

That means the player loses 2.7p per £1 bet on average. Understanding EV helps identify which bets lose money fastest and which are safer over time.

Use normal distributions or Monte Carlo simulations to model expected results of long-term sessions and visualize how often wins, losses, and profit/loss cycles occur.

Applied to European Roulette using RL

The Concept Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions by interacting with an environment, receiving rewards or penalties, and updating its strategy (policy) over time.

Review Key Terms in the Context of European Roulette

Term	**Explanation (Roulette Context)**
Agent	The AI player (or your algorithm) learning to bet smartly
Environment	The roulette table, wheel, and outcomes
Action	The bet placed (e.g., Red, Black, Number 17, Voisins, etc.)
Reward	Win = +X money, Lose = -1 or 0 depending on outcome
State	Current game condition (e.g., previous 5 outcomes, balance, etc.)
Policy	The agent's betting strategy (e.g., "bet red after 2 blacks")
Episode	One full game or betting session (e.g., 100 spins in a row)

To actually train an RL agent on roulette:

- You define states (like recent spin history)
- Use a method like Q-learning or Deep Q Networks (DQN)
- Simulate 10,000+ episodes
- Reward = money won or lost

Q-learning & DQN Explained (with Roulette Analogy)

Identify the Concept Q-learning is a model-free reinforcement learning algorithm. It learns the value of an action in a given state without needing a model of the environment. It learns by trial and error: Take an action → Get a reward → Update the Q-value

What Happens When the Q-table Gets Too Big? Q-learning uses a Q-table to store all possible (state, action) pairs and their expected future rewards.

When the number of states/actions becomes huge (like in complex environments), the Q-table becomes too big to store or search efficiently.

So, instead of storing values in a table, we use a neural network to approximate those values. This is called a Deep Q Network (DQN).

Deep Q Networks (DQN) extend Q-learning by using a neural network to approximate the Q-values when the state/action space is too large to store in a table.

When the state/action space is too big to store in a table

When the state/action space is too big to store in a table (like in image games or pattern-rich data), we replace the table with a neural network. The DQN predicts Q-values instead of looking them up.

Neural Network
Used in DQN to approximate Q-values when the state/action is large

You'd have millions or billions of possible situations (states), each needing its own entry in the Q-table.

Key Comparison: Q-Table vs. DQN (Neural Network)

Feature	**Q-learning**	**Deep Q-Learning (DQN)**
Memory	Needs full Q-table (big matrix)	Learns to estimate Q-values with a NN
State complexity	Works for small/simple problems	Works for large/complex problems
Example use case	Grid world, small games	Video games, trading, roulette patterns
Representation	Exact lookup	Generalized prediction via function

---

Summary in One Sentence

When there are too many state-action combinations to store in a Q-table, we use a neural network to approximate the Q-values — this makes it Deep Q-Learning (DQN).
In complex environments (like playing a video game or roulette with thousands of possible states), we can't store Q-values in a table.
So instead, we approximate the function Q(s, a) using a neural network:

$Q(s, a) \approx \text{NeuralNetwork}(s, a)$
The neural network learns the mapping from state-action pairs to predicted future rewards.

Q-learning Update Rule

Q-learning Formula $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max(Q(s', a')) - Q(s, a)$

$Q(\text{state, action}) \leftarrow Q(\text{state, action}) + \alpha \times (\text{reward} + \gamma \times \max(Q(\text{next\_state, a})) - Q(\text{state, action}))$
This is the core Q-learning update rule.

Now Compare With DQN (Neural Network)

When the states are too many (e.g., last 5 spins, balance, bet history), we can't use a Q-table like above.

So instead, we use a neural network:

(State, Action) → Q-value
DQN Flow
Input: Game state (e.g., spin history, balance)
Output: Q-values for all possible bets
Agent chooses action with highest Q-value

Feature	Q-table	DQN (Deep Q-Network)
Stores values	In a 2D table (state x action)	Uses a neural network to predict them
Good for	Small, simple problems	Large, complex problems
Limitation	Memory explodes with many states/actions	Needs training time and GPU sometimes

---

Q-table: A grid where each (State, Action) pair maps to a numeric Q-value.

DQN (Deep Q Network): A neural network that takes the state as input and predicts Q-values for all actions.

This diagram helps you understand that:

Q-tables are simple lookup tables (great for small problems).
DQNs are powerful neural networks used when the state/action space is too large for a table.

Q-table: (State, Action) → Q-value

	Bet Red	Bet Black
LastSpin=Red	0.35	0.65
LastSpin=Black	0.60	0.40
LastSpin=Zero	0.50	0.50

DQN: Neural Network Approximation

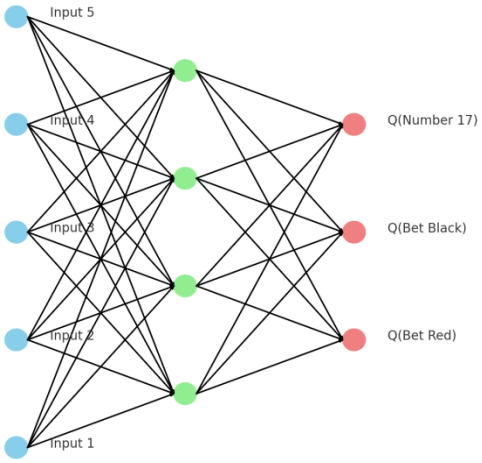


Diagram Summary

Q-table

- Excel-style table of Q-values
- Easy to understand and update
- Bad for high-dimensional state spaces

DQN (Deep Q-Network)

- Neural network that predicts Q-values
- Input: Game data
- Output: Predicted rewards

1. Q-table (Left side of the image)

What It Is:

- A **table (like Excel)** that stores the value of every **(State, Action)** combination.

Example from the Image:

State	Bet Red	Bet Black
LastSpin = Red	0.35	0.65
LastSpin = Black	0.60	0.40
LastSpin = Zero	0.50	0.50

- If the last spin was **Red**, and you **bet on Black**, the expected reward is **0.65** → better choice.
- If the last spin was **Black**, and you **bet on Red**, the expected reward is **0.60**.
- The values improve over time as the agent learns from playing roulette.

Q-table is good when:

- Number of **states and actions is small**
- Easy to update and store

---

2. DQN (Right side of the image)

What It Is:

- A **neural network** (a series of connected neurons/layers) that **predicts Q-values**.
- It replaces the Q-table when you have **too many states/actions** to store in a table.

What You See in the Image:

- **Input nodes** (Input 1 to Input 5):
  These could be game information like:
  - Last few spins
  - Your current balance
  - Last action
  - Time of day
  - etc.
- **Hidden layer**:
  The neural network processes the input here, finding patterns and relationships.
- **Output nodes**:
  Each one gives the **predicted Q-value** for an action:
  - Q(Bet Red)
  - Q(Bet Black)
  - Q(Bet Number 17)

How It Works:

- You pass the **state** as input (e.g., "Last spin was Red, balance is £200").
- The network **outputs the Q-values** for each possible bet.
- The agent chooses the action with the **highest predicted Q-value**.

Simple Analogy to Remember

Q-table	**DQN Neural Network**

Q-table	DQN Neural Network
Like a **cheat sheet**	Like a **thinking brain**
Lookup answers for known cases	Learns to **generalize** from patterns
Easy for small problems	Needed for big, complex decision problems

---

Summary

Feature	Q-table	DQN (Neural Net)
Stores	A grid of state-action values	A function that **predicts** values
Use Case	Simple environments (few states)	Complex environments (many states)
Memory	Can grow very large	Compact and scalable
Learning method	Value lookup and update	Pattern recognition via training

How RL Interacts with an Environment (Roulette)

What this means:

In RL, the **agent** (player) **talks** to the **environment** (roulette game):

- It **observes** the current state (e.g., last spin result)
- It **takes an action** (e.g., bets on Red)
- The environment **responds** with:
    - New state (next spin result)
    - Reward (win or lose money)

Why It's Hard to Beat Roulette Using RL

Real-world truth:

Roulette is a **luck-based game with a fixed house edge (2.7%)**.
There is **no strategy** that can change the physics of the wheel.

But what RL can do:

- Learn **patterns** in simulated data
- Try to find **profitable streaks**
- But it will **never truly beat** roulette long-term

How Neural Networks Generalize Strategies (DQN)

What this means:

Instead of storing Q-values in a table, a **DQN**:

- Takes complex state input (like history, balance, odds)
- Outputs **predicted Q-values** for actions
- Learns **patterns and general rules** from lots of data

A Q-table is like memorizing every situation and response.
A DQN is like understanding **why** something works and applying it in **new** situations.

In roulette:

- If the last 3 spins were Red, and balance is £100 → network learns **whether it's smart to bet on Black**
- DQN **approximates** and **generalizes** across unseen states.
- **Summary Table**

What You Learn	**What It Teaches You**	**Analogy**
RL interaction with environment	Agent learns from actions and feedback	Puppy learning fetch
Q-values and updates	Learn which action gives most reward	Mental notes on what works
Exploration vs. Exploitation	Try new vs. repeat best actions	Trying new food vs. favorites
RL in luck-based games like roulette	Why strategies can't beat pure randomness	Coin flipping with AI
DQN strategy generalization	Learn patterns from complex data, not just memorize	Understanding vs. memorizing

What's Included:

Section	**Description**
Roulette Environment	Simulates spins with Red, Black, and Zero
Q-table Setup	Tracks values for betting actions based on previous spin
Q-learning Algorithm	Updates Q-values with exploration & exploitation
Graph	Visualizes learning progress (average reward)
Final Q-table	Displays learned strategy after training

Why Roulette is a Luck-Based Game

This **bias is the house edge**:
2.7%

Key Point:

Even if you play forever, your **expected loss per £1 bet is 2.7p.**

This is **built into the math** — it's not a strategy problem, it's a **statistical law**.

In **probability and statistics**, the **expected value** is the **average outcome** you can expect if you **repeat an experiment many times**.

What Is Expected Value?

Expected Value (EV) is the **average amount you expect to win or lose** per bet, **in the long run**, based on all possible outcomes and their probabilities.

Expected value is the long-term average outcome of repeated trials.

- It doesn't guarantee the result of one trial.
- But if you play 1,000 times, your average gain or loss per bet will approach the EV.

- It's not what **will** happen in one trial, but what you'd expect to happen **on average** over **many** trials.

Law of Large Numbers & EV

If you repeat the game thousands of times, your **average winnings per game** will get closer and closer to the **expected value**.

So even if you win £35 once in a while.

Concept 1: **Expected Value (EV)**

The average result you expect **in the long run**.

Concept 2: **Law of Large Numbers (LLN)**

As you repeat an experiment **many times**, your **actual average** will get closer to the **expected value**.

One-Line Summary to Tell Students:

Concept	**Analogy**	**Summary**
Expected Value	Average value in candy box	What math says **should happen** on average

Concept	Analogy	Summary
Law of Large Numbers	Flipping more and more coins	As you do it **many times**, your real results **match** the expected average

---

- **Expected Value**: The center of the target (where you aim)
- **Law of Large Numbers**: The more arrows you shoot, the **closer your average hits get to the bullseye**

RL Concepts: Q-learning, SARSA, DQN, PPO, Actor-Critic explain it

Here's a clear, beginner-friendly explanation of each **Reinforcement Learning (RL)** concept — **Q-learning, SARSA, DQN, PPO, and Actor-Critic** — with analogies and differences:

Fundamental Idea of Reinforcement Learning

Think of an **agent** (like a robot or AI player) in an **environment** (like a game or real-world scenario) learning to make decisions by **trial and error** to **maximize rewards** over time.

---

1. Q-learning (Off-Policy)

"Learn what's best, even if you don't do it right now."

- **What it is:** A model-free **value-based** method.
- **Key idea:** Learns the **maximum future reward (Q-values)** for state-action pairs.
- **Equation:** Q[s, a] = Q[s, a] + alpha * (reward + gamma * max(Q[next_state, a_prime]) - Q[s, a])

Variables:

- Q: Q-table (e.g., a dictionary or 2D array)
- s: current state
- a: action taken in state s
- alpha: learning rate
- reward: reward received after action a
- gamma: discount factor
- next_state: resulting state after action
- a_prime: possible actions from next_state (used inside max()

- **Off-policy**: Updates Q-value using the **best possible action** (even if the agent didn't take it).

Analogy: Like learning by watching the best chess moves on YouTube, even if you didn't try them yourself.

---

2. SARSA (On-Policy)

"Learn from what you actually do."

- **What it is:** A model-free **value-based** method similar to Q-learning.
- **Key difference:** Uses the **actual action taken** instead of the best one.
- **Equation:**

  $$Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \cdot Q(s', a') - Q(s,a)]$$ Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \cdot Q(s', a') - Q(s, a) \right]$$Q(s,a) \leftarrow Q(s,a) + \alpha [r + \gamma \cdot Q(s', a') - Q(s,a)]$$

- **On-policy**: Learns from the **policy it is currently using**, including exploratory moves.

Analogy: Like learning from your own chess games, even when you make mistakes.

---

3. DQN (Deep Q-Network)

"Use neural networks to estimate Q-values for complex environments."

- **What it is:** A **deep learning version of Q-learning**.
- **Problem solved:** Handles **large/continuous state spaces** using neural networks.
- **Techniques used:**
  - Experience replay (store past transitions and sample randomly)
  - Target network (a slowly-updating network for stability)

Analogy: Like using a smart assistant that remembers your past mistakes and gradually gets better at a complex video game.

---

4. PPO (Proximal Policy Optimization)

"Improve your policy a little at a time — not too much, not too little."

- **What it is:** A **policy-gradient** method (directly learns policy).
- **Key idea:** Optimizes the **policy network** to maximize expected reward while keeping updates stable.
- Uses **clipping** in its objective function to avoid large, unstable updates.

Analogy: Like learning to ride a bike — you adjust your balance slowly rather than making huge swings that could make you fall.

---

5. Actor-Critic

"One model decides, another evaluates."

- **What it is:** Combines both **policy-based (actor)** and **value-based (critic)** approaches.
- **Actor:** Suggests actions (policy).
- **Critic:** Evaluates how good the action was (value function).
- Can be used in both deterministic and stochastic policy learning.

Analogy: Like a student (actor) doing homework and a tutor (critic) giving feedback after each step.

Summary Table

Concept	Type	Learns What?	Policy Type	Key Feature
Q-learning	Value-based	Optimal Q-values	Off-policy	Learns best action even if not taken
SARSA	Value-based	Actual Q-values	On-policy	Learns from actual path taken
DQN	Value-based	Q-values via NN	Off-policy	Scales Q-learning to large state spaces
PPO	Policy-based	Policy parameters	On-policy	Stable policy updates with clipping
Actor-Critic	Hybrid	Policy + value func.	Both	Actor suggests; Critic critiques

---

on-policy vs. off-policy in reinforcement learning using **simple analogies**

What Does "Policy" **Mean?**

In RL, a **policy** is the strategy that the agent uses to decide what action to take in a given state.

On-Policy**: "Learn from your own experiences."**

The agent **follows a policy** to interact with the environment

And it **learns from the same policy** it uses to explore

Example: SARSA, PPO

- Learns from what it actually does, even if it makes mistakes

---

Off-Policy**: "Learn from someone else's experiences (or the best path), not what you just did."**

The agent follows one policy (e.g., explores randomly)
But learns from a **different, better policy** (e.g., optimal path)

Analogy:

You're watching a master chef cook on YouTube while you're still learning.
You're not cooking exactly like them, but you're updating your brain with **their** best techniques.

- **Example**: Q-learning, DQN
- Learns from the best possible actions — not necessarily the ones it actually took

---

Comparison Table with Analogies:

Concept	Type	Learns From	Analogy
On-Policy	Your own behavior	Like learning to play piano by actually playing it (even with mistakes)	
Off-Policy	Someone else's (better) behavior	Like learning football by watching Messi's moves, not your own practice	

Phase 1: Problem Definition and Research Question

1. Working Title

"Profit Optimization in European Roulette: A Reinforcement Learning Approach to Minimizing Loss and Maximizing Return in Stochastic Gambling Environments"

This is a clear and concise working title, highlighting:

- The problem domain: European Roulette
- The methodology: Reinforcement Learning
- The goal: Loss minimization and profit maximization

---

2. Research Problem / Motivation

What is the broad question?
Can reinforcement learning (RL) agents learn to minimize losses and maximize profits in European Roulette, despite the inherent house edge and randomness of the game?

Why is it important?

Real-world relevance:

- This problem reflects decision-making under uncertainty, common in financial trading, investment management, dynamic pricing strategies, and risk management.
- In such domains, the outcome is uncertain, but intelligent agents must find strategies to optimize returns while mitigating losses.

Academic gap:

- Most reinforcement learning research focuses on environments with clearly defined reward structures and positive or balanced expected value.
- European Roulette is a loss-heavy, high-variance environment with a fixed house edge (2.70%) and unpredictable, stochastic outcomes.
- Few studies explore whether RL can adaptively optimize performance in such adverse conditions.
- This opens new research into reward function design and policy learning in biased environments.

---

3. Research Objectives / Hypotheses

What are the specific aims?

1. Develop a simulation environment for European Roulette that supports dynamic betting options and includes realistic limitations.
2. Design a reward function to promote profit, reduce risk, and maintain balance between volatility and return.
3. Compare performance of RL-based strategies (Q-learning, DQN) with fixed strategies (Martingale, Flat Betting, Keefer).
4. Investigate whether RL agents can detect subtle patterns or anomalies in spin outcomes and exploit them.

Any hypotheses or expected patterns?

1. RL agents will outperform fixed strategies in short-run simulations.
2. RL agents will learn conservative, bankroll-preserving strategies.
3. RL models can detect exploitable patterns in data with RNG flaws or human error.

Research Project Checklist: European Roulette and Reinforcement Learning

1. Identified 2–3 Potential Research Asertationreas

- Reinforcement Learning (RL) in Gambling
- Finding Patterns in Random Games
- Using Simulations to Study Decision-Making

2. Why These Areas Are Interesting

- RL is useful for building smart systems that learn from experience, like how people play games.
- Finding patterns in random data helps us understand real-life randomness.
- Simulations let us test ideas without needing to gamble real money.

3. What I Already Know That Helps

- I know Python, machine learning, and basic statistics.
- I've worked with data and used tools like pandas and NumPy.
- I have some experience with RL frameworks like TensorFlow or Keras.

4. Main Focus Area

- Reinforcement Learning (RL) in Gambling Environments

5. Research Question

Can reinforcement learning (RL) agents learn to minimize losses and maximize profits in European Roulette, despite the inherent house edge and randomness of the game?

6. Why This Question Matters

- It's useful in real-world decision-making, like finance or trading.
- Helps understand how AI can learn in tough situations where there's usually a loss.

7. Is This Question the Right Size?

- Yes, it focuses on one game (roulette) and one method (RL).
- There's enough room to explore agent behavior, rewards, and comparisons with traditional strategies.

8. Can I Actually Answer This Question?

- Yes. I already have a dataset of 100,000 roulette rounds.
- I can use free tools like Python, Google Colab, and RL libraries.
- I have the time and skills to do it.

European Roulette consists of 37 pockets (1–36 and a single zero), offering better odds than its American counterpart, but still maintains a fixed house edge of 2.70%. This edge arises because the game pays out based on 36:1 odds, even though there are 37 outcomes—making every bet statistically unfavorable over time.

To address this limitation, the project will explore and simulate popular progressive betting strategies like Martingale, D'Alembert, and the Keefer progression, all of which attempt to recover losses through structured wager increments. While these strategies may provide short-term gains, they are generally unsustainable due to exponential risk or limited recovery.

Therefore, to improve upon these systems, I will implement and train a Reinforcement Learning (RL) agent using models such as Deep Q-Networks (DQN) or Recurrent Neural Networks (RNNs). These models will be used to dynamically optimize betting actions by learning from historical spin data.

The RL agent will be trained with states such as previous spin outcomes, bankroll, and bet history; actions like choosing different bet types (e.g., dozens, colors, numbers); and rewards based on the profit or loss per spin. Over thousands of episodes, the agent will aim to maximize long-term expected value rather than short-term wins—offering a more robust and data-driven approach to evaluating roulette strategies.

This study will help assess whether intelligent models can outperform traditional betting heuristics and will be simulated strictly for academic and statistical purposes, not for real-money gambling.