Reinforcement learning

Uses:

→ Autonomous helis → aerobatic maneouvers
→ drones

Prob:

Exact: position of heli → how to move control sticks

Abstraction: states s → action a

Supervised learning

$x \rightarrow y$

Issue: one correct action a is hard to determine

- No concept of delayed rewards
- Can't learn from trial and error.

RL is better suited to handle the complexity

RL → good dog, bad dog routine

what to do instead of how to do

reward function

+ve reward: heli flying well +1

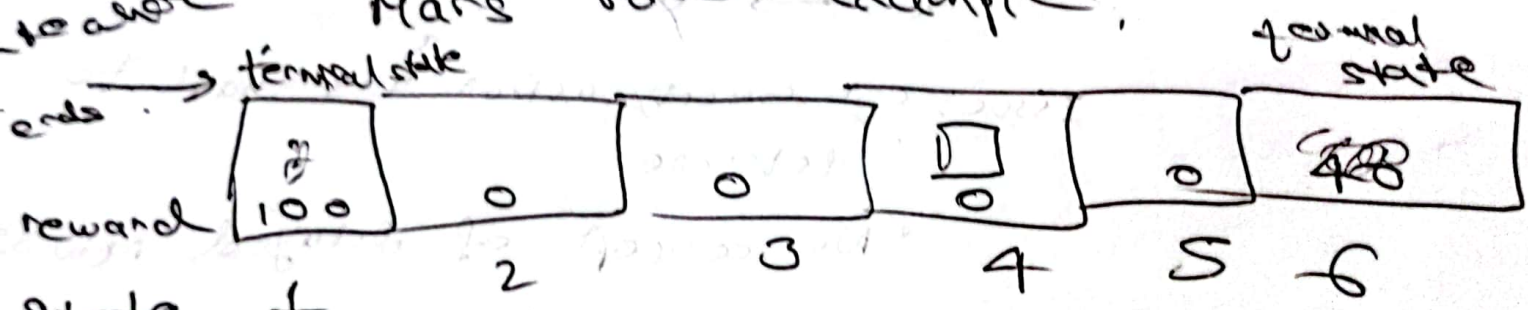
-ve reward: heli crashes −1000

RL applications

- Controlling robots
- Factory optimization
- Financial (stock) trading
- Playing games (video games too).

RL formalism:

Mars rover example: D = rover



Choice: ← left, → right

Possible paths

4 — 3 — 2 — 1

0 0 0 100

0 0 40

0 0 0 0 0 100

At every time step:

robot is at state s

$$(s, a, R(s), s')$$

current state — action chosen from state s — reward — new state, s

Return in reinforcement learning:

~~Answers~~ Measures effort given vs for a reward.

In prev example, if rover goes left for 100 reward we get:

$$\text{Return} = 0 + (0.9)0 + (0.9)^2 0 + (0.9)^3 100 = 72.9$$

discount factor

Abstract:

$$\text{return} = R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$$

$\gamma = 0.9$

∴ Rewards obtainable earlier are more attractive

Usual discount factors : 0.99, 0.999

after incorporating discount factor

return = total reward

reward = for each state

Note? Important in finance as a dollar today is more important than a dollar tomorrow (get e.g. can get interest if saved today)

- Reward varies when starting from diff states.

Mars rover: Rewards starting from each state and $\gamma = 0.5$

⇒ Always go left

1	2	3	4	5	6
100	50	25	12.5	6.25	40
100	← 0	← 0	← 0	← 0	40

← return
← reward

⇒ Always go right

100	2.5	5	10	20	40
100	→ 0	→ 0	→ 0	→ 0	40

⇒ Max ~~reward~~ return if we can go left or right at every state

100 100	50 ← 0	25 ← 0	12.5 ← 0	20 → 0	40 40

→ There is a concept of -ve rewards as well

↳ financial apps. where we want to delay payment

Policy: Function π that take a state s and maps it to an action

$\pi(s) = a$

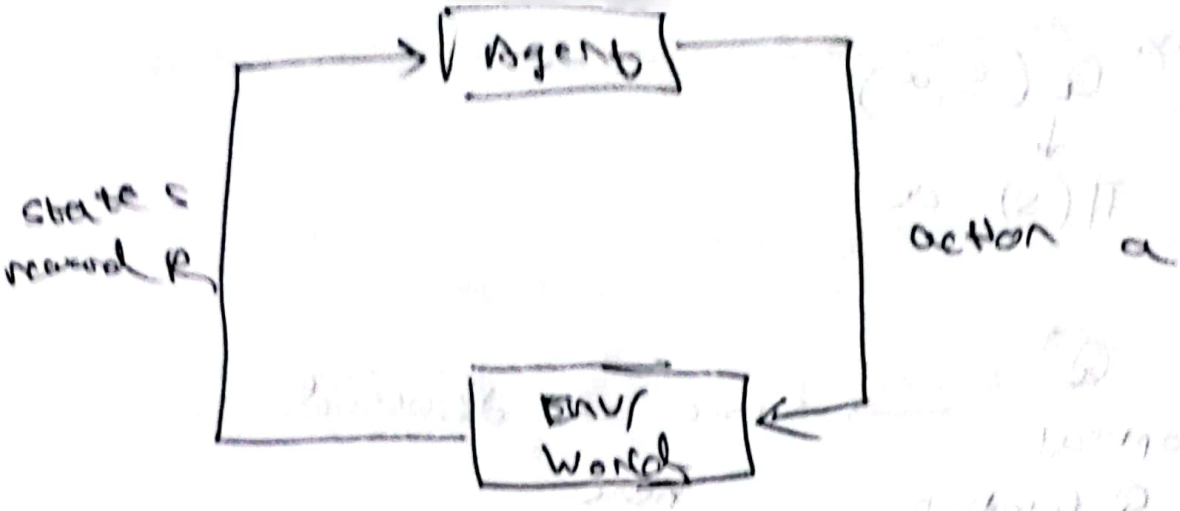
Goal of RL → Find π ~~which for every state~~ which for every state can tell us an action to maximize the return.

	Mars Rover	Helicopter	Chess
States	6	position of heli	pieces on board
actions	← →	how to move control stick	possible moves
rewards	100, 0, 40	+1, −1000	+1, −1, 0
discount factor γ	0.5	0.99	0.995
return	$R_1 + \gamma R_1 + R_2 \gamma^2 \ldots$	$R_1 + \gamma R_2 + \gamma R_3$	$R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots$
policy π	100 ← ← ← → 40	find $\pi(s) = a$	find $\pi(s) = a$

Markov Decision Process (MDP) ~~$\pi(s) = a$~~

↑

future only depends on current state and not how we got to that state



State-action value function / Q-function

$\hookrightarrow$ $Q(s, a)$ = return, if you

- Start in state s
- take action a (once)
- act optimally after that

state we are in

action we may take

We can come with a Q before we have ever calculated π

100	100	100	50	0	2.5	25	0	6.25	12.5	0	10	6.25	0	20	40	40	40
1			2			3			4			5			6		

$Q(2, \leftarrow)$ $Q(2, \rightarrow)$

max

$\downarrow$

return at state 2

$$\max_a Q(s,a)$$
$$\downarrow$$
$$\pi(s) = a$$

Q^* → same as discussed here

optimal Q function

Bellman equations

s = current state $R(s)$ = reward of s.

a = current state

s' = state after taking a

a' = action that you take in s'

$$Q(s,a) = \underbrace{R(s)}_{\text{immediate reward}} + \underbrace{\gamma \max_{a'} Q(s',a')}_{\text{return from behaving optimally starting from } s'.}$$

at terminal states

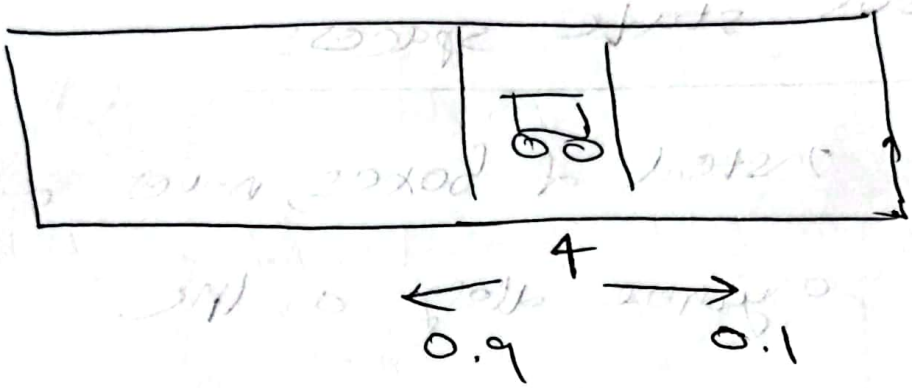
$$Q(s,a) = R(s)$$

Best return from s' is $\max_{a'} Q(s',a')$

Random (stochastic) environment:

Randomness in ~~eva~~ env. → wheels slipping, wind blowing
↓
exact action not taken

Stochastic environment:



tell to go left but 90% of going left and 10%.

Expected return.

- Return is random so we don't maximize that

- Expected Return = Average $(R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots)$

$$= E\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots\right]$$

Modified Bellman equation:

s' is random

$$\therefore Q(s,a) = R(s) + \gamma E\left[\max_{a'} Q(s',a')\right]$$

# of states may be much larger

Continuous state spaces:

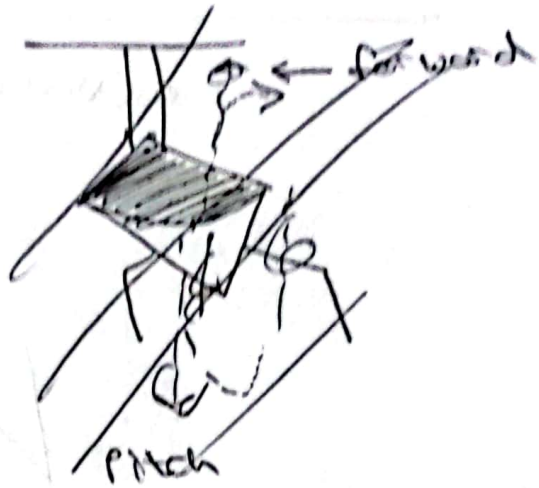
Instead of boxes, now can be anywhere along a line

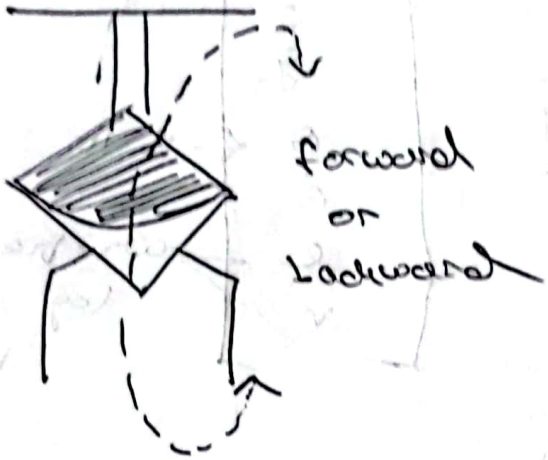
For a truck, continuous state may be

$$s = \begin{bmatrix} x \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \quad \left.\begin{matrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{matrix}\right\} \text{velocity}$$

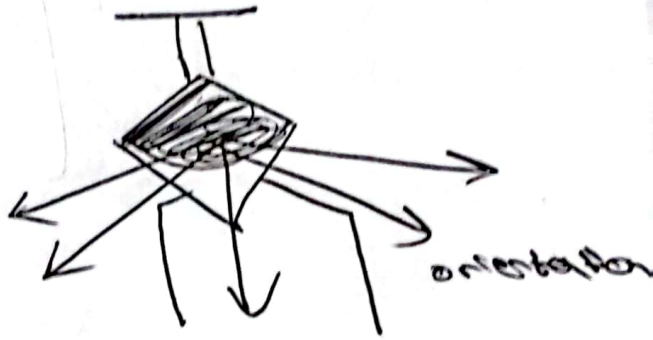
For autonomous helicopter:

roll





pitch



yaw

$$S = \begin{bmatrix} x \\ y \\ z \\ \phi \\ \theta \\ \omega \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\omega} \end{bmatrix}$$

roll — ϕ

pitch — θ

yaw — ω

position

orientation

velocity

angular velocity

Lunar Lander

actions: do nothing
left thruster
main thruster
right thruster

$$s = \cancel{\begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \end{bmatrix}$$

θ ← tilt

l, r ← whether left/right leg is sitting are touching the ground

Reward function:

- getting to landing pad: 100 – 140
- ~~additional~~ reward for moving toward/away from pad
- Crash: –100
- Soft landing: +100
- leg grounded: +10
- Fire ~~engine~~ main engine: –0.3
- Fire side thruster: –0.03

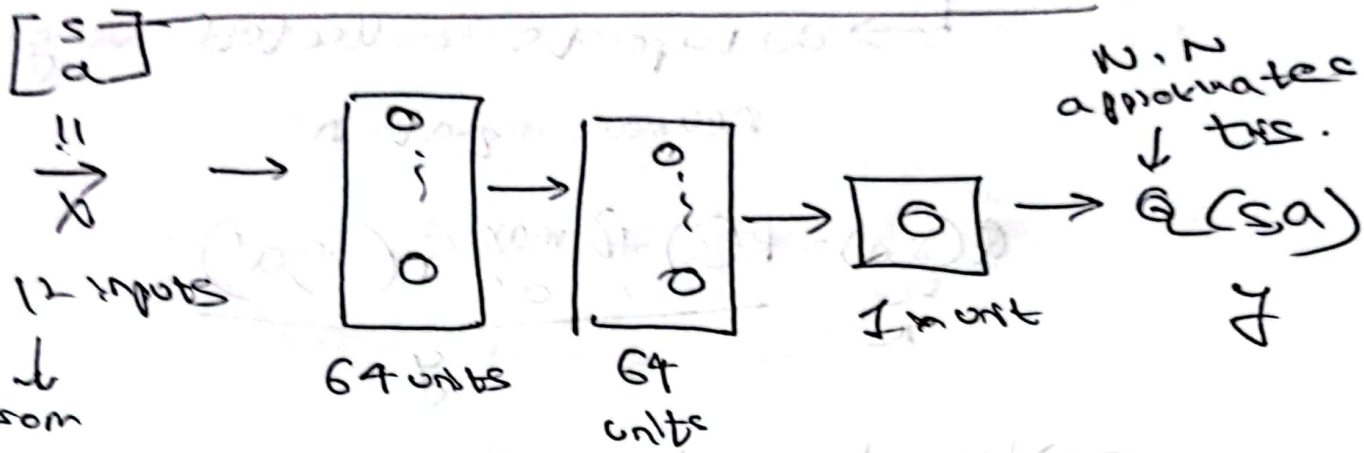
- ~~to~~ not to waste too much fuel

Learn π given s so $a = \pi(s)$ to maximize return

$\gamma = 0.985$

Learning the state-value function.

Deep reinforcement learning



s from prev page + one hot vector to rep. action taken like $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

In a state s, use N.N to compute Q(s, nothing), Q(s, left), Q(s, main), Q(s, right)

Pick a action that maximizes ~~Q(s)~~ Q(s,a)

calculate these and choose a with largest Q

The NN calculates state-action value function (~~max rel~~ return if we take action a at state s and behave optimally afterwards) for a give state-action pair [Q(s,a) from s, a]

↳ datapoints collected using Bellman equation

$$\underbrace{Q(s,a)}_{x} = \underbrace{R(s) + \gamma \max_{a'} Q(s',a')}_{y}$$

→ we try out different things in Lunar Lander

→ If we dont have a good policy, we try random actions

X	Y
$x^{(1)} = (s^{(1)}, a^{(1)})$	$y^{(1)}$
$x^2 = s^2, a^2$	y^2
10,000	10,000

$(s, a, R(s), s')$

$s^{(1)}$ $\underbrace{(s', a')}_{x}, \overbrace{R(s'), s'}^{y})$

$(s^2, a^2, R(s^2), s'^2)$

$(s^3, a^3, R(s^{(3)}), s'(3))$

$y^{(1)} = R(s') + \gamma \max_{a'} Q(s'^{(1)}, a'^{(1)})$

PT: how to get this?

we can start with random Q

↳ it will get better over time

Learning algo:

- Initialize N.N randomly as guess of $Q(s,a)$

Repeat {

- Take action in lunar lander. Get $(s, a, R(s), s')$
- Store 10k most recent $(s, a, R(s), s')$ tuples ← replay buffer

Train N.N:

Create training set of 10k most recent examples

$x = (s,a)$ and $y = R(s) + \gamma \max_{a'} Q(s', a')$

Train Q_{new} such that

$Q_{new}(s,a) \approx y$

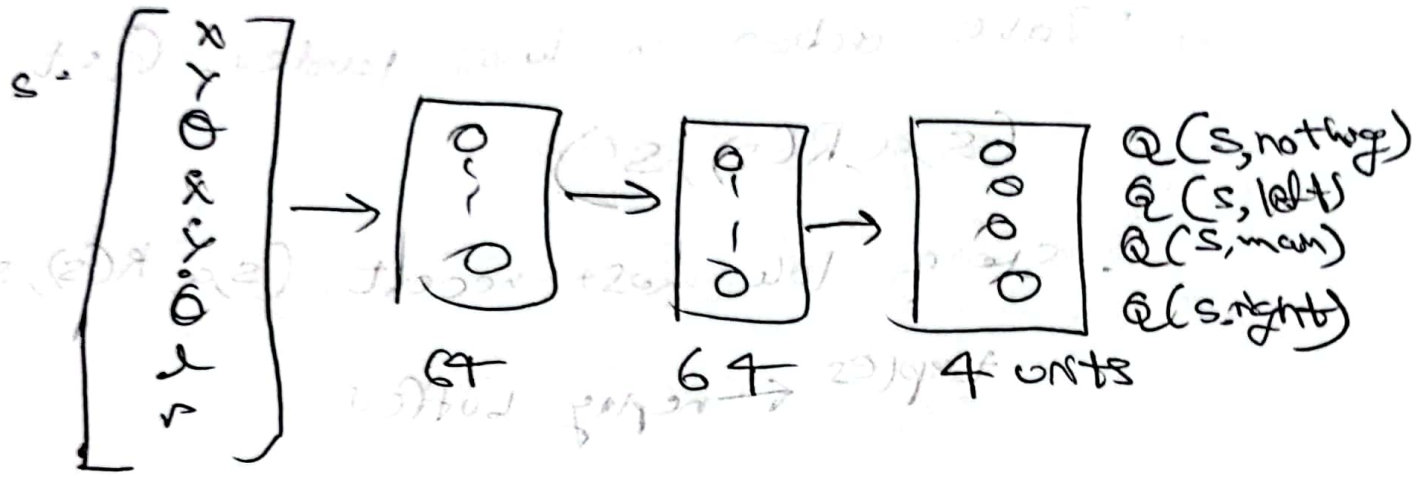
set $Q := Q_{new}$

↑ aka DQN algo.

Refining the DQN algo:

We need to com do inferences 4 times to pick an action

Instead we change the architecture:



8 inputs

ε-greedy policy

How to pick action to take while learning

Option 1:

Pick action a that maximizes Q(s,a)

Option 2:

→ with prob 0.95, pick a that maximizes Q(s,a) → greedy/exploitation

→ with prob 0.05, pick action a randomly → ~~exploitation~~ exploration

eg. by chance N.N may think $Q(s, max)$ is low

↳ randomness helps to break out of this.

Option - 2 is ε-greedy policy

Trick: start ε high, then decrease gradually.

RL is more susceptible to hyperparameter tuning than supervised learning

MM-batches:

- Same idea as seen before

Soft update

Set $Q = Q_{new}$ → Q_{new} may be worse

Sol^n:

$$w = 0.01\, w_{new} + 0.99\, w$$

$$B = 0.01\, B_{new} + 0.99\, B$$

Limitations of RL:

- Much easier to get something working in a simulation than on real robot
- Few applications that surpass supervised/unsupervised algos