Artificial Intelligence

Lab File

Name – Tushya Gupta
Roll number – UE233106
Group – 6

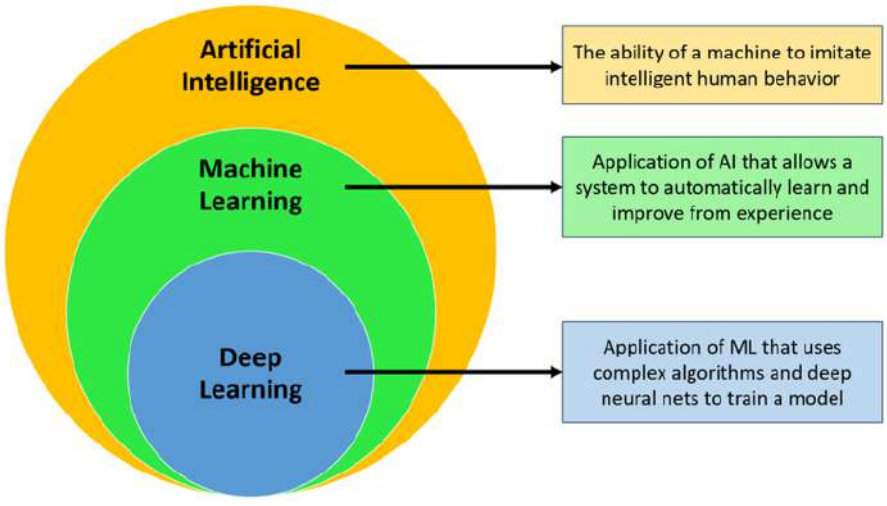
Lab 1

Q1. Introduction to Artificial Intelligence.

Artificial Intelligence
Artificial intelligence is the capability of computational systems to perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making. It is a field of research in computer science that develops and studies methods and software that enable machines to perceive their environment and use learning and intelligence to take actions that maximize their chances of achieving defined goals.

Machine Learning
Machine learning (ML) allows computers to learn and make decisions without being explicitly programmed. It involves feeding data into algorithms to identify patterns and make predictions on new data. It is used in various applications like image recognition, speech processing, language translation, recommender systems, etc.

Deep Learning
In machine learning, deep learning focuses on utilizing multilayered neural networks to perform tasks such as classification, regression, and representation learning. The field takes inspiration from biological neuroscience and is centered around stacking artificial neurons into layers and "training" them to process data. The adjective "deep" refers to the use of multiple layers in the network. Methods used can be supervised, semi-supervised or unsupervised.



Different types of ML algorithms

- Supervised machine learning - Supervised machine learning is a type of machine learning that uses labeled data to train models, allowing them to learn the relationship between input features and output labels. This approach is commonly used for tasks like classification and regression, where the model predicts outcomes based on new, unseen data.
- Unsupervised machine learning - Unsupervised learning is a branch of machine learning that deals with unlabeled data. Unlike supervised learning, where the data is labeled with a specific category or outcome, unsupervised learning algorithms are tasked with finding patterns and relationships within the data without any prior knowledge of the data's meaning. Unsupervised machine learning algorithms find hidden patterns and data without any human intervention and is usually seen in clustering algorithms.
- Reinforced machine learning - 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. This type of machine learning is mostly seen in video games.

History of machine learning

- **1950s**: Foundations laid
  Alan Turing introduces the idea of the Turing Test to measure machine intelligence. Early research explores symbolic AI and logic-based systems, focusing on reasoning through formal rules and symbolic representations of knowledge.
- **1956**: Dartmouth Conference
  The term Artificial Intelligence is formally coined at the Dartmouth Summer Research Project on AI. Researchers concentrate on problem-solving, theorem proving, and symbolic reasoning, establishing AI as a distinct field of study.
- **1960s–1970s**: Expert systems and limitations
  Development of expert systems such as DENDRAL (chemistry) and MYCIN (medical diagnosis). These systems use rule-based reasoning to mimic human decision-making in narrow domains.

Progress is constrained by limited computational resources and lack of large datasets.

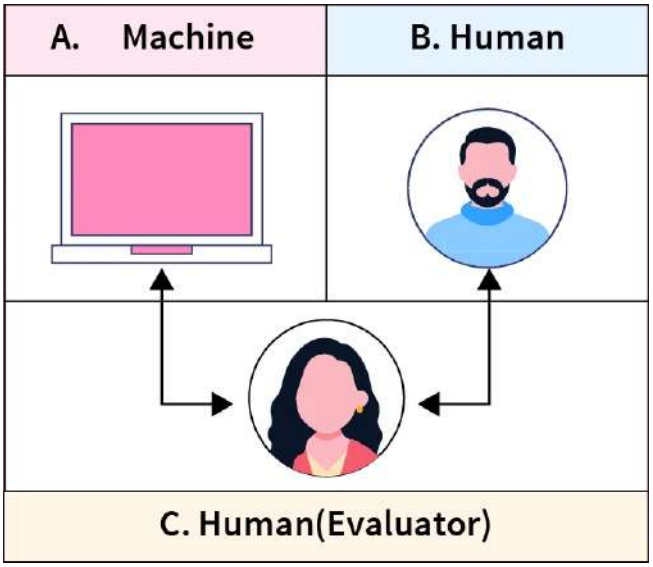
- **1980s**: Rise of *machine learning* as a subset of AI. Neural networks regain interest with the backpropagation algorithm.
- **1990s**: Probabilistic models and practical AI
  Hidden Markov models, and support vector machines gain traction, enabling probabilistic reasoning under uncertainty. AI begins achieving practical success in areas like speech recognition, handwriting recognition, and recommendation systems.
- **2000s**: Big data and scalable methods
  Growth of the internet and affordable computing drives large-scale data collection. Ensemble methods (boosting, bagging, random forests) and kernel methods improve predictive performance. AI becomes central to search engines, spam filtering, and personalization.
- **2010s**: Deep learning revolution
  Advances in GPU computing and large datasets fuel breakthroughs in deep neural networks. Landmark achievements include AlexNet (image recognition), breakthroughs in NLP with transformers, and reinforcement learning milestones like AlphaGo. AI begins powering consumer applications such as virtual assistants, translation, and image tagging.
- **2020s**: Foundation models and generative AI
  Emergence of large-scale models like GPT and DALL·E enables generative text, images, and multimodal AI. Adoption accelerates across industries from healthcare to finance. Discussions expand to include AI ethics, fairness, bias, regulation, and scalability, making governance as important as technical progress.

What is the Turing Test

The Turing Test is one of the most well-known and debated concepts in artificial intelligence (AI). It was proposed by the British mathematician and computer scientist Alan Turing in 1950 in his seminal paper, "Computing Machinery and Intelligence." He proposed that the "Turing test is used to determine whether or not a computer(machine) can think intelligently like humans"?

The Turing Test is a widely recognized benchmark for evaluating a machine's ability to demonstrate human-like intelligence. The core idea is

simple: A human judge engages in a text-based conversation with both a human and a machine. The judge's task is to determine which participant is human and which is the machine. If the judge is unable to distinguish between the human and the machine based solely on the conversation, the machine is said to have passed the Turing Test.



Lab 2

Q2. Write a Program to Implement Depth First Search.

Code –

```
graph = {
    "A": [("B", 2), ("C", 2)],
    "B": [("D", 2), ("E", 2)],
    "C": [("F", 2), ("G", 2)],
    "D": [],
    "E": [],
    "F": [],
    "G": [],
}
path = []


def recc_cost(graph, visited, curr, goal, cost):
    path.append(curr)
    visited.add(curr)
    if curr == goal:
        return cost
    for node in graph[curr]:
        if node[0] not in visited:
            v = recc_cost(graph, visited, node[0], goal, cost + node[1])
            if v != -1:
                return v
    path.pop()
    return -1


inp = input("Entry: ").upper()
out = input("Goal: ").upper()
print("Cost:",recc_cost(graph, set(), inp, out, 0))
print(" -> ".join(path))
```

```
Entry: A
Goal: F
Cost: 4
A -> C -> F
```

Lab 3

Q3. Implementation of Breadth First Search.

Code –

```
import queue

graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["F", "G"],
    "D": [],
    "E": [],
    "F": [],
    "G": [],
}


def bfs(graph, start, goal):
    visited = set()
    q = queue.Queue()
    q.put(start)

    while not q.empty():
        vertex = q.get()
        if vertex == goal:
            print(f"{vertex}")
            return
        if vertex not in visited:
            print(f"{vertex} > ", end="")
            visited.add(vertex)

            for node in graph[vertex]:
                if node not in visited:
                    q.put(node)


start = input("Starting Node: ").upper()
goal = input("Ending Node: ").upper()
bfs(graph, start, goal)
```

```
Starting Node: A
Ending Node: F
A > B > C > D > E > F
```

Lab 4

Q4. Write a Program to Implement Best First Search.

Code –

```
import heapq

graph = {
    "A": [("B", 3), ("C", 5)],
    "B": [("D", 4), ("E", 10)],
    "C": [("F", 20), ("G", 9)],
    "D": [],
    "E": [],
    "F": [],
    "G": [],
}


def best_first(graph, start, goal):
    pq = [(0, start, [start])]
    visited = set()

    while pq:
        _, node, path = heapq.heappop(pq)
        print(f"{node} ", end="")
        if node == goal:
            return path

        visited.add(node)

        for n, cost in graph[node]:
            if n not in visited:
                heapq.heappush(pq, (cost, n, path + [n]))


path = best_first(graph, "A", "E")
print()
print(f"Path = {path}")
```

```
A B D C G E
Path = ['A', 'B', 'E']
```

Lab 5

Q5. Introduction to AI Languages such as LISP, PROLOG.

LISP(list processing)

Lisp is a family of programming languages known for its unique parenthesized syntax and is primarily used for artificial intelligence and symbolic processing. It was created in the late 1950s by John McCarthy and is notable for its ability to manipulate code as data, which allows for powerful macro systems and dynamic programming capabilities. It was designed mainly for Artificial Intelligence (AI) research, symbolic computation, and list processing.

Features –

- **List-based language** – all data and code are represented as lists.
- **Homoiconicity** – code and data share the same structure, making meta-programming easy.
- **Dynamic typing** – types are checked at runtime, not at compile time.
- **Garbage collection** – automatic memory management.
- **Recursion support** – functions naturally use recursion for control flow.
- **Functional programming** – emphasizes functions as first-class objects.
- **Interactive environment (REPL)** – allows immediate execution and testing.
- **Macros** – powerful mechanism to extend the language by transforming code.
- **Portability and extensibility** – widely adaptable for AI, symbolic processing, and research.

Syntax –

General rules:
Expressions are written in prefix notation: (operator operand1 operand2 ...)
Everything is enclosed in parentheses.
Functions are defined with defun.
Comments start with ;

```
; Arithmetic
(+ 3 4)        ; Output: 7
(* 2 5)        ; Output: 10

; Defining a function
(defun square (x) (* x x))
(square 6)      ; Output: 36

; Conditional
(defun absval (x)
  (if (< x 0)
     (- x)
     x))
(absval -7)     ; Output: 7

; List operations
(setq mylist '(1 2 3 4))
(car mylist)    ; Output: 1
(cdr mylist)    ; Output: (2 3 4)
(cons 0 mylist)  ; Output: (0 1 2 3 4)
```

Operators –

1. Arithmetic Operators
   + : Addition
   - : Subtraction
   * : Multiplication
   / : Division
   mod : Modulus
2. Relational Operators
   = : Equal
   /= : Not equal
   < : Less than
   > : Greater than
   <= : Less than or equal
   >= : Greater than or equal
3. Logical Operators
   and : Logical AND
   or : Logical OR
   not : Logical NOT

4. List Operators
   car : Returns first element of list
   cdr : Returns rest of list
   cons : Constructs a new list

Applications –

- **Artificial Intelligence** – expert systems, natural language processing, reasoning engines.
- **Symbolic Computation** – computer algebra systems like Macsyma and Maxima.
- **Machine Learning Research** – early AI research, neural networks, and symbolic ML.
- **Robotics** – planning, pathfinding, and knowledge-based control systems.
- **Natural Language Processing** – parsing, semantic analysis, and chatbots.
- **Knowledge Representation** – encoding facts, rules, and reasoning about them.
- **Academic Research** – prototyping new programming paradigms and algorithms.
- **Industrial Systems** – used historically in companies like Symbolics and Franz Inc. for specialized AI tools.

PROLOG(PROgramming in LOGic)

Prolog is a logic programming language that has its origins in artificial intelligence, automated theorem proving, and computational linguistics. Prolog has its roots in first-order logic, a formal logic. Unlike many other programming languages, Prolog is intended primarily as a declarative programming language: the program is a set of facts and rules, which define relations. A computation is initiated by running a query over the program.

Features –

- **Declarative Language** – programs describe what the problem is, not how to solve it.
- **Based on First-Order Predicate Logic** – facts and rules form the knowledge base.
- **Automatic Backtracking** – explores possible solutions by undoing and retrying choices.

- **Unification** – pattern matching mechanism to bind variables with values.
- **Recursion Support** – natural way to express iterative logic.
- **Built-in Search Mechanism** – systematically searches through rules and facts.
- **Symbolic and Non-Numeric Processing** – works well with text, relations, and symbolic data.
- **Dynamic Database** – facts and rules can be added or removed at runtime.
- **Strong for AI** – reasoning, problem-solving, and knowledge representation are native strengths.

Syntax –

1. Facts – state relationships or properties.
   parent(john, mary).
   likes(alice, pizza).
2. Rules – define logical relationships using :- (if).
   grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
3. Queries – questions asked to the system, ending with ?.
   ?- parent(john, mary).
   ?- grandparent(john, susan).
4. Variables – start with uppercase letters or underscore _.
   X = john.
5. Comments
   % Single-line comment
   /* Multi-line
   comment */

Operators –

1. Arithmetic Operators
   + : Addition
   - : Subtraction
   * : Multiplication
   / : Division (floating point)
   // : Integer division
   mod : Modulus
   ?- X is 10 + 5.
   X = 15.
2. Relational Operators
   = : Unification (matches terms)

\= : Not unifiable
== : Exactly identical
\== : Not identical
<, >, =<, >= : Numeric comparisons
?- 4 = 4.
% true
?- 4 == 4.0.
% false

3. Logical Operators
   , AND
   ; OR
   \+ NOT
   ?- parent(john, mary), parent(mary, susan).
   % true
4. Control Operators
   -> If-then
   ; Else
   ! Cut (prevents backtracking beyond this point)
   max(X,Y,Max) :- X >= Y -> Max = X ; Max = Y.

Applications –

- **Expert Systems** – medical diagnosis, legal reasoning, and decision support (e.g., MYCIN, DENDRAL).
- **Natural Language Processing (NLP)** – parsing, grammar checking, semantic analysis, and chatbots.
- **Knowledge Representation** – encoding facts and rules for reasoning tasks.
- **Theorem Proving** – automated reasoning and mathematical proof systems.
- **Constraint Satisfaction Problems** – puzzles, scheduling, planning, and resource allocation.
- **Artificial Intelligence** – symbolic reasoning, planning, and problem-solving.
- **Databases** – deductive databases that support complex queries.
- **Robotics** – reasoning-based planning and decision-making modules.
- **Education and Research** – teaching logic, AI, and computational linguistics.

Lab 6

Q6. Implementation of Min-Max or Minimax search.

Code –

```python
import sys

graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["F", "G"],
    "D": [],
    "E": [],
    "F": [],
    "G": [],
}

values = {
    "A": 0,
    "B": 0,
    "C": 0,
    "D": 1,
    "E": 2,
    "F": 3,
    "G": 4,
}

def min_max(
    values: dict[str, int],
    graph: dict,
    maximizingPlayer: bool,
    currNode: str,
) -> int:
    if len(graph[currNode]) == 0:
        return values[currNode]

    if maximizingPlayer:
        best = -sys.maxsize
        for n in graph[currNode]:
            val = min_max(values, graph, not maximizingPlayer, n)
            best = max(best, val)
    else:
        best = sys.maxsize
        for n in graph[currNode]:
            val = min_max(values, graph, not maximizingPlayer, n)
            best = min(best, val)

    return best

print(min_max(values, graph, True, "A"))
```

```
❯ uv run minmax.py
3
```

Lab 7

Q7. Implementation of Tic-Tac-Toe Algorithm.
Code –

```
import copy
import math
from dataclasses import dataclass
from enum import Enum


class Player(Enum):
    human = "human"
    computer = "computer"


class GameWinner(Enum):
    human = "human"
    computer = "computer"
    tie = "tie"


class Pattern(Enum):
    cross = "X"
    zero = "O"
    blank = " "


@dataclass
class Board:
    board: list[Pattern]
    player: Player
    current_move: Pattern
    first_move: Player
    game_over: bool = False

    def available_moves(self):
        return [i for i, p in enumerate(self.board) if p == Pattern.blank]


def print_board(board: Board):
    for i in range(3):
        for j in range(3):
            print(board.board[3 * i + j].value, end="|" if j != 2 else "\n")
        if i != 2:
            print("-" * 5)


def minimax(board, maximizing):
    state = game_over(board)
    if state is not None:
        if state == GameWinner.computer:
            return 1, None
        elif state == GameWinner.human:
            return -1, None
```

```
        elif state == GameWinner.tie:
            return 0, None

    best_val = -math.inf if maximizing else math.inf
    best_move = None

    for move in board.available_moves():
        new_board = copy.deepcopy(board)
        do_move(new_board, move)
        val, _ = minimax(new_board, not maximizing)
        if maximizing and val > best_val or not maximizing and val < best_val:
            best_val, best_move = val, move

    return best_val, best_move


def game_over(board: Board):
    for i in range(0, 9, 3):
        if (
            board.board[i] == board.board[i + 1]
            and board.board[i] == board.board[i + 2]
            and board.board[i] != Pattern.blank
        ):
            if board.board[i] == Pattern.cross:
                return board.first_move
            else:
                return (
                    GameWinner.computer
                    if board.first_move == Player.human
                    else GameWinner.human
                )

    for i in range(3):
        if (
            board.board[i] == board.board[i + 3]
            and board.board[i] == board.board[i + 6]
            and board.board[i] != Pattern.blank
        ):
            if board.board[i] == Pattern.cross:
                return board.first_move
            else:
                return (
                    GameWinner.computer
                    if board.first_move == Player.human
                    else GameWinner.human
                )

    if (
        board.board[0] == board.board[4]
        and board.board[0] == board.board[8]
        and board.board[0] != Pattern.blank
    ):
        if board.board[0] == Pattern.cross:
            return board.first_move
        else:
            return (
                GameWinner.computer
                if board.first_move == Player.human
                else GameWinner.human
            )
```

```
    if (
        board.board[2] == board.board[4]
        and board.board[2] == board.board[6]
        and board.board[2] != Pattern.blank
    ):
        if board.board[2] == Pattern.cross:
            return board.first_move
        else:
            return (
                GameWinner.computer
                if board.first_move == Player.human
                else GameWinner.human
            )

    if all(p != Pattern.blank for p in board.board):
        return GameWinner.tie

    return None


def do_move(board: Board, move: int):
    board.board[move] = board.current_move
    board.player = Player.computer if board.player == Player.human else Player.human
    board.current_move = (
        Pattern.cross if board.current_move == Pattern.zero else Pattern.zero
    )


def get_move(board: Board):
    print_board(board)
    print(f"Current move by {board.player.value}, place a {board.current_move.value}")
    can_move = False
    move = None
    while not can_move:
        move = int(input("Move to where: "))
        if board.board[move] != Pattern.blank:
            print("Can only place at blank spots")
        else:
            can_move = True
    if can_move and move is not None:
        do_move(board, move)
        state = game_over(board)
        if state is not None:
            board.game_over = True
            print_board(board)
            print(f"Game is won by {state.value}")


def main():
    player = input("First move goes to (H/C): ").upper()
    board = Board(
        [Pattern.blank] * 9,
        Player.human if player == "H" else Player.computer,
        Pattern.cross,
        Player.human if player == "H" else Player.computer,
    )

    while True:
        if board.game_over:
```

```
            break
        if board.player == Player.human:
            get_move(board)
        else:
            _, move = minimax(board, True)
            if move is not None:
                do_move(board, move)
                state = game_over(board)
                if state is not None:
                    board.game_over = True
                    print_board(board)
                    print(f"Game is won by {state.value}")


if __name__ == "__main__":
    main()
```

```
❯ uv run main.py
First move goes to (H/C): H
 | |
-----
 | |
-----
 | |
Current move by human, place a X
Move to where: 4
O| |
-----
 |X|
-----
 | |
Current move by human, place a X
Move to where: 2
O| |X
-----
 |X|
-----
O| |
Current move by human, place a X
Move to where: 3
O| |X
-----
X|X|O
-----
O| |
Current move by human, place a X
Move to where: 1
O|X|X
-----
X|X|O
-----
O|O|
Current move by human, place a X
Move to where: 8
O|X|X
-----
X|X|O
-----
O|O|X
Game is won by tie
```

```
First move goes to (H/C): C
X| |
-----
 | |
-----
 | |
Current move by human, place a O
Move to where: 2
X| |O
-----
X| |
-----
 | |
Current move by human, place a O
Move to where: 6
X| |O
-----
X|X|
-----
O| |
Current move by human, place a O
Move to where: 8
X| |O
-----
X|X|X
-----
O| |O
Game is won by computer
```

Lab 8

Q8. Write a program to implement Min-Max or Minimax search with Apha-Beta pruning.

Code –

```
import math

graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["F", "G"],
    "D": [],
    "E": [],
    "F": [],
    "G": [],
}

values = {
    "A": 0,
    "B": 0,
    "C": 0,
    "D": 4,
    "E": 3,
    "F": 2,
    "G": 1,
}


def alpha_beta(
    values: dict[str, int],
    graph: dict,
    maximizingPlayer: bool,
    currNode: str,
    alpha: float,
    beta: float,
) -> float:
    if len(graph[currNode]) == 0:
        return values[currNode]

    if maximizingPlayer:
        best = -math.inf
        for n in graph[currNode]:
            val = alpha_beta(values, graph, not maximizingPlayer, n, alpha, beta)
            alpha = max(alpha, val)
            best = max(best, val)
            if alpha >= beta:
                print(f"Pruning at {currNode}")
                break
    else:
        best = math.inf
        for n in graph[currNode]:
            val = alpha_beta(values, graph, not maximizingPlayer, n, alpha, beta)
            beta = min(beta, val)
            best = min(best, val)
            if alpha >= beta:
```

```
                print(f"Pruning at {currNode}")
                break

    return best

print(alpha_beta(values, graph, True, "A", -math.inf, math.inf))
```

```
❯ uv run searching_algo/alpha_beta.py
Pruning at C
3
```