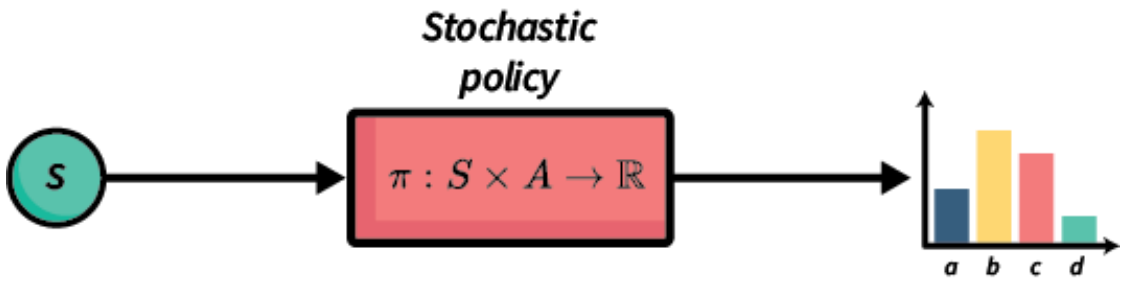
Policy

- The strategy to take actions base on the current states
  - **Deterministic policy**: always selects the same action for a state
  - **Stochastic policy**: selects actions based on probabilities
- The policy is what the agent learns and improves over time

$$\pi(a|s) \stackrel{\text{def}}{=} P[A_t = a|S_t = s]$$



Usually start from a random policy → find the optimal policy through learning $\pi_*(a|s)$

Value functions

Evaluates the expected return (i.e., cumulative future rewards) from a given state or state-action pair, under a particular policy π

- **State-value function $v_\pi(s)$**
- **Action-value function $q_\pi(s, a)$**

For example:

At each square of the maze, the mouse may think: *Hemm, this spot is worth something --- not because there's cheese there, but because it leads to cheese.*

- It evaluates how good is the current situation.
- It helps decide where to go next.

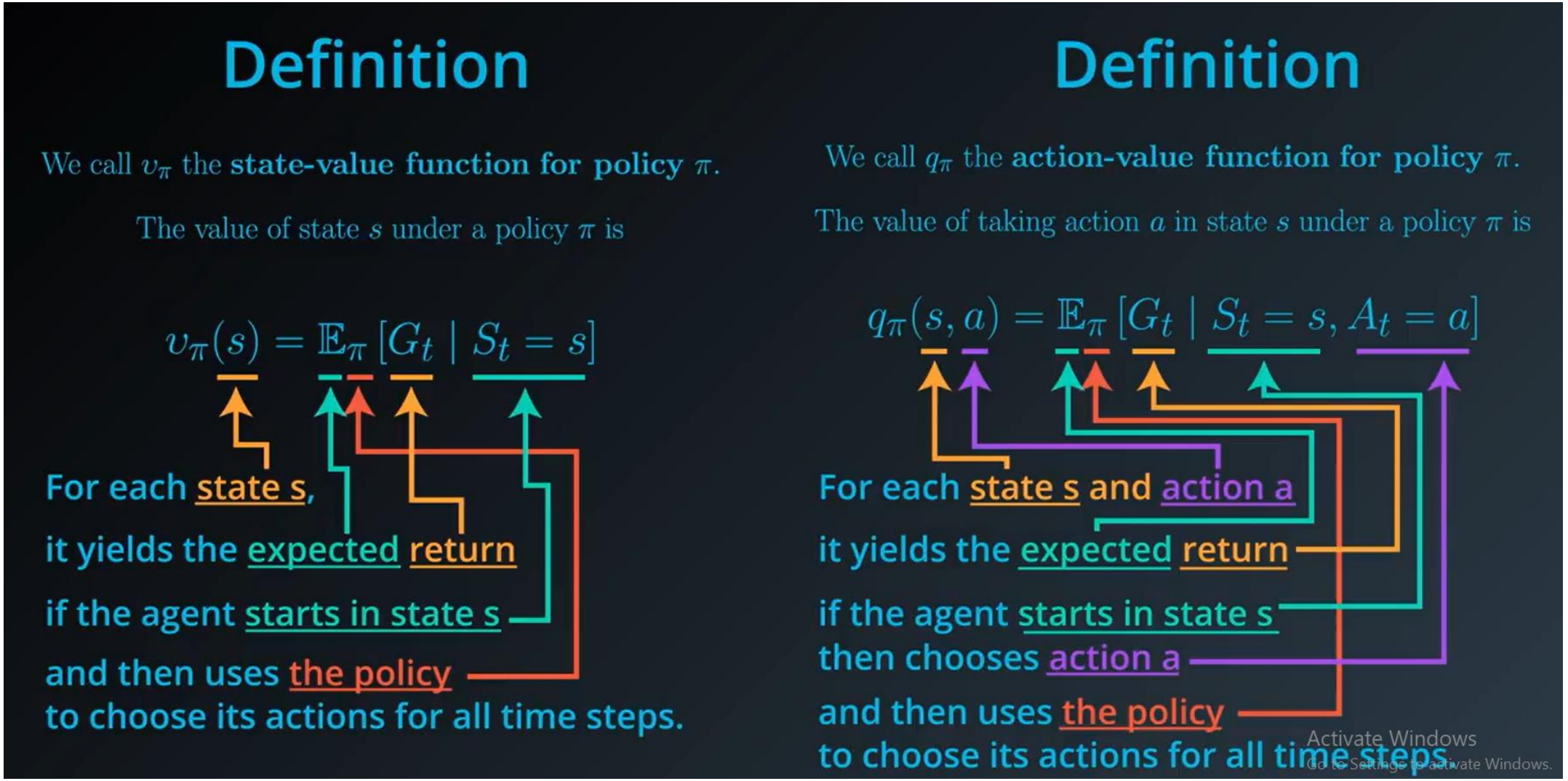
•Value functions help estimate long-term benefits.

•Example: In a chess game, choosing a move leads to different possible future states. Some states look more promising than others — the value function estimates how good each state is.

•Unlike reward, which is given only at the end (win/loss), the value function reflects the *potential* to win from a given state.

Two possible attacking moves for the black pawn: the white rook or the white bishop. They have different consequences which may lead to very different winning probability

Value functions



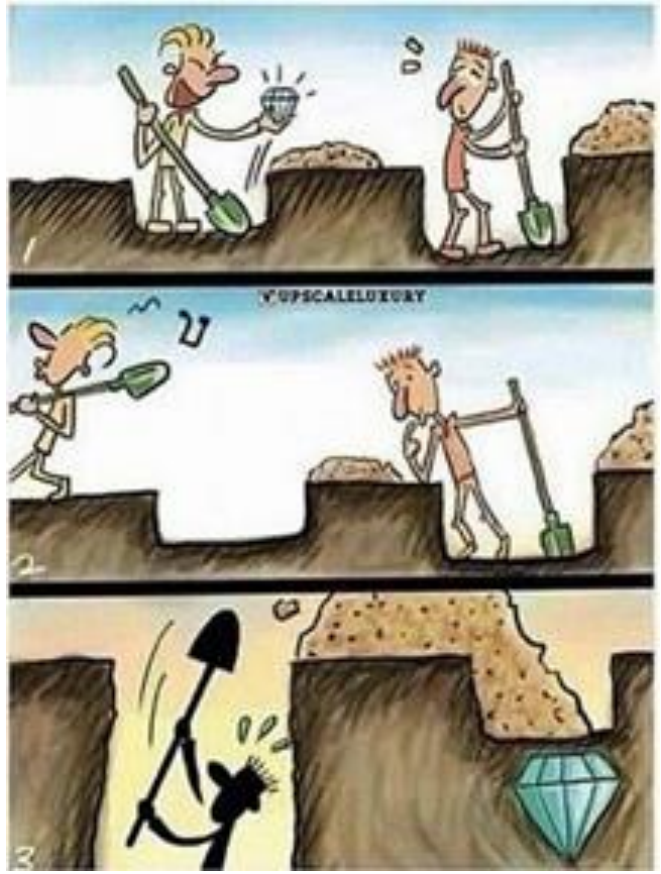
Exploration vs. Exploitation

Exploration: trying new actions to discover their effects

Exploitation: using known information to maximize reward

The agent must balance exploration and exploitation

Too much exploitation can miss better strategies; too much exploration slows learning



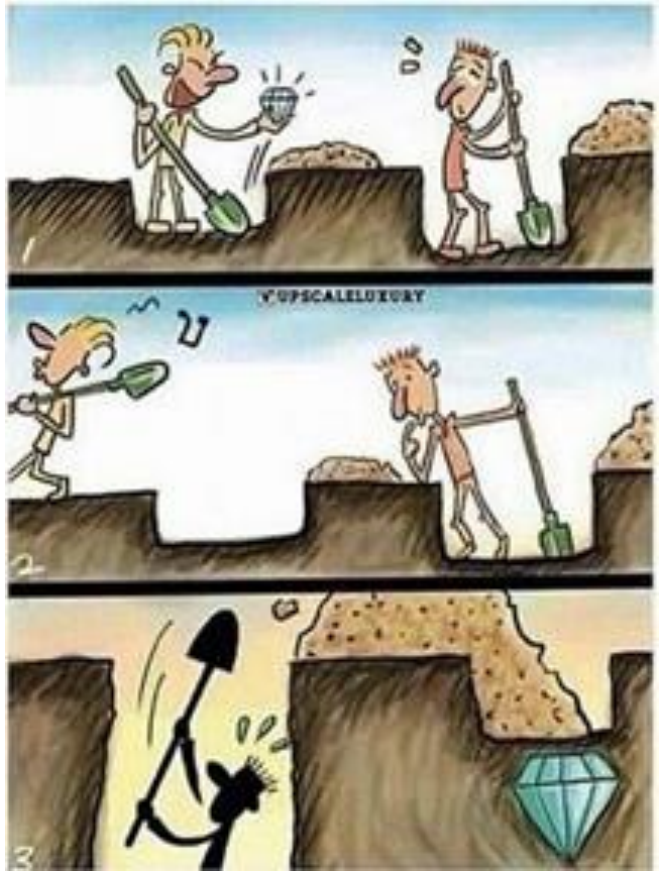
Exploration vs. Exploitation

If you only dig at the same spot where your friend found a diamond, you might get something — but you're relying on partial information. That spot worked once, but you don’t know if there are bigger or more diamonds elsewhere.

To find better rewards, you need to take the risk of exploring new areas. This is the essence of exploration vs. exploitation:

- **Exploitation** is digging where you already know there’s some reward.

- **Exploration** is venturing into the unknown, hoping to discover something even better



Markov Decision Process (MDP)

RL problems are typically modelled as MDPs, defined by the **transition probability**

$$P(s' \mid s, a)$$

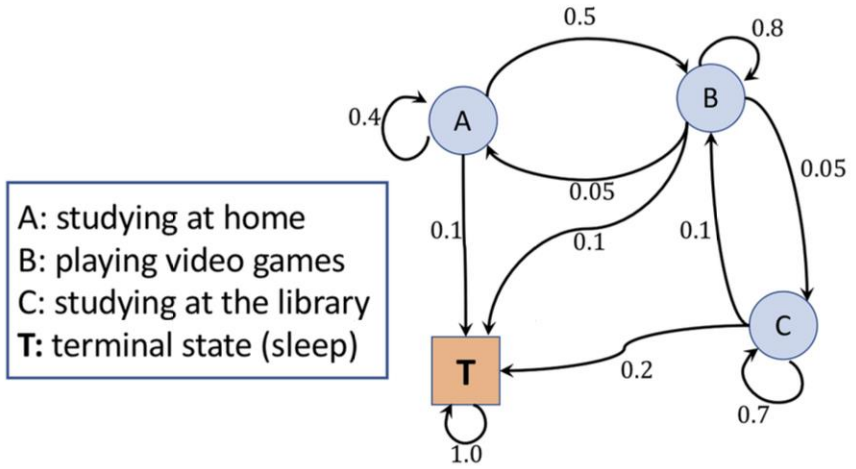
the probability of moving to a new state s', when taking action a from state s

Markov: The future depends only on the current state and action, not on the entire history

No memory on the past states/actions!

An example of Markov decision process

A student deciding between three different situations: A, B, C, near the final exam



Transition probabilities:

	A	B	C	T
A	0.4	0.5	0.0	0.1
B	0.05	0.8	0.05	0.1
C	0.0	0.1	0.7	0.2
T	0.0	0.0	0.0	1.0

MDPs: The transition probability only depends on the preceding time step

The sum of each row = 1

The decision are made every hour. After a few hours, it always ends at the **terminal state (T):** sleep
This is called an **episode.** A few examples:

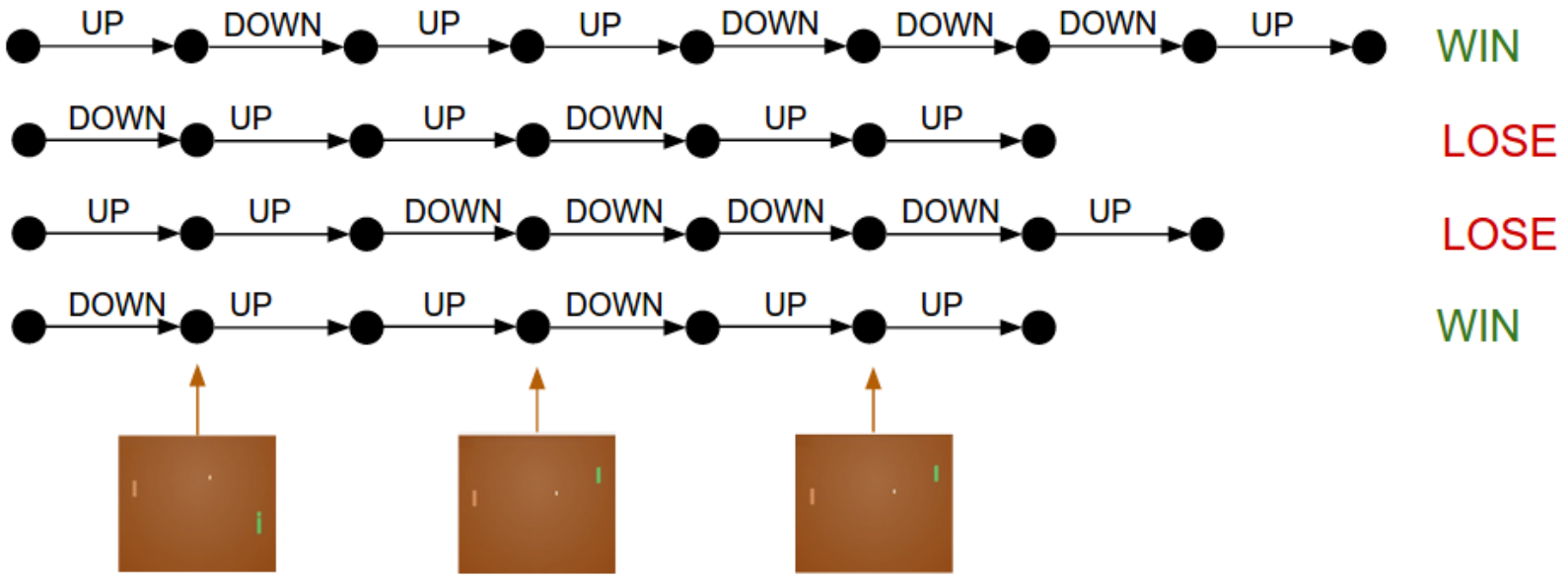
$$\text{Episode 1: } BBCCCCBAT \rightarrow \text{pass (final reward} = +1)$$
$$\text{Episode 2: } ABBBBBBBBBBT \rightarrow \text{fail (final reward} = -1)$$
$$\text{Episode 3: } BCCCCCT \rightarrow \text{pass (final reward} = +1)$$

Learning from experience

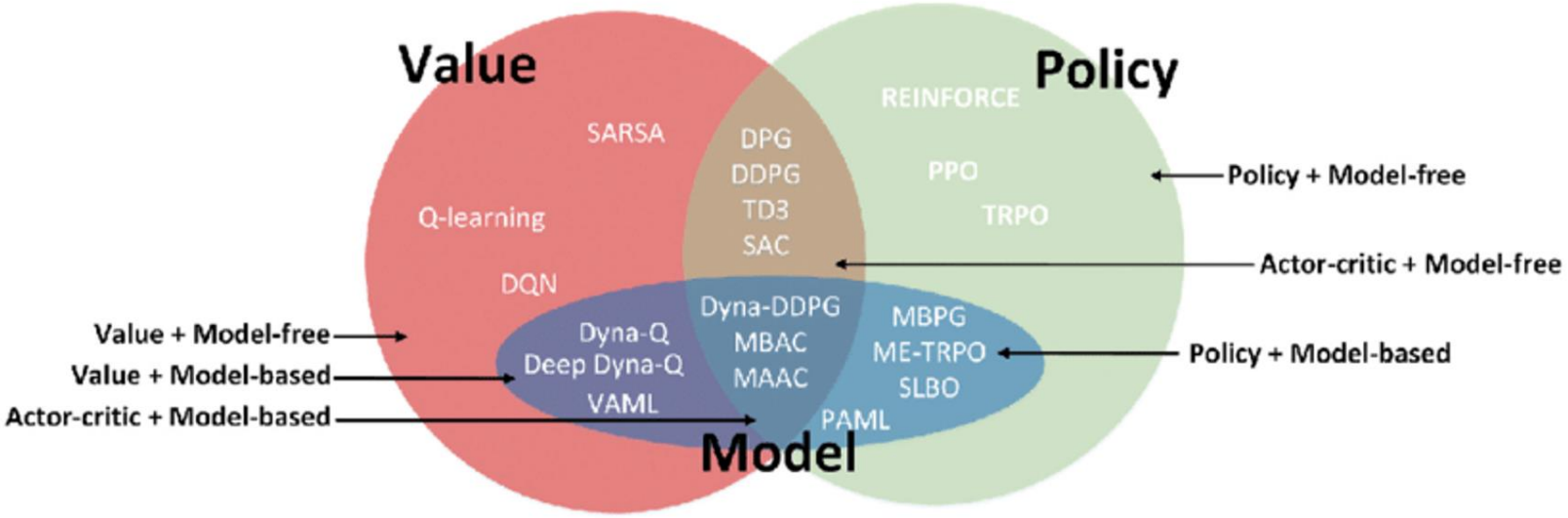
The agent learns by interacting with the environment over many episodes

Experience: a sequence of (state, action, reward, next state) tuples

The agent updates its policy based on experience



Popular approaches in reinforcement learning



For example,

- **Q-learning** is a value-based method: it learns optimal value functions
- **Policy Gradient**, such as TRPO: it learns optimal policy, works well for continuous or high-dimensional actions

Policy gradient

and

TRPO (trust-region policy optimization)

Policy Gradient Methods

- **Goal**: learn a parameterized policy $\pi_\theta(a|s)$ directly

- **How:**
  - Maximize the expected return $J(\theta) = \mathbb{E}_{\pi_\theta}[R]$
  - Use gradient ascent to update policy parameters

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot Q^{\pi_\theta}(s,a)\right]$$

Action-value function
It's weight when taking the average
So we increase log-probability of actions that lead to higher return (better than using total return R)

Why the log?
- **log-derivative trick**
- It transforms the gradient of expectations into expectations of gradients

A stochastic policy that gives the probability of choosing action a in state s, depending on parameters θ

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[R] \Rightarrow \mathbb{E}_{\pi_\theta}\left[R \cdot \nabla_\theta \log \pi_\theta(a|s)\right]$$

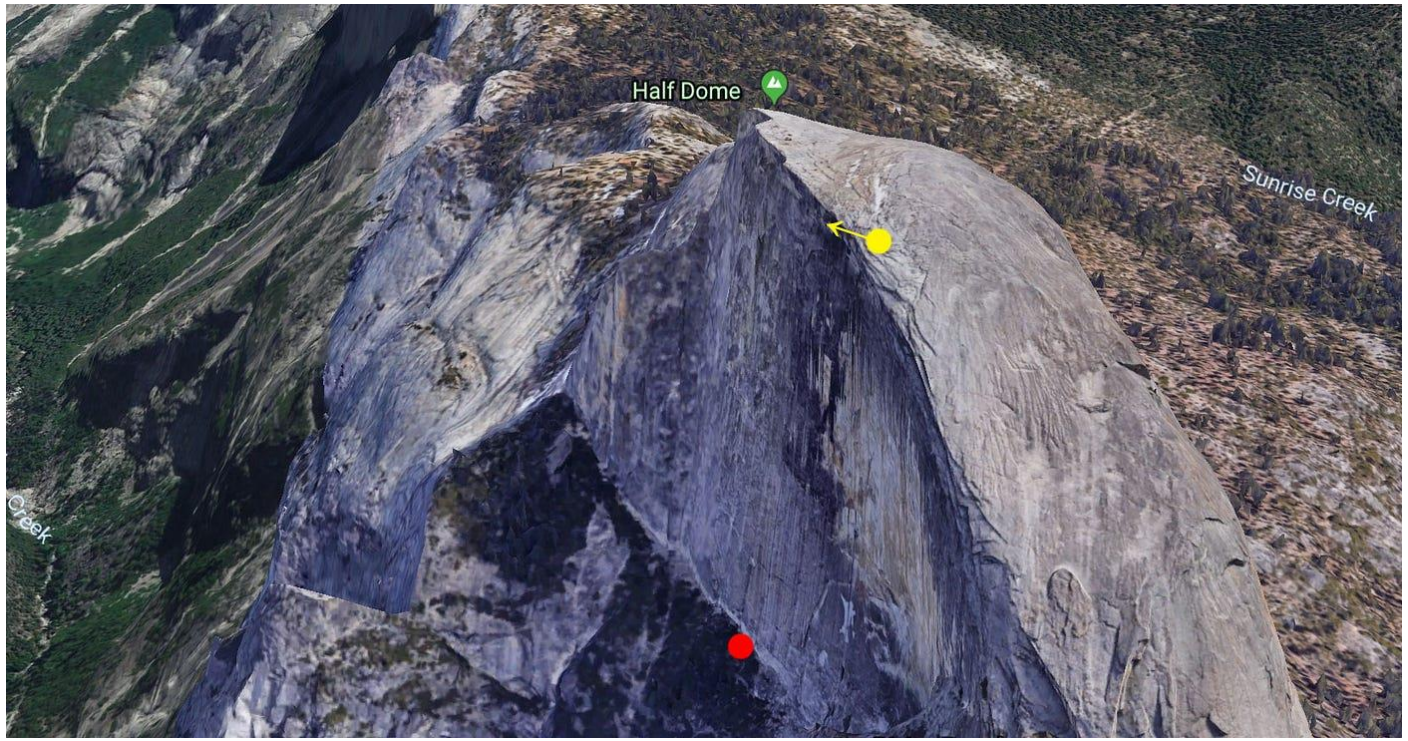
Limitations of naive Policy Gradient

- Too large a step leads to a disaster: fall off the cliff
- Too small a step leads to a very slow process

https://jonathan-hui.medium.com/rl-trust-region-policy-optimization-trpo-explained-a6ee04eeeee9

Limitations of Naive Policy Gradient

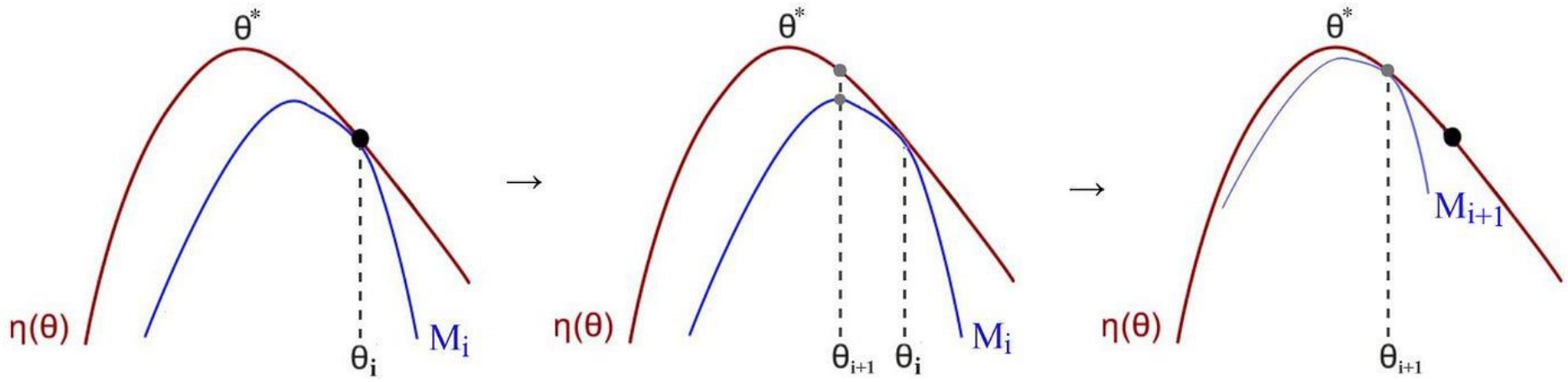
- After falling off the cliff, it **takes a long time to resume the exploration**
- It's very hard to have a **proper learning rate** in RL



- How to avoid aggressive moves?

Minorize-Maximization (MM) algorithm

- Instead of maximizing the **expected reward (red)**, let's maximize a "**lower bound function" (blue) (the minorizing function)**

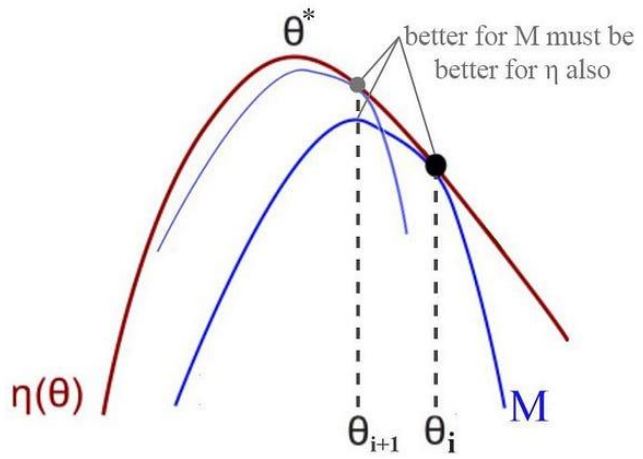


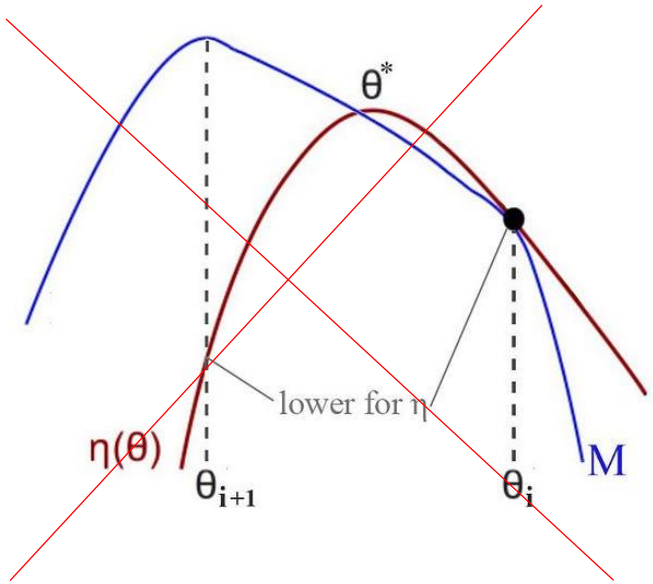
1. Start with an initial policy guess θ_i
2. Find a lower bound M that approximate the expected reward locally at the current guess
3. Locate the optimal point for M and use it as the next guess
4. Eventually, our guess will converge to the optimal policy

How to choose the "lower-bound function" M

- M should be easier to optimize than the true objective policy
- Usually a quadratic function $ax^2 + bx + c$
- …but in vector form, which is a convex function and we know how to optimize it

$$g \cdot (\theta - \theta_{\text{old}}) - \frac{\beta}{2}(\theta - \theta_{\text{old}})^T F (\theta - \theta_{\text{old}})$$





By optimizing a lower bound function approximating η locally, it guarantees policy improvement every time and lead us to the optimal policy eventually

From MM to TRPO

- We want **to improve our policy** while **staying close to the old one** (to avoid big surprises)
- TRPO uses the MM principle: optimize an alternative objective that's safer to change

Line search
(like gradient ascent)

Trust region

Trust region: A "safe zone" that limits how far the new policy can move from the old one

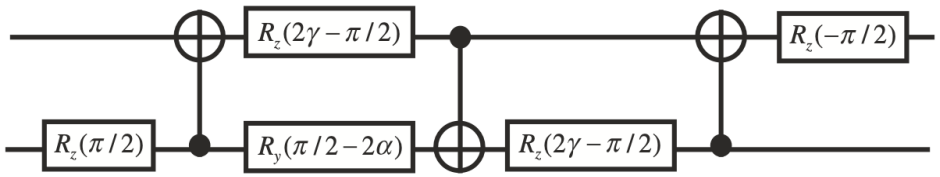
TRPO (Optimization + Safety)

- Typically add the KL divergence constraint

RL for quantum gate control

Quantum Gate Design

- **Goal:** learn high fidelity, low-leakage, fast one-qubit and two-qubit gates
- Control the system using parameterized pulses applied to qubits
- The **environment**: quantum simulator governed by time-dependent Hamiltonian
- The agent's task: discover a control strategy to implement target gates

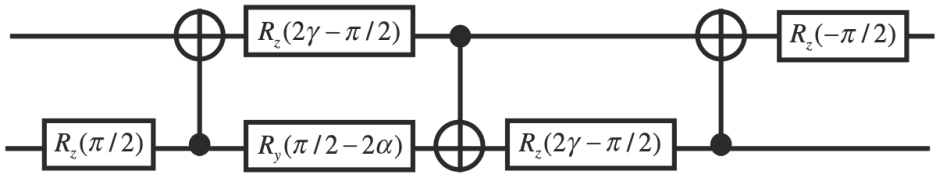


Why traditional methods struggle

- Manual pulse design is labor-intensive and not scalable
- Gradient-based optimization often fails with noisy or high-dimensional systems
- Quantum dynamics are non-linear and sensitive to noise
- RL offers adaptive learning and can discover non-intuitive solutions

Why TRPO for quantum control

- Quantum systems are **sensitive** and **high-dimensional**
- TRPO provides **stable updates** needed for such delicate environments
- The RL agent must learn **smooth control pulses** – large jumps are problematic
- TRPO helps find fast, high-fidelity, low-leakage gate sequences



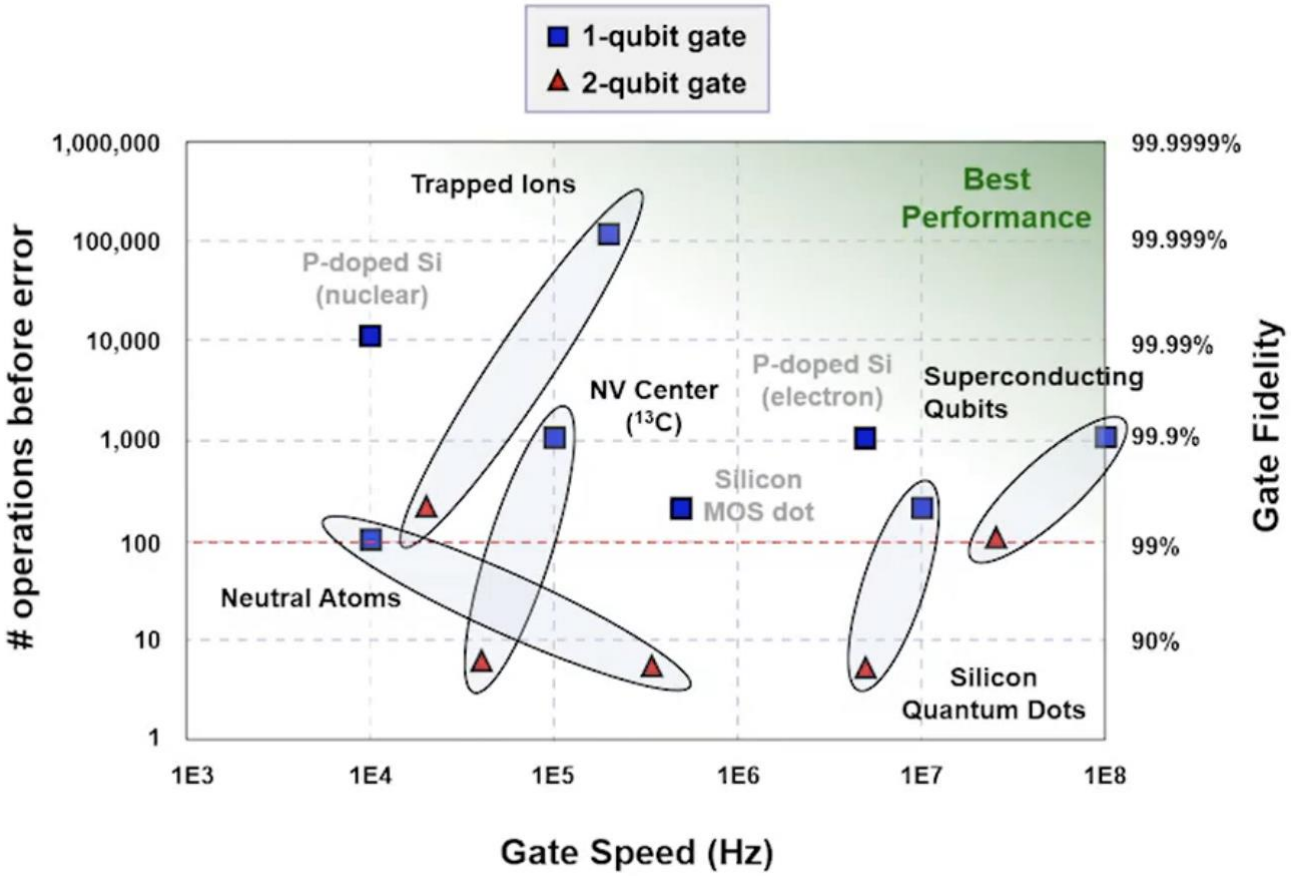
A quantum circuit
- Depth: 7
- Three two-qubit gates
- Five single qubit gates

To realize a unitary transformation

$$\mathcal{N}(a,a,\gamma) = \exp[i(a\sigma_1^x\sigma_2^x + a\sigma_1^y\sigma_2^y + \gamma\sigma_1^z\sigma_2^z)],$$

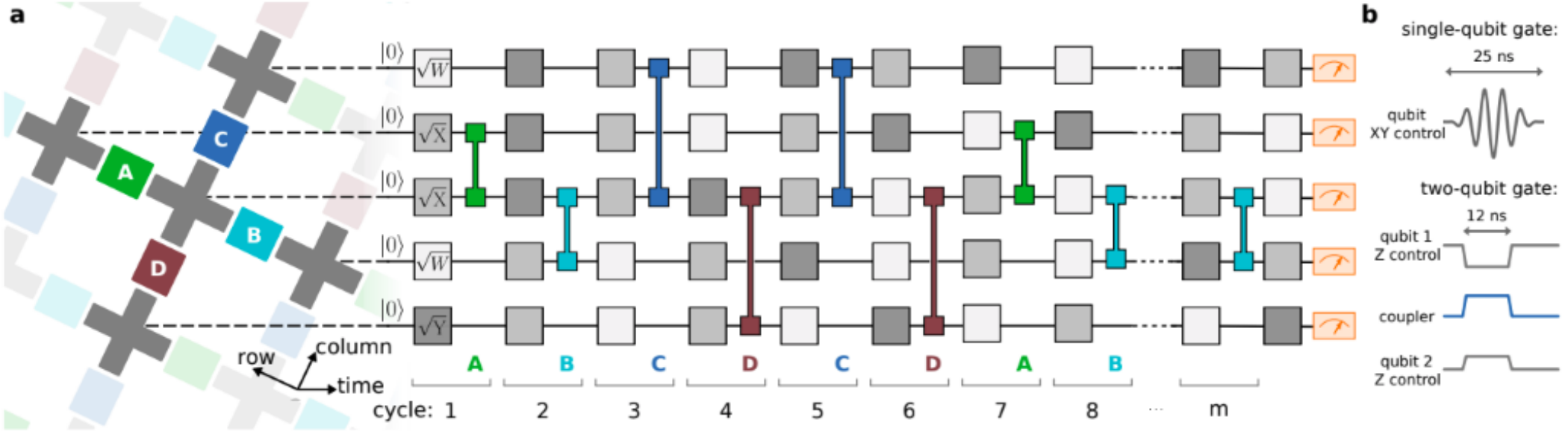
Quantum circuits and their control

Qubit modalities



Superconducting circuits and **trapped ion** quantum computers are leading with respect to the key metrics: **gate fidelity** and **gate speed**.

Google's quantum computer



- Superconducting circuits, **bosonic** qubits
- It has demonstrated the **quantum supremacy**
- The sequence of **two-qubit gates** are chosen according to a tiling pattern, coupling each qubit sequentially to its four nearest-neighbor qubits