MARKOV DECISION PROBLEM/PROCESS

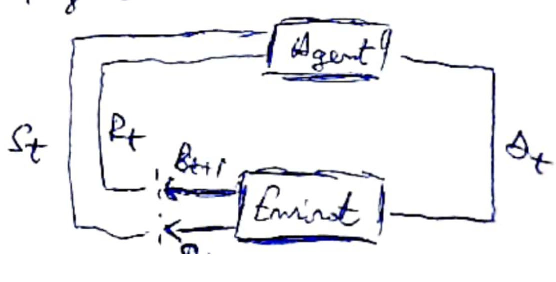
- avem un 'factor de decizie' → ~~AGEN~~ Agent care interacționează cu mediul înconjurător

- interacțiunile se ... pas cu pas
- la fiecare pas agentul are o reprezentare a stării mediului
- agentul selectează o acțiune iar mediul face tranziția către o nouă stare, iar agentul primește o recompensă

Componentele unui MDP
- mediul
- agentul
- toate stările posibile ale mediului ~~actuale~~
- toate acțiunile pe care le poate lua agentul
- toate recompensele pe care agentul le poate lua în funcție de acțiunile alese

Scopul: să maximizeze recompensa totală pe care o primește în funcție de acțiunile alese

Notații:
- S -> stări
- A -> Acțiuni
- R -> reward
- t -> timestep
- $S_t \in S$ -> starea primită de agent la momentul t
- A_t -> acțiunea selectată la momentul t pornind de la S_t
- (S_t, A_t) -> face tranziția la $S_{t+1} \in S$, și primește un Reward $R_{t+1} \in R$
- procesul primirii unei recompense poate fi scris ca o funcție f : $f(S_t, A_t) = R_{t+1}$
- Traiectoria reprezintă procesul secvențial:
  $S_0, A_0, R_1, S_1, A_1, R_2 \ldots$



Scanned with CamScanner

Goal MDP is to maximize it's cumulative reward. So, we will define <u>Expected Return</u> of the rewards at a <u>given time</u>

- we can define $G_t = R_{t+1} + R_{t+2} + \dots R_{T}$, T - the final step
  - the agent goal is to maximize G
- but G_t could be infinite so we need to modify it:
  - ~~discounted~~ the agent goal is to maximize the <u>discounted</u> reward
  - discount rate $= \gamma \in [0, 1]$
  - $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$
  - $\boxed{G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}}$ $\boxed{G_t = R_{t+1} + \gamma G_{t+1}}$

  - obs: the agent will care more about the immediate reward rather than the future ones because they will be heavily discounted

#2

What is the probability that an agent will select a specific action from a specific state?	"How good" is a specific action or a specific state for the agent?
⇓	⇓
Policies (π)	Value Functions

Policy: a function that maps a given state to probabilities of selecting each possible action from that state

- an agent follows a policy
  - e.g. if an agent follows a policy π at time t, then $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$.
    This means that, at time t, under policy π, the probability of taking action a in state s is $\boxed{\pi(a \mid s)}$
- obs: for each state $s \in S$, π is a probability distribution over $a \in A(s)$

Value Function: function of states, or of state, action pairs, that estimates how good it is for an agent to be in a given state, or how good is the agent to perform a given action in a given state

- Value function $\xrightarrow{\text{given in terms of}}$ expected Return $\longrightarrow$ the way the agent acts $\xrightarrow[\text{is that}]{\text{i.e.}}$ policy

- $\boxed{(\text{states})}$ or $\boxed{(\text{state, action})}$

  $(\text{states}) \downarrow$ state-value function

  $(\text{state, action}) \downarrow$ action-value function

State-Value Function : v_π

- how good any given state is for an agent following policy π
- in other words, it gives the value of <u>a state</u> under π
- $\boxed{v_\pi(s) = E[G_t \mid S_t = s]} = E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]$

Action-Value Function : q_π

- how good it is for the agent to take any given action from any given state while following policy π
- in other words, it gives the value of <u>an action</u> under π
- $\boxed{q_\pi(s,a) = E[G_t \mid S_t = s, A_t = a]} = E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$

- $q_\pi(s,a)$ - is referred as Q-function
- the output of $q_\pi(s,a)$ - is refered as Q-value

Optimal Policies.

- a policy π is better than a policy π' if the expected return of π is $\geq$ the expected return for π' for all states

  $\pi > \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S$
- a policy that is better or at least the same as all policies is called "optimal policy"

Optimal Value-state function

- $\boxed{v_*(s) = \max_\pi v_\pi(s)} \quad \forall s \in S$

Optimal Action-Value function

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

Bellman Optimality Equation for q_* : it must satisfy the equation

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

*3.

Q-learning

- it's a reinforcement learning technique used for learning the optimal policy and a Markov Decision Process
- it's objective is to find a policy that is optimal in the sense that the expected return over all successive timesteps is the maximum achievable
- in other words the goal of Q-learning is to find the optimal policy by learning the optimal Q values for each (state, action) pair

How Does It Works?

- iteratively updates the Q values for each state, action pair using the Bellman Equation while the Q value converges to the optimal

Ex.

- at the start of the game the lizard has no idea of how good any given action is from any going state, it's not aware of anything besides the current state of the environment
- therefore the Q values for each (state, action) pair will all be initialized with 0 since the lizard know nothing about environment at the start
- trough the game the Q values will be iteratively updated using Value Iteration
- we will be using use of a table, called Q-Table to store the Q-values for each (state, action) pair

End Ex

- Exploration vs Exploitation
- to get this balanced we use an ε-greedy strategy

ε - Greedy Strategy

- helps us balance the exploration and exploitation
- we have an exploration rate ε that we initially set to 1
- this ε is the probability that our agent will explore the environment rather than exploit it
- ε decrease over time

Update Q-Value

- $q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$
- $\boxed{loss = q_*(s,a) - q(s,a)}$, we try to minimize this

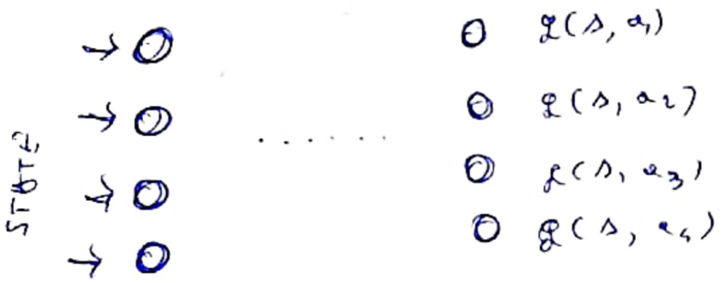
$$loss = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right]$$

- λ -> learning rate

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

DEEP Q-LEARNING / DEEP Q-NETWORKS

- instead of using value iteration to directly compute the q values and find the optimal q function, we instead use a function approximator to estimate the optimal q function. What could it be?... Neural Networks
- we will be using a deep neural network to estimate the q values for each (state, action) pair in a given environment, and in turn the n.n. will approximate the optimal q value
- the action combining q learning with a deep n.n. is called Deep Q-Learning
- a deep n.n. that approximates a q function is called a Deep Q-Network / D.Q.N
- n.n. receives states as an input and the network estimates q values for each action that can be taken from that state



- it's objective is to approximate the optimal q function, and the optimal q will satisfy the Bellman Equation
- the loss of the network is calculated comparing the outputted q values to the targeted q values ~~from~~, and the objective is to minimize this loss
- in a complex environment, we'll use images as our input to the networks
- because a single frame does not contain sufficient information (eg where the ball is moving), we will be using ~~more~~ a stack of frames which will represent a single input

Experience Replay / Replay Memory

- utilized in the traing process of a ~~E~~ DQN
- we store agent experiences at each timestep in the dataset called Replay Memory
- at time t the agent experiences $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$
- all of the agent expertes at each timestep over all episodes played by the agent are stored in the Replay Memory
- in practice will usualy see the replay memory set to some fite size limit lig N (=6 for ex) and will store the last N experiences
- this replay memory dataset is actualy what will readily be sampling from to train the network
- the act of ~~gaining~~ gaining experience and sampling from the Replay Memory that stores this experiences is actualy what Experience Replay is
- why use Replay Memory? and not train sequentialy?
  - to break the corelation between consecutive samples

How it works:

- initialise replay memory capacity
- intialise the network with random weights
- for each episode
  - intialize the starting state
  - for each timestep
    - select an action (via exploration or exploitation)
    - execute selected action in an emulator
    - observe reward and next state
    - store experience in replay memory
    - sample random batch from replay memory
    - process states from batch
    - pass batch of preprocessed states to policy network
    - calculate loss between output Q-values and target Q-Val
      - requires a second pass to the network for the next sta
    - gradient descend updates weights in the policy network to minize loss

25.

Problems when Using Single Network

- when we update the weights, we also update the target ~~term~~ values ~~outputted q value~~ because they are calculated using the same weights
- our q values will be updated with each iteration to move closer to the target q values, but the target q value will also be moving in the same direction
- rather than doing a second pass to the policy network to calculate the targeted q values, we instead obtain the targeted q values from a completely separate network called The Target Network
- this network is a clone of the policy network, it's weights are frozen with the original policy network's weights and we update the weights in the target network to the policy network's new weights every certain amount of timesteps
- this removes much of the instability introduced by only using one network to calculate both the q values as well as the target q values
- these values won't stay fixed the entire time, after x amount of time steps, will update the weights in the target network with the weight from our policy network which will intern update the target q values with respect to what it's learned over those past timesteps; this will cause the policy network to start to approximate the updated targets

The new steps will look like:

- initialize replay memory capacity
- initialize the Policy Network with random weights
- ★ clone the policy network, and call it the Target Network
- for each episode
  - initialize the starting state
  - for each time step
    - select an action via exploration or exploitation
    - execute selected action in an emulator
    - observe reward and next state
    - store experience in replay memory
    - sample random batch from replay memory
    - preprocess states from batch
    - pass batch of preprocessed states to Policy Network
    - calculate loss between output Q-Values and target Q-Values
      - ★ require a pass to the Target Network for the next st
    - gradient descent updates weight in the Policy Network to minimize loss
      - ★ after x time steps, weights in the Target Network are updated to the weights in the policy network