Week 03

01] What is reinforcement learning

*) Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment to achieve a goal.

*) Instead of learning from labled data (SL), the agent learns through trial and error, by recieving rewards or penalties for its actions

Mars Rover Example

*) Goal → find valuable rocks and avoid terrain.
*) Environment → The martian landscape
*) Agent → The rover
*) Actions → Move forward, turn left/right, ...
*) State → The rover's current location

*) Reward → +10 (Finds valuable rock)
-5 (if it gets stuck)
-1 (for wasting energy on unnecessary moves)

*) The rover starts not knowing anything about which actions are good or bad.

*) Each time it recieves a reward, it updates its understanding of what makes best in certain situations

*) Over time, it learns a strategy (policy) to maximize total reward; to explore efficiently and safely

2] The return in Reinforcement Learning

*] The return (G_t) is the total future reward the agent can expect starting from time t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

where:

*) R_{t+1} ; R_{t+2} , ... = Future Rewards

*) γ = discount factor ($0 \leq \gamma \leq 1$)
- which makes future rewards slightly less important than immediate ones.

ex: If the rover finds a rock after 3 moves, with $\gamma = 0.9$ & rewards (0, 0, 0, +10)

$$G_0 = 0 + 0.9(0) + (0.9)^2 0 + 0.9^3(10)$$
$$= 7.29$$

*) So, the return represents how valuable the current situation is in the long run.

3) Making Decisions (Policies in RL)

*) A policy (π) is the decision-making strategy that tells the agent what action to take in each state

$\pi(a|s) = P(\text{take action } a \text{ in state } s)$

*) Deterministic policy => always pics the same action for a given state

ex: Always move towards a rock if it saws nearby

*) Stochastic policy => picks action based on probabilities

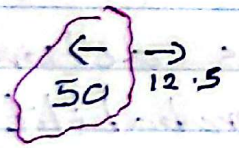
ex: 70% chance move forward, 30% turn right

*) The goal of RL is to find optimal policy (π^*) that maximizes expected return

4] State - action Value

$Q(s,a)$ = Return if you
- *) start in state s.
- *) take action a (once)
- *) then behave optimally after that

100	50	25	25	20	40	return
	←	←	←	→		action
100	0	0	0	0	40	reward
1	2	3	4	5	6	



$$Q(2,\rightarrow) = 0 + 0.5(0) + 0.5^2(0) + 0.5^3 \cdot 100$$
$$= 12.5$$

$$Q(2,\leftarrow) = 0 + 0.5(100)$$
$$= 50$$

*) The best possible return from state s is $\max_a Q(s,a)$

*) The best possible action in state s is the action a that gives $\max_a Q(s,a)$

5] Bellman Equation

S : current state
R(s) : reward of current state
a : current action
s' : state you get to after taking action a
a' : action that you take in s'

*) The best possible return from state:

*) S is $\max_a Q(s,a)$

*) S' is $\max_{a'} Q(s',a')$

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

Immediate Reward

Return from behaving optimally starting from state s'

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

$$Q(4,\leftarrow) = 0 + 0.5(0) + 0.5^2(0) + 0.5^3 \cdot 100$$
$$= R(4) + 0.5\underbrace{(0 + 0.5(0) + (0.5)^3 100)}_{\text{optimal return of state 3 } (s')}$$

$$= R(4) + 0.5\left(\max_{a'} Q(3,a')\right)$$

6) Stochastic Environment

*) A stochastic (random) environment means actions do not always lead to the same results (There's uncertinity)

ex : Rover commands move forward
- *) 80% chance : moves safely (+2)
- *) 15% it slips slightly (0)
- *) 5% it gets stuck (-10)

The rover must learn to act optimally despite uncertainty (chosing actions that maximize expected / average rewards, not guranteed ones

Thus...

$$Q(s,a) = R(s) + \gamma E\left[\max Q(s',a')\right]$$

s' can be anything

7] Continuos state space

*) In earlier examples (like rover) states were discrete (ex: cell x,y on grid)

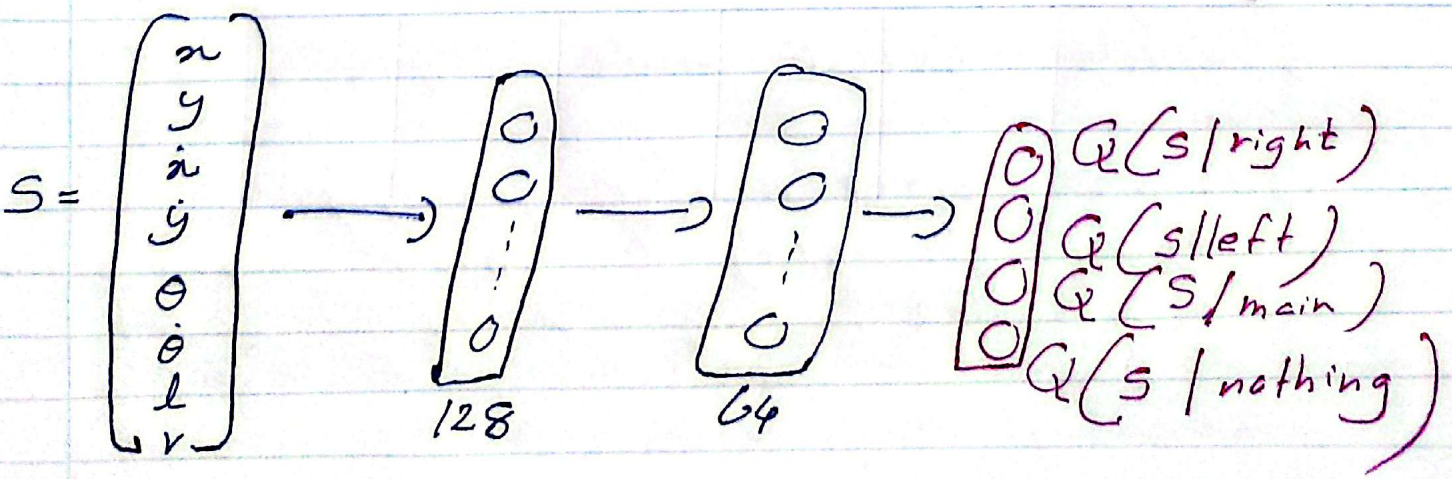
*) But in real world, states are continuos meaning they can take any real value

*) We can't use a table to store all Q(s,a) values (Infinite). Instead we use a function approximator, usually a neural network to estimate $\hat{Q}(s,a)$

ex: Lunar Lander Environment

8] Deep Q network (DQN)

*) DQN replaces the Q-table with a deep neural network. The network takes the current state as input and outputs a vector of Q-values, one for each possible actions



Training process & key techniques

*) The agent learns through trial and error, interacting with the environment and updating the neural network's weights (θ).

ε-Greedy Policy

*) The agent must balance trying new things (exploration) and using what it has learned. (exploitation)

*) With a small probability ε (epsilon), the agent chooses a random action (exploration)

*) With probability $1-\varepsilon$, it chooses the action with the highest Q-value predicted by the network for the current state (exploitation)

*) ε usually decays over time, meaning the agent explores more at the start and exploits more once its gained experience.

Experience Replay

*) The agent's experiences (state, action, reward, next state) are stored in a large memory buffer called the "replay buffer".

*) Instead of training the network on consecutive, highly correlated experiences, the agent randomly samples mini batches of past experiences from this buffer to train the network

*) Benifits → ·) Breaks the corretation consecutive samples, which stabilizes training

·) Allow the agent to reuse past experiences, improving sample efficiency.

Target Network

*) DQN use two neural networks

1) The Main Q-Network (weights θ)
- Used to choose the next action and is updated at every training step.

2) The target Q-Network (weights θ^-)
- Used to calculate the target Q-value, which is the ground truth" the main network tries to match.

Stabalizing :-

- The target network's weights θ^- are only updated periodically by copying the weights from the main network. (ex:- every few 1000 steps)
- This freezes the target for a while, providing a stable reference point to learn from.

Step by Step Flow

1) Initialize networks
   - *) Main network $Q(s, a; \theta)$ with random weights
   - *) Target network $Q(s, a; \theta^-)$ copy main networks weights

2) Initialize Replay Buffer
   - *) Empty at start
   - *) Stores experiences (s, a, r, s')

3) Observe Initial State S_0

4) Action selection (ε-greedy)
   - *) With probability ε : chose random action
   - *) Otherwise : $\arg\max_a Q(s, a; \theta)$

5] Take action and Observe Reward & Next state

*) Receive r and s'

6] Store experiences in Replay buffer

* (s, a, r, s')

7] Sample mini batch from Replay Buffer

*) Random sample of size N (64)

8] Compute target Q-Values using Target Network:

$$y_i = r_i + \gamma \max_{a'} Q(s_i', a'; \theta^-)$$

9] Update Main network (θ)

*) Minimize loss (Adam / Gradient Descent)

$$L(\theta) = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i; \theta))^2$$

10] Soft update Target variable

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^- \quad (0 \leq \tau \leq 1)$$

or

$$\theta^- \leftarrow \theta \text{ (every 1000 steps)}$$

11] Move to Next state : $s \rightarrow s'$ and repeat

Over time DQN Converges to a policy that maximizes expected return