REVA
UNIVERSITY
Bengaluru, India

M23DE0201 – Machine Learning

II Semester MCA C Academic Year : 2024 - 2025

School of Computer Science and Applications

Dr.P SREE LAKSHMI
Assistant Professor

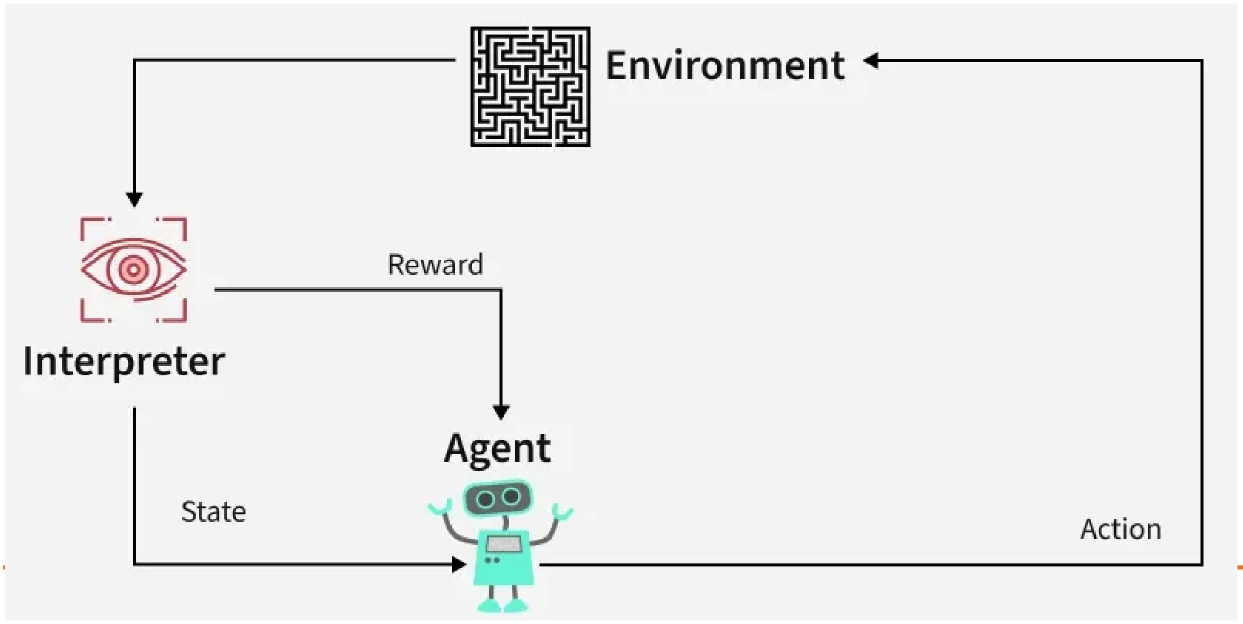
UNIT 4 CONTENTS

REINFORCEMENT LEARNING- INTRODUCTION

Reinforcement Learning (RL) is a branch of machine learning that focuses on how agents can learn to make decisions through trial and error to maximize cumulative rewards.

RL allows machines to learn by interacting with an environment and receiving feedback based on their actions.

This feedback comes in the form of **rewards or penalties**.



REINFORCEMENT LEARNING

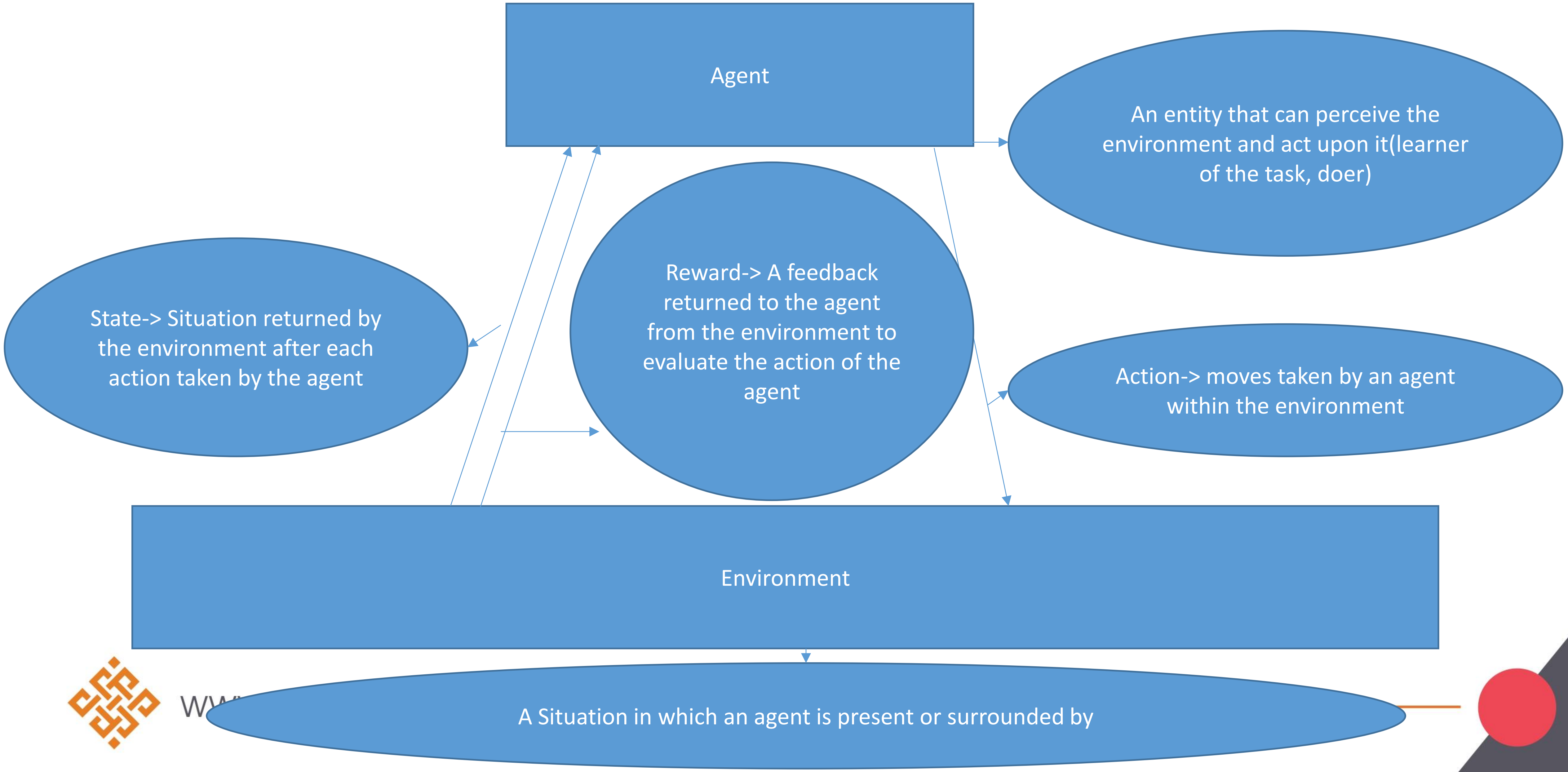
The core components of RL include:

1. **Agent**: The learner or decision-maker.
2. **Environment**: Everything the agent interacts with.
3. **Action**: The choices made by the agent.
4. **State**: A representation of the current situation of the agent in the environment.
5. **Reward**: Feedback from the environment based on the action taken.

The goal of reinforcement learning is

to develop a policy—a strategy that defines how an agent behaves at a given time—that maximizes the expected cumulative reward over time.

SOME TERMINOLOGIES TO UNDERSTAND IN RL



Agent
An entity that can perceive the environment and act upon it(learner of the task, doer)
Reward-> A feedback returned to the agent from the environment to evaluate the action of the agent
State-> Situation returned by the environment after each action taken by the agent
Action-> moves taken by an agent within the environment
Environment
A Situation in which an agent is present or surrounded by

HOW REINFORCEMENT LEARNING WORKS?

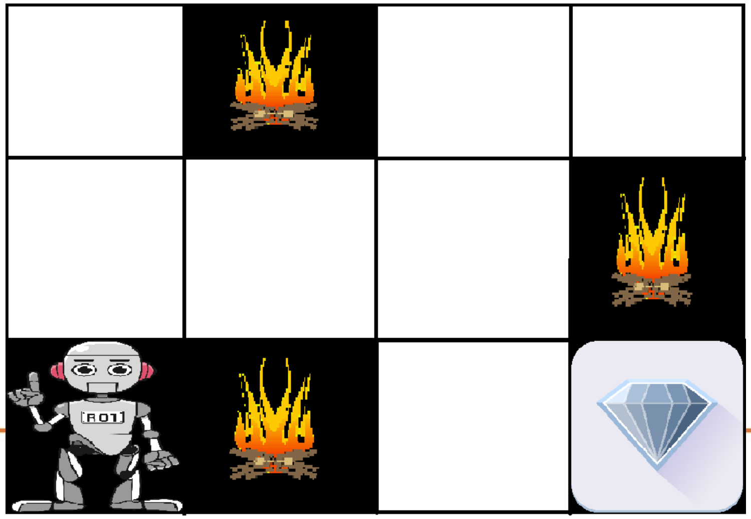
The RL process involves an agent performing actions in an environment, receiving rewards or penalties based on those actions, and adjusting its behavior accordingly. This loop helps the agent improve its decision-making over time to maximize the **cumulative reward**.

Here's a breakdown of RL components:

1. **Policy**: A strategy that the agent uses to determine the next action based on the current state.
2. **Reward Function**: A function that provides feedback on the actions taken, guiding the agent towards its goal.
3. **Value Function**: Estimates the future cumulative rewards the agent will receive from a given state.
4. **Model of the Environment**: A representation of the environment that predicts future states and rewards, aiding in planning.

REINFORCEMENT LEARNING EXAMPLE: NAVIGATING A MAZE

- Imagine a robot navigating a maze to reach a diamond while avoiding fire hazards. The goal is to find the optimal path with the least number of hazards while maximizing the reward:
- Each time the robot moves correctly, it receives a reward.
- If the robot takes the wrong path, it loses points.
- The robot learns by exploring different paths in the maze. By trying various moves, it evaluates the rewards and penalties for each path. Over time, the robot determines the best route by selecting the actions that lead to the highest cumulative reward.



The robot's learning process can be summarized as follows:

- **Exploration**: The robot starts by exploring all possible paths in the maze, taking different actions at each step (e.g., move left, right, up, or down).
- **Feedback**: After each move, the robot receives feedback from the environment:
  - A positive reward for moving closer to the diamond.
  - A penalty for moving into a fire hazard.
- **Adjusting Behavior**: Based on this feedback, the robot adjusts its behavior to maximize the cumulative reward, favoring paths that avoid hazards and bring it closer to the diamond.
- **Optimal Path**: Eventually, the robot discovers the optimal path with the least number of hazards and the highest reward by selecting the right actions based on past experiences.

LEARNING HOW TO OPTIMIZE REWARDS

What is Reward in RL?

- In RL, an **agent** interacts with an **environment**.
- At each step, the agent takes an **action**.
- The environment gives back a **reward** (a number that tells the agent how good or bad that action was).
- The agent's **goal** is to learn a strategy (called **policy**) that **maximizes the total reward** in the long run.

So **optimizing rewards** means:

The agent learns which actions will lead to the best outcomes over time.

Example – Game of Tic-Tac-Toe

Environment: The game board.

Agent: The player (our RL model).

Actions: Placing X in one of the empty spots.

Reward:

+1 for **winning**

0 for **draw**

-1 for **losing**

If the agent always takes random moves, sometimes it wins, sometimes it loses → average reward is low.
But if it learns from experience (which moves often lead to wins), it can **optimize its reward** by playing smarter.

How Does the Agent Learn to Optimize Rewards?

There are different methods (algorithms):

- **Value-based (Q-Learning):**

  The agent learns the "value" of each action in each state.

  Example: In the maze, Q-values tell the robot which move (up, down, left, right) will likely give the best future reward.

- **Policy-based:**

  Instead of storing values, the agent directly learns the **policy** (probability of choosing an action).

- **Actor-Critic (Hybrid):**

  Combines both: actor learns the policy, critic evaluates actions

POLICY SEARCH

What is a Policy in RL?

A **policy** is the strategy an agent uses to decide **which action to take** in each state.

Formally: $\pi(a|s) = P(\text{action } a \mid \text{state } s)$

I,e., the probability of choosing action a when in state s.

Example: In a self-driving car:
•**State:** Traffic light is red.
•**Policy:** Always choose "Stop".
•**State:** Green light + no car in front.
•**Policy:** Choose "Move forward".

Policy search means **directly finding the best policy** (instead of learning value functions like Q-learning).

There are two main ways:

1. Discrete Policy Search:

If the policy is simple (few parameters), we can try different policies and keep the best one.

Example: A robot with 2 possible speeds (slow, fast). We test both and keep whichever gives more reward.

2. Gradient-based Policy Search (Policy Gradient):

We represent the policy with some parameters (like weights in a function).

Then we **adjust parameters step by step** using gradient ascent to maximize expected reward.

Example: In the maze, the robot tries moving with certain probabilities. If one direction leads to the exit more often, gradient updates will increase the probability of choosing that action.

Policy search is especially useful when the **action space is continuous** (like choosing how much force to apply, not just "left" or "right").

The Need for Policy Search:

In many complex environments, especially those with high-dimensional state and action spaces, finding an optimal policy directly through value-based methods (like Q-learning) can be computationally expensive or infeasible.

This leads to the necessity for **policy search methods**, which focus on optimizing policies directly rather than estimating value functions.

Applications of Policy Search: Policy search methods are particularly useful in scenarios where:

- The environment has continuous action spaces (e.g., robotics).
- The dynamics are complex or unknown.
- There are constraints or requirements that make traditional value-function approaches less effective.

For example, in robotic control tasks, where precise movements are required, policy gradient methods can effectively learn policies that yield smooth and efficient trajectories.

Challenges in Policy Search:

While powerful, policy search methods face several challenges:

- **High Variance**: Estimating gradients using sample trajectories can lead to high variance in updates, making convergence slow.
- **Sample Efficiency**: Many algorithms require a significant number of samples to achieve good performance.
- **Local Optima**: These methods may converge to local optima rather than global ones due to their reliance on stochastic sampling.

NEURAL NETWORK POLICIES

- In reinforcement learning (RL), policies are crucial as they define the behavior of an agent interacting with an environment.
- A policy can be represented using neural networks, which allows for complex decision-making processes based on high-dimensional input data.

POLICY REPRESENTATION USING NEURAL NETWORKS

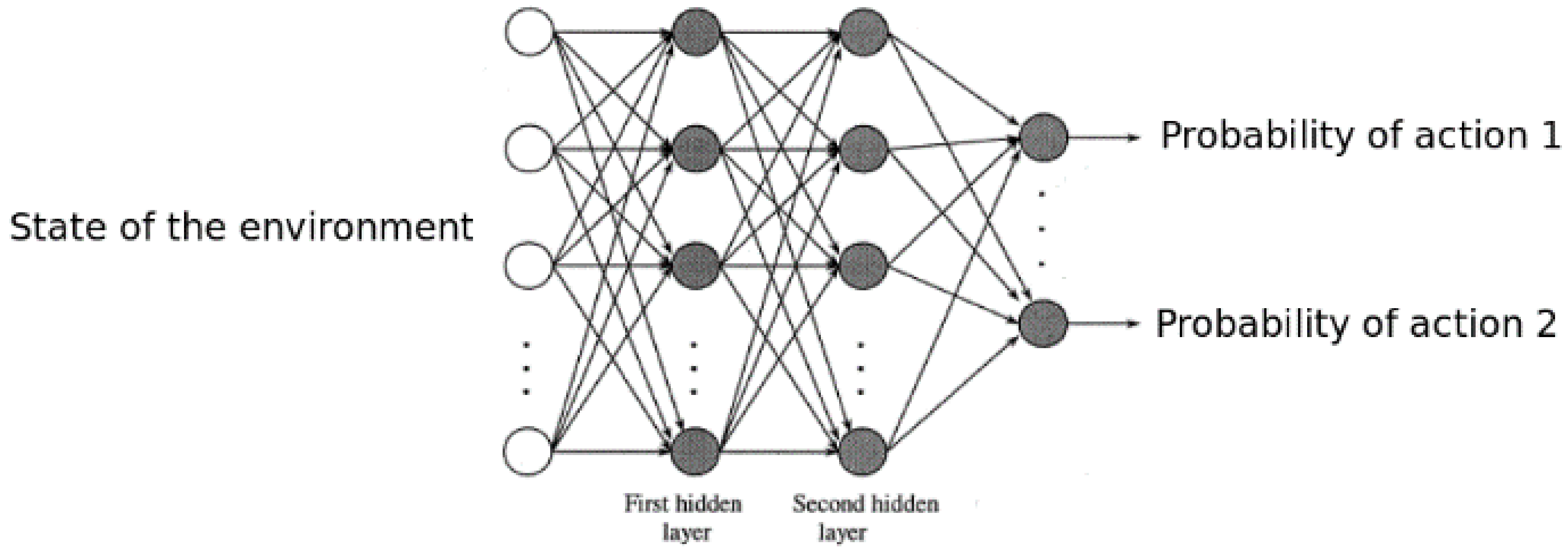
Neural Network Architecture: Neural networks serve as function approximators for policies in RL. The architecture typically consists of:

1. Input Layer: Represents the state of the environment.
2. Hidden Layers: Process the input through various transformations.
3. Output Layer: Depending on whether the policy is deterministic or stochastic, this layer will either output a single action or a probability distribution over actions.

Output Configuration

1. For **deterministic policies**, the output layer has one node representing the action directly.
2. For **stochastic policies**, if there are $\text{Loss} = -\log(P(a|s)) \cdot R$ where $P(a|s)$ is the probability of taking action a given state s, and R is the return received after taking that action.

POLICY REPRESENTATION USING NEURAL NETWORKS



ADVANTAGES OF NEURAL NETWORK POLICIES

Using neural networks to represent policies offers several advantages:

1. They can handle high-dimensional inputs (like images).
2. They allow for flexible representations capable of capturing complex relationships within data.
3. They facilitate exploration through stochastic outputs, which helps prevent local optima during training.

NEURAL NETWORK POLICIES- SUMMARY

A **neural network policy** is when we use a **neural network to represent the policy**.

Input: **state** (observations from the environment).

Output: **action probabilities** (or continuous action values).

Example: Self-driving car 🚗

Input (state): camera image, car speed, distance to obstacle.

The neural network processes this.

Output:

- 70% probability → go straight
- 20% probability → turn left
- 10% probability → turn right

This is called a **policy network**.

ACTION EVALUATION

In RL, the agent needs to **evaluate how good its actions are**.

If the agent takes an action in a state, we ask:
“Did this action help me get closer to my goal?”

There are different ways to evaluate:

State Value Function (V(s)) → how good it is to be in a state.

Action Value Function (Q(s, a)) → how good it is to take action a in state s.

👉 Example (Maze):

If robot moves right and reaches a dead end → action evaluation says it was a **bad action.**

If robot moves up and gets closer to the goal → **good action.**

This evaluation helps the agent refine its policy.

CREDIT ASSIGNMENT PROBLEM

This is a **big challenge in RL**.

When rewards come **much later**, how do we know **which earlier actions actually caused them**?

👉 Example (Student Exam):

Reward = good marks at the end of semester.

But… was it because:

They studied regularly?

They revised one day before?

They drank coffee to stay awake?
We don't know which action to give **credit** to.

👉 Example (Game of Chess):

Agent makes 40 moves, but only at the end it knows **win (+1)** or **loss (-1)**.

Which move(s) led to the win/loss? That's the **credit assignment problem**.

CREDIT ASSIGNMENT PROBLEM

- The credit assignment problem (CAP) is a fundamental challenge in reinforcement learning (RL) that deals with determining which actions taken by an agent are responsible for the rewards it receives.
- This problem arises because rewards are often delayed and may not be directly linked to specific actions, making it difficult for the agent to learn which actions to repeat or avoid in the future.

The main challenges associated with CAP include:

- Delayed Rewards: When rewards are received after a series of actions, determining which specific action(s) led to that reward can be ambiguous.
- Multiple Contributing Actions: In cases where several actions contribute to achieving a reward, distinguishing their individual impacts becomes problematic.
- Exploration vs. Exploitation: The agent must balance between exploring new strategies and exploiting known successful actions based on past experiences

CREDIT ASSIGNMENT PROBLEM

Several approaches have been developed within reinforcement learning frameworks to address CAP effectively:

1. **Temporal Difference (TD) Learning:**

TD learning combines ideas from dynamic programming and Monte Carlo methods. It updates value estimates based on other learned estimates without waiting for final outcomes. By using bootstrapping, TD learning can assign credit to past actions even when immediate rewards are not available.

Concept: Updates value estimates before the episode ends (bootstrapping).

Example: Imagine a chess game. You don't have to wait until checkmate to know if a move was good. If a move improves your position right now (e.g., capturing an opponent's piece), TD learning updates its value immediately based on the new state.

👉 This way, earlier moves also get partial credit even before the final win/lose is known.

How it helps CAP: Assigns credit to recent actions immediately by *bootstrapping* from the estimated value of the next state, even before the episode ends.

2. Monte Carlo Methods:

Monte Carlo methods evaluate policies based on complete episodes of experience. They calculate returns from states by averaging rewards obtained over multiple episodes, allowing agents to assign credit retrospectively once they receive feedback at episode completion.

Concept: Waits until the end of an episode to assign credit (no bootstrapping).

Example:
Think about **playing a full football match**. You only know whether your strategy worked after the referee blows the final whistle. Then you look back at the whole game and assign credit:

1. If your team won, passes and shots during the game get positive credit.
2. If you lost, those same actions get negative credit.

👉 Learning happens only after the entire episode (the match) is finished.

How it helps CAP: Assigns credit retrospectively at the end of an episode by averaging the total return for each visited state-action pair.

3. Eligibility Traces:

Concept: Mixes both TD and Monte Carlo – remembers recent actions with a decaying trace.

Example:
Imagine **teaching a dog a trick (like rolling over)**.

The dog sits, then lies down, then rolls over.

You give a treat **only at the end**.

With **eligibility traces**, the algorithm assigns more credit to the last action (rolling over), but also gives some decayed credit to earlier actions (sitting, lying down) because they led to the reward.
👉 It helps distribute credit smoothly across the sequence of actions.

How it helps CAP:

Provides a **memory of recent actions** with a decaying weight, so when a reward is finally observed, it can be distributed across the whole sequence of contributing actions.

USING POLICY GRADIENTS

- **Credit Assignment Problem Recap**

In RL, rewards often come **much later** than the actions that caused them.

Hard to know **which actions deserve credit (or blame)**.

- **How Policy Gradients Work**

Instead of explicitly figuring out which state/action led to the reward, we directly adjust the **probability distribution over actions** (the policy).

If the final outcome is good → increase probabilities of actions taken.

If the outcome is bad → decrease probabilities of those actions.

- Policy gradient methods **don't try to assign exact credit to each action immediately**. Instead, they:
  - Look at the **whole trajectory** (sequence of states → actions → reward).
  - Use the final outcome (return R) to update the probabilities of actions taken.
  - Actions that occurred in a **successful trajectory** get their probabilities increased.
  - Actions in **unsuccessful trajectories** get their probabilities decreased.

Example (Policy Gradient solving CAP):

CartPole balancing game

The agent must balance a pole on a cart by pushing left/right.

Reward = +1 for every time step the pole stays upright.

Failure = pole falls.

👉 CAP: Which past actions (left/right pushes) deserve credit for balancing longer?

Policy Gradient solution:

- Collect full trajectory (episode).
- Compute total reward (return).
- For each action taken, increase its probability if return was high, decrease if return was low.
- This way, even actions early in the episode get credit if they contributed to balancing successfully.

Formula (REINFORCE — Monte Carlo Policy Gradient)

The update rule: $\nabla J(\theta) \approx \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot G_t\right]$

Where:

- $\pi_\theta(a_t|s_t)$ = probability of action a_t under policy parameters θ.
- $\log \pi_\theta(a_t|s_t)$ = log-probability, tells us how much to adjust action likelihood.
- G_t = return (sum of future rewards from time t).
- The gradient **automatically attributes credit** across all actions proportional to the rewards obtained.

Why Policy Gradients help CAP?

They **bypass the need** to explicitly assign credit to individual actions.

Instead, the gradient scales the probability of all actions in proportion to the eventual return.

High-return episodes → reinforce chosen actions.

Low-return episodes → discourage chosen actions.

Example – Maze Robot 🧩

The robot has a policy network that gives probabilities for moves (up, down, left, right).

Episode 1: Robot goes right → right → down → dead end → reward = −10.

Update: reduce probabilities of those actions.

Episode 2: Robot goes up → right → right → exit → reward = +100.

Update: increase probabilities of those actions.

👉 Even though the reward came **only at the end**, policy gradient pushes credit backward to all actions in the successful trajectory.

This way, the **credit assignment problem** is handled gradually:

- Bad trajectories teach the agent what *not* to do.
- Good trajectories reinforce the *right* sequence of actions.

Policy gradients solve the **credit assignment problem** by using the **final reward signal** to adjust the probabilities of all actions taken in the episode — reinforcing good trajectories and discouraging bad ones.

Imagine a student preparing for exams:

They try **different study strategies** (random policy).

If they get **good marks (reward)**:

- Policy gradient says: “Increase probability of those study habits (practice problems, early revision).”

If they fail:

- Policy gradient says: “Decrease probability of those habits (procrastination, only last-minute study).”

Even though the **reward comes only after the exam**, over multiple exams the student learns **which actions deserve credit**.

MARKOV DECISION PROCESSES (MDPS)

Reinforcement learning problems are often modeled as Markov Decision Processes (MDPs).

A **Markov Decision Process (MDP)** is a mathematical framework used to model decision-making situations where outcomes are partly random and partly under the control of a decision-maker.

MDPs provide a formalism for defining environments in which an agent operates, making them essential in the field of reinforcement learning (RL).

The key components of an MDP include:

1. **States (S)**: The various situations or configurations that the agent can be in.
2. **Actions (A)**: The set of all possible actions that the agent can take.
3. **Transition Probabilities (P)**: The probabilities of moving from one state to another given a specific action.
4. **Rewards (R)**: The immediate return received after transitioning from one state to another due to an action.
5. **Discount Factor (γ)**: A factor between 0 and 1 that determines the importance of future rewards compared to immediate rewards.

MARKOV DECISION PROCESSES (MDPS)

To find optimal policies within MDPs, several algorithms can be employed:

1. **Dynamic Programming:**

Dynamic programming methods such as value iteration and policy iteration systematically evaluate policies and improve them until convergence on an optimal policy is achieved.

2. Monte Carlo Methods:

These methods use random sampling to estimate value functions based on episodes generated by following policies.

3. Temporal-Difference Learning:

Temporal-difference methods combine ideas from dynamic programming and Monte Carlo methods by updating estimates based on other learned estimates without waiting for final outcomes.

MARKOV DECISION PROCESSES (MDPS)

MDPs are widely used across various domains within reinforcement learning:

1. In robotics for navigation tasks where robots must learn optimal paths.
2. In game playing AI where agents learn strategies through trial-and-error interactions with their environment.
3. In finance for portfolio management decisions where agents must balance risk against potential returns over time.

MARKOV DECISION PROCESS- EXAMPLE

Self-Driving Car

States (S):

Red light, Green light, Yellow light, Pedestrian crossing, Empty road.

Actions (A):

Stop, Move, Turn left, Turn right, Slow down.

Transition Probabilities (P):

If state = “Green light” and action = “Move”, then with 0.9 probability → new state = “Empty road”, with 0.1 probability → “Pedestrian crossing”.

Rewards (R):

+100 for reaching destination safely.

−100 for accident.

−1 for waiting too long at signals.

👉 Again, this is an MDP since each new state depends only on **current state + action**, not on how the car got there.

Q LEARNING

Q-Learning is a **model-free reinforcement learning (RL) algorithm** that learns the **optimal action-value function** for a given environment.

Model-free → does not require knowledge of environment dynamics (P(s'|s,a)

Off-policy → learns the optimal policy independently of the agent's actions (e.g., can learn optimal greedy policy while exploring with ε-greedy).

Value-based → learns state-action values, not direct policies (contrast with policy gradient methods).

Theoretical Foundation

i) Markov Decision Process (MDP)

Environment is modeled as an MDP with:

- S: set of states
- A: set of actions
- $P(s'|s,a)$: transition probability
- $R(s,a)$: reward function
- $\gamma \in [0,1]$: discount factor

ii) Action value function:

It’s called the **Q-function** because it assigns a **quality** (Q) to taking action a in state s.

It tells us the **expected total return** if:

The agent starts at state s.

Takes action a.

Then follows policy π\pi afterwards.

👉 In plain words: *“If I do action a in state s and then behave according to π\pi, how much reward will I collect in the long run?”*

Action-Value Function (Q-function)

The **Q-function** under policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \,\middle|\, s_0 = s, a_0 = a\right]$$

- $\mathbb{E}_{\pi}[\cdot]$: Expectation (average) over randomness in environment and the chosen policy π.
- r_{t+1}: reward obtained at time $t+1$.
- γ^t: discount factor ($0 \leq \gamma \leq 1$) reduces the value of distant rewards.
- $\sum_{t=0}^{\infty} \gamma^t r_{t+1}$: the **discounted return** — total reward collected from time 0 onwards.
- Condition $s_0 = s, a_0 = a$: we start in state s and first action is a.

The **optimal Q-function**:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

And it satisfies the **Bellman Optimality Equation**:

$$Q^*(s,a) = \mathbb{E}\Big[R(s,a) + \gamma \max_{a'} Q^*(s',a')\Big]$$

Q-Learning Algorithm

Since environment dynamics are unknown, Q-learning estimates $Q^*(s, a)$ **iteratively**:

Update Rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

Where:

- $\alpha \in (0, 1]$: learning rate
- r_{t+1}: reward after taking action a_t in state s_t
- γ: discount factor
- The term in brackets is the **Temporal Difference (TD) error**.

Q LEARNING ALGORITHMIC STEPS

1. **Initialization**: Create a Q-table initialized with arbitrary values (often zeros). This table will store Q-values for each state-action pair.

2. **Observation**: The agent observes its current state s.

3. **Action Selection**: The agent selects an action a based on its current policy, which may involve exploration (trying new actions) or exploitation (choosing known rewarding actions). A common strategy used here is the epsilon-greedy method, where most actions are chosen based on maximum Q-values but some are chosen randomly.

4. **Environment Interaction**: The agent performs action a, transitioning to a new state s' and receiving a reward r.

5. **Q-value Update**: The Q-value for the previous state-action pair is updated using the formula:

$Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma \max_{a'} Q(s',a')-Q(s,a)]$ where α: Learning rate (determines how much new information overrides old information) , γ: Discount factor (determines the importance of future rewards).

6. **Iteration**: Steps 2-5 are repeated until convergence or until a predefined number of episodes are completed.

7. **Policy Derivation**: After sufficient training, derive an optimal policy from the learned Q-values by selecting actions that yield the highest Q-values in each state.

Q LEARNING- WORKING

1. **Initialize** all Q(s, a) = 0 (or small random values).
2. For **each episode**:
   - Start at the initial state (e.g., `S`).
   - At each step:
     - Choose an action (using ε-greedy: mostly the best action, sometimes random).
     - Execute the action → get reward `r` and next state `s'`.
     - Update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

     - Move to `s'`.
   - Episode ends when terminal state is reached (Goal or Pit).
3. **Repeat episodes** until the Q-values stabilize (i.e., changes between updates become very small).

Q LEARNING EXAMPLE

Setup

Grid world:

```
+---+---+
| S | A |
+---+---+
| B | G |
+---+---+
```

- **S** = Start (top-left)
- **A** = top-right
- **B** = bottom-left
- **G** = Goal (bottom-right)
- Actions = {Up, Down, Left, Right}
- Rewards = -1 for each move, +10 when reaching Goal
- Parameters:
  - Learning rate $\alpha = 0.5$
  - Discount factor $\gamma = 0.9$

We'll update **Q(s,a)** using:

$$Q(s,a) \;\leftarrow\; Q(s,a) + \alpha\Big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big]$$

Episode 1: Agent starts at S.

1. **Action = Right** → moves to **A**, reward = -1.

$$Q(S, Right) = 0 + 0.5\left[-1 + 0.9 \cdot 0 - 0\right] = -0.5$$

Q-table:

State	Up	Down	Left	Right
S	0	0	0	**-0.5**

2. From **A**, **Action = Down** → moves to **G**, reward = +10.

$$Q(A, Down) = 0 + 0.5\left[10 + 0.9 \cdot 0 - 0\right] = 5$$

Q-table:

State	Up	Down	Left	Right
S	0	0	0	-0.5
A	0	**5**	0	0

Episode 2: Start at S again.

1. **Action = Down** → moves to **B**, reward = -1.

$$Q(S, Down) = 0 + 0.5\big[-1 + 0.9 \cdot 0 - 0\big] = -0.5$$

2. From **B**, **Action = Right** → moves to **G**, reward = +10.

$$Q(B, Right) = 0 + 0.5\big[10 + 0.9 \cdot 0 - 0\big] = 5$$

Q-table:

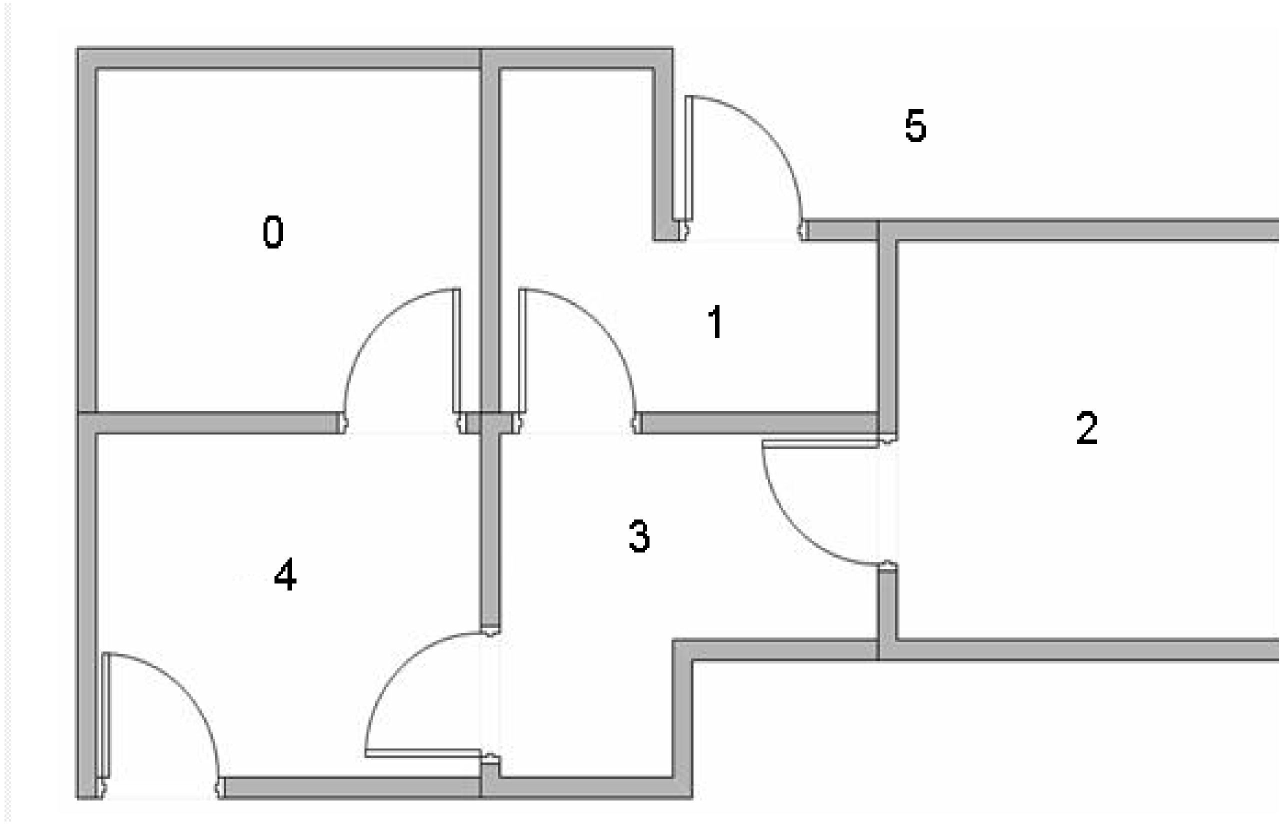
State	Up	Down	Left	Right
S	0	**-0.5**	0	-0.5
A	0	5	0	0
B	0	0	0	**5**

Q LEARNING ALGORITHM CONTD…

- Set the Gamma Parameter and Environment rewards in matrix R
- Initialize matrix Q to Zero
- Select a random initial state
- Set current state=initial state
- Select one among all the possible actions for the current state
- Using this possible action, consider going to the next state
- Get Maximum Q value for this next state based on all the possible actions
- Compute: Q(s, a)=R(s, a)+Gamma * max[Q(next state, all actions)]
- Repeat above steps until current state=goal state

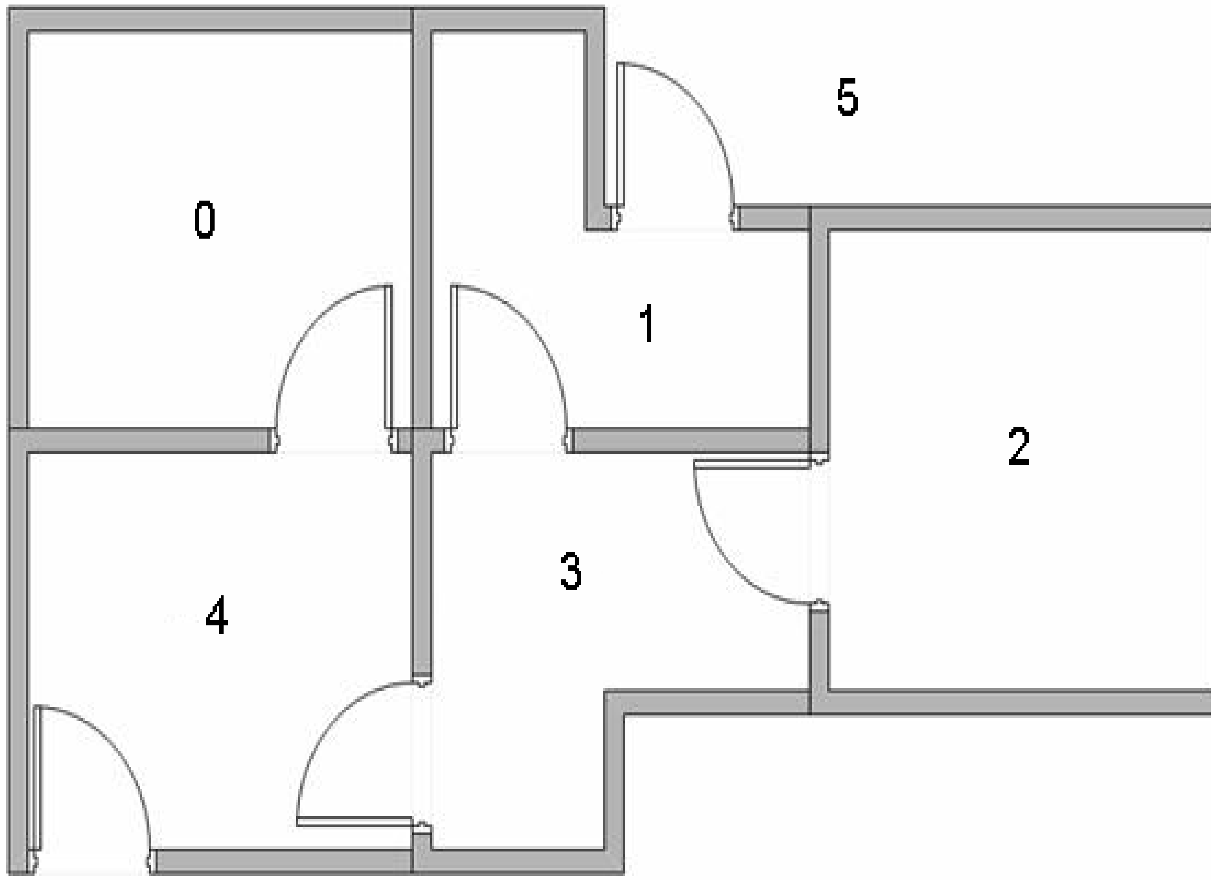
CONSIDER THIS EXAMPLE TO UNDERSTAND Q LEARNING ALGORITHM

- Place an agent in any one of the rooms(0,1,2,3,4) and the goal is to reach outside the building(room 5)



CONSIDER THIS EXAMPLE TO UNDERSTAND Q LEARNING ALGORITHM - GENERATING REWARD MATRIX

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{array}$$



Q MATRIX – REPRESENTING THE MEMORY OF WHAT THE AGENT HAS LEARNT THROUGH EXPERIENCE

Use the Formula-

Q(State, Action) = R(state,action)+Gamma*Max[Q(next state, all actions)]

i. e Q(s, a)=r(s, a)+Gamma*Max[Q(next state, all actions)]

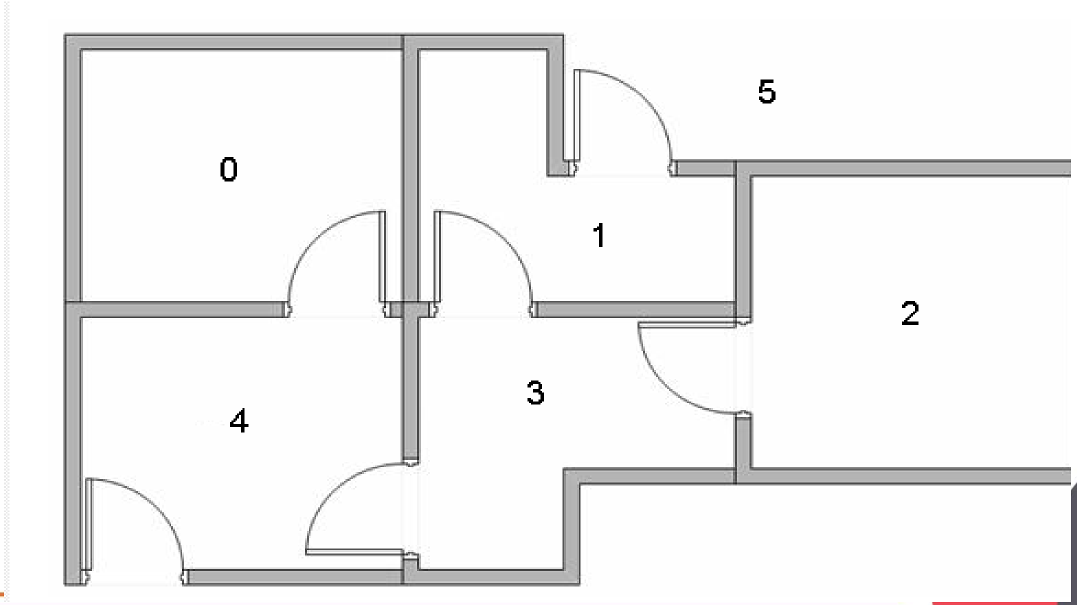
HOW TO CALCULATE THE Q MATRIX ELEMENTS

- set the learning parameter, Gamma=0.8 and the initial state as Room 1
- From Room1, you can go either go to Room 3 or 5, lets select Room 5
- From Room 5, calculate maximum Q Value for this next state based on all the possible actions.
- Q(State, Action)=R(state,action)+Gamma*Max[Q(next state, all actions)]

Q(1,5)=R(1,5)+0.8*Max[Q(5,1),Q(5,4)]

=100+0.8*0

=100



USING DEEP Q-LEARNING TO LEARN HOW TO PLAY PACMAN

Deep Q-Learning (DQN) is a powerful reinforcement learning algorithm that combines Q-learning with deep neural networks, enabling agents to learn optimal policies directly from high-dimensional sensory inputs, such as images and video frames.

1. **State Representation:** Each frame or series of frames from Pac-Man serves as input.
2. **Neural Network Architecture:** A convolutional neural network processes these frames and outputs Q-values for each possible action.
3. **Experience Replay:** To stabilize training, experiences are stored in memory and sampled randomly during updates.
4. **Target Networks:** Two separate networks—the main network and target network—are used to improve stability during training by decoupling action selection from value estimation.

Through repeated gameplay, adjusting weights based on observed rewards, and leveraging experience replay, Deep Q-Learning enables an agent to learn effective strategies for playing Pacman autonomously. In the context of playing Pacman, DQN can effectively learn how to navigate the game environment, avoid ghosts, and collect pellets by leveraging visual information from the game screen.

USING DEEP Q-LEARNING TO LEARN HOW TO PLAY PACMAN

Step 1: Understanding the Game Environment:

- The first step in applying DQN to Pacman is understanding the game mechanics and environment.
- Ms. Pacman is an arcade game where the player controls Ms. Pacman through a maze filled with dots (pellets) while avoiding ghosts.
- The objective is to eat all the pellets while maximizing points by consuming power pellets that turn ghosts vulnerable for a limited time.

Game State Representation :

The state of the game can be represented using frames captured from the game screen. Each frame is a 210x160 RGB image that contains information about Ms. Pacman's position, the locations of pellets, and ghost positions.

Step 2: Preprocessing Input Data: To make it feasible for a neural network to process these images, we need to preprocess them:

- Grayscale Conversion: Convert RGB images to grayscale to reduce complexity.
- Resizing: Resize images to a smaller dimension (e.g., 88x80 pixels) while maintaining important features.
- Normalization: Normalize pixel values to fall within a specific range (e.g., -1 to 1) for better training performance.

USING DEEP Q-LEARNING TO LEARN HOW TO PLAY PACMAN

Step 3: Building the Deep Q-Network:

The core of DQN involves creating a neural network that approximates the Q-value function:

- Convolutional Layers: These layers extract features from input images.
- Fully Connected Layers: These layers combine features learned by convolutional layers and output Q-values for each possible action.

Network Architecture: A typical architecture might include:

- Three convolutional layers followed by max-pooling layers.
- One or more fully connected layers leading to an output layer representing Q-values for each action.

Step 4: Implementing Experience Replay:

Experience replay helps stabilize training by storing past experiences in a replay buffer and sampling random batches during training:

- Store transitions in the format .
- Sample mini-batches from this buffer when updating the network weights.

USING DEEP Q-LEARNING TO LEARN HOW TO PLAY PACMAN

Step 5: Training Process: Training involves several key steps:

- Initialize the environment and reset it.
- For each episode:
  - ✓ Capture frames and preprocess them.
  - ✓ Use an epsilon-greedy policy for action selection (exploration vs exploitation).
  - ✓ Execute actions in the environment and store transitions in memory.
  - ✓ Periodically sample from memory and update network weights based on Bellman's equation.

Epsilon-Greedy Policy: This policy balances exploration (trying new actions) with exploitation (choosing known rewarding actions):

epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)

Step 6: Evaluating Performance:

After training over many episodes (often thousands), evaluate performance by running Ms. Pacman in real-time and observing its ability to maximize scores while avoiding ghosts.

Visualizing Results: You can visualize gameplay using libraries like Matplotlib or OpenAI Gym's built-in rendering capabilities.

```
env.render()
```

RELATIONSHIP TO DYNAMIC PROGRAMMING

Reinforcement Learning: is a type of machine learning where an agent learns to make decisions by interacting with an environment.

- The agent receives feedback in the form of rewards or penalties based on its actions, allowing it to learn the optimal strategy over time.
- It is often used in scenarios where the environment is not fully known or changes dynamically, making it suitable for tasks such as game playing, robotics, and autonomous driving.
- It's key feature is that it involves trial-and-error learning, where the agent explores different actions to understand which ones lead to the best outcomes.

RELATIONSHIP TO DYNAMIC PROGRAMMING

Dynamic Programming: is **a mathematical optimization approach typically used to improvise recursive algorithms and** It basically involves simplifying a large problem into smaller sub-problems.

- It involves solving each subproblem only once and storing the solution to avoid redundant calculations.
- Unlike reinforcement learning, dynamic programming requires a model of the environment or system being studied.
- This model allows for efficient computation of optimal solutions through a process of backward induction.
- It is commonly used in optimization problems where the problem can be divided into overlapping subproblems that can be solved independently.

USING DEEP Q-LEARNING TO LEARN HOW TO PLAY PACMAN

Step 5: Training Process: Training involves several key steps:

- Initialize the environment and reset it.
- For each episode:
  - ✓ Capture frames and preprocess them.
  - ✓ Use an epsilon-greedy policy for action selection (exploration vs exploitation).
  - ✓ Execute actions in the environment and store transitions in memory.
  - ✓ Periodically sample from memory and update network weights based on Bellman’s equation.

Epsilon-Greedy Policy: This policy balances exploration (trying new actions) with exploitation (choosing known rewarding actions):

```
epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
```

Step 6: Evaluating Performance: After training over many episodes (often thousands), evaluate performance by running Ms. Pacman in real-time and observing its ability to maximize scores while avoiding ghosts. Visualizing Results: You can visualize gameplay using libraries like Matplotlib or OpenAI Gym’s built-in rendering capabilities.

```
env.render()
```