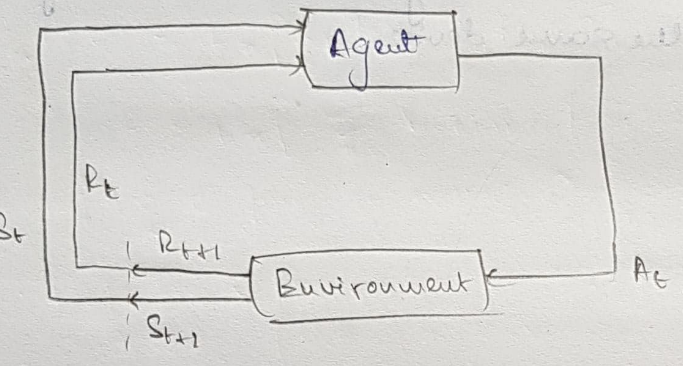
Reinforcement Learning
(Deep Lizard)

Markov decision Process (MDP): It is a way to formalize sequential decision making.

Components of MDP:
- agent
- environment
- state
- action
- reward

* Agents wants to maximize the cumulative reward.



Steps:
- At t, env. is in state S_t.
- Agent observes the state and takes action A_t
- Env changes to S_{t+1} with a reward R_{t+1} is given to agent.
- Now, $t+1$ is the current time step.

$S_0\, A_0\, R_1\, S_1\, A_1\, R_2\, S_2\, A_2 \ldots$

$S_t, S_{t+1} \in S$
$A_t, A_{t+1} \in A$ — finite sets
$R_t, R_{t+1} \in R$

Finite MDPs

If infinite, then Infinite MDPs.

Hence, S_t, R_t (random vars) will have well defined prob. dist. (every value of S_t, R_t have associated prob).
The prob. dist of S_t and R_t depend on the prev. time step (action and state)

* Probability of transition to state s' with reward r from taking action a in state s.

$$p(s', r \mid s, a) = Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

Expected Return:
* Sum of future rewards.

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T$$

$T \rightarrow$ final time step

If $T \rightarrow$ finite, then the agent environment interaction can be broken into sub_sequences called episodes.
Each episode ends in a terminal state T

and the env. is reset after that. Next episode is independent of how prev. one ended. These are called as episodic tasks.

G_t will be finite if each reward is finite.

If $T = \infty$

$$G_t = R_{t+1} + R_{t+2} + \cdots \quad , \text{ here } G_t = \infty.$$

Continuing tasks (can't be broken into episodes)

Discounted Returns:

Future rewards are more heavily discounted, so the agent cares more about the immediate reward.

discount rate

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \quad 0 \leq \gamma \leq 1$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

If $\gamma = 0 \Rightarrow G_t = R_{t+1}$ (the agent will choose greedy action to optimize the immediate reward)

* Smaller the $\gamma \Rightarrow$ more miopic is the agent

Now; if $R_t < \infty$ and $0 < \gamma < 1 \Rightarrow G_t < \infty$

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

$$= R_{t+1} + \gamma \left[R_{t+2} + \gamma R_{t+3} + \cdots \right]$$

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Policies (what is the prob. of the agent to select any given action in a particular state).

For each state $s \in S$, π is a prob. dist over $a \in A(s)$.

$\pi(a|s)$: Prob. of taking action a in state s.

Value functions (how good is it for an agent to be in a state or how good is it for the agent to choose an action in a given state).

Value functions depends on actions taken and hence on policies and thus on expected return.

- **State value func** (how good is any state for an agent following policy π)

$$v_\pi(s) = E_\pi[G_t \mid S_t = s]$$
$$= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]$$

v_π: Expected return from starting from state s and following policy π thereafter.

- **Action value func** (how good it is for an agent to take any given action in a given state when following policy π)

Q-func

$$q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a]$$
$$= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

* output of Q-func is called Q-value.

q_π: Expected return when starting from state s, taking action a and then following π thereafter.

Optimality:

RL algorithms seek to find a policy that will yield more returns to the agent if it follows that policy.

optimal Policy:

$\underbrace{\pi \geq \pi'}$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ $\forall s \in S$.

Policy π is better than π'

* A policy π is better than π' if and only if, the state value functions for all states under policy π is greater than that under π'.

optimal state value func:

The optimal policy has an associated optimal state-value func

$$v_*(s) = \underbrace{\max_\pi v_\pi(s)}_{\text{maximise } v_\pi(s) \text{ over all possible } \pi} \quad \forall s \in S.$$

Optimal Action value func:

optimal policy also has optimal action value func (Q func.)

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in S,\ a \in A(s)$$

Bellman optimality eq. for q_*:

q_* must satisfy

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

The optimal action value func. of a state action pair is equal to the expected reward of choosing a in state s plus the max. expected discounted return from the next state action pair.

Q-learning:

An RL algo that finds the optimal policy by learning optimal Q-values for each state action pair.

Q-learning with value iteration:

Iteratively update Q-values for each state action pair using bellman equ. until the Q-func. converges to optimal Q-func (q_*).

This is called value iteration.

The Q-values are stored in Q-table.

Q table — actions →

↑ states ↓ | dim = [no of actions × no. of states]

Exploration vs Exploitation:

Epsilon greedy strategy: ϵ → exploration rate

Initially $\epsilon = 1$ and is reduced after every time step towards 0

At every time step, generate a random number 'x' between 0 and 1.

If $x < \epsilon$ ⇒ explore

$x > \epsilon$ → exploit (become greedy)

(choose action with highest Q-value)

Updating the Q-value:

We want the Q-value to eventually converge to optimal Q-value,

$$\overbrace{q_*(s,a)}^{\text{optimal Q-value}} - \overbrace{q(s,a)}^{\text{old Q-value}} = \text{loss}$$

$$\rightarrow E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] = \text{loss}$$

We update the Q-value of a state-action pair iteratively to minimise the loss whenever we encounter the same state action pair.

Learning Rate (α):

A number between 0 and 1 which tells how quickly agent abandons the prev Q-value in the Q-table for the new value.

$\alpha = 1 \Rightarrow$ prev value is totally discarded.

$$\underbrace{q^{new}(s,a)}_{\text{new Q-value}} = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\underbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}_{\text{learned value}}$$

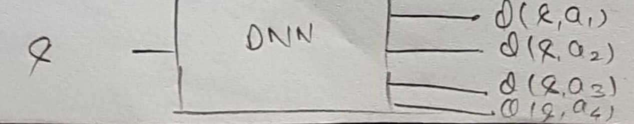
Creating Lizard Game:

+1 (0)	-1 (1)	+1 (2)
-1 (3)	-10 over (4)	-1 (5)
S -1 (6)	-1 (7)	+10 over (8)

Deep Q-learning:

When state-spaces increase in size, iterative approaches such as value iteration become computationally expensive. Instead of using value iteration to compute optimal Q-func, we can use Neural Networks to approximate the optimal Q-func.

* The DNN accepts state of the env. as input and outputs estimated Q-values for each action in that state. The DNN tries to approximate the optimal Q-func that satisfies Bellman's equ.



* loss is calculated by comparing the output Q-values of the network to the target Q-values from the RHS of Bellman Eqn.

Based on the loss, weights of the network are updated using back prop. The process is repeated over and over for each state until loss is minimized.

* Consecutive frames are often used as input to the NN after some preprocessing.
* Activation func. are not added at the output layer since we want raw Q-values from the network.

Replay Memory:

We store the agent's experiences at each time step into a dataset called replay memory.

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

agent's experience at time step 't'.

Last N experiences are stored in replay memory (finite size)

* we randomly sample from replay memory to train the DQN and not train from sequential experience to break the correlation between consecutive samples. (for more efficient learning)

Deep Q-Network is also called the policy network.

Training a Policy Network:

Assuming that we input a single state as input (batch size = 1)

For eg., we pass s_4 as input.

$$e_4 = (s_4, a_4, r_5, s_5)$$

The network outputs Q-values for all the action we select the Q-value for the action a_4 from the exp. tuple. Loss is then calculated by comparing this estimated Q-value with the target Q-value for a_4.

Now, target Q-value is given by the RHS of Bellman eqn.

$$q_*(s_4, a_4) = E\left[R_{t+1} + \gamma \max_{a_5} q_*(s_5, a_5)\right]$$

To compute $\max_{a_5} q_*(s_5, a_5)$, we pass s_5 to the policy network and choose the max Q-value output over all the actions.

Therefore, two forward passes are done before calculating the loss and updating the weights.

*In case of value iteration, $\max_{a'} q_*(s', a')$ was found directly from the Q-table.

Training Issues with DQNs:

The first pass through the DQN (Policy Network) is done to find the estimated Q-value and 2nd pass for target Q-value. Both of these passes are done through the same network before updating the weights. $q(s,a)$ approaches $q_*(s,a)$ ~~before updation~~ after updating the weights but $q_*(s,a)$ also moves away. This introduces instability during training.

The Target Network:

A clone of policy net called target network is used to do the 2nd pass to find the target Q-value. The weights of this network is updated to the same as Policy Network after some time steps. This reduces instability.