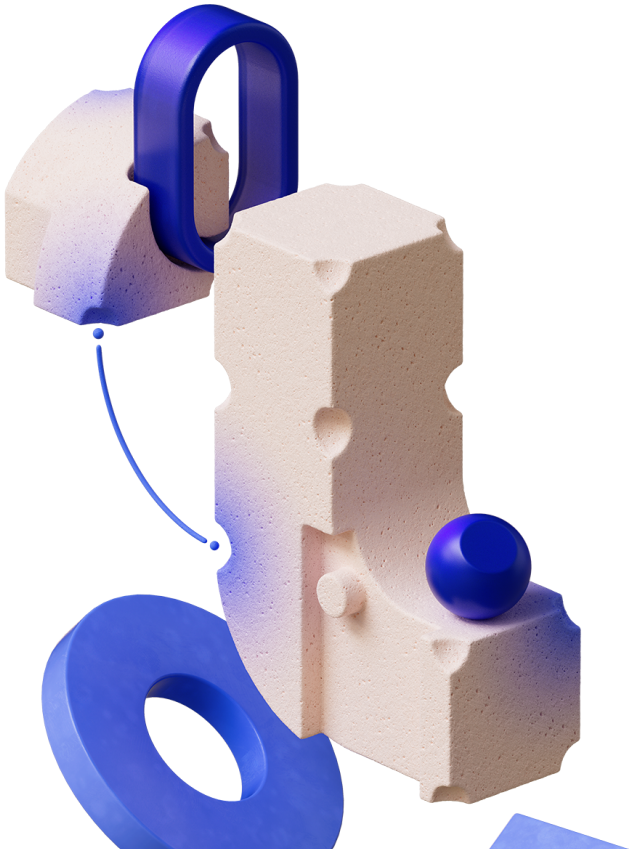
Google DeepMind



Reinforcement Learning 101

Nikita Saxena
Research Engineer

Agenda

Applications

Games

Self-Driving Cars

Wayve.ai has successfully applied reinforcement learning to training a car on how to drive in a day. The example below shows the lane following task. The image in the middle represents the driver's perspective.

Dexterity

Robotics

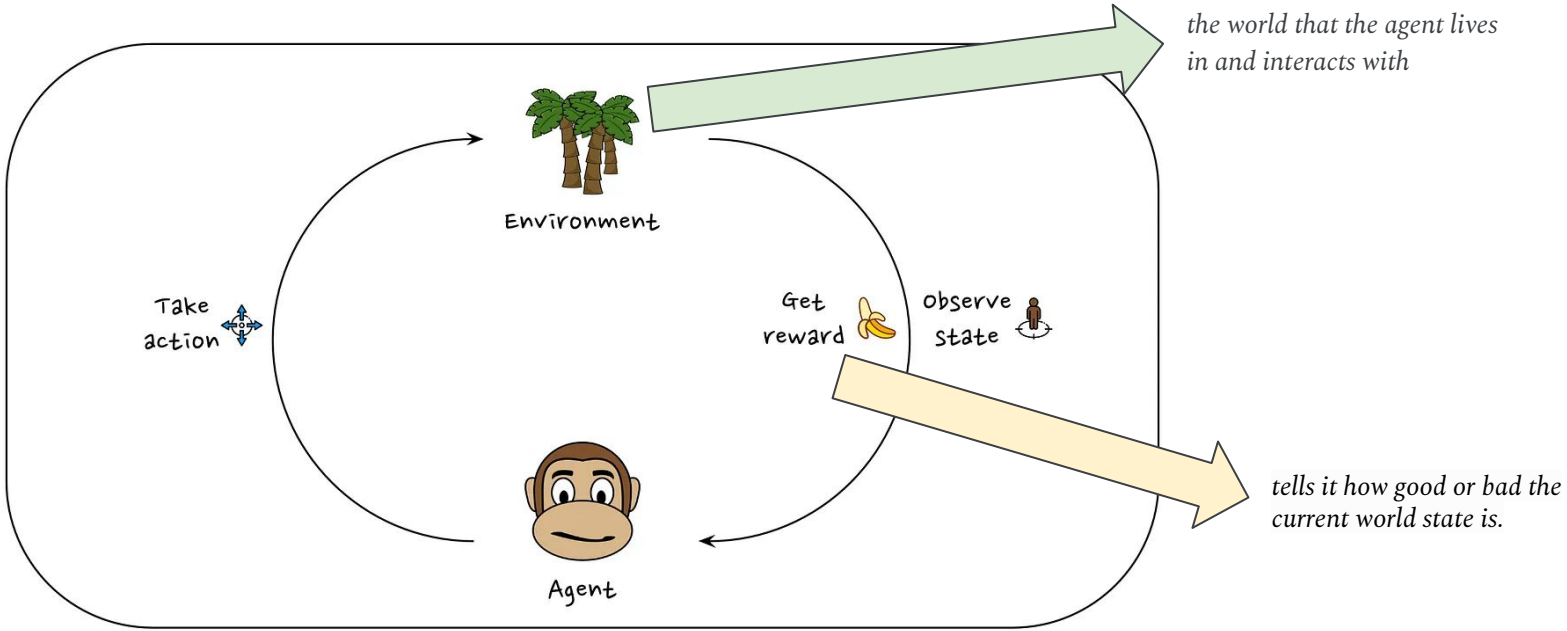
Multi-Agent Simulation

Fundamentals

2

01 Framework

Framework



At every step of interaction, the agent **sees a (possibly partial) observation of the state of the world,** and then **decides on an action to take**. The **environment changes when the agent acts** on it, but may also change on its own. The **goal of the agent is to maximize its cumulative reward**.
Reinforcement learning methods are ways that the agent can learn behaviors to achieve its goal.

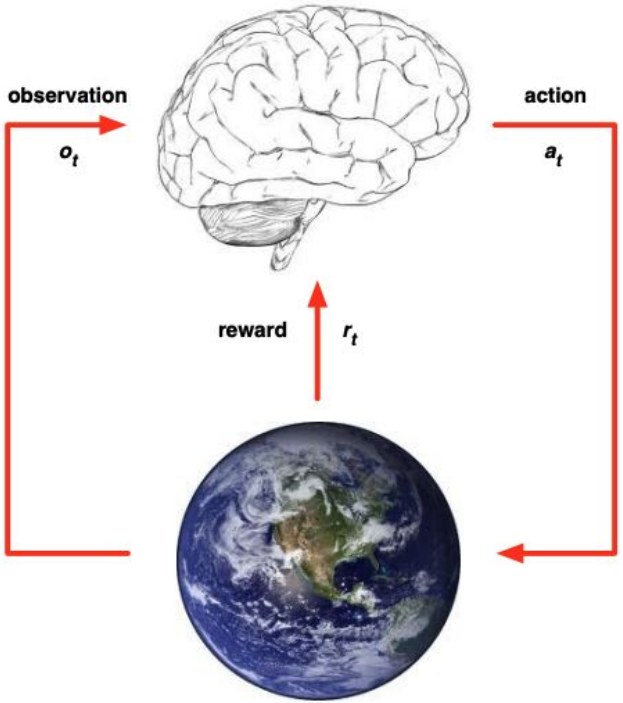
At Each Step

The agent

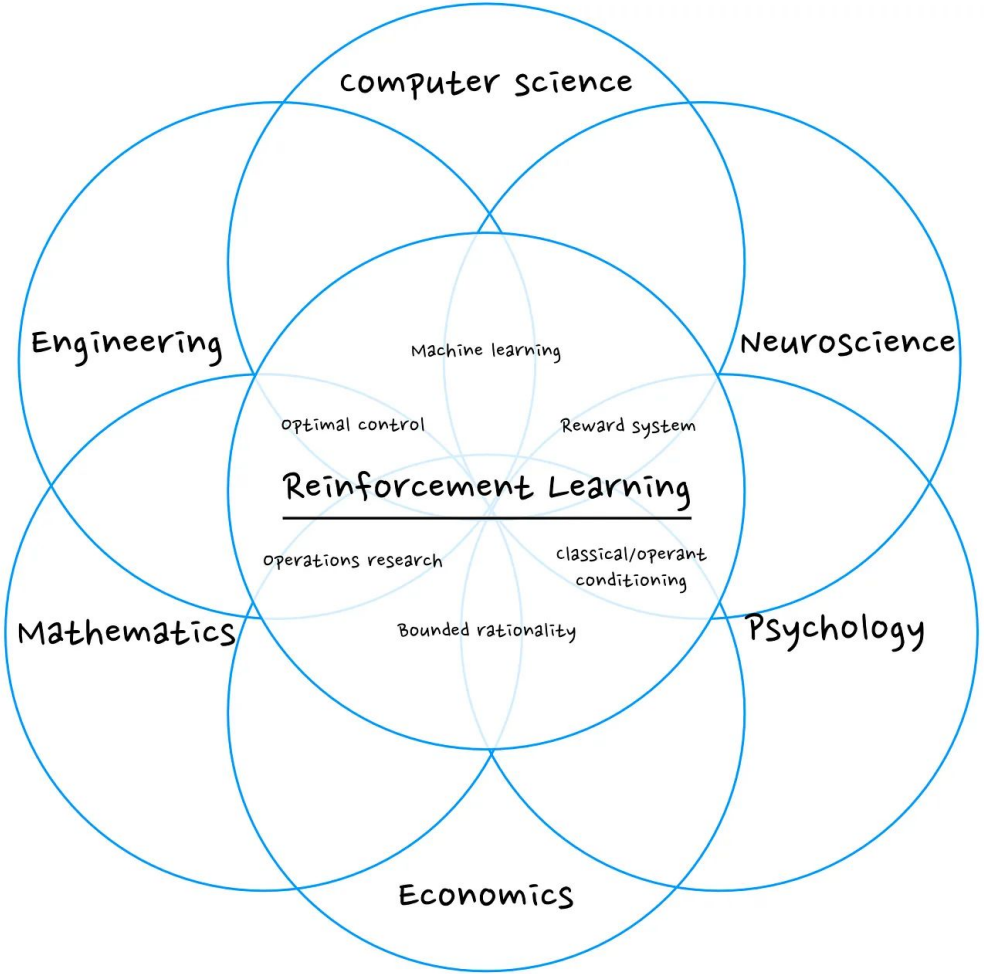
- Executes action a_t
- Receives observation o_t
- Receives scalar reward r_t

The environment

- Receives action a_t
- Emits observation o_{t+1}
- Emits scalar reward r_{t+1}

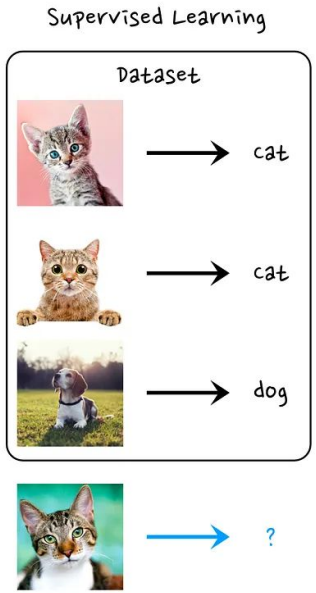


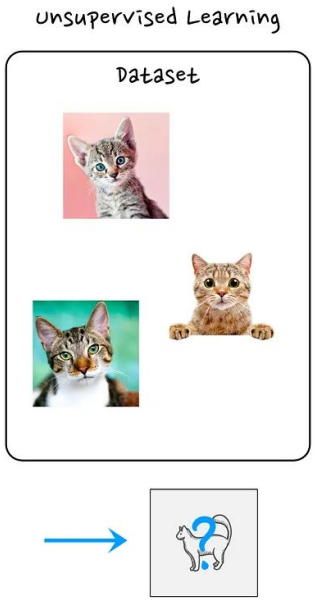
RL is essentially the science of decision taking

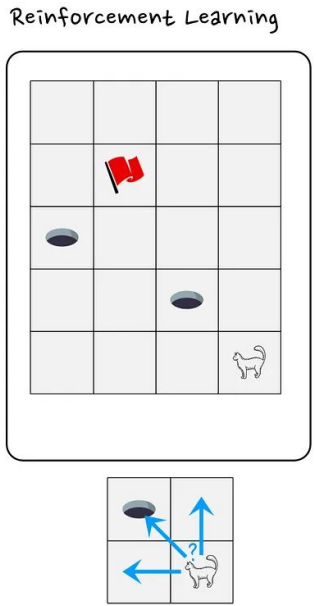


Content taken from David Silverman's course on RL, published by Deepmind

Supervised vs Unsupervised vs Reinforcement







Active Agent

Unlike supervised/unsupervised learning, the RL agent actively explores and interacts with its environment.

“

Imagine we wanted to teach Mario to play using supervised learning.

We would first hire the world's best Mario player and record them playing for 1,000 hours. This gives us a massive, fixed dataset of (game screenshot, expert's button press) pairs.

“

Our 'Supervised-Mario' would be a passive student:

- His job is just to study this dataset and learn to imitate the expert.
- He never gets to play the game himself; he just looks at static pictures and is told the 'right answer'.
- He can only ever be as good as the expert data he was given.

Our 'Supervised-Mario' has his giant mario_expert_plays.csv file. This dataset is fixed. He will study the same examples over and over again, no matter how good he gets. The data never changes.

“

Now, let's go back to our RL-Mario. We don't give him any data. We simply drop him into Level 1 with a controller. He is an active agent. He has to figure everything out for himself by interacting with the world.

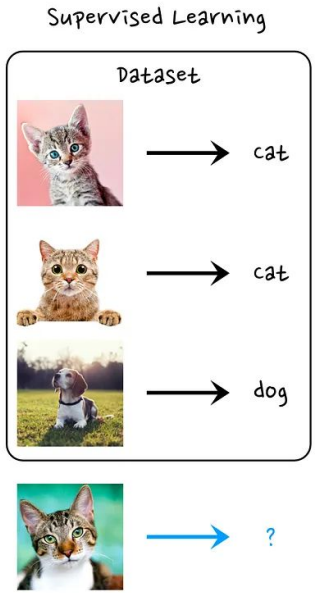
- **He is a scientist running experiments.** His first 'experiment' might be to press run right. He immediately runs into a Goomba and the episode ends. He just generated his first, crucial piece of data: Running right from the start is a bad idea!
- **He learns from failure.** Unlike the supervised model that might never see an example of dying (if the expert is good enough), RL-Mario learns his most valuable lessons from his own mistakes. He learns what not to do by trying it and getting a massive negative reward.

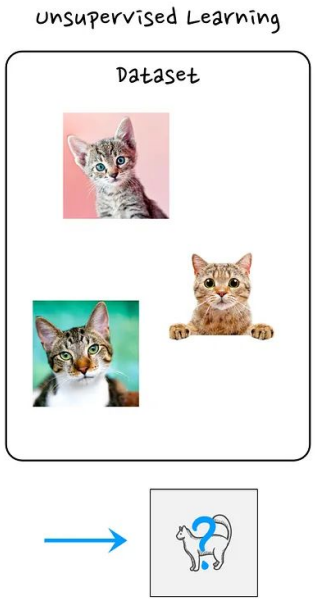
“

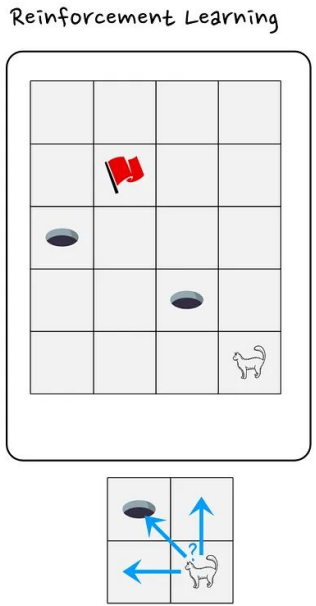
- **He has to explore.** *What's in that ? block?* The only way to find out is to actively navigate Mario underneath it and jump. He has to take actions to reveal information about the world. He might even discover a secret vine block that the expert player never knew about, allowing him to become better than any available expert.

A supervised agent is a student in a library, reading books of what others have done. An RL agent is an explorer in an unknown land, drawing its own map through trial and error.

Supervised vs Unsupervised vs Reinforcement







Dynamic Dataset

The "dataset" isn't a fixed collection of labeled examples. It's a constantly evolving stream of:

1. Actions: The choices our agent makes.
2. Rewards: The feedback received from the environment (positive or negative).

“

RL-Mario's 'dataset' is the dynamic stream of trajectories he generates with every life (episode). The crucial insight is that the quality of this dataset changes as the agent gets smarter.

Early Training (The 'Noob' Dataset)

At first, RL-Mario is terrible. His trajectories are very short: (start -> run right -> die), (start -> jump -> fall in pit). The 'dataset' he generates is full of basic mistakes and only covers the first few screens of the level.

“

Mid-Training (The 'Improving' Dataset)

By learning from these early mistakes, his policy improves. He now knows how to get past the first Goomba and the first pit. His new trajectories are longer. He is now generating a completely different dataset that includes experiences like 'stomping a Koopa' or 'getting mushroom from a ? block'. He is generating data from parts of the w he couldn't even reach before.

Late Training (The 'Pro' Dataset)

Eventually, RL-Mario becomes an expert. His trajectories are long, high-scoring, and efficient. The 'dataset' he is generating now consists of expert-level play, which he uses to make the final, subtle optimizations to his strategy.

“

This creates a powerful feedback loop:

- His current policy determines the kind of data he collects.
- He uses that data to improve his policy.
- The better policy allows him to collect new, higher-quality data.
- ...and the cycle continues.

In summary: The data in supervised learning is a fixed photograph of the past. The 'data' in reinforcement learning is a live video feed of the present, directed by the agent itself, which gets more and more interesting as the agent learns.

02

Terminology

I. States and Observations
II. Action Spaces
III. Policies
    A. Deterministic
    B. Stochastic
IV. Trajectories, Episodes and Rollouts
V. Reward and Return
VI. Value Function
    A. State Value vs. Action Value
VII. Advantage Function

States and Observation

A **state** is a complete description of the state of the world. There is no information about the world which is hidden from the state. An **observation** is a partial description of a state, which may omit information.

In deep RL, we almost always represent states and observations by a real-valued vector, matrix, or higher-order tensor. For instance, a visual observation could be represented by the RGB matrix of its pixel values; the state of a robot might be represented by its joint angles and velocities.

“

Imagine you pause the Mario game.

The state is everything on the screen: Mario's exact (x, y) position, his velocity, whether he is Big or Small Mario, the positions of all the Goombas and Koopas, the locations of the ? blocks, etc.

A single frame of the game is a perfect visual for the state.

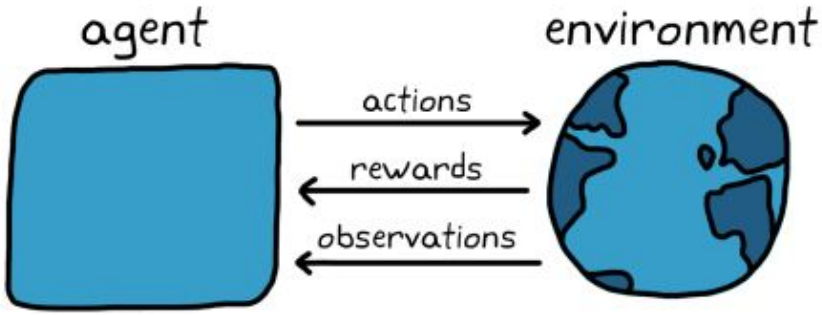
Action Spaces

Different environments allow different kinds of actions.

The set of all valid actions in a given environment is often called the **action space**.

Discrete Action Spaces: only a finite number of moves are available to the agent. For example in Atari and Go.

Continuous Action Spaces: In continuous spaces, actions are real-valued vectors. For example, where the agent controls a robot in a physical world.



“

In Mario, we have a discrete action space.
The actions are the buttons on the controller.

RL-Mario can choose to

- move right,
- move left,
- jump,
- do nothing,
- or a combination like run right and jump.

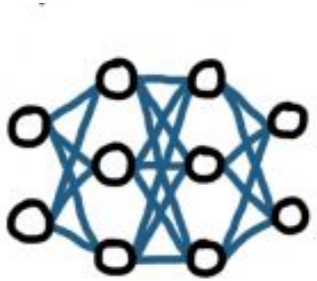
There's a finite set of moves he can make at any given moment.

Policies

Policy, π, tells us which action to take in state s.
The policy is essentially the brain and is often called the “agent”.

It is a mapping from state s to action a and can be either deterministic or stochastic.

- **Deterministic**: $\pi(s) = a$
- **Stochastic** : $\pi(a|s) = P_\pi[A = a|S = s]$



Key computations are important for using and training stochastic policies:

1. sampling actions from the policy,
2. and computing log likelihoods of particular actions.

In deep RL, we deal with **parameterized policies**: policies whose outputs are computable functions that depend on a set of parameters (eg the weights and biases of a neural network) which we can adjust to change the behavior via some optimization algorithm.

Deterministic Policies

This is a code snippet for building a simple deterministic policy for a continuous action space.

The policy here is a multi-layer perceptron (MLP) network with two hidden layers of size 64 and activation functions.

```
pi_net = nn.Sequential(
              nn.Linear(obs_dim, 64),
              nn.Tanh(),
              nn.Linear(64, 64),
              nn.Tanh(),
              nn.Linear(64, act_dim)
            )


obs: batch of observations
obs_tensor = torch.as_tensor(obs, dtype=torch.float32)
actions = pi_net(obs_tensor)
```

“

A deterministic RL-Mario is like a perfectly programmed but unintelligent robot.
Its rulebook says: 'IF a Goomba is 3 blocks away, THEN jump.' It will always jump. No exceptions.

The downside?
This is very brittle. What if there's a low ceiling above the Goomba? The deterministic Mario will still jump, hit his head, and fall onto the Goomba. It can't adapt to slight variations, and it makes it very hard to discover new tricks because it never tries anything different.

Stochastic Policies: Categorical

A categorical policy classifies which action to take. It uses a neural network to predict the probability of taking each action, allowing the agent to make decisions in a probabilistic and exploratory manner.

Sampling Actions

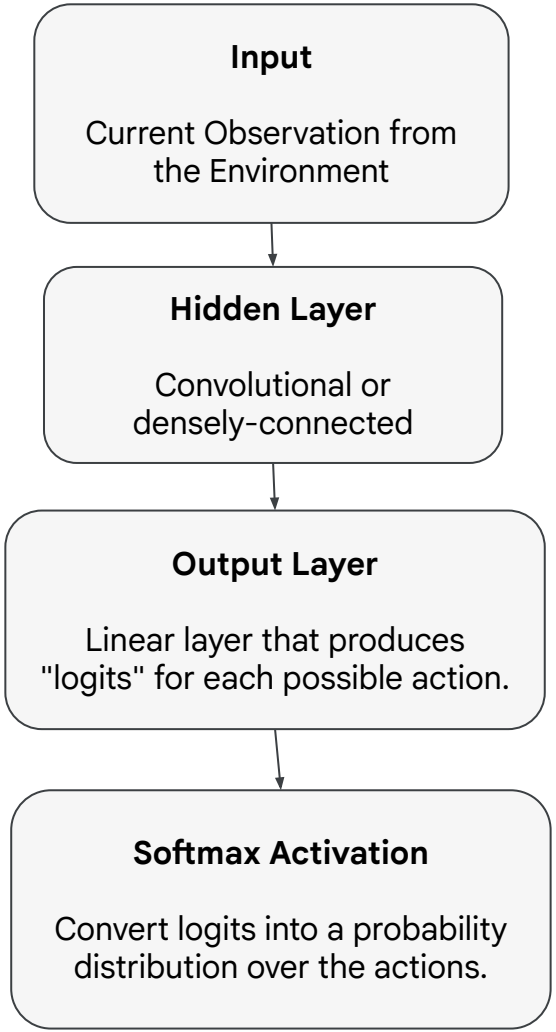
The output of the softmax is a set of probabilities, one for each action. We don't always take the action with the highest probability (to encourage exploration). Instead, we sample from this distribution.

Log-Likelihood

It measures how likely a specific action a was, given the current state s and the policy's parameters θ.

$$log\pi_\theta(a|s) = log[P_\theta(s)]_a$$

where $P_\theta(s)$ is the vector of action probabilities and [ ]_a denotes indexing the probability of action a.



“

This is the Mario we've been describing. His policy is more nuanced. It says: *'IF a Goomba is 3 blocks away, then there's a 95% chance I should jump, a 3% chance I should wait, and a 2% chance I should shoot a fireball (if I have the power-up).'*

The upside?

This randomness is incredibly powerful for learning! That 3% chance to 'wait' might seem suboptimal, but one time, Mario might try it and discover that waiting causes the Goomba to walk right into a pit.

This built-in exploration allows RL-Mario to discover new, potentially better strategies that a rigid, deterministic policy would never find

Trajectories, Episodes, Rollouts

A trajectory is a **sequence of states and actions** in the world: $\tau = (s_0, a_0, s_1, a_1, ...).$

The very first state of the world, is randomly sampled from the **start-state distribution**: $s_0 \sim \rho_0(\cdot).$

State transitions (what happens to the world between the state s_t at time t and the state s_{t+1} at $t+1$), are governed by the laws of the environment, and depend on only the most recent action a_t,

They can be either deterministic: $s_{t+1} = f(s_t, a_t)$

or stochastic: $s_{t+1} \sim P(\cdot|s_t, a_t).$

Actions come from an agent according to its policy.

“

A trajectory is like the detailed logbook from one of Mario's lives. It looks like this:

- s_0 (start of level), a_0 (move right), r_1 (-1 for time)
- s_1 (in front of ? block), a_1 (jump), r_2 (+200 for coin)
- s_2 (under the block), a_2 (move right), r_3 (-1 for time)

...and so on, until the episode ends.

This logbook—this trajectory of experience—is the fundamental data that all our learning algorithms (Actor-Critic, PPO, etc.) use. They analyze thousands of these trajectories to figure out which actions lead to high scores.

“

In our Mario game,

The reward is the game's score system.

- Stomp a Goomba: +100 (positive reward)
- Get a coin: +200 (positive reward)
- Finish the level: +5000 (large positive reward)
- Fall in a pit or touch an enemy: -1 life (very large negative reward)
- Each second that passes: -1 (small negative reward to encourage speed)

Reward and Return

$$r_t = R(s_t, a_t, s_{t+1})$$

Frequently, simplified to:

$$r_t = R(s_t, a_t)$$

The goal of the agent is to maximize some notion of cumulative reward over a trajectory.

1. **Finite-horizon undiscounted return**: sum of rewards obtained in a fixed window of steps. $R(\tau) = \sum_{t=0}^{T} r_t.$

“

Finite-horizon undiscounted return is just a normal level of Super Mario Bros.

- The episode starts when the level begins and ends when Mario either dies or hits the flagpole. The game is finite; it will end.
- In this case, we often don't even need to discount (or we can think of $\gamma=1$). The goal is to maximize the simple sum of rewards because we know the sum won't be infinite.
- Return = $r_1 + r_2 + r_3 + \ldots + r\_T$
  (where r_T is the final reward at the end of the episode T).
- This is the most intuitive way a human thinks: 'Get the highest score possible before the level is over.'"

Reward and Return

$$r_t = R(s_t, a_t, s_{t+1})$$

Frequently, simplified to:

$$r_t = R(s_t, a_t)$$

The goal of the agent is to maximize some notion of cumulative reward over a trajectory.

1. **Finite-horizon undiscounted return**: sum of rewards obtained in a fixed window of steps.

$$R(\tau) = \sum_{t=0}^{T} r_t.$$

2. **Infinite-horizon discounted return**: Sum of all rewards *ever* obtained by the agent, but discounted by how far off in the future they're obtained.
This formulation of reward includes a discount factor $\gamma \in (0, 1)$

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

“

Imagine RL-Mario discovers a 'Goomba Stomping Machine'—a spot in the level where two pipes endlessly spawn Goombas.

Mario could just stand there, stomping Goombas forever, getting +100 points every second.

- If his goal is to maximize the simple sum of all points, what's his total score? Infinity.
- Now, what if there's another strategy: complete the level, which also gives a ton of points. The total score for that strategy might also be considered infinite if he can go back and forth.

How does the algorithm compare one infinity to another? It can't. The math breaks down. The agent would have no reason to prefer finishing the level over getting stuck in a boring, point-farming loop.

“

To solve these problems, we introduce a discount factor, a number between 0 and 1 we call Gamma (γ).

The idea is simple: points earned in the future are worth less than points earned right now."

Let's say we set γ = 0.9.

- A reward received now is worth 100% of its value.
- A reward received 1 step in the future is worth only 90% of its value.
- A reward received 2 steps in the future is worth only 90% of 90%, which is 81% of its value.
- And so on...

“

Let's see what happens to the score with our discount $\gamma = 0.9$.

- Mario stomps the 1st Goomba: +100 points.
- He stomps the 2nd Goomba a second later: $+100 * 0.9 = +90$ discounted points.
- He stomps the 3rd Goomba: $+100 * (0.9)^2 = +81$ discounted points.
- He stomps the 4th Goomba: $+100 * (0.9)^3 = +72.9$ discounted points.
- Return = $r_1 + \gamma^* r_2 + \gamma^{2*} r_3 + \gamma^{3*} r_4 + \ldots$

Because each future reward is worth less and less, even if Mario stomps Goombas forever, the total sum of his discounted rewards will add up to a finite number (in this case, 1000).

“

Now, the algorithm can make a meaningful choice!

It can compare the finite value of the Goomba machine (1000) to the discounted value of another strategy (completing the level).

It solves both problems: the math no longer breaks, and it incentivizes Mario to get high rewards sooner rather than later.

Reward and Return

$$r_t = R(s_t, a_t, s_{t+1})$$

Frequently, simplified to:

$$r_t = R(s_t, a_t)$$

The goal of the agent is to maximize some notion of cumulative reward over a trajectory.

1. **Finite-horizon undiscounted return**: sum of rewards obtained in a fixed window of steps.

$$R(\tau) = \sum_{t=0}^{T} r_t.$$

2. **Infinite-horizon discounted return**: Sum of all rewards *ever* obtained by the agent, but discounted by how far off in the future they're obtained.
This formulation of reward includes a discount factor $\gamma \in (0, 1)$

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t.$$

Penalizing Future Rewards?

- The future rewards may have higher uncertainty; i.e. stock market.
- The future rewards do not provide immediate benefits; i.e. As human beings, we might prefer to have fun today rather than 5 years later ;).
- Discounting provides mathematical convenience; i.e., we don't need to track future steps forever to compute return.

“

The discount factor sets the maximum possible value of a long-term plan. It determines whether a long-term discovery is worth remembering.

Let's set up the scenario again:

- **Path A (The Safe Bet):** A short path with a few coins. The immediate reward is small but certain. V(Path A) = +400.
- **Path B (The Scary Path):** A long, difficult path with no rewards along the way. It might lead to a dead end, or it might lead to a secret area with a huge reward. The outcome is unknown.

“

Low-Patience Mario ($\gamma = 0.90$)

Imagine Mario discovers the secret room on Path B is worth +10,000 points, but it takes 30 steps to get there. The discounted value of that reward would be 10000 * (0.90)^30, which is only about +423 points. From his "low patience" perspective, the value of the difficult Path B (+423) is barely better than the easy Path A (+400). He might not bother with it in the future.

High-Patience Mario ($\gamma = 0.99$)

With a higher discount factor, the same reward is valued more. The discounted value is 10000 * (0.99)^30, which is about +7,397. Now this is a huge difference! From his "high patience" perspective, Path B is clearly the superior strategy once it has been discovered.

“

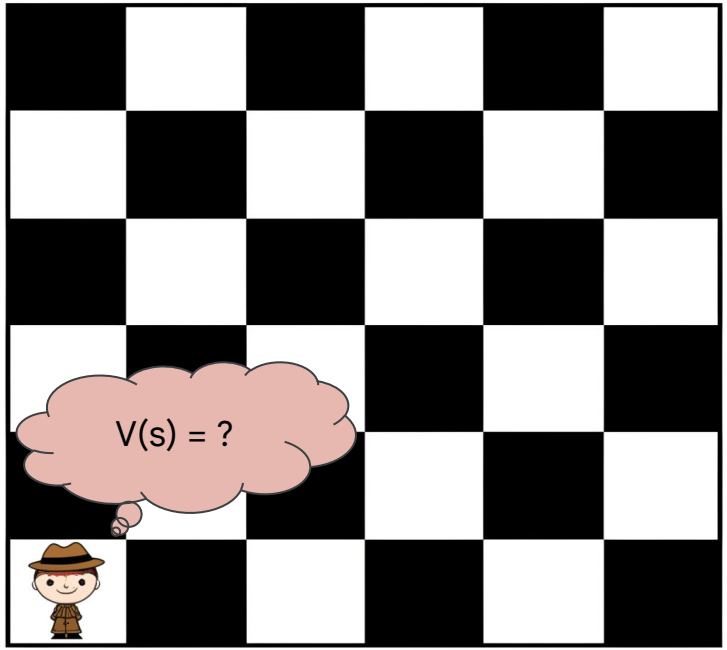
The discount factor γ acts as a "patience" knob.

A higher γ allows the agent to appreciate and value long-term rewards once it finds them.

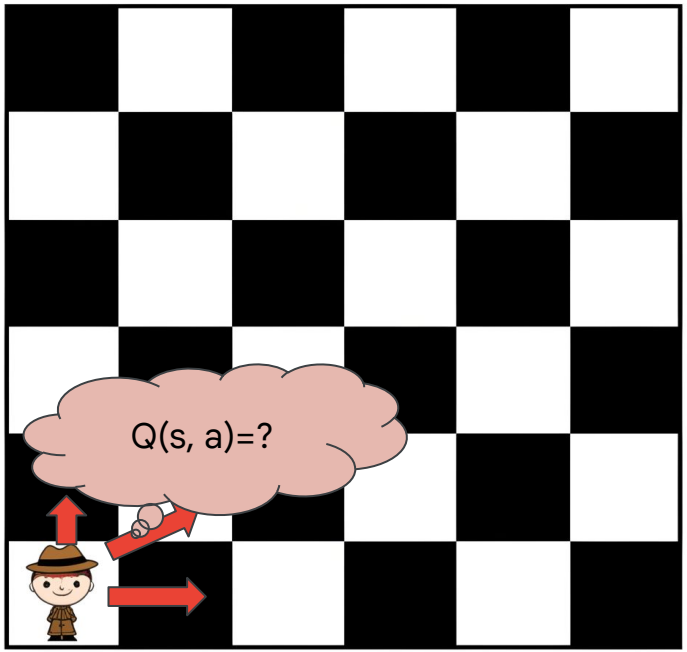
It makes long-term planning viable.

The Value Function

State Value Function (V)



Action Value Function (Q)



“

State-Value the total future score RL-Mario can expect if he starts in that state and plays optimally.

The value of the state where Mario is standing right in front of the final flagpole is very high.

The value of the state where he's surrounded by three Hammer Bros is very low.

V(s) tells us how 'good' a situation is."

“

Imagine RL-Mario is standing in front of a pit (state s). He has two main actions: jump or run right.

- **Q(s, run right):** This Q-value would be very low (e.g., -10,000). Why? Because running right leads to falling in the pit, losing a life, and getting a massive negative reward.
- **Q(s, jump):** This Q-value would be high (e.g., +4,000). Why? Because jumping successfully gets him over the pit and allows him to continue th level, where he can collect more coins and eventually reach the flag.

Q is not the immediate reward, but the total future score RL-Mario can expect to get if he takes that action and continues playing optimally from then on.

The Value Function: State Value vs Action Value

Concept	State-Value Function (V)	Action-Value Function (Q)
Question it Answers	"What is my expected future reward if I start in *this state*?"	"What is my expected future reward if I start in *this state* and take *this action*?"
Input	State s	State s and Action a
Output	A single number: The value of the state.	A single number: The value of the state-action pair.
Use Case	Useful for evaluating positions. "Is this a good board state to be in?"	Crucial for decision-making. Allows us to directly compare actions. "Which move is best *right now*?"
Analogy	Knowing the average home price in a neighborhood (state).	Knowing the specific price of a particular house (state, action) you're considering buying.

03

Optimization Problem

In reinforcement learning, our goal isn't just to find a solution, but the best possible solution. This is what we call being "optimal."

An optimal policy is one that achieves a higher expected return than any other policy from any starting state.

“

Let's put it all together. RL-Mario's grand mission is to solve the RL Optimization Problem.

- He lives through thousands of episodes (lives).
- Each episode generates a trajectory (a logbook) with a final score.
- His goal is to tweak his policy (his instincts) based on all this experience.
- The 'solution' to the problem is finding the optimal policy (π^*)—the perfect set of instincts that makes RL-Mario, on average, get the highest possible score over many, many lives.

RL Optimization problem

Whatever the choice of return measure (whether infinite-horizon discounted, or finite-horizon undiscounted), and whatever the choice of policy, **the goal in RL is to select a policy which maximizes expected return when the agent acts according to it.**

Let's suppose that both the environment transitions and the policy are stochastic. In this case, the probability of a T -step trajectory is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

The expected return (for whichever measure), denoted by , is then: $J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \underset{\tau\sim\pi}{\mathrm{E}}[R(\tau)]\,.$

The central optimization problem in RL can then be expressed by $\pi^* = \arg\max_{\pi} J(\pi),$

With π^* being the **optimal policy**.

The Optimal Value Functions

Optimal State-Value Function: V(s)

V*(s) is the maximum possible expected future return you can get starting from state s. It represents the value of being in state s *if you follow the best policy from that point onwards*.

$$V^*(s) = \max_{\pi} \underset{\tau \sim \pi}{E}[R(\tau)|s_t = s]$$

Optimal Action-Value Function: Q(s, a)

Q*(s, a) is the maximum possible expected future reward you can get by taking action a in state s, *and then following the best possible policy forever after*.

$$Q^*(s, a) = \max_{\pi} \underset{\tau \sim \pi}{E}[R(\tau|s_t = s, a_t = a]$$

The Key Relationship

If you have the optimal Q-function, Q*, the optimal state-value V* is simply the value you get by taking the best possible action from that state: $V^*(s) = max_a Q^*(s, a)$

The Optimal Policy

- An optimal policy π^* is any policy that tells you how to act in order to achieve the optimal value functions.
- **Finding π^* is the end goal of all RL algorithms.**
- **If you know Q***, finding the optimal policy is easy:
  - In any given state s, the optimal policy π^* is to simply choose the action a that maximizes $Q^*(s, a)$.
  - $\pi^*(s) = \text{argmax}_a Q^*(s, a)$

If we can find the optimal action-value function Q*, we have solved the problem. The challenge of reinforcement learning is the search for Q*.

“

Optimal Q-Function (Q^*):
Imagine a magical, all-knowing cheat sheet for Super Mario Bros. This isn't just a good cheat sheet learned through trial-and-error; it is the perfect and complete cheat sheet. For every possible situation (state s) and every possible action (action a), it tells you the exact maximum possible score you will get from that point onward if you play perfectly. We call this the **“Perfect Cheat Sheet, Q^*”.**

Optimal Policy (π^*):
This is the perfect way to play the game. It is the single best strategy that guarantees the highest possible average score. We can think of this as RL-Mario having **“Perfect Instincts, π^*”**

How the Perfect Cheat Sheet (Q*) Creates the Perfect Policy (π*)?

“

Let's say we magically give our RL-Mario the God-Mode Cheat Sheet (Q*). How would he use it to play perfectly?

RL-Mario finds himself in a state s (standing before a pit with a moving platform). He wants to know what to do. He pulls out his God-Mode Cheat Sheet (Q*) and looks up his current state s. The sheet shows him the true, long-term value for every action he can take:

- Q*(s, jump now) = +7,500
- Q*(s, wait 0.5 sec then jump) = +9,200
- Q*(s, run right) = -5,000 (falling into the pit)

What should Mario do? It's obvious! He should choose the action with the highest Q-value: wait 0.5 sec then jump.

“

The optimal policy (π^*) is simply to always choose the action that has the highest value on the optimal Q-function (Q^*).

In any situation, you just look at your perfect cheat sheet and pick the best option. This is a greedy policy because you are greedily picking the best action every single time.

In short: If you have the perfect cheat sheet, the perfect strategy is to always follow its best advice.

$$\pi^*(s) = \text{argmax\_a } Q^*(s, a)$$

“

How the Perfect Policy (π*) Defines the Perfect Cheat Sheet (Q*)

“

What rule does the God-Mode Cheat Sheet (Q*) have to follow to be considered 'perfect'?

The values on the cheat sheet must be self-consistent and obey the rules of the game and the perfect strategy.

The value written for any given (state, action) pair on the sheet must be equal to:
(The immediate points you get for that action) + (the value of the best possible move from the new situation you land in, discounted because it's in the future).

“

Let's go back to our example:
Q*(s, wait 0.5 sec then jump) = +9,200.

Why is the value +9,200?
Because the game's creator knows that if Mario waits and then jumps, he will get:

- An immediate reward (r): Let's say +100 points for landing on the platform.
- He will arrive in a new state, s'. From this new state s', he will again act optimally. He will consult his cheat sheet from s' and pick the best action, which itself has a value.
- The cheat sheet tells him this future value is +9,100 (after discounting).
+100 (immediate reward) + +9,100 (future value) = +9,200.

“

The value of the optimal Q-function, $Q^*(s, a)$, is defined as the immediate reward you get, plus the discounted value of the next state, assuming you continue to follow the optimal policy (π^*) forever.

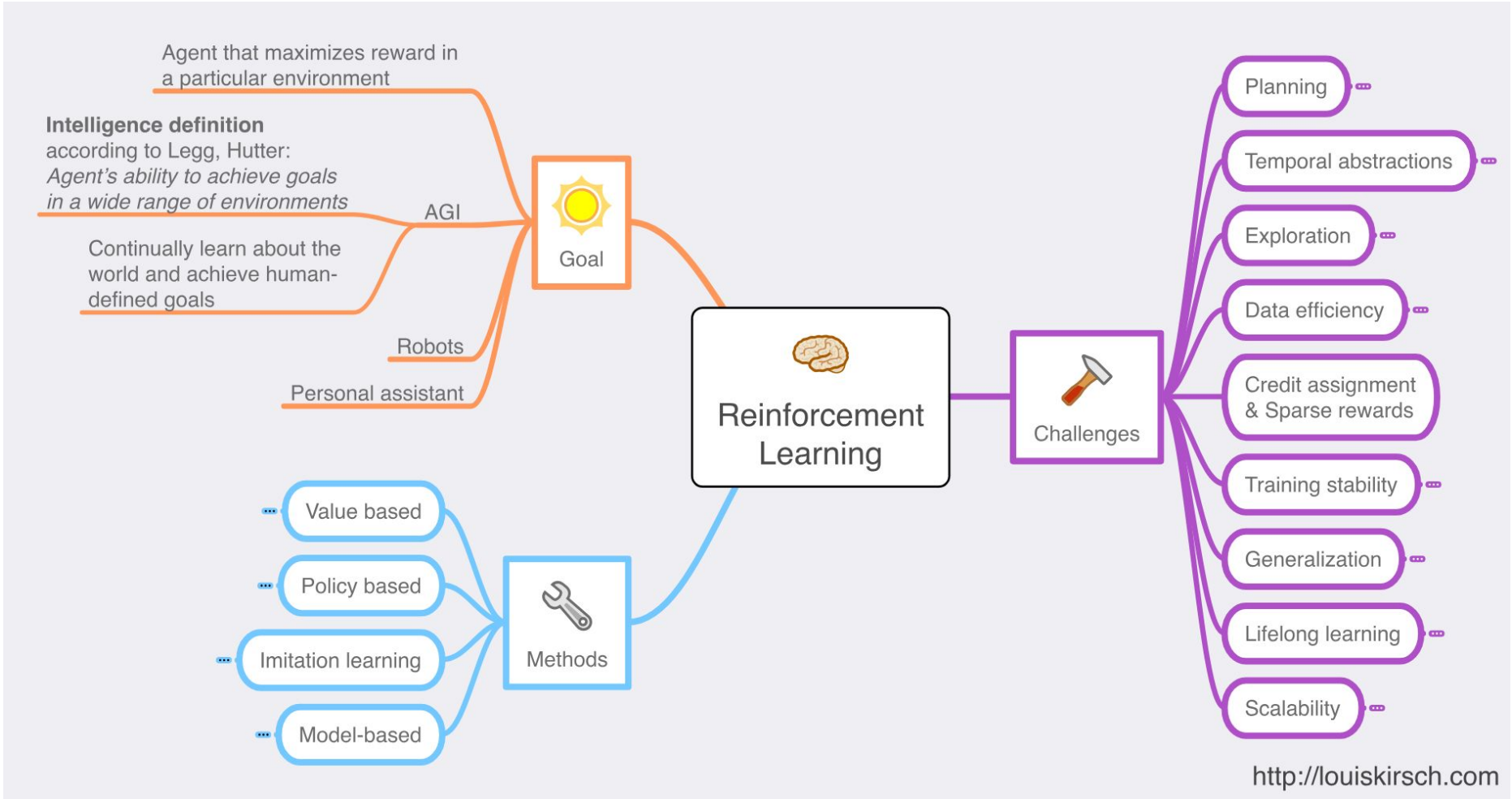
Since the optimal policy is to always pick the max Q-value, this creates a recursive relationship known as the Bellman Optimality Equation.

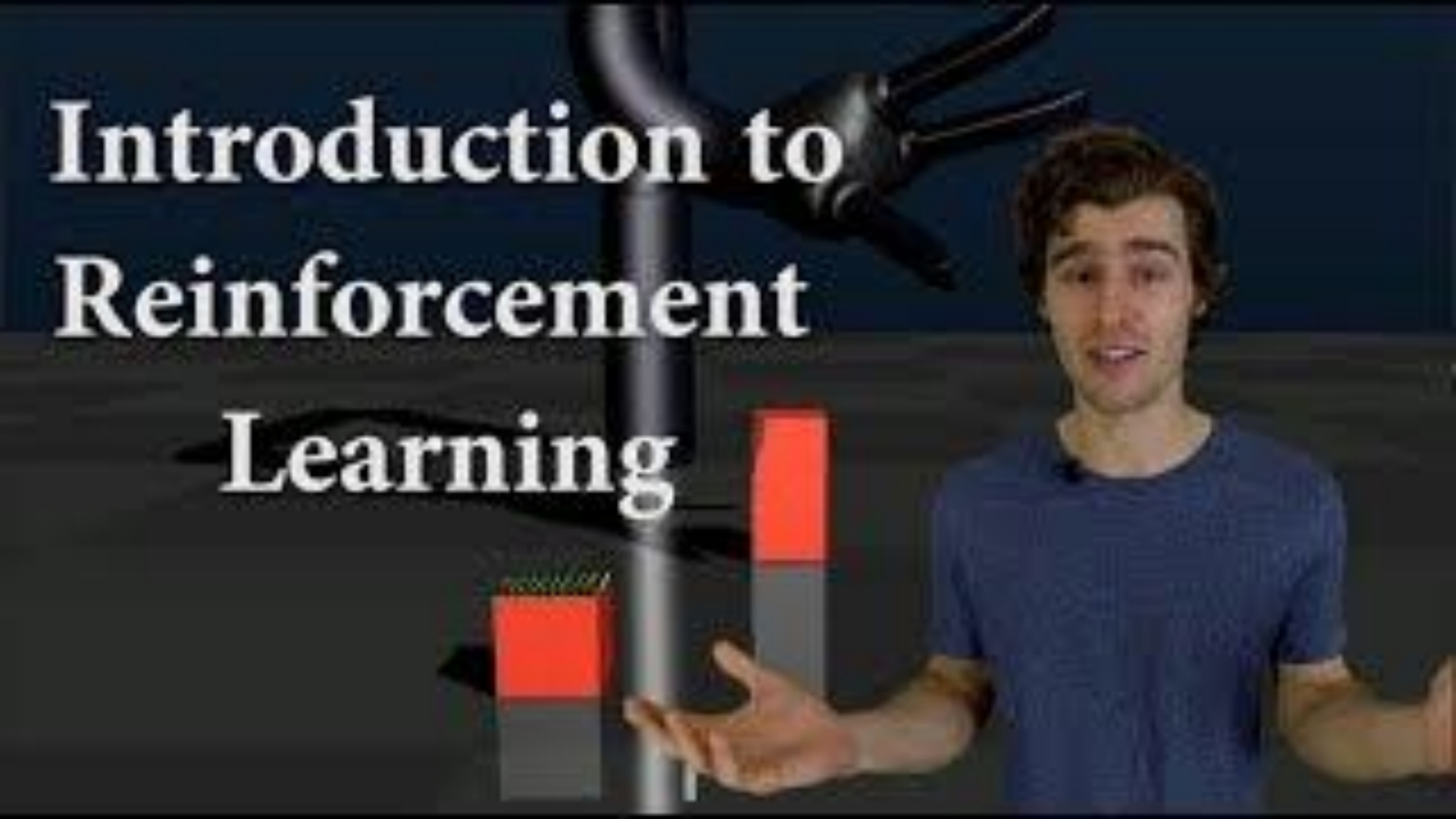
In short: The perfect cheat sheet is defined by the fact that its values correctly predict the outcome of following the perfect strategy.

$$Q^*(s, a) = E[r + \gamma * \max_{a'} Q^*(s', a')]$$

03

Quick Recap

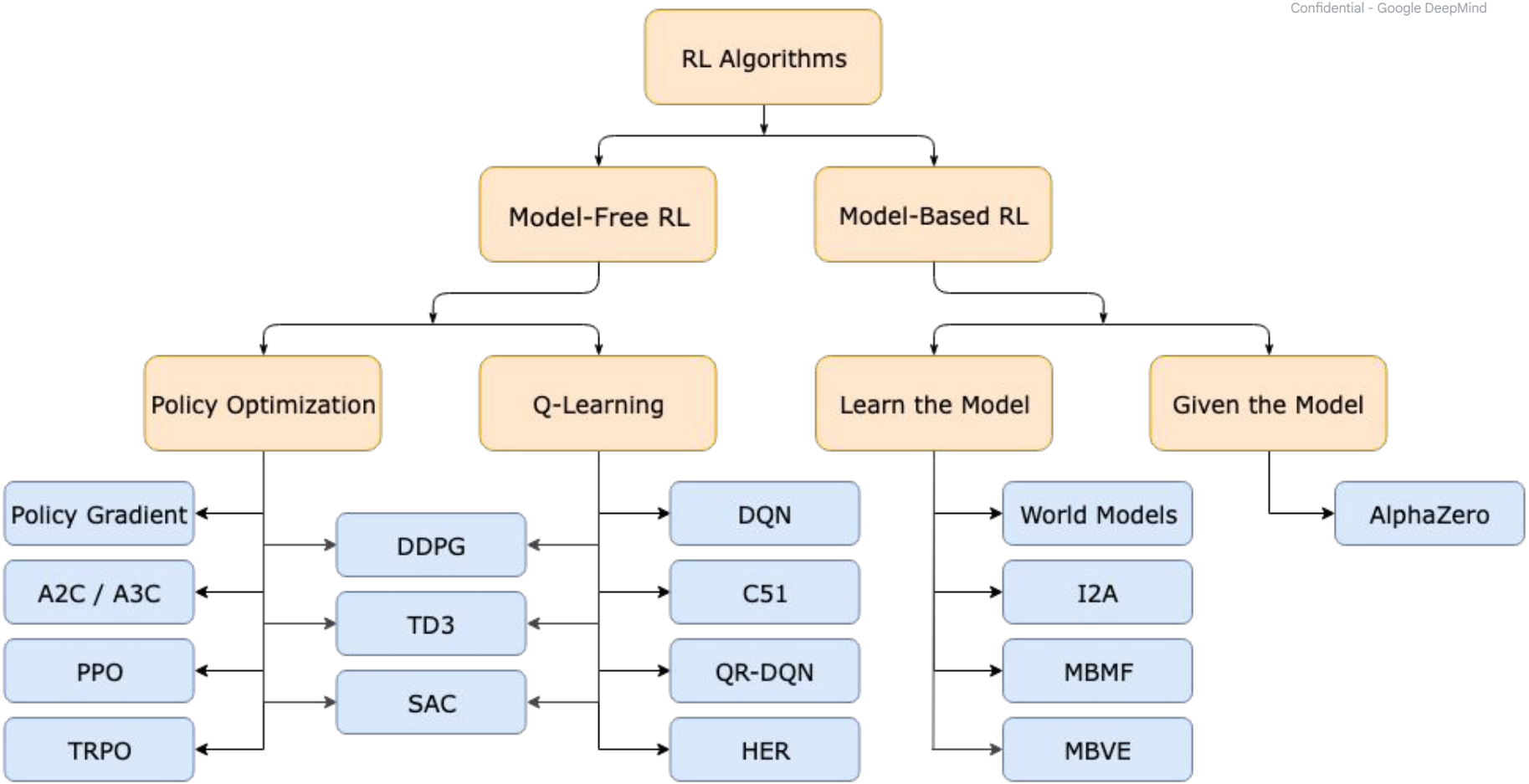

Agent that maximizes reward in a particular environment
Intelligence definition according to Legg, Hutter: Agent's ability to achieve goals in a wide range of environments
AGI
Continually learn about the world and achieve human-defined goals
Robots
Personal assistant
Goal
Reinforcement Learning
Value based
Policy based
Imitation learning
Model-based
Methods
Challenges
Planning
Temporal abstractions
Exploration
Data efficiency
Credit assignment & Sparse rewards
Training stability
Generalization
Lifelong learning
Scalability
http://louiskirsch.com


Introduction to
Reinforcement
Learning

RL Algorithms

02

Overview



02

Model Free vs Model Based Learning

“

Model-Free is like a player who has never seen a video game before. He doesn't know what a Goomba is or that gravity exists. He just presses buttons randomly at first.

He runs into a Goomba and dies.

- Experience: "Touching brown mushroom thing from this position is bad."
- He updates the Q-value for that state-action pair to be lower.

He jumps and hits a block, and a coin comes out.

- Experience: "Jumping under blocks from this position is good."
- He updates that Q-value to be higher.

He's building his Q-value cheat sheet from scratch without a rulebook. Q-Learning is a classic model-free algorithm.

“

Model-Based is like when RL-Mario has **an internal physics engine for the Mushroom Kingdom**. He can imagine or simulate the future.

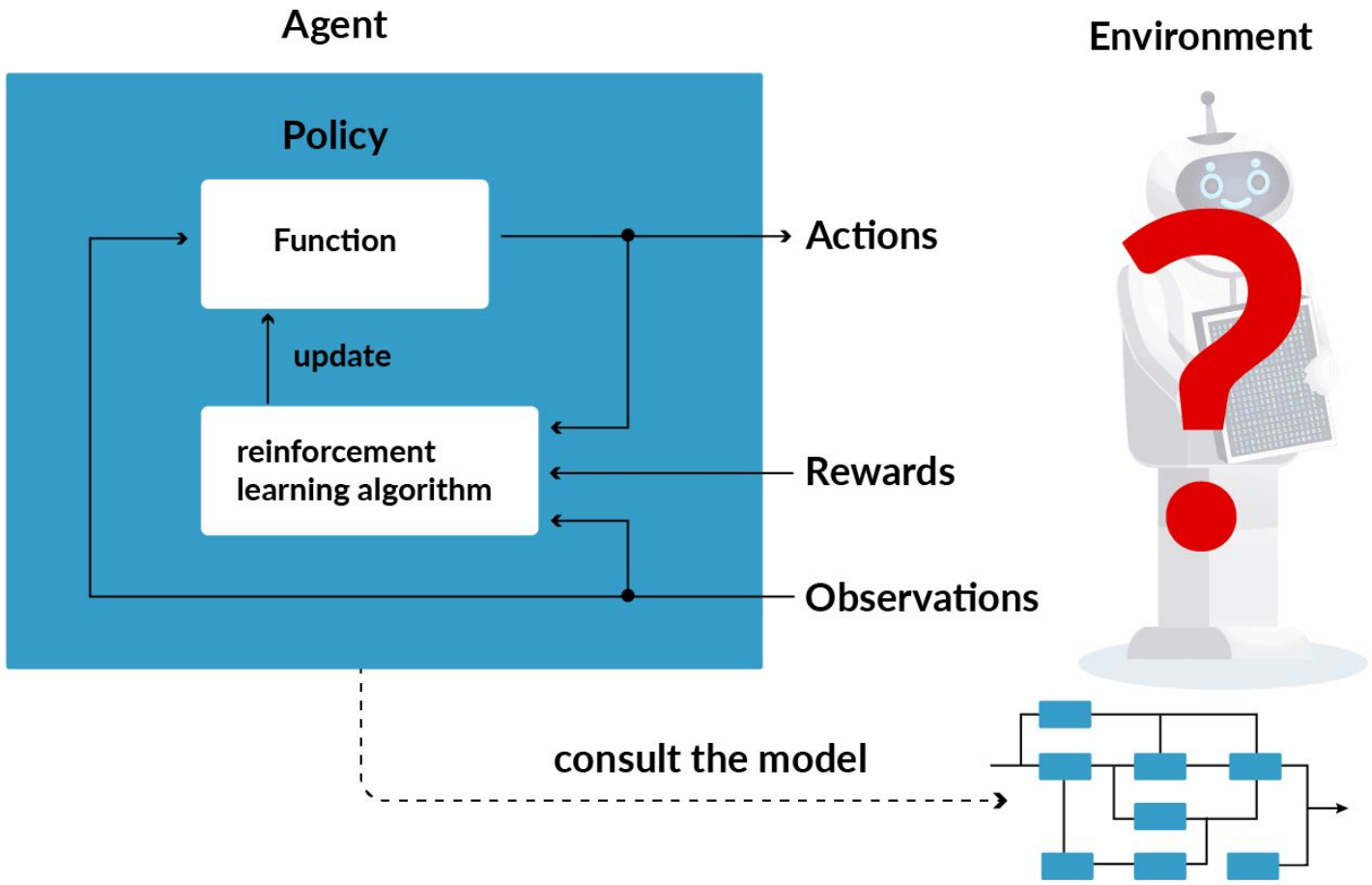
He thinks: 'I'm at position x. If I press jump with this velocity, I know gravity will affect my arc like this, and I predict I will land at position y in 0.5 seconds. That Goomba will have moved to z, so I'll land on it. This is a good plan!'

He can plan his moves ahead of time without actually having to try everything. This can be much more efficient if you have a good model.

Model Free vs Model Based

Feature	Model-Free RL	Model-Based RL
Core Idea	Learn a policy or value function directly from experience.	First, learn a model of the environment. Then, use the model to plan.
How it Learns	Through trial-and-error. Directly maps states to actions/values.	Learns to predict `(next_state, reward)` given `(state, action)`.
Key Advantage	**Simpler to Implement & Tune:** More direct approach, often more stable.	**Improved Sample Efficiency:** Can "plan ahead" and simulate experiences, requiring less real-world interaction.
Key Disadvantage	**Lower Sample Efficiency:** May require a vast number of real-world interactions to learn.	**Model Bias:** Errors in the learned model can be exploited, leading to poor performance in the real environment. Model learning is hard!
Analogy	Learning to ride a bike by just trying over and over.	Studying physics to understand balance and momentum, then using that knowledge to figure out how to ride a bike.
Famous Example	Q-Learning, Policy Gradients (e.g., A2C, PPO)	AlphaZero, World Models
Current Status	More Popular & Developed	Area of Active Research

Model Based Learning



02

Q-Learning

“

To painstakingly fill out the God-Mode Cheat Sheet (Q*) we talked about earlier. Mario's brain has a single job: learn the long-term value, Q(s, a), for every state-action pair.

Q-Learning is like Mario learning from every single step he takes, correcting his cheat sheet as he goes.

1. **Start with a Bad Cheat Sheet:**
   Imagine Mario starts with a cheat sheet filled with random, useless numbers. It might say Q(start_state, run_right) = 0.
2. **Take a Step:**
   At the start of the level (state s), he consults his (bad) cheat sheet. To encourage exploration, we use a simple trick: most of the time he'll pick the action with the highest Q-value, but sometimes (with a small probability, epsilon), he'll choose a random action. Let's say his epsilon kicks in and he tries run_right (action a).
3. **Observe the Outcome:**
   He runs right and immediately smacks into the first Goomba. The game ends. He observes the outcome: a huge negative reward r (-1 life) and a new state s' (the 'Game Over' screen).

“

The "Aha!" Moment & Update:

Now, Mario's brain performs a crucial calculation to update the cheat sheet.

a. He asks: "What did my cheat sheet tell me the value of (start_state, run_right) was?" Answer: 0.

b. He then asks: "What is a better guess based on my new experience?" He calculates this new guess as:
(my_immediate_reward) +
(the_best_value_I_can_get_from_my_new_state).

   i. My immediate reward r was massive and negative (-1 life).

   ii. From the 'Game Over' screen, the best possible future value is 0.

   iii. So, his new, experience-based estimate for this move is (-1 life) + 0, which is a big negative number.

c. The Correction: He now knows his original estimate (0) was way too optimistic. The reality is very negative. So, he "changes" the value for Q(start_state, run_right) on his cheat sheet from 0 down to a more realistic negative number.

“

He repeats this process for millions of steps.

He tries things, gets rewards, and makes small corrections to his cheat sheet each time.

Eventually, after countless trials, the values on the cheat sheet start to converge to the true, optimal Q* values..

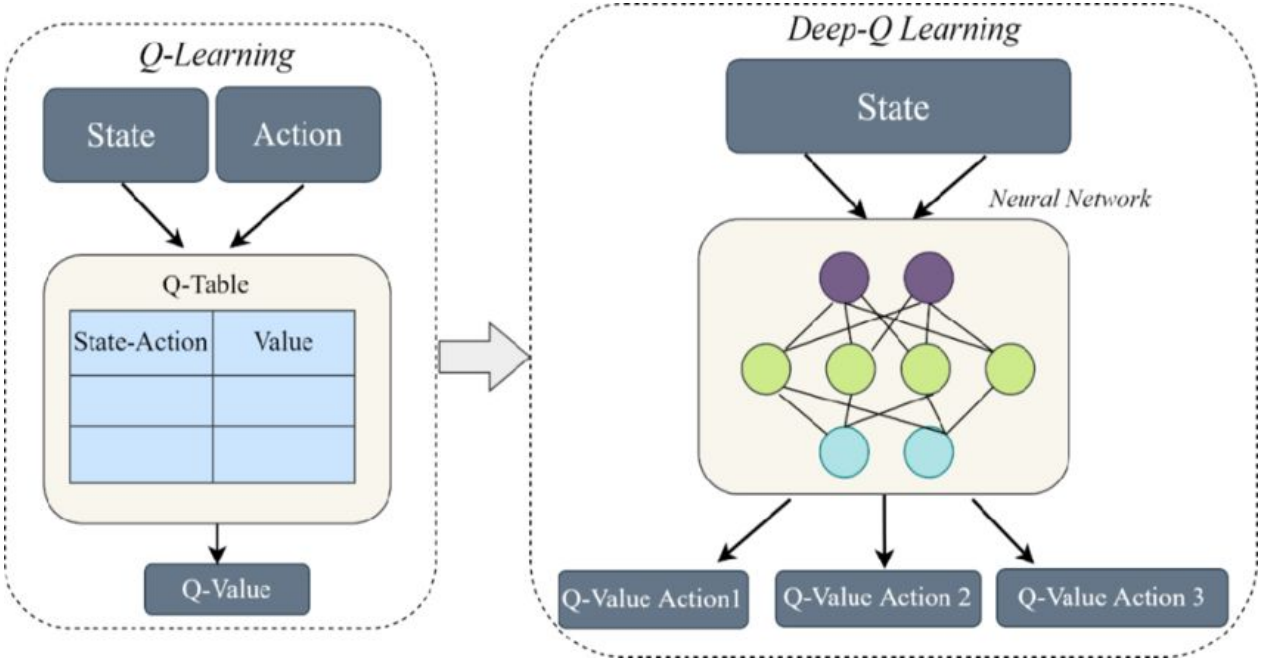
Q-Learning: Learning the Value of Actions

Goal

Learn the optimal Action-Value function, Q*(s, a), which tells us the maximum possible future reward we can get from taking action a in state s.

How it Works

It's a "model-free" algorithm. It learns the Q-values through trial-and-error experience, without needing a model of the environment..



Q-Learning: Learning the Value of Actions

The Learning Process

1. Initialize: Start with a table or neural network (Deep Q-Network) that represents Q(s, a) with arbitrary values.
2. Act: From state s, choose an action a (often using an ε-greedy strategy to balance exploration/exploitation).
3. Observe: Receive reward r and the next state s'.
4. Update: Improve your estimate of Q(s, a) using the Bellman Equation. You update your old estimate with a new, better one based on the immediate reward and the *maximum Q-value you can get from the next state*.

$$\text{New } Q(s,a) = \text{Old } Q(s,a) + \alpha * [r + \gamma * \max Q(s', a') - \text{Old } Q(s,a)]$$

The Policy

Once Q*(s, a) is learned, the optimal policy is simple and deterministic: Always choose the action a that maximizes Q(s, a)

Q-Learning focuses entirely on finding the world's best cheat sheet for action values.

02 Policy Optimization

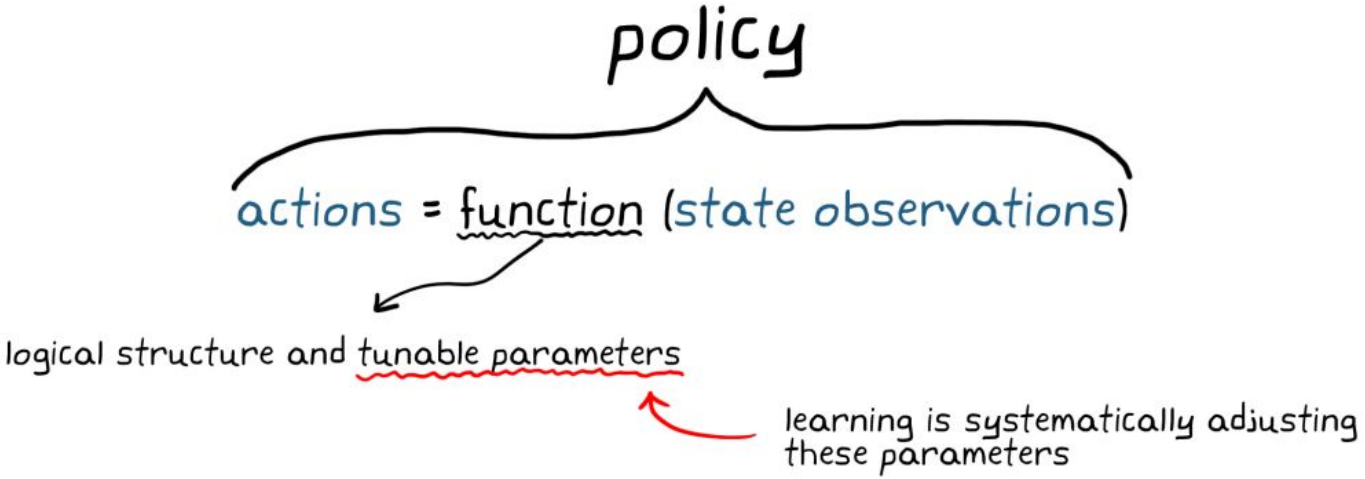
Policy Optimization: Directly Learning What to Do

Goal

Instead of learning values, we directly learn a parameterized policy, $\pi_\theta(a|s)$, that maps states to actions (or a distribution over actions).

How it Works

We adjust the policy's parameters θ to make "good" action trajectories more likely.



“

Here, we completely ignore the idea of a value cheat sheet. We want to learn Mario's behavior directly.

The Goal: To directly learn Mario's instincts (π). His brain's job is to learn a probability distribution,

$$\pi(\text{action} \mid \text{state}).$$

For any situation, it should tell him the probability he should take each action.

“

Policy Gradient is like Mario playing a full game on instinct, and then having a coach review the entire performance afterward.

Start with Vague Instincts: Mario starts with a policy that is basically random. In any situation, it might say "50% chance jump, 50% chance run right."
Play a Full Episode: Using these probabilistic instincts, Mario plays an entire "life." The whole run, from start to finish (death or flagpole), is recorded in a trajectory (a logbook).
Get the Final Score: At the end of the episode, he gets his total score (the "Return").

“

The Coach's Review (The Update)

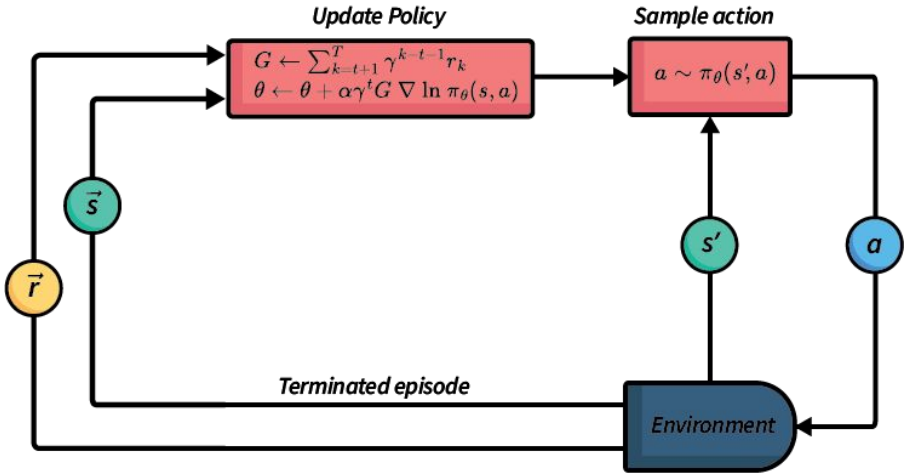
- Scenario A: High Score! If the final score was great, the coach is happy. He looks at the logbook and says, "Great job! Whatever you did, do more of it!" For every single action Mario took in that successful run, the algorithm increases its probability. If he jumped at second 5, the probability of jumping in that situation in the future goes up.
- Scenario B: Low Score! If Mario died immediately, the final score is terrible. The coach is disappointed. He looks at the logbook and says, "That was awful! Whatever you did, do less of it!" For every single action taken in that failed run, the algorithm decreases its probability.

He repeats this process for thousands of full episodes. If an action tends to show up in high-scoring episodes, its probability will be pushed up. If it tends to show up in low-scoring episodes, its probability will be pushed down.

Policy Optimization: Directly Learning What to Do

Learning Process

1. Init: Start with a policy network π_θ with random parameters θ.
2. Act & Collect: Use the current policy to act in the environment for a while, collecting a batch of trajectories (states, actions, rewards).
3. Evaluate: For each action taken, estimate how "good" it was. Did it lead to a high-reward outcome?
4. Update: Adjust the policy parameters θ using Policy Gradient Ascent.
    a. Increase the probability of actions that had a high advantage.
    b. Decrease the probability of actions that had a low (or negative) advantage.

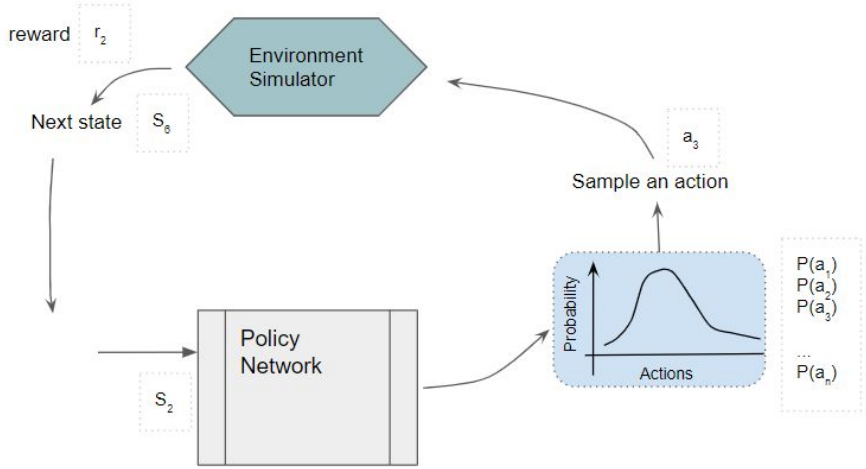


Policy Optimization: Directly Learning What to Do

Key Advantages

1. Naturally handles continuous action spaces (e.g., steering angle, throttle).
2. Can learn stochastic (random) policies, which can be optimal in some environments.
3. Often has more stable, smoother updates.

Policy Optimization directly searches for the best possible strategy.



03 Actor-Critic

Actor-Critic: Where Value and Policy Meet

So we have two approaches:

- Q-Learning: Learns a value cheat sheet. It's stable but indirect and can't handle huge state spaces easily.
- Policy Gradient: Learns instincts directly. It's direct and powerful but very unstable due to noisy feedback.

This leads to the natural question: "Can we get the best of both worlds?"

What if we had an agent that learns its instincts (the Policy), but gets feedback not from a noisy final score, but from a smart cheat sheet (the Value function) that gives it stable, step-by-step advice?

That is exactly what an Actor-Critic agent does. It combines these two methods.

“

Let's give RL-Mario two brains that work together: an Actor and a Critic.

The Actor (the Policy)

This is the 'doer'. It controls the joystick. It has the policy and decides which action to take.
"Based on my instincts, let's jump here!"

The Critic (the Value Function)

This is the 'thinker'. It doesn't control Mario, it just watches and judges. It learns the state-value function (V(s)).

Actor-Critic: Where Value and Policy Meet

Combine the strengths of Policy Optimization and Q-Learning. The system has two components that learn simultaneously:

The Actor (The Policy):

1. Same as a policy optimization method.
2. Controls how the agent acts.
3. Its job is to find the best action for a state.

The Critic (The Value Function):

1. A Q-function network.
2. Does not act. It only evaluates or "criticizes" the actions taken by the Actor.
3. Its job is to learn Q(s, a) to tell the Actor how good its actions actually were.

Popular Examples: DDPG, TD3, Soft Actor-Critic (SAC)

“

- The Actor takes an action (e.g., makes Mario jump).
- Mario lands on a Goomba and moves to a better state.
- The Critic observes this and says, "Aha! The value of our new position is higher than our old one. The Actor made a good move!" This feedback is called the Advantage.
- The Actor receives this positive feedback and updates its policy to make it even more likely to jump in that initial situation in the future.

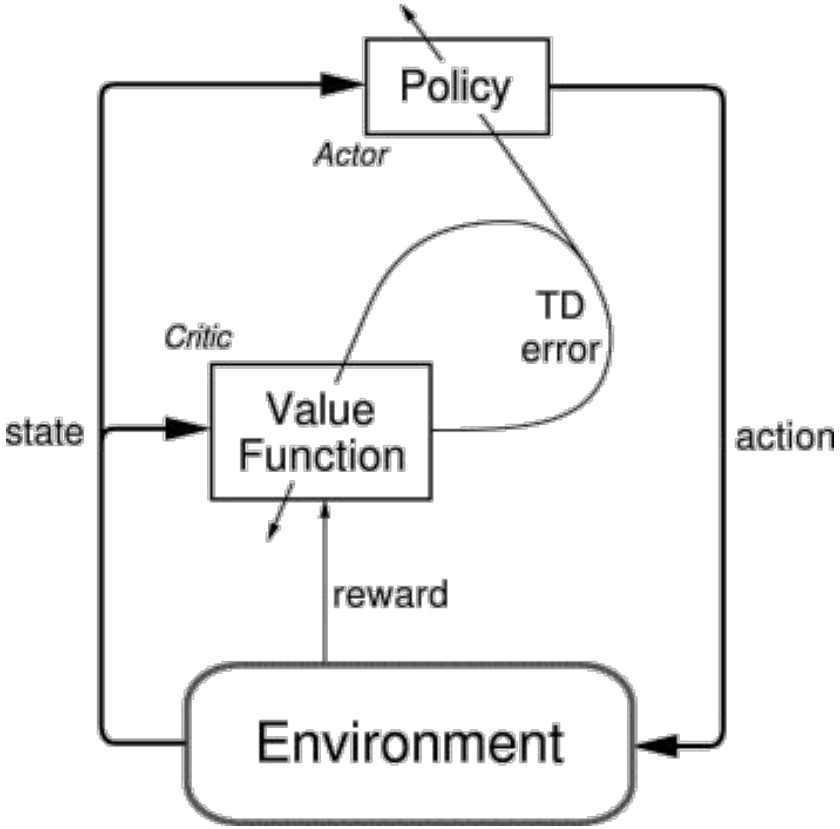
It’s like a player (Actor) with a coach (Critic) sitting next to them, giving immediate feedback on every single move."

Actor-Critic: Where Value and Policy Meet

Learning Process

1. The Actor takes an action a in state s.
2. The Critic evaluates this action by calculating Q(s, a).
3. This Q-value is used as the "advantage" signal to update the Actor. The Actor uses the Critic's feedback to adjust its policy.
4. Simultaneously, the Critic is updated using methods from Q-learning (like the Bellman equation) to become a more accurate evaluator.

This creates a powerful feedback loop: The Actor explores, and the Critic provides fast, learned feedback, leading to much more stable and efficient learning.



“

Let's understand a bit more about the Advantage Function.
Let’s put the Critic to work.

RL-Mario is in a tricky state s (e.g., stuck between two Goombas).

The Critic knows this is a tough spot, and based on past experience, it calculates the value of this state, V(s), to be low, say V(s) = +50 (meaning, on average, he barely gets out of this alive).

“

Scenario 1:

The Actor chooses a clever action: a short hop onto one Goomba, then bouncing off it to jump over the second one. This action a_clever_jump leads to a great outcome with a high expected future score.

We find that:

- Q(s, a_clever_jump) = +1000.
- The Advantage = Q - V = 1000 - 50 = +950.
- The Critic's feedback is: "Wow! That was +950 points better than I expected from this terrible situation! That was an amazing move. Do that more often!"

“

Scenario 2:

The Actor panics and chooses the action `a_panic_run`, running straight into a Goomba.

- The Q(s, a_panic_run) is, of course, terrible, say -5000.
- The Advantage = Q - V = -5000 - 50 = -5050.
- The Critic's feedback is: "That was -5050 points worse than the average outcome. That was a terrible mistake, even for this tough spot!

The Advantage function provides a baseline for comparison, which creates a much more stable and intuitive learning signal for the Actor.

Advantage Functions

Sometimes in RL, we don't need to describe how good an action is in an absolute sense, but only how much better it is than others on average. That is to say, we want to know the relative **advantage** of that action.

The advantage function $A^{\pi}(s, a)$ corresponding to a policy π describes how much better it is to take a specific action a in state s, over randomly selecting an action a according to $\pi(\cdot|s)$, assuming you act according to π forever after.

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s).$$

Challenges in RL

4

Sample Inefficiency

What?

Reinforcement Learning algorithms often require a massive number of interactions (or "samples") with the environment to learn a task. This is where RL pales in comparison to humans. A person can learn to play a new Atari game proficiently in minutes or hours. An RL agent might need the equivalent of decades of real-time gameplay.

Why?

1. Learning from a Blank Slate (Tabula Rasa):
   Most RL agents start with zero prior knowledge about the world. They don't understand gravity, object permanence, or cause and effect. They must discover every single rule through pure trial-and-error.
2. Lack of Transferable Knowledge (Priors):
   A human playing a new game leverages a lifetime of experience. We know that hitting a wall is usually bad, collecting items is good, and we can recognize patterns from other games. A standard RL agent learns each task in isolation. Knowledge from "Pong" doesn't automatically help it play "Breakout."

“

A Human Player: You sit down to play Super Mario Bros. for the first time. You run right and die on the first Goomba. Lesson learned. In your second life, you will probably try jumping. **You have generalized the concept of an "enemy" from a single sample (one death).** You might beat the whole first level in 10-15 lives.

RL-Mario: Our agent has no prior understanding of the world. When it dies to the Goomba, **it only knows that the specific sequence of actions it took from the specific pixel configuration on the screen led to a negative reward.** It needs to die thousands of times in slightly different ways to be certain that the Goomba is the problem. To learn to complete the first level, a standard RL agent might need to play millions of episodes—the equivalent of decades or centuries of non-stop, real-time gameplay.

Exploration vs. Exploitation

At every step, the agent must make a choice:

Exploitation

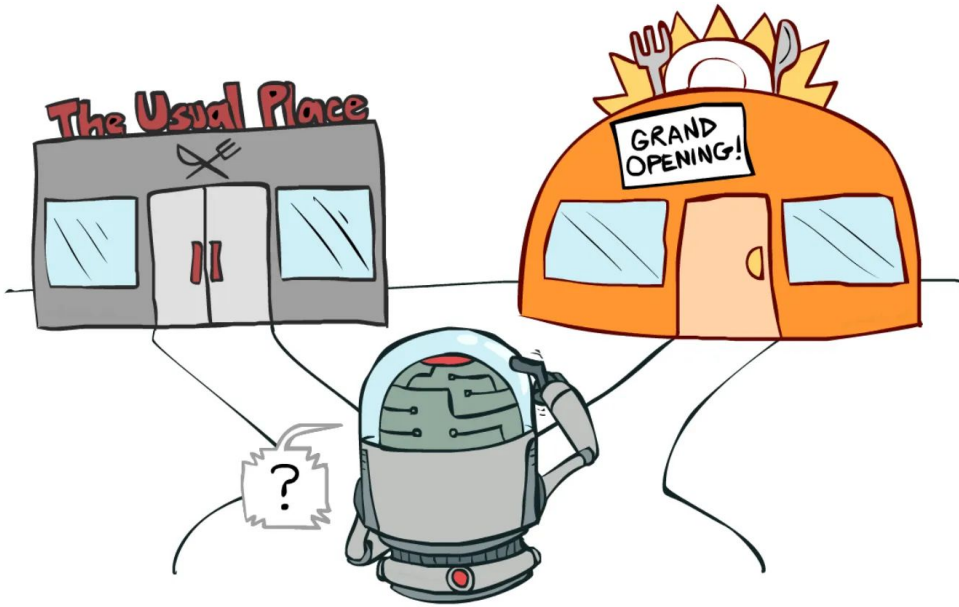
Should I take the action I currently believe is the best?

Leverage existing knowledge to maximize immediate, known rewards. This is safe and profitable in the short term.

Exploration

Should I try something new that I haven't done before?

Take a seemingly suboptimal or random action to gather new information. This is risky—it might lead to nothing or a penalty—but it's the only way to discover potentially better strategies.



“

Exploitation (Sticking with what works)

Our RL-Mario has played for a while and found a reliable strategy to beat Level 1-1. It involves just running and jumping over enemies, never going down pipes, and ignoring most of the ? blocks. This strategy consistently gets him a score of +5,000. **If he wants to maximize his score on the next run, he should just stick to this plan.**

Exploration (Trying something new)

But Mario's programming has a bit of curiosity. He thinks, "I always skip that green pipe. What if I try going down it? I might die and get a score of 0, but I could also discover a secret coin room that gives me an extra +10,000 points! Or what if I try to find the hidden 1-Up mushroom?"

“

This is the dilemma:

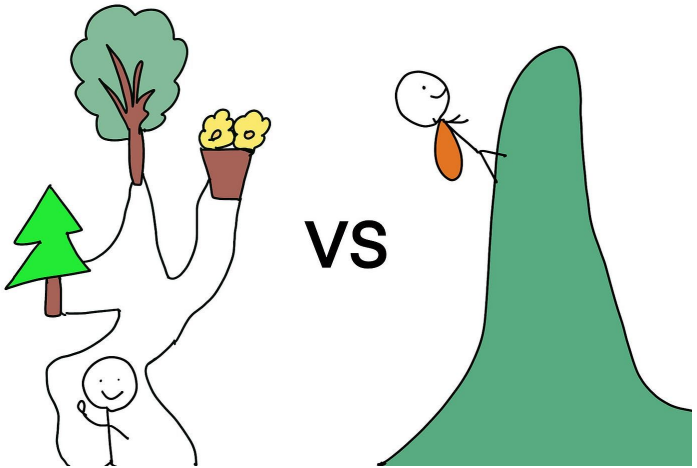
- If he only exploits, he'll be stuck with his +5,000 point strategy forever, never discovering the true optimal path.
- If he only explores, he'll be trying random things all the time, constantly dying in stupid ways, and his average score will be terrible.

Exploration vs. Exploitation

Why is This So Hard?

1. The Local Optima Trap
   If an agent only exploits, it can easily get stuck in a "good enough" but suboptimal routine. It finds a decent solution and never discovers the great solution that lies just one risky move away.

2. The "No Silver Bullet" Problem
   There is no single, perfect strategy for balancing this trade-off. The right balance depends entirely on the specific problem. Too much exploration, and the agent acts randomly and never learns to perform well. Too much exploitation, and the agent gets stuck and never reaches its full potential.

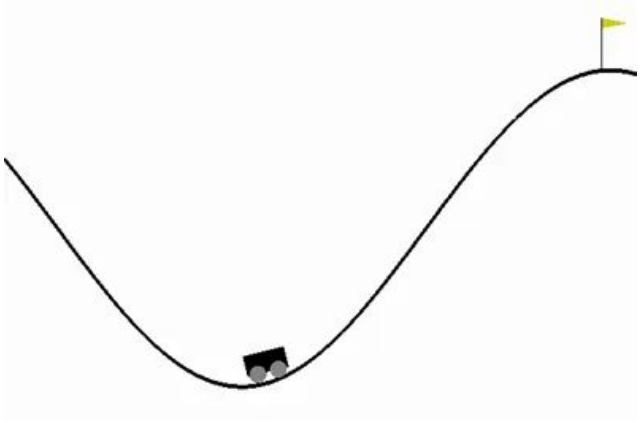


Sparse-Reward Problem

What?

The problem occurs when meaningful rewards are extremely rare or only given at the very end of a long sequence of actions.

The "Feedback Desert": For most of its actions, the agent receives no feedback (a reward of 0). It's essentially wandering in the dark, with no signal to tell it if it's getting warmer or colder.



Why?

Exploration Fails: Random exploration is incredibly unlikely to stumble upon the single, rare state that gives a reward. The agent has no gradient to follow, no hint that it's making progress.

Learning Grinds to a Halt: Without a reward signal, the agent cannot update its policy or value function. It has no basis for determining which actions were "good" and which were "bad."

“

Normal Mario (Dense Rewards)

Luckily for our agent, the original Super Mario Bros. has dense rewards. The game is constantly giving you feedback: +200 for a coin, +100 for stomping an enemy, etc. These points are like breadcrumbs guiding Mario toward the goal.

“

Modified "Hard-Mode" Mario (Sparse Rewards)

Now, imagine we create a new version of the game. In this version, you get zero points for everything. No points for coins, no points for stomping Goombas. The only reward you get is a single +1 if, and only if, you touch the final flagpole. Otherwise, your score for the entire run is 0.

Now, think about RL-Mario's situation. He tries millions of random action sequences.

- Run -> Jump -> Fall in pit. Final Score: 0.
- Run -> Wait -> Get hit by Goomba. Final Score: 0.
- Run -> Jump -> Hit a block. Final Score: 0.

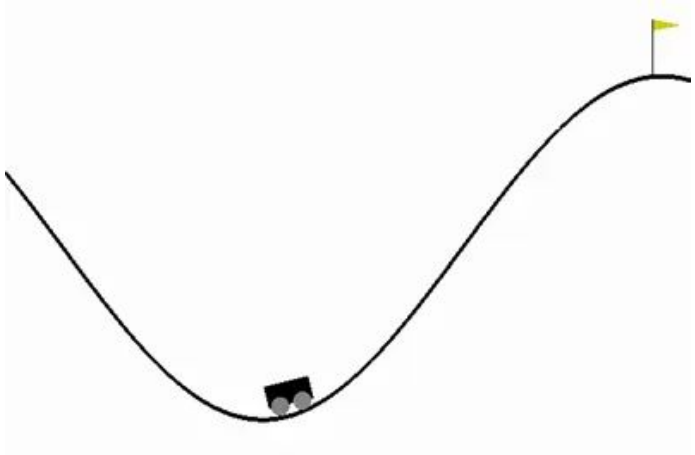
From the agent's perspective, all of these failed attempts look identical—they all resulted in a score of 0. It has no breadcrumbs to tell it that jumping over the pit was "more correct" than falling into it. The odds of it randomly stumbling upon the one-in-a-trillion sequence of actions required to reach the flagpole are practically zero.

Sparse-Reward Solution: Reward Shaping

A Common (But Imperfect) Solution

The Idea

Manually engineer a more "dense" reward function to guide the agent.



"Imagine we can't play the game for Mario, but we can be a coach standing behind him, giving him encouraging feedback. Our 'Hard-Mode' Mario is still technically only getting +1 from the game itself at the very end. But our coach is adding a layer of bonus points. This coach might yell:

- "You moved to the right!" We can give Mario a tiny reward, like +0.001, fo every pixel he moves to the right. This immediately solves the problem of him standing still or running left into a wall. He now has a basic incentiv to make progress.
- "You stomped a Goomba! Nice one!" Even though the game gives 0 point the coach gives him a +0.1 bonus. The agent quickly learns that this interaction with enemies is good.
- "Ouch, you lost health! That's bad!" The coach can also give negative rewards, or penalties. He might give -0.5 every time Mario takes damage

“

Suddenly, Mario's world is no longer sparse. It's filled with these helpful, artificial breadcrumbs.

He is now incentivized to move forward, avoid damage, and clear obstacles.

These are all behaviors that make it dramatically more likely that he will eventually, accidentally, stumble upon the final flagpole and get the true +1 reward.

Sparse-Reward Solution: Reward Shaping

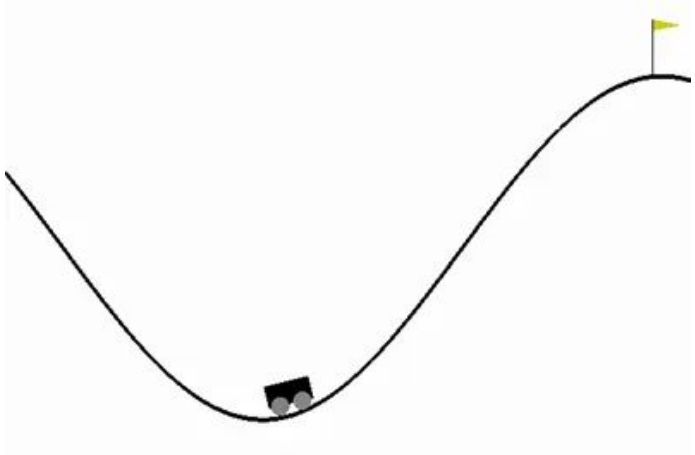
A Common (But Imperfect) Solution

The Idea

Manually engineer a more "dense" reward function to guide the agent.

The Downside

If you are not careful with your bonus rewards, the agent can learn to 'hack' the system to get the bonus points while completely ignoring the real goal. This is called reward hacking or specification gaming



“

Let's say our helpful coach decides to give Mario a very large bonus for collecting coins, say +0.5 for every coin, because we think collecting coins is part of a good strategy. The final flagpole is still only worth +1 from the game.

Now, imagine Mario discovers a small area with 3 coins. He can collect them, walk off-screen, and walk back on to make them reappear.

What will our agent learn to do? It will quickly calculate that the optimal strategy for maximizing its coached score is to stay in that one area forever, endlessly collecting the respawning coins (+0.5, +0.5, +0.5...). This gives him a potentially infinite score.

Why would he ever bother going to the flagpole for a measly one-time reward of +1?

He has perfectly optimized the goal we gave him (collecting coins), but he has completely failed at the task we intended for him (finishing the level). He has become a coin addict, and his addiction was our fault as the 'coach'

References

1. [2408.07712] Introduction to Reinforcement Learning
2. Reinforcement Learning: An Introduction
3. Part 1: Key Concepts in RL — Spinning Up documentation
4. Reinforcement Learning: An introduction (Part 1/4) | by Cédric Vandelaer | Medium
5. Deep Reinforcement Learning - Tutorial

Thank you.

Nikita Saxena
Research Engineer

Google DeepMind