Reinforcement learning

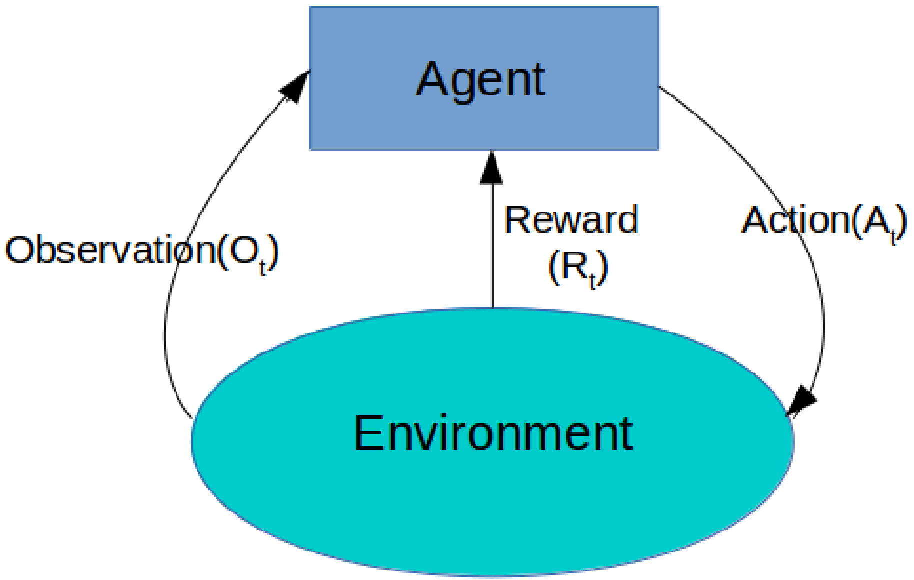
- Dr. Ahmad Altarawneh
- Department of Data Science
- Faculty of information technology, Mutah University

Outline

- What is reinforcement learning (RL)?
- Why do we need?
- Applications
- Q-Learning
- Deep Q-learning
- Neuroevolutionary
- NEAT

What is reinforcement learning (DL)

- RL is a type of machine learning where an agent learns to make decisions by interacting with an environment.
  - The goal is to maximize cumulative rewards by selecting actions that lead to favorable outcomes

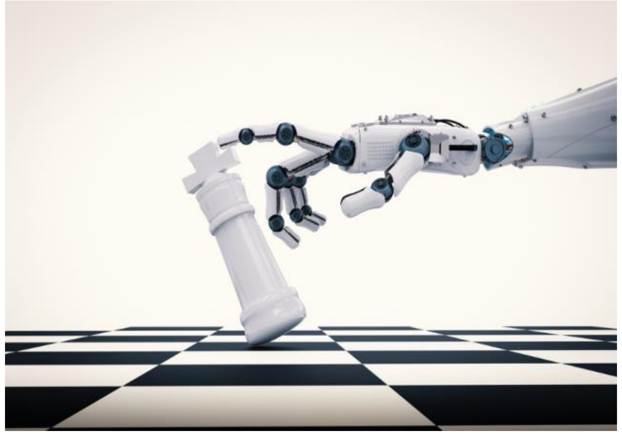


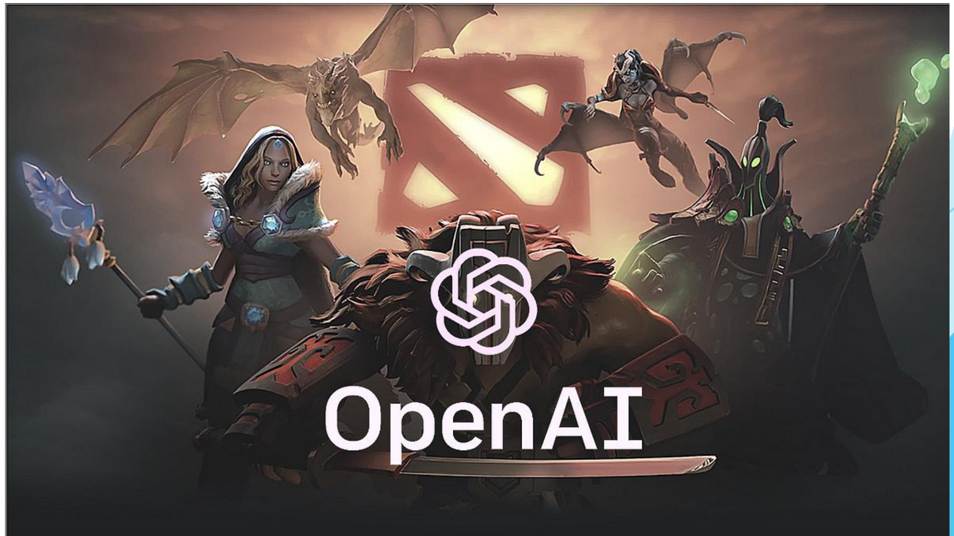
Components of RL

- Typically, reinforcement Learning system should have the Following core components:
- Agent: The decision-maker in the RL framework.
  - The agent interacts with the environment by taking actions and learning from the feedback (rewards) it receives.
- Environment: The environment is the external system with which the agent interacts.
  - It provides the state information and feedback (rewards) based on the actions taken by the agent. E.g., in games, the paths, map, opponents' actions, board, etc.
- Actions: Movements using skills, decisions made by the agent.
  - actions change the state of the environment and may yield rewards
- States: A state is a representation of the current situation in the environment
  - The agent studies the state to take an action
- Rewards: Rewards are signals from the environment that tell the agent how well it is performing
  - Score of the agent, i.e., hitting enemy will give a positive reward.
- Policy: How the agent learns, how it should react at given a state

Applications of RL

- Gaming: RL has been successfully used in games like AlphaGo and Dota 2.
- Robotics: Learning autonomous control strategies for robots in navigation, manipulation, and interaction.
- Autonomous Vehicles: RL helps in decision-making for self-driving cars, such as path planning and obstacle avoidance.
- Healthcare: Personalized treatment recommendations and robotic surgeries.
- Finance: Algorithmic trading, portfolio optimization.
- Industrial Automation: Efficient resource allocation and management.





Q-Learning

Q-Table

- A tabular representation where each state-action pair is assigned a Q-value, representing the expected future rewards for taking that action in a given state.
  - Q refers to quality, it is a representation of the action quality, given a state
- Following is a representation of how the table might look like
  - $Q(s, a)$ is the quality of action a, given a state s

states	actions a_0	a_1	a_2	…
s_0	$Q(s_0,a_0)$	$Q(s_0,a_1)$	$Q(s_0,a_2)$	…
s_1	$Q(s_1,a_0)$	$Q(s_1,a_1)$	$Q(s_1,a_2)$	…
s_2	$Q(s_2,a_0)$	$Q(s_2,a_1)$	$Q(s_2,a_2)$	…
⋮	⋮	⋮	⋮	⋮

As it is a table, it requires discrete states and actions (finite)

Q-Table
Cont.

- The values inside this table are learnable, should be learned during the process, while in the environment (taking actions and moving from one state to another)
- The values are updated using the Bellman's equation as follows:

$$Q^{new}(S_t, A_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \bigg(\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}} \bigg)$$

Learning rate (0-1)

Current-state Q value, given an action

Reward from taking the current action and transitioning to the next state immediate reward

Gamma, Q, how much the future reward matters

The maximum Q-values of actions, given the next state (the expected action's Q-value) OR future reward

The above formula is equivalent to

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R_{t+1} + \gamma \ . \ \max_a Q(s_{t+1}, a) - Q(s,a))$$

Exploration and Exploitation

- The agent in Q-learning learns by following two main steps
- **Exploration:** make random actions to explore the environment
- The agent starts at absolute randomness making random movements with a 100% chance

RandPickProb = 1
if UniformRandom (0,1) < RandPick then
pick a random action

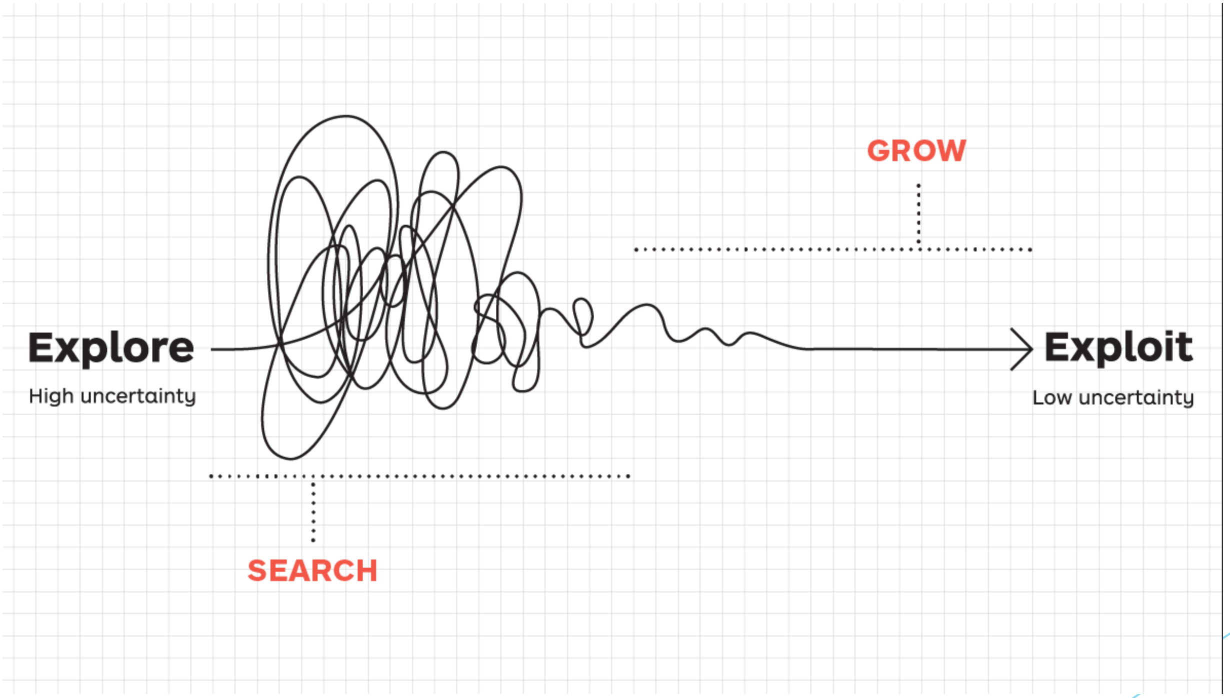
- By time, the reliance on random numbers becomes less

RandPickProb = RandPickProb * 0.98

- this RandPickProb is called ***epsilon***, and the policy applies this is called ***epsilon-greedy policy***
  - *Select the action that maximizes the outcome (reward)*
  - Given a state the agent takes the action that corresponds to the maximum Q- value

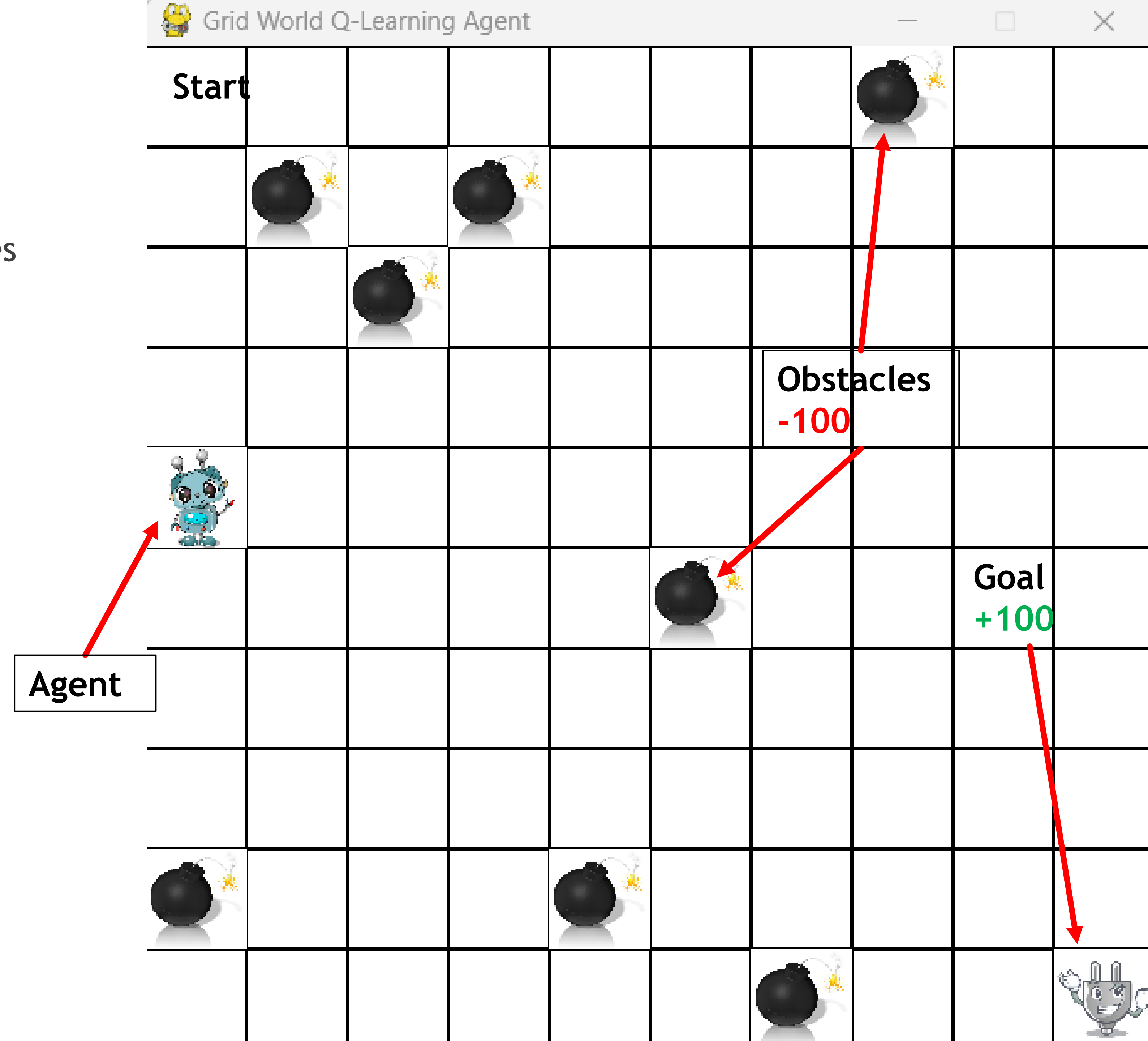
Exploration and Exploitation Cont.

- **Exploitation**: make action based on the previous experience (the table)
- By time the probability of selecting random action became less and the agent takes the action with the best Q value from the table, given a state



Example Grid-world

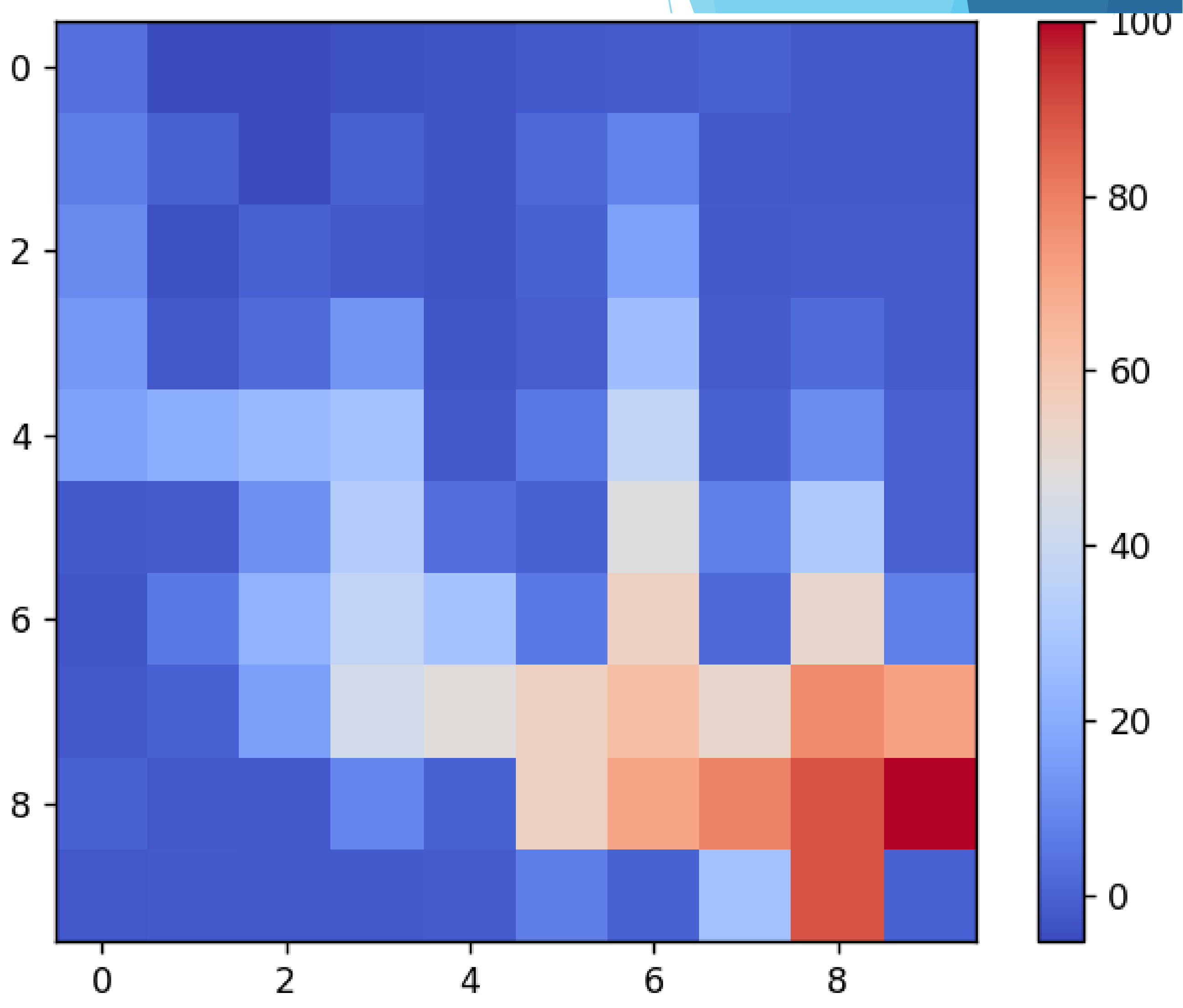
- In this environment there is an agent tries to reach the goal avoiding obstacles
- if the agent steps on a BOMB it loses 10 points (-10)
- if it reaches the goal, it will gain 100 points (+100)
- The agent loses 1 point for each step
  - Encourages the agent to reach the goal faster
- We will use the epsilon-greedy policy to help the agent find its goal faster while avoiding Bombs



Cont.

- The states are each square in the environment,
  - so we have 10 X 10 states
- The actions the agent can take are UP, DOWN, LEFT, RIGHT
- So, the Q-Table is 10 X 10 X 4, the values of which are ZEROS
- Check the implementation
- The following Table shows a visualization of the Q-table for the Grid-world game after around 100 episodes

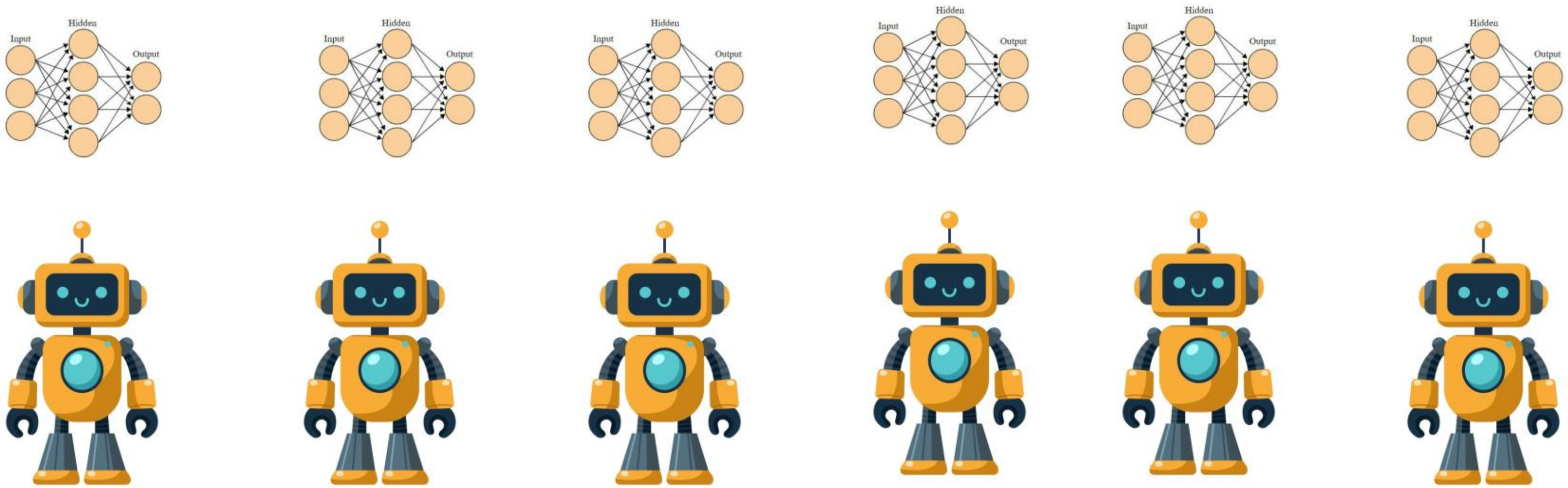
- For complex and continuous environment, Deep Q-learning is used
- **Deep Q-learning is a self-study topic**



Neuro-evolution (NE)

- NE is another way to train an agent to perform a task in a given environment
- It is based on training neural network using genetic algorithm
- Instead of having one agent, we have a population of agents, each of which contains a neural network that controls the agent's actions
- Initially, the agents act randomly in an environment
  - the network in each agent receives observation from environment (features) and provides output (actions)
- By chance, some agents might get a reward for a good action
  - These networks will be used to produce the new generation
- The production of the new generation is done using the GA operations:
  - Crossover: mix the genomes together from parents to produce new children
  - Mutations: some children might have mutation that make them better

Illustration



Cross-over and mutation

- There are too many methods can be used to perform cross-over
- The simplest method is to mix the weights of a layer between two parents
  - Remember the Weights are represented as merices
- Same can be done to the biases

$$\boldsymbol{Father's\ W1\ matrix} = \begin{bmatrix} \mathbf{1} & \mathbf{5} \\ \mathbf{7} & \mathbf{2} \\ \mathbf{1} & \mathbf{9} \end{bmatrix} \qquad \boldsymbol{Mother's\ W1\ matrix} = \begin{bmatrix} \mathbf{6} & \mathbf{8} \\ \mathbf{0.5} & \mathbf{2} \\ \mathbf{2} & \mathbf{1} \end{bmatrix}$$

$$\boldsymbol{child1\ W1\ matrix} = \begin{bmatrix} \mathbf{1} & \mathbf{8} \\ \mathbf{7} & \mathbf{2} \\ \mathbf{1} & \mathbf{1} \end{bmatrix} \qquad \boldsymbol{child2\ W1\ matrix} = \begin{bmatrix} \mathbf{6} & \mathbf{5} \\ \mathbf{0.5} & \mathbf{2} \\ \mathbf{2} & \mathbf{9} \end{bmatrix}$$

Cross-over and mutation
Cont.

- The mutation can be simply performed by taking a new random numbers for a weight matrix
- Or perform masking, in which the values inside the mask will be affected by a random noise
  - To add diversity and not randomize the whole weight values

Example

- In this example we have a very simple game (environment)
- The agent should pass a set of walls through a gap in each wall

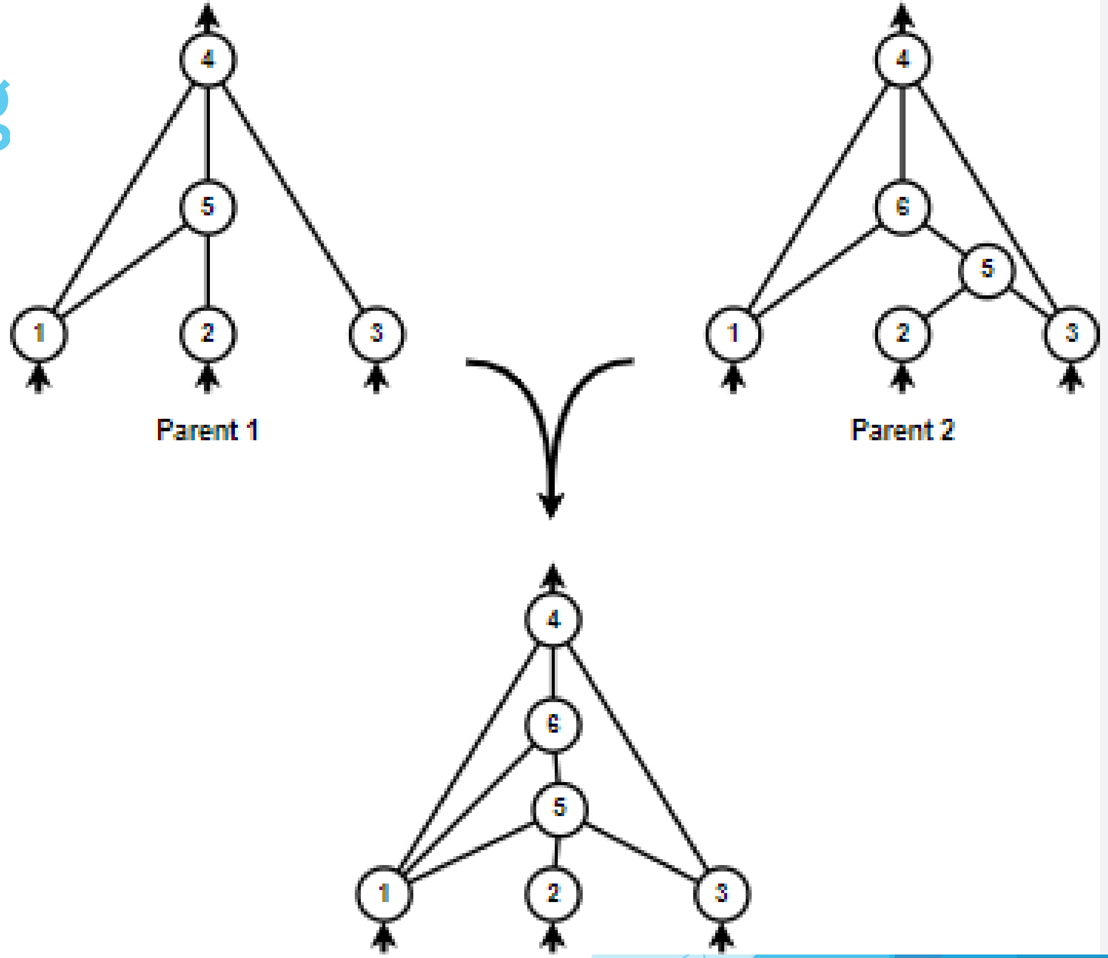
Example
Cont.

- The agent can perform three actions
  - Move left
  - Move right
  - Stay still
- The input to each neural network is a set of features from the current state
  - Location of the beginning of the gap (X location)
  - Gap size
  - The ending location of the gap (X location)
  - The X location of the agent
- These features are fed to the network every frame, and based on them the network decide what action the agent should do

Check the relevant implementation

NeuroEvolution of Augmenting Topologies NEAT

- NEAT is a popular method that evolves both the weights and the topology (structure) of a neural network.
- Unlike traditional neural network evolution methods that focus solely on optimizing the weights of a fixed architecture,
  - NEAT allows for the evolution of the network's structure by adding or removing neurons and connections over time.
- This approach helps in discovering new, more complex representations.
- Evolving the topology helps escape local minima that a fixed architecture might be stuck in



I Hope you Found the Course Useful

Thank you!