# **AI & Agents**

## **Introduction to AI & Agents**

**What is AI?**

Artificial Intelligence (AI) refers to the development of computer systems that can perform tasks that typically require human intelligence, such as learning, problem-solving, decision-making, perception, and reasoning. AI enables machines to mimic cognitive functions, process data, and adapt to new information, often with the goal of automating tasks or automating human capabilities.

AI systems rely on algorithms, data, and computational power to analyze patterns, make predictions, or generate outputs. They are used in diverse applications, from voice assistants and recommendation systems to autonomous vehicles and medical diagnostics.

**Prelude to the History & Evolution of AI**

The concept of AI has evolved over decades, shaped by philosophical ideas, technological advancements, and scientific breakthroughs. Below is a concise overview of its historical development.

1. Philosophical roots (Pre-20th Century)
   * The idea of creating artificial beings with intelligence dates back to ancient myths (e.g., Greek automata) and philosophical debates about the nature of light.
   * In the 17th-19th centuries, thinker like Rene Descartes and Gottfried Leibniz explored the possibility of mechanizing reasoning, laying conceptual groundwork for AI.
2. Early Foundations (1950’s to 1950s):
   * 1943: Warren McCulloch and Walter Pitts proposed a model of artificial neurons, an early precursor to neural networks.
   * 1950: Alan Turing introduced the "Turing Test" to evaluate machine intelligence and published Computing Machinery and Intelligence, formalizing the question, "Can machines think?"
   * 1956: The term "Artificial Intelligence" was coined by John McCarthy at the Dartmouth Conference, marking the birth of AI as a field. The conference aimed to explore how machines could simulate human intelligence.
3. Early AI Systems and Symbolic AI (1960s to 1980s)
   * AI research focused on symbolic or rule-based systems, where human experts encoded knowledge into programs.
   * 1960s: Systems like the Logic Theorist (Herbert Simon and Allen Newell) proved mathematical theorems, and ELIZA (Joseph Weizenbaum) simulated conversation.
   * 1980s: Expert systems, such as MYCIN for medical diagnosis, became prominent. However, limitations in computational power and data led to an "AI winter" (periods of reduced funding and interest).
4. Rise of Machine Learning (1990s to 2000s )
   * AI shifted from rule-based systems to data-driven approaches, particularly machine learning (ML), where systems learn patterns from data.
   * 1997: IBM’s Deep Blue defeated chess champion Garry Kasparov, showcasing AI’s potential in specialized tasks.
   * 2000s: Algorithms like support vector machines and decision trees gained traction, supported by growing computational power and data availability.
5. Deep Learning and Modern AI (2010s to Present)
   * The advent of deep learning, powered by neural networks, large datasets, and GPUs, revolutionized AI.
   * 2012: AlexNet, a deep convolutional neural network, achieved a breakthrough in image recognition, sparking a deep learning boom.
   * 2014–2016: AI systems like AlphaGo (DeepMind) defeated human champions in Go, demonstrating advanced strategic reasoning.
   * 2020s: Large language models (LLMs) like GPT-3 and multimodal AI systems (e.g., DALL·E) enabled natural language processing, image generation, and cross-domain tasks. AI became integral to industries, from healthcare to finance.
6. Current Trends (2025)
   * AI is increasingly autonomous, with systems like AI agents performing complex tasks with minimal human intervention.
   * Ethical concerns, such as bias, privacy, and job displacement, drive research into responsible AI.
   * Advances in neuromorphic computing, quantum computing, and energy-efficient AI aim to push the boundaries of scalability and performance.

The evolution of AI reflects a shift from rigid, rule-based systems to adaptive, learning-based models, fueled by data, computation, and interdisciplinary innovation.

## **What are AI Agents?**

AI agents are autonomous or semi-autonomous systems that perceive their environment, make decisions and take actions to achieve specific goals. Unlike traditional AI models that performs narrowly defined tasks (e.g., image classification), AI agents are designed to interact dynamically with their environment, often in real-time, and adapt to changing conditions.

An AI agent typically consists of:

**Perception:** Sensing or collecting data from the environment (e.g., cameras, microphones or text inputs)

**Reasoning:** Processing data to make decisions, often using algorithms or trained models.

**Action:** Executing tasks, such as moving a robotic arm, generating text, or adjusting system parameters.

**Learning:** Improving performance over time based on feedback or experience (optional, depending on the agent type).

AI agents are used in applications like robotics, virtual assistants, autonomous vehicles and automated workflows.

### **Types of AI Agents**

AI agents are classified based on their complexity, autonomy and decision making capabilities. Below are the main types as commonly categorized in AI literature:

#### **Simple Reflex Agents:**

**Description:** These agents act based on predefined rules or conditions, responding directly to current inputs without considering past or future states.

**Characteristics:**

* No memory or internal state.
* Rule-based: “if condition X, then action Y.”
* Limited to simple environments

**Example:**

* A thermostat that turns on heating when the temperature drops below a threshold
* A spam filter the flags emails based on specific keywords

**Use Case:** Basis automation task with predictable inputs.

#### **Model-Based Reflex Agents:**

**Description:** These agents maintain an internal model of the environment, allowing them to consider the current state and some history to make decisions.

**Characteristics:**

* Uses a model to track how the environment evolves
* Can handle partially observable environments
* More complex than simple reflex agents

**Examples:**

* A vacuum cleaner robot that maps a room to avoid obstacles
* A traffic light system that adjusts timings based on sensor data

**Use Case:** Systems requiring awareness of environmental changes

#### **Goal-Based Agents**

**Description:** These agents make decisions to achieve specific goals, evaluating possible actions based on their outcomes.

**Characteristics:**

* Incorporates planning and search algorithms
* Considers future consequences of actions
* More flexible than reflex agents

**Examples:**

* A navigation app that calculates the fastest route to a destination
* An AI playing chess, aiming to checkmate the opponent.

**Use Case:** Tasks requiring optimization or strategic planning.

#### **Utility-Based Agents**

**Description:** These agents aim to maximize a utility function, which quantifies the desirability of different outcomes, allowing for nuanced decision-making.

**Characteristics:**

* Evaluates trade-offs between multiple goals
* Uses a utility metric (e.g., efficiency, safety, or user satisfaction.)
* Suitable for complex, uncertain environments.

**Examples:**

* An autonomous car balancing speed, safety, and fuel efficiency
* A recommendation system optimizing user engagement and relevance.

**Use Case:** Systems requiring multi-criteria optimization

#### **Learning Agents**

**Description:** These agents improve their performance over time by learning from experience, adapting to new data or feedback

**Characteristics:**

* Incorporates machine learning or reinforcement learning
* Can operate in dynamic, unpredictable environments
* Highly autonomous and adaptive.

**Examples:**

* A chatbot like Deepseek, Grok, ChatGPT, that refines responses based on user interactions.
* A reinforcement learning agent mastering video games (e.g., DeepMind’s AlphaStar)

**Use Case:** Applications requiring continuous improvement or adaption.

#### **Multi-Agent Systems (Emerging Type)**

**Description:** These involve multiple AI agents interacting or collaborating to achieve individual or collective goals.

**Characteristics:**

* Agents may cooperate (e.g., swarm robotics) or compete (e.g., trading bots)
* Requires coordination, communication or negotiation
* Complex due to inter-agent dynamics

**Examples:**

* A fleet of delivery drones coordinating routes
* AI agents in a simulated economy negotiating resources

**Use Case:** Distributed systems or collaborative tasks.

AI is a field dedicated to creating intelligent systems that emulate human cognitive abilities, evolving from philosophical ideas to modern data-driven models like deep learning. Its history spans early symbolic systems, machine learning advancements, and today’s autonomous AI agents.

AI agents are systems that act autonomously to achieve goals, ranging from simple reflex agents to sophisticated learning agents (adaptive). The types – simple reflex, model-based, goal-based, utility-based, learning and multi-agent systems – reflect increasing complexity and autonomy, enabling applications from basic automation to complex decision-making in dynamic environments.

## **Simple Reflex Agents:**

Explain Simple Reflex Agents with an hands on example

**Simple Reflex Agents:**

**Explanation**

Simple reflex agents are the most basic type of AI agent. They operate by directly mapping specific inputs (perceptions from the environment) to predefined actions using condition-action rules (if-then statements). These agents:

* **React to the current state**: They only consider the immediate input or perception, without maintaining memory of past states or planning for future outcomes.
* **Use predefined rules**: Their behavior is governed by a set of hardcoded "if condition, then action" rules.
* **Are limited in scope**: They work best in fully observable, simple, and predictable environments where the correct action depends only on the current input.

**Key Characteristics**:

* No internal state or memory.
* Fast and efficient due to their simplicity.
* Inflexible, as they cannot adapt to new situations or handle complex, dynamic environments.

### **Real-World Analogy**:

A simple reflex agent is like a vending machine. If you insert the correct amount of money and press a button (condition), it dispenses the selected item (action). It doesn’t "remember" previous transactions or plan for future ones.

### **Hands-On Example:** Simple Reflex Agent for a Smart Light Bulb

Let’s design a simple reflex agent for a **smart light bulb** that turns on or off based on the ambient light level detected by a sensor. This example will include a conceptual explanation, pseudocode, and a Python implementation to make it hands-on.

**Scenario**

* **Environment**: A room with a smart light bulb and a light sensor.
* **Perception**: The light sensor measures the ambient light level in lux (a unit of illuminance).
* **Actions**: The light bulb can either turn ON or turn OFF.
* **Rules**:
  + If the ambient light level is below 100 lux (e.g., it’s dark), turn the light ON.
  + If the ambient light level is 100 lux or above (e.g., it’s bright), turn the light OFF.

This is a classic simple reflex agent because the action (ON/OFF) depends only on the current sensor reading, with no memory of past states or future planning.

#### **Step-by-Step Explanation**

1. **Perception**: The agent reads the current light level from the sensor.
2. **Condition-Action Rules**:
   * Rule 1: IF light\_level < 100 lux, THEN turn\_light\_ON.
   * Rule 2: IF light\_level ≥ 100 lux, THEN turn\_light\_OFF.
3. **Action**: The agent sends a command to the light bulb to either turn ON or OFF.
4. **No Memory**: The agent doesn’t track whether the light was previously ON or consider time of day, user preferences, or energy costs.

FUNCTION SimpleReflexLightAgent(light\_level)

IF light\_level < 100 THEN

RETURN "Turn Light ON"

ELSE

RETURN "Turn Light OFF"

END IF

END FUNCTION

MAIN

LOOP

light\_level = ReadSensor()

action = SimpleReflexLightAgent(light\_level)

ExecuteAction(action)

END LOOP

END MAIN

#### **Python Implementation**

Below is a Python program simulating the smart light bulb agent. The program mimics sensor readings with random values for demonstration, but in a real system, it would interface with a physical sensor.

import random

import time

# Simple reflex agent function

def simple\_reflex\_light\_agent(light\_level):

if light\_level < 100:

return "Turn Light ON"

else:

return "Turn Light OFF"

# Simulate the environment and agent loop

def main():

print("Smart Light Bulb Agent Simulation")

print("--------------------------------")

while True:

# Simulate sensor reading (random light level between 0 and 200 lux)

light\_level = random.randint(0, 200)

# Get action from the agent

action = simple\_reflex\_light\_agent(light\_level)

# Display the result

print(f"Light Level: {light\_level} lux | Action: {action}")

# Wait for a second before the next reading

time.sleep(1)

# Optional: Break the loop after a few iterations for demo purposes

# Remove this for continuous operation

if input("Continue? (y/n): ").lower() == 'n':

break

# Run the simulation

if \_\_name\_\_ == "\_\_main\_\_":

main()

**How to Run the Code**

1. Copy the Python code into a .py file (e.g., smart\_light\_agent.py).
2. Ensure you have Python installed (version 3.x).
3. Run the script using a terminal or IDE: python smart\_light\_agent.py.
4. The program will:
   * Generate random light levels (simulating sensor data).
   * Apply the reflex rules to decide whether to turn the light ON or OFF.
   * Print the light level and action every second.
   * Prompt you to continue or stop the simulation.

**Sample Output**

Smart Light Bulb Agent Simulation

--------------------------------

Light Level: 85 lux | Action: Turn Light ON

Light Level: 120 lux | Action: Turn Light OFF

Light Level: 45 lux | Action: Turn Light ON

Light Level: 150 lux | Action: Turn Light OFF

Continue? (y/n): n

**Why This is a Simple Reflex Agent**

* **Direct Mapping**: The agent maps the light level (input) to an action (ON/OFF) using fixed rules.
* **No Memory**: It doesn’t consider whether the light was ON previously or track patterns over time.
* **No Planning**: It doesn’t account for future states, like saving energy or adjusting brightness.
* **Simple Environment**: The decision depends only on the current light level, assuming the sensor provides accurate, immediate data.

**Limitations of This Agent**

* **Inflexibility**: It can’t adapt to new conditions (e.g., user preferences for dimming instead of ON/OFF).
* **No Context**: It doesn’t consider time of day, occupancy, or energy costs.
* **Fully Observable Assumption**: If the sensor fails or the environment is noisy, the agent may make incorrect decisions.

To address these limitations, a more advanced agent (e.g., a model-based or learning agent) would be needed, which could maintain an internal state, learn from user behavior, or optimize for multiple factors.

**Extending the Example (Optional)**

If you want to make the example more interactive or realistic, here are ideas to try:

* **Add a Physical Sensor**: If you have a microcontroller (e.g., Raspberry Pi) and a light sensor, modify the code to read real sensor data.
* **Expand Rules**: Add more conditions, like turning the light ON only if it’s nighttime (requires a clock input).
* **Simulate Energy Usage**: Track how often the light is ON to estimate power consumption.

## **Model-Based Reflex Agents:**

**Model-Based Reflex Agent** is an intelligent agent that makes decisions based on a model of the world, combining current perceptions with an internal state to choose actions. Unlike simple reflex agents, which react solely to current percepts using condition-action rules, model-based reflex agents maintain an internal state to track aspects of the environment not directly observable. This internal state is updated based on the agent’s understanding of how the world evolves (transition model) and how its actions affect the world. The agent uses this state, along with its percepts, to select actions from a predefined set of rules.

### **Key Components**

1. **Percepts**: Inputs from the environment (e.g., sensor data).
2. **Internal State**: A representation of the world, including past and current information.
3. **Transition Model**: Knowledge of how the world changes over time or due to actions.
4. **Rules**: Condition-action mappings that consider both percepts and internal state to select actions.
5. **Actions**: Outputs that affect the environment.

### How It Works

* The agent perceives the environment through sensors.
* It updates its internal state based on the current percept and its transition model.
* It evaluates rules that match the combination of the internal state and percept.
* It selects and performs an action based on the matched rule.

### **Hands-On Example: Vacuum Cleaner Agent**

Let’s implement a simple model-based reflex agent for a vacuum cleaner in a two-room environment (Room A and Room B). The vacuum can move between rooms and clean dirt. The agent maintains an internal state to track which rooms are clean or dirty, even when it only perceives its current room.

#### **Scenario**

* **Environment**: Two rooms (A and B). Each can be clean or dirty.
* **Percepts**: The agent can sense:
  + Its current location (Room A or B).
  + Whether the current room is dirty.
* **Actions**: Suck (clean the current room), Move Left (to Room A), Move Right (to Room B).
* **Internal State**: Tracks the cleanliness of both rooms (e.g., {A: Clean, B: Dirty}).
* **Goal**: Keep both rooms clean.

#### **Agent Logic**

* Update the internal state based on the current percept (e.g., if Room A is dirty, update state to reflect this).
* Use rules to decide the action:
  + If the current room is dirty, suck.
  + If the current room is clean and the other room is dirty, move to the other room.
  + If both rooms are clean, do nothing (or stop).

#### **Python Implementation**

Below is a Python program simulating the vacuum cleaner agent.

class VacuumAgent:

def \_\_init\_\_(self):

# Internal state: tracks cleanliness of rooms

self.state = {'A': 'Unknown', 'B': 'Unknown'}

# Current location of the agent

self.location = 'A'

def update\_state(self, percept):

# Percept: {'location': 'A' or 'B', 'status': 'Dirty' or 'Clean'}

self.location = percept['location']

self.state[self.location] = percept['status']

def choose\_action(self):

# Rule-based decision making

if self.state[self.location] == 'Dirty':

return 'Suck'

elif self.location == 'A' and self.state['B'] == 'Dirty':

return 'Move Right'

elif self.location == 'B' and self.state['A'] == 'Dirty':

return 'Move Left'

else:

return 'NoOp' # Do nothing if both rooms are clean or unknown

def act(self, percept):

# Update state and choose action

self.update\_state(percept)

action = self.choose\_action()

return action

# Simulate the environment and agent

def simulate\_vacuum():

agent = VacuumAgent()

# Example sequence of percepts

percepts = [

{'location': 'A', 'status': 'Dirty'}, # Room A is dirty

{'location': 'A', 'status': 'Clean'}, # After sucking, A is clean

{'location': 'B', 'status': 'Dirty'}, # Move to B, B is dirty

{'location': 'B', 'status': 'Clean'}, # After sucking, B is clean

]

print("Initial state:", agent.state)

for percept in percepts:

action = agent.act(percept)

print(f"Percept: {percept}, Action: {action}, State: {agent.state}")

# Run the simulation

simulate\_vacuum()

**Output**

Initial state: {'A': 'Unknown', 'B': 'Unknown'}

Percept: {'location': 'A', 'status': 'Dirty'}, Action: Suck, State: {'A': 'Dirty', 'B': 'Unknown'}

Percept: {'location': 'A', 'status': 'Clean'}, Action: NoOp, State: {'A': 'Clean', 'B': 'Unknown'}

Percept: {'location': 'B', 'status': 'Dirty'}, Action: Suck, State: {'A': 'Clean', 'B': 'Dirty'}

Percept: {'location': 'B', 'status': 'Clean'}, Action: NoOp, State: {'A': 'Clean', 'B': 'Clean'}

**Explanation of Execution**

1. **First Percept**: The agent is in Room A, which is dirty. It updates the state to {A: Dirty, B: Unknown} and chooses to suck.
2. **Second Percept**: Room A is now clean (after sucking). The state updates to {A: Clean, B: Unknown}. Since B’s status is unknown and A is clean, the agent does nothing (NoOp).
3. **Third Percept**: The agent moves to Room B (simulated by the environment), which is dirty. The state updates to {A: Clean, B: Dirty}, and the agent chooses to suck.
4. **Fourth Percept**: Room B is now clean. The state updates to {A: Clean, B: Clean}, and the agent does nothing (NoOp).

**Why Model-Based?**

* The agent maintains an internal state ({A: Clean, B: Dirty}) to track the cleanliness of both rooms, even though it can only perceive the current room.
* It uses this state to make informed decisions, like moving to Room B when A is clean and B is dirty, unlike a simple reflex agent that would only react to the current room’s status.

**Advantages**

* Handles partial observability by maintaining a model of the world.
* Makes informed decisions based on both current percepts and historical knowledge.

**Limitations**

* Requires an accurate transition model and initial state assumptions.
* Can be computationally expensive for complex environments with large state spaces.
* Still relies on predefined rules, limiting adaptability compared to learning agents.

This example demonstrates how a model-based reflex agent uses an internal state to make decisions in a partially observable environment, providing a practical hands-on understanding of the concept.

## **Goal-Based Agents**

A **Goal-Based Agent** is a type of intelligent agent in artificial intelligence that makes decisions based on evaluating actions that help achieve a specific goal. Unlike simpler agents (e.g., reflex agents), goal-based agents consider future outcomes and select actions that bring them closer to their desired state. They use a model of the environment, a goal to achieve, and a search or planning mechanism to determine the best course of action.

### **Key Components of a Goal-Based Agent**

1. **Sensors**: Perceive the current state of the environment.
2. **Knowledge Base**: Represents the agent's understanding of the environment and how actions affect it.
3. **Goal**: A desired state or outcome the agent aims to achieve.
4. **Decision-Making Mechanism**: Evaluates possible actions by predicting their outcomes and selecting those that lead toward the goal.
5. **Actuators**: Execute the chosen actions in the environment.

### **How It Works**

* The agent perceives the environment through sensors.
* It compares the current state to the goal.
* Using a model of the environment, it predicts the outcomes of possible actions.
* It selects the action that minimizes the "distance" to the goal (often using search algorithms or heuristics).
* The action is executed, and the process repeats until the goal is achieved.

### **Hands-On Example: A Goal-Based Agent for a Simple Maze Solver**

Let’s implement a goal-based agent that navigates a 2D maze to reach a target position using Python. The maze is represented as a grid, where:

* 0 represents an open path.
* 1 represents a wall.
* The agent starts at a given position and aims to reach a goal position.

#### **Problem Description**

* **Environment**: A 5x5 maze grid.
* **Goal**: Move from the start position (0,0) to the goal position (4,4).
* **Actions**: Move up, down, left, or right (if not blocked by a wall or out of bounds).
* **Decision Mechanism**: Use a Breadth-First Search (BFS) algorithm to find the shortest path to the goal.

#### **Python Implementation**

from collections import deque

# Define the maze (0 = open, 1 = wall)

maze = [

[0, 0, 0, 1, 0],

[1, 1, 0, 1, 0],

[0, 0, 0, 0, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 1, 0]

]

# Define start and goal positions

start = (0, 0) # (row, col)

goal = (4, 4)

# Possible actions: up, right, down, left

actions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

action\_names = ['Up', 'Right', 'Down', 'Left']

# Goal-Based Agent using BFS

def goal\_based\_agent(maze, start, goal):

rows, cols = len(maze), len(maze[0])

# Check if position is valid

def is\_valid(pos):

r, c = pos

return 0 <= r < rows and 0 <= c < cols and maze[r][c] == 0

# BFS to find the shortest path

queue = deque([(start, [])]) # (position, path)

visited = set([start])

while queue:

(current\_r, current\_c), path = queue.popleft()

# Check if goal is reached

if (current\_r, current\_c) == goal:

return path

# Explore possible actions

for i, (dr, dc) in enumerate(actions):

next\_r, next\_c = current\_r + dr, current\_c + dc

next\_pos = (next\_r, next\_c)

if is\_valid(next\_pos) and next\_pos not in visited:

visited.add(next\_pos)

queue.append((next\_pos, path + [action\_names[i]]))

return None # No path found

# Run the agent

path = goal\_based\_agent(maze, start, goal)

if path:

print("Path to goal:", path)

else:

print("No path found!")

# Simulate the agent's movement

def simulate\_movement(maze, start, path):

current = start

print("Starting at:", current)

for action in path:

if action == 'Up':

current = (current[0] - 1, current[1])

elif action == 'Right':

current = (current[0], current[1] + 1)

elif action == 'Down':

current = (current[0] + 1, current[1])

elif action == 'Left':

current = (current[0], current[1] - 1)

print(f"Move {action} to: {current}")

if path:

simulate\_movement(maze, start, path)

#### **Explanation of the Code**

1. **Environment Representation**:
   * The maze is a 5x5 grid where 0 is a free cell and 1 is a wall.
   * The agent starts at (0,0) and aims to reach (4,4).
2. **Agent Logic**:
   * The goal\_based\_agent function uses BFS to explore the maze.
   * It maintains a queue of positions and the paths taken to reach them.
   * For each position, it evaluates possible moves (up, right, down, left).
   * A move is valid if it stays within bounds and lands on a free cell (0).
   * The agent stops when it reaches the goal, returning the sequence of actions.
3. **Simulation**:
   * The simulate\_movement function traces the path, showing how the agent moves step-by-step from the start to the goal.

**Example Output**

For the given maze, the output might look like:

Path to goal: ['Right', 'Right', 'Down', 'Down', 'Down', 'Right', 'Right']

Starting at: (0, 0)

Move Right to: (0, 1)

Move Right to: (0, 2)

Move Down to: (1, 2)

Move Down to: (2, 2)

Move Down to: (3, 2)

Move Right to: (3, 3)

Move Right to: (3, 4)

Move Down to: (4, 4)

**How This Demonstrates a Goal-Based Agent**

* **Goal**: Reach position (4,4).
* **Perception**: The agent "senses" the maze layout and its current position.
* **Decision-Making**: BFS evaluates all possible paths, selecting the one that achieves the goal efficiently.
* **Action**: The agent executes moves (up, right, down, left) to navigate the maze.
* **Future Consideration**: The agent plans the entire path by considering future states, not just immediate moves.

**Real-World Analogy**

A goal-based agent is like a GPS navigation system:

* **Goal**: Reach a destination.
* **Environment**: Roads, traffic, and obstacles.
* **Actions**: Turn left, right, go straight, etc.
* **Decision**: The GPS evaluates possible routes and selects the one that minimizes time or distance to the destination.

This example illustrates how goal-based agents plan and act to achieve specific objectives, making them suitable for tasks requiring foresight and strategic decision-making.

## **Utility-Based Agents**

A **Utility-Based Agent** is an intelligent agent in artificial intelligence that makes decisions by evaluating possible actions based on a **utility function**. The utility function assigns a numerical value (utility) to each possible outcome, reflecting its desirability. The agent selects the action that maximizes the expected utility, balancing goals, preferences, and uncertainties.

### Key Characteristics

* **Goal**: Maximize expected utility.
* **Components**:
  + **Utility Function**: Maps states or outcomes to a real number indicating desirability.
  + **Decision-Making**: Chooses actions based on the highest utility, considering probabilities if outcomes are uncertain.
  + **Environment**: Typically complex, with multiple possible outcomes and trade-offs.
* **Use Case**: Suitable for scenarios where outcomes have varying degrees of preference, unlike simple goal-based agents that aim for a binary success/failure.

### **Hands-On Example: Autonomous Vacuum Cleaner**

Let’s design a simple utility-based agent for an **autonomous vacuum cleaner** navigating a room. The vacuum cleaner must decide its next action (e.g., move forward, turn left, turn right, or clean) based on a utility function that considers factors like dirt cleaned, battery usage, and avoiding obstacles.

#### **Scenario**

* **Environment**: A 3x3 grid room with dirt in some cells and walls at boundaries.
* **States**: The vacuum’s position, battery level, and dirt status of each cell.
* **Actions**: Move forward, turn left, turn right, clean.
* **Utility Function**: Combines:
  + +10 for each cell cleaned.
  + -1 for each unit of battery consumed per action.
  + -50 for hitting a wall (obstacle penalty).
* **Objective**: Maximize utility by cleaning as much dirt as possible while conserving battery and avoiding walls.

#### **Python Implementation**

Below is a simplified Python simulation of the vacuum cleaner agent.

import random

class VacuumAgent:

def \_\_init\_\_(self, grid\_size=3, battery=20):

self.grid\_size = grid\_size

self.position = [0, 0] # Starting at (0,0)

self.battery = battery

self.direction = 'up' # Possible: up, right, down, left

# Grid: 0 = clean, 1 = dirty

self.grid = [[random.choice([0, 1]) for \_ in range(grid\_size)] for \_ in range(grid\_size)]

self.grid[0][0] = 0 # Starting position is clean

self.total\_utility = 0

def is\_valid\_position(self, pos):

return 0 <= pos[0] < self.grid\_size and 0 <= pos[1] < self.grid\_size

def get\_next\_position(self):

if self.direction == 'up':

return [self.position[0] - 1, self.position[1]]

elif self.direction == 'right':

return [self.position[0], self.position[1] + 1]

elif self.direction == 'down':

return [self.position[0] + 1, self.position[1]]

elif self.direction == 'left':

return [self.position[0], self.position[1] - 1]

def calculate\_utility(self, action):

utility = -1 # Battery cost for any action

if action == 'clean' and self.grid[self.position[0]][self.position[1]] == 1:

utility += 10 # Reward for cleaning dirt

elif action == 'move':

next\_pos = self.get\_next\_position()

if not self.is\_valid\_position(next\_pos):

utility -= 50 # Penalty for hitting a wall

return utility

def choose\_action(self):

actions = ['move', 'clean', 'turn\_left', 'turn\_right']

utilities = [self.calculate\_utility(action) for action in actions]

max\_utility = max(utilities)

best\_actions = [actions[i] for i in range(len(actions)) if utilities[i] == max\_utility]

return random.choice(best\_actions) # Randomly pick among best actions

def perform\_action(self, action):

if self.battery <= 0:

print("Battery depleted!")

return False

self.battery -= 1

if action == 'clean' and self.grid[self.position[0]][self.position[1]] == 1:

self.grid[self.position[0]][self.position[1]] = 0

self.total\_utility += 10

print(f"Cleaned at {self.position}, Utility: +10")

elif action == 'move':

next\_pos = self.get\_next\_position()

if self.is\_valid\_position(next\_pos):

self.position = next\_pos

print(f"Moved to {self.position}")

else:

self.total\_utility -= 50

print(f"Hit wall at {self.position}, Utility: -50")

elif action == 'turn\_left':

directions = ['up', 'left', 'down', 'right']

self.direction = directions[(directions.index(self.direction) + 1) % 4]

print(f"Turned left, facing {self.direction}")

elif action == 'turn\_right':

directions = ['up', 'right', 'down', 'left']

self.direction = directions[(directions.index(self.direction) + 1) % 4]

print(f"Turned right, facing {self.direction}")

self.total\_utility -= 1 # Battery cost

return True

def display\_grid(self):

for i in range(self.grid\_size):

row = ''

for j in range(self.grid\_size):

if [i, j] == self.position:

row += 'V '

else:

row += str(self.grid[i][j]) + ' '

print(row)

print(f"Battery: {self.battery}, Total Utility: {self.total\_utility}")

# Simulate the agent

agent = VacuumAgent()

steps = 10

for \_ in range(steps):

agent.display\_grid()

if agent.battery <= 0:

break

action = agent.choose\_action()

print(f"Chosen action: {action}")

agent.perform\_action(action)

print("-" \* 20)

#### **Explanation of the Code**

1. **Initialization**:
   * The vacuum starts at position (0,0) with a 3x3 grid, some cells randomly dirty (1) or clean (0).
   * Battery is set to 20 units, and the agent faces "up."
2. **Utility Function**:
   * Cleaning a dirty cell: +10 utility.
   * Any action: -1 utility (battery cost).
   * Hitting a wall: -50 utility.
3. **Decision-Making**:
   * The agent evaluates the utility of each action (move, clean, turn left, turn right).
   * It selects the action with the highest utility (randomly among ties).
4. **Actions**:
   * **Move**: Attempts to move in the current direction, penalized if it hits a wall.
   * **Clean**: Cleans the current cell if dirty, earning utility.
   * **Turn Left/Right**: Changes the direction without moving.
5. **Simulation**:
   * The agent runs for 10 steps or until the battery is depleted.
   * It prints the grid, chosen action, and updates (position, utility, battery).

**Sample Output**

0 1 0

1 0 1

0 1 V

Battery: 20, Total Utility: 0

Chosen action: clean

Cleaned at [2, 2], Utility: +10

--------------------

0 1 0

1 0 1

0 1 V

Battery: 19, Total Utility: 9

Chosen action: move

Moved to [1, 2]

--------------------

0 1 0

1 0 V

0 1 0

Battery: 18, Total Utility: 8

Chosen action: clean

--------------------

**Why Utility-Based?**

* The vacuum doesn’t just aim to clean (like a goal-based agent) but evaluates trade-offs:
  + Cleaning is valuable but costs battery.
  + Moving risks hitting walls, which is heavily penalized.
  + Turning adjusts direction but doesn’t directly clean.
* The utility function guides the agent to prioritize cleaning when possible, avoid walls, and conserve battery.

**Real-World Applications**

* **Robotics**: Autonomous drones choosing paths to maximize coverage while minimizing fuel use.
* **Finance**: Trading bots maximizing profit while minimizing risk.
* **Gaming**: NPCs deciding actions to balance health, resources, and objectives.

This example demonstrates how a utility-based agent evaluates multiple factors to make optimal decisions in a dynamic environment. You can extend this by adding more complex utilities (e.g., time penalties, dirt priorities) or a larger grid.

## **Learning Agents**

A **Learning Agent** in artificial intelligence is an agent that improves its performance over time by learning from experience. Unlike utility-based agents that rely on a predefined utility function, learning agents adapt their behavior based on interactions with the environment, using feedback to refine their decision-making process. They are particularly useful in dynamic or uncertain environments where explicit programming of all scenarios is impractical.

### **Key Components of a Learning Agent**

1. **Performance Element**: Selects actions based on current knowledge (similar to a utility-based agent).
2. **Learning Element**: Updates the agent's knowledge based on feedback (e.g., rewards, errors).
3. **Critic**: Evaluates the agent's actions, providing feedback on how well they align with goals.
4. **Problem Generator**: Suggests new actions or experiments to explore the environment and improve learning.

### **Types of Learning**

* **Supervised Learning**: Learning from labeled data (e.g., input-output pairs).
* **Unsupervised Learning**: Finding patterns in unlabeled data.
* **Reinforcement Learning**: Learning through trial and error, maximizing a reward signal.

For this hands-on example, we’ll focus on a **reinforcement learning agent** using Q-learning, a popular algorithm for learning optimal actions in a dynamic environment.

### **Hands-On Example: Grid World Navigation**

We’ll create a learning agent that navigates a 4x4 grid world to reach a goal while avoiding obstacles. The agent learns an optimal policy using Q-learning, updating its knowledge based on rewards received from the environment.

**Scenario**

* **Environment**: A 4x4 grid where:
  + The agent starts at position (0,0).
  + The goal is at (3,3) with a reward of +100.
  + Obstacles (e.g., walls) at specific cells yield a reward of -50.
  + Each move costs a small penalty (-1) to encourage efficiency.
* **Actions**: Move up, down, left, or right.
* **Learning**: The agent uses Q-learning to update a Q-table, which stores the expected future rewards for each state-action pair.
* **Objective**: Learn a policy to reach the goal efficiently, avoiding obstacles.

#### **Python Implementation**

Below is a Python implementation of the Q-learning agent navigating the grid world.

import numpy as np

import random

class GridWorldAgent:

def \_\_init\_\_(self, grid\_size=4, alpha=0.1, gamma=0.9, epsilon=0.1):

self.grid\_size = grid\_size

self.state = [0, 0] # Start at (0,0)

self.actions = ['up', 'down', 'left', 'right']

self.q\_table = np.zeros((grid\_size, grid\_size, len(self.actions))) # Q-table: state-action values

self.alpha = alpha # Learning rate

self.gamma = gamma # Discount factor

self.epsilon = epsilon # Exploration rate

self.goal = [3, 3] # Goal position

self.obstacles = [[1, 1], [2, 1], [1, 3]] # Wall positions

def is\_valid\_state(self, state):

return 0 <= state[0] < self.grid\_size and 0 <= state[1] < self.grid\_size

def get\_next\_state(self, action):

next\_state = self.state.copy()

if action == 'up':

next\_state[0] -= 1

elif action == 'down':

next\_state[0] += 1

elif action == 'left':

next\_state[1] -= 1

elif action == 'right':

next\_state[1] += 1

return next\_state if self.is\_valid\_state(next\_state) else self.state

def get\_reward(self, state):

if state == self.goal:

return 100 # Reward for reaching the goal

if state in self.obstacles:

return -50 # Penalty for hitting an obstacle

return -1 # Small penalty for each move

def choose\_action(self):

# Epsilon-greedy policy: explore or exploit

if random.random() < self.epsilon:

return random.choice(self.actions) # Explore: random action

else:

state\_idx = tuple(self.state)

return self.actions[np.argmax(self.q\_table[state\_idx])] # Exploit: best action

def update\_q\_table(self, action, reward, next\_state):

action\_idx = self.actions.index(action)

current\_q = self.q\_table[tuple(self.state)][action\_idx]

next\_max\_q = np.max(self.q\_table[tuple(next\_state)])

# Q-learning update rule

self.q\_table[tuple(self.state)][action\_idx] = current\_q + self.alpha \* (

reward + self.gamma \* next\_max\_q - current\_q

)

def step(self):

# Choose and perform an action

action = self.choose\_action()

next\_state = self.get\_next\_state(action)

reward = self.get\_reward(next\_state)

self.update\_q\_table(action, reward, next\_state)

if next\_state not in self.obstacles: # Don't move into obstacles

self.state = next\_state

return reward, self.state == self.goal

def display\_grid(self):

for i in range(self.grid\_size):

row = ''

for j in range(self.grid\_size):

if [i, j] == self.state:

row += 'A ' # Agent

elif [i, j] == self.goal:

row += 'G ' # Goal

elif [i, j] in self.obstacles:

row += 'X ' # Obstacle

else:

row += '. ' # Empty

print(row)

# Simulate the agent

agent = GridWorldAgent()

episodes = 1000

max\_steps = 50

for episode in range(episodes):

agent.state = [0, 0] # Reset to start

total\_reward = 0

for step in range(max\_steps):

reward, done = agent.step()

total\_reward += reward

if done: # Reached the goal

break

if episode % 100 == 0:

print(f"Episode {episode}, Total Reward: {total\_reward}")

agent.display\_grid()

print("-" \* 20)

# Test the learned policy

print("Testing learned policy:")

agent.epsilon = 0 # Disable exploration

agent.state = [0, 0]

steps = 0

while agent.state != agent.goal and steps < max\_steps:

agent.display\_grid()

action = agent.choose\_action()

print(f"Action: {action}")

reward, done = agent.step()

steps += 1

print("Final state:")

agent.display\_grid()

#### **Explanation of the Code**

1. **Initialization**:
   * The agent starts at (0,0) in a 4x4 grid.
   * The Q-table is a 3D array storing Q-values for each state (x, y) and action.
   * Hyperparameters:
     + alpha (0.1): Learning rate for Q-table updates.
     + gamma (0.9): Discount factor for future rewards.
     + epsilon (0.1): Exploration probability for epsilon-greedy policy.
   * Goal at (3,3), obstacles at (1,1), (2,1), (1,3).
2. **Q-Learning Algorithm**:
   * **Choose Action**: Uses an epsilon-greedy policy to balance exploration (random actions) and exploitation (best-known action from Q-table).
   * **Perform Action**: Moves to the next state unless it’s invalid (grid boundary).
   * **Reward**: +100 for the goal, -50 for obstacles, -1 for each move.
   * **Update Q-Table**: Updates Q-value using the Q-learning formula: Q(s,a)←Q(s,a)+α⋅(r+γ⋅max⁡Q(s′,a′)−Q(s,a))Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (r + \gamma \cdot \max Q(s', a') - Q(s, a))Q(s,a)←Q(s,a)+α⋅(r+γ⋅maxQ(s′,a′)−Q(s,a)) where sss is the current state, aaa is the action, rrr is the reward, s′s's′ is the next state, and a′a'a′ is the next action.
3. **Training**:
   * Runs for 1000 episodes, each with up to 50 steps.
   * The agent resets to (0,0) at the start of each episode.
   * Prints the grid and total reward every 100 episodes.
4. **Testing**:
   * Sets epsilon = 0 to use the learned policy (no exploration).
   * Demonstrates the agent’s path from (0,0) to the goal.

**Sample Output**

Episode 0, Total Reward: -73

A . . .

. X . X

. X . .

. . . G

--------------------

Episode 100, Total Reward: 92

A . . .

. X . X

. X . .

. . . G

--------------------

...

Testing learned policy:

A . . .

. X . X

. X . .

. . . G

Action: right

. A . .

. X . X

. X . .

. . . G

...

Final state:

. . . .

. X . X

. X . .

. . . A

**Why Learning Agent?**

* **Adaptability**: The agent starts with no knowledge (zeroed Q-table) and learns an optimal path through trial and error.
* **Dynamic Environments**: Q-learning handles uncertainty (e.g., obstacles) without requiring a predefined utility function.
* **Improvement Over Time**: Early episodes show negative rewards (hitting obstacles), but later episodes yield high rewards as the agent learns to reach the goal efficiently.

**Key Observations**

* **Learning Element**: The Q-table updates reflect the learning process, storing expected rewards for state-action pairs.
* **Critic**: The reward function (+100, -50, -1) evaluates actions, guiding the agent toward the goal.
* **Performance Element**: The epsilon-greedy policy selects actions, improving as the Q-table converges.
* **Problem Generator**: Random exploration (via epsilon) ensures the agent tries new paths to discover the goal.

**Real-World Applications**

* **Robotics**: Robots learning to navigate warehouses or homes.
* **Gaming**: NPCs learning strategies in dynamic game environments.
* **Autonomous Vehicles**: Cars learning to optimize routes while avoiding obstacles.
* **Recommendation Systems**: Learning user preferences through interaction feedback.

**Comparison with Utility-Based Agents**

* **Utility-Based Agent** (from your previous query):
  + Uses a fixed utility function (e.g., +10 for cleaning, -50 for walls).
  + No learning; decisions are static based on predefined values.
  + Suitable for well-defined environments with known trade-offs.
* **Learning Agent**:
  + Starts with no knowledge and learns utility (Q-values) from experience.
  + Adapts to changes in the environment (e.g., new obstacles).
  + Ideal for complex or dynamic environments where manual utility design is challenging.

This example illustrates how a learning agent improves its behavior over time using reinforcement learning. You can extend it by adding more obstacles, a larger grid, or a more complex reward structure to simulate real-world scenarios.