**Python programing**

**8 puzzle**

import heapq

class PuzzleNode:

def \_\_init\_\_(self, state, parent=None, move=None):

self.state = state

self.parent = parent

self.move = move

self.cost = 0 if parent is None else parent.cost + 1

def \_\_lt\_\_(self, other):

return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

def \_\_eq\_\_(self, other):

return self.state == other.state

def \_\_hash\_\_(self):

return hash(str(self.state))

def heuristic(self):

# Manhanttan distance heuristic

goal\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

h = 0

for i in range(3):

for j in range(3):

if self.state[i][j] != 0:

x, y = divmod(self.state[i][j] - 1, 3)

h += abs(x - i) + abs(y - j)

return h

def is\_goal(self):

goal\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

return self.state == goal\_state

def get\_neighbors(self):

neighbors = []

zero\_x, zero\_y = None, None

for i in range(3):

for j in range(3):

if self.state[i][j] == 0:

zero\_x, zero\_y = i, j

break

possible\_moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

for dx, dy in possible\_moves:

new\_x, new\_y = zero\_x + dx, zero\_y + dy

if 0 <= new\_x < 3 and 0 <= new\_y < 3:

new\_state = [list(row) for row in self.state]

new\_state[zero\_x][zero\_y], new\_state[new\_x][new\_y] = new\_state[new\_x][new\_y], new\_state[zero\_x][zero\_y]

neighbors.append(PuzzleNode(new\_state, parent=self, move=(dx, dy)))

return neighbors

def solve\_8\_puzzle(initial\_state):

initial\_node = PuzzleNode(initial\_state)

open\_list = [initial\_node]

closed\_set = set()

while open\_list:

current\_node = heapq.heappop(open\_list)

if current\_node.is\_goal():

path = []

while current\_node:

path.append(current\_node.state)

current\_node = current\_node.parent

return list(reversed(path))

closed\_set.add(current\_node)

for neighbor in current\_node.get\_neighbors():

if neighbor in closed\_set:

continue

if neighbor not in open\_list:

heapq.heappush(open\_list, neighbor)

return None

# Example usage:

initial\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

solution = solve\_8\_puzzle(initial\_state)

if solution:

for step, state in enumerate(solution):

print(f"Step {step}:")

for row in state:

print(row)

print("\n")

else:

print("No solution found.")

**8 queens**

def is\_safe(board, row, col, n):

# Check if there is a queen in the same column

for i in range(row):

if board[i][col] == 1:

return False

# Check upper-left diagonal

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check upper-right diagonal

for i, j in zip(range(row, -1, -1), range(col, n)):

if board[i][j] == 1:

return False

return True

def solve\_n\_queens\_util(board, row, n, solutions):

if row == n:

solutions.append(["".join("Q" if cell else "." for cell in row) for row in board])

return

for col in range(n):

if is\_safe(board, row, col, n):

board[row][col] = 1

solve\_n\_queens\_util(board, row + 1, n, solutions)

board[row][col] = 0

def solve\_n\_queens(n):

board = [[0] \* n for \_ in range(n)]

solutions = []

solve\_n\_queens\_util(board, 0, n, solutions)

return solutions

def print\_solutions(solutions):

for i, solution in enumerate(solutions):

print(f"Solution {i + 1}:")

for row in solution:

print(row)

print("\n")

# Example usage:

n = 8 # Change this to the desired board size

solutions = solve\_n\_queens(n)

if solutions:

print(f"Found {len(solutions)} solutions for {n}-Queens:")

print\_solutions(solutions)

else:

print(f"No solutions found for {n}-Queens.")

**water jug**

from collections import deque

class State:

def \_\_init\_\_(self, jug1, jug2):

self.jug1 = jug1

self.jug2 = jug2

def \_\_eq\_\_(self, other):

return self.jug1 == other.jug1 and self.jug2 == other.jug2

def \_\_hash\_\_(self):

return hash((self.jug1, self.jug2))

def \_\_str\_\_(self):

return f"({self.jug1}, {self.jug2})"

def water\_jug\_problem(capacity1, capacity2, target):

initial\_state = State(0, 0)

visited = set()

queue = deque([(initial\_state, [])])

while queue:

current\_state, actions = queue.popleft()

if current\_state in visited:

continue

visited.add(current\_state)

if current\_state.jug1 == target or current\_state.jug2 == target:

return actions

# Fill jug1

queue.append((State(capacity1, current\_state.jug2), actions + ["Fill jug1"]))

# Fill jug2

queue.append((State(current\_state.jug1, capacity2), actions + ["Fill jug2"]))

# Empty jug1

queue.append((State(0, current\_state.jug2), actions + ["Empty jug1"]))

# Empty jug2

queue.append((State(current\_state.jug1, 0), actions + ["Empty jug2"]))

# Pour from jug1 to jug2

pour\_amount = min(current\_state.jug1, capacity2 - current\_state.jug2)

queue.append((State(current\_state.jug1 - pour\_amount, current\_state.jug2 + pour\_amount),

actions + [f"Pour {pour\_amount} from jug1 to jug2"]))

# Pour from jug2 to jug1

pour\_amount = min(current\_state.jug2, capacity1 - current\_state.jug1)

queue.append((State(current\_state.jug1 + pour\_amount, current\_state.jug2 - pour\_amount),

actions + [f"Pour {pour\_amount} from jug2 to jug1"]))

return None

# Example usage:

capacity\_jug1 = 4

capacity\_jug2 = 3

target\_volume = 2

result = water\_jug\_problem(capacity\_jug1, capacity\_jug2, target\_volume)

if result:

print(f"Actions to reach {target\_volume} liters in one of the jugs:")

for action in result:

print(action)

else:

print(f"Cannot reach {target\_volume} liters in one of the jugs.")

**cryptarthimatic**

from itertools import permutations

def is\_valid(puzzle, solution):

# Check if leading digits are not zero

for word in puzzle:

if solution[word[0]] == '0':

return False

# Evaluate the equation

left\_operand = sum(int(solution[word]) for word in puzzle[:-1])

right\_operand = int(solution[puzzle[-1]])

return left\_operand == right\_operand

def solve\_cryptarithmetic(puzzle):

# Extract unique letters from the puzzle

letters = set("".join(puzzle))

if len(letters) > 10:

print("Too many unique letters for a valid solution.")

return

# Generate all possible permutations of digits from 0 to 9

digit\_permutations = permutations("0123456789", len(letters))

# Iterate through permutations and check for a valid solution

for perm in digit\_permutations:

solution = dict(zip(letters, perm))

if is\_valid(puzzle, solution):

print("Solution found:")

for word in puzzle:

print(word, "=", int("".join(solution[c] for c in word)))

return

print("No solution found.")

# Example usage:

puzzle = ["SEND", "MORE", "MONEY"]

solve\_cryptarithmetic(puzzle)

**Missionaries Cannibal**

def is\_valid\_state(state):

# Check if the number of cannibals is greater than missionaries on either side

if state[0] < state[1] and state[0] != 0:

return False

if state[3] < state[4] and state[3] != 0:

return False

return True

def is\_goal\_state(state, goal\_state):

return state == goal\_state

def generate\_next\_states(state):

next\_states = []

moves = [(1, 0), (0, 1), (2, 0), (0, 2), (1, 1)]

for move in moves:

if state[2] == 'left':

new\_state = (

state[0] - move[0],

state[1] - move[1],

'right',

state[3] + move[0],

state[4] + move[1]

)

else:

new\_state = (

state[0] + move[0],

state[1] + move[1],

'left',

state[3] - move[0],

state[4] - move[1]

)

if 0 <= new\_state[0] <= 3 and 0 <= new\_state[1] <= 3 and 0 <= new\_state[3] <= 3 and 0 <= new\_state[4] <= 3:

next\_states.append(new\_state)

return next\_states

def dfs\_missionaries\_and\_cannibals(initial\_state, goal\_state):

stack = [(initial\_state, [])]

visited = set()

while stack:

current\_state, path = stack.pop()

visited.add(current\_state)

if is\_goal\_state(current\_state, goal\_state):

return path

for next\_state in generate\_next\_states(current\_state):

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

stack.append((next\_state, path + [next\_state]))

return None

# Example usage:

initial\_state = (3, 3, 'left', 0, 0)

goal\_state = (0, 0, 'right', 3, 3)

solution = dfs\_missionaries\_and\_cannibals(initial\_state, goal\_state)

if solution:

print("Solution found:")

for state in solution:

print(state)

else:

print("No solution found.")

**Travelling Salesman**

import itertools

import sys

def calculate\_total\_distance(path, cities):

total\_distance = 0

for i in range(len(path) - 1):

city1 = cities[path[i]]

city2 = cities[path[i + 1]]

total\_distance += calculate\_distance(city1, city2)

return total\_distance

def calculate\_distance(city1, city2):

x1, y1 = city1

x2, y2 = city2

return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) \*\* 0.5

def solve\_tsp(cities):

num\_cities = len(cities)

if num\_cities <= 2:

return cities

shortest\_path = None

shortest\_distance = sys.float\_info.max

for path in itertools.permutations(range(num\_cities)):

distance = calculate\_total\_distance(path, cities)

if distance < shortest\_distance:

shortest\_distance = distance

shortest\_path = path

return shortest\_path, shortest\_distance

# Example usage:

cities = [(0, 0), (1, 2), (2, 4), (3, 1)]

shortest\_path, shortest\_distance = solve\_tsp(cities)

if shortest\_path:

print("Shortest TSP path:", shortest\_path)

print("Shortest TSP distance:", shortest\_distance)

else:

print("No solution found.")

**vacuum cleaner world**

class VacuumCleaner:

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

self.location = "A" # Starting location is Room A

self.cleaned = {"A": False, "B": False}

def clean(self, room):

self.cleaned[room] = True

print(f"Room {room} is now clean.")

def move(self, room):

if room == self.location:

print(f"Already in Room {room}.")

else:

print(f"Moving from Room {self.location} to Room {room}.")

self.location = room

def is\_clean(self, room):

return self.cleaned[room]

def main():

vacuum = VacuumCleaner()

while not all(vacuum.cleaned.values()):

current\_room = vacuum.location

if not vacuum.is\_clean(current\_room):

vacuum.clean(current\_room)

else:

next\_room = "B" if current\_room == "A" else "A"

vacuum.move(next\_room)

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

main()

**breadth first search**

from collections import defaultdict, deque

class Graph:

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

self.graph = defaultdict(list)

def add\_edge(self, u, v):

self.graph[u].append(v)

self.graph[v].append(u)

def bfs(self, start):

visited = set()

queue = deque()

visited.add(start)

queue.append(start)

while queue:

vertex = queue.popleft()

print(vertex, end=" ")

for neighbor in self.graph[vertex]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

# Example usage:

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

g = Graph()

g.add\_edge(0, 1)

g.add\_edge(0, 2)

g.add\_edge(1, 2)

g.add\_edge(2, 0)

g.add\_edge(2, 3)

g.add\_edge(3, 3)

print("Breadth-First Traversal starting from vertex 2:")

g.bfs(2)

**uniform cost search**

import heapq

class Graph:

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

self.graph = {}

def add\_edge(self, node1, node2, cost):

if node1 not in self.graph:

self.graph[node1] = []

if node2 not in self.graph:

self.graph[node2] = []

self.graph[node1].append((node2, cost))

self.graph[node2].append((node1, cost))

def uniform\_cost\_search(graph, start, goal):

visited = set()

priority\_queue = [(0, start)]

while priority\_queue:

cost, current\_node = heapq.heappop(priority\_queue)

if current\_node in visited:

continue

visited.add(current\_node)

if current\_node == goal:

return cost

for neighbor, neighbor\_cost in graph.graph[current\_node]:

if neighbor not in visited:

heapq.heappush(priority\_queue, (cost + neighbor\_cost, neighbor))

return float("inf")

# Example usage:

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

g = Graph()

g.add\_edge('A', 'B', 4)

g.add\_edge('A', 'C', 2)

g.add\_edge('B', 'C', 5)

g.add\_edge('B', 'D', 10)

g.add\_edge('C', 'D', 3)

g.add\_edge('D', 'E', 7)

g.add\_edge('E', 'A', 8)

start\_node = 'A'

goal\_node = 'D'

cost = uniform\_cost\_search(g, start\_node, goal\_node)

if cost != float("inf"):

print(f"Shortest path cost from {start\_node} to {goal\_node}: {cost}")

else:

print(f"No path found from {start\_node} to {goal\_node}")

**depth first search**

class Graph:

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

self.graph = {}

def add\_edge(self, node, neighbors):

self.graph[node] = neighbors

def dfs\_recursive(self, node, visited):

if node not in visited:

print(node, end=" ")

visited.add(node)

for neighbor in self.graph[node]:

self.dfs\_recursive(neighbor, visited)

def dfs(self, start\_node):

visited = set()

self.dfs\_recursive(start\_node, visited)

# Example usage:

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

g = Graph()

g.add\_edge('A', ['B', 'C'])

g.add\_edge('B', ['D', 'E'])

g.add\_edge('C', ['F'])

g.add\_edge('D', [])

g.add\_edge('E', ['F'])

g.add\_edge('F', [])

start\_node = 'A'

print("Depth-First Traversal starting from node", start\_node + ":")

g.dfs(start\_node)

**minmax search tree**

import math

# Representation of the game board (example: Tic-Tac-Toe)

# 'X' represents the player, 'O' represents the opponent, and '.' represents an empty space.

board = [

['X', 'O', 'X'],

['O', 'X', 'O'],

['.', '.', 'X']

]

def is\_terminal(board):

# Check if the game is over (terminal state)

for row in board:

if all(cell == 'X' for cell in row) or all(cell == 'O' for cell in row):

return True

for col in range(3):

if all(board[row][col] == 'X' for row in range(3)) or all(board[row][col] == 'O' for row in range(3)):

return True

if all(board[i][i] == 'X' for i in range(3)) or all(board[i][i] == 'O' for i in range(3)):

return True

if all(board[i][2 - i] == 'X' for i in range(3)) or all(board[i][2 - i] == 'O' for i in range(3)):

return True

# Check if the game is a draw (no empty spaces left)

for row in board:

if '.' in row:

return False

return True

def minimax(board, depth, is\_maximizing):

if is\_terminal(board):

if is\_maximizing:

return -1 # Opponent wins

else:

return 1 # Player wins

return 0 # Draw

if is\_maximizing:

best\_score = -math.inf

for i in range(3):

for j in range(3):

if board[i][j] == '.':

board[i][j] = 'X'

score = minimax(board, depth + 1, False)

board[i][j] = '.'

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = math.inf

for i in range(3):

for j in range(3):

if board[i][j] == '.':

board[i][j] = 'O'

score = minimax(board, depth + 1, True)

board[i][j] = '.'

best\_score = min(score, best\_score)

return best\_score

def find\_best\_move(board):

best\_move = None

best\_score = -math.inf

for i in range(3):

for j in range(3):

if board[i][j] == '.':

board[i][j] = 'X'

score = minimax(board, 0, False)

board[i][j] = '.'

if score > best\_score:

best\_score = score

best\_move = (i, j)

return best\_move

# Example usage:

best\_move = find\_best\_move(board)

print("Best Move:", best\_move)

**alpha beta purning**

import math

# Define the game tree as a dictionary

# The keys represent nodes, and the values represent (min, max) for the player (1 for MAX, -1 for MIN).

game\_tree = {

'A': (0, 0),

'B': (0, 0),

'C': (0, 0),

'D': (0, 0),

'E': (0, 0),

'F': (0, 0),

'G': (0, 0),

'H': (0, 0),

'I': (0, 0),

}

# Define the game tree edges

edges = {

'A': ['B', 'C', 'D'],

'B': ['E', 'F'],

'C': ['G', 'H'],

'D': ['I'],

}

def alpha\_beta(node, alpha, beta, maximizing\_player):

if node not in edges:

return game\_tree[node][0] # Leaf node, return its value

if maximizing\_player:

value = -math.inf

for child in edges[node]:

value = max(value, alpha\_beta(child, alpha, beta, False))

alpha = max(alpha, value)

if alpha >= beta:

break # Beta pruning

game\_tree[node] = (value, game\_tree[node][1]) # Update node value

return value

else:

value = math.inf

for child in edges[node]:

value = min(value, alpha\_beta(child, alpha, beta, True))

beta = min(beta, value)

if alpha >= beta:

break # Alpha pruning

game\_tree[node] = (game\_tree[node][0], value) # Update node value

return value

# Initial call to alpha\_beta

alpha\_beta('A', -math.inf, math.inf, True)

# Display the final game tree with updated values

for node, (min\_val, max\_val) in game\_tree.items():

print(f'Node {node}: (MIN, MAX) = ({min\_val}, {max\_val})')

**Design a web page**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta name="description" content="Optimizing Anchor Tags and Titles for SEO">

<title>SEO Optimization Example</title>

<!-- Search Engine Optimization (SEO) Meta Tags -->

<meta name="keywords" content="SEO, Anchor Tags, Titles, Search Engine Optimization">

<meta name="author" content="Your Name">

<meta name="robots" content="index, follow">

<link rel="canonical" href="https://www.yourwebsite.com/seo-optimization-example.html">

<!-- Open Graph Protocol Meta Tags (for social media sharing) -->

<meta property="og:title" content="SEO Optimization Example">

<meta property="og:description" content="Optimizing Anchor Tags and Titles for SEO">

<meta property="og:image" content="https://www.yourwebsite.com/images/seo-image.jpg">

<meta property="og:url" content="https://www.yourwebsite.com/seo-optimization-example.html">

<meta property="og:type" content="article">

<!-- Twitter Card Meta Tags (for Twitter sharing) -->

<meta name="twitter:card" content="summary\_large\_image">

<meta name="twitter:title" content="SEO Optimization Example">

<meta name="twitter:description" content="Optimizing Anchor Tags and Titles for SEO">

<meta name="twitter:image" content="https://www.yourwebsite.com/images/seo-image.jpg">

<!-- Favicon -->

<link rel="icon" href="favicon.ico" type="image/x-icon">

<!-- CSS Styles (you can link your stylesheet here) -->

<link rel="stylesheet" href="styles.css">

</head>

<body>

<header>

<h1>Welcome to Our SEO Optimization Example</h1>

</header>

<nav>

<ul>

<li><a href="#section1">Section 1</a></li>

<li><a href="#section2">Section 2</a></li>

<li><a href="#section3">Section 3</a></li>

</ul>

</nav>

<main>

<section id="section1">

<h2>Section 1: Optimizing Anchor Tags and Titles</h2>

<p>Anchor tags are an essential part of SEO. Ensure that your anchor text is descriptive and relevant to the linked content. Use meaningful titles and alt attributes for images and links. This improves accessibility and helps search engines understand your content better.</p>

</section>

<section id="section2">

<h2>Section 2: Title Optimization</h2>

<p>The title of your web page is a crucial SEO element. It should accurately represent the content and include relevant keywords. Keep it concise and compelling to attract users and improve click-through rates in search results.</p>

</section>

<section id="section3">

<h2>Section 3: Conclusion</h2>

<p>Optimizing anchor tags and titles is a fundamental step in enhancing your website's search engine visibility. Follow best practices and stay updated with SEO guidelines to achieve better rankings and user engagement.</p>

</section>

</main>

<footer>

<p>&copy; 2023 Your Website. All rights reserved.</p>

</footer>

</body>

</html>

**Map colouring**

def is\_safe(node, color, graph, color\_assignment):

for neighbor in graph[node]:

if color\_assignment.get(neighbor) == color:

return False

return True

def map\_coloring(graph, colors, node, color\_assignment):

if node not in graph:

return True # All nodes are colored

for color in colors:

if is\_safe(node, color, graph, color\_assignment):

color\_assignment[node] = color

if map\_coloring(graph, colors, next\_node(node, graph), color\_assignment):

return True

color\_assignment[node] = None # Backtrack

return False

def next\_node(node, graph):

# Find the next uncolored node

for n in graph:

if n not in color\_assignment:

return n

return None

# Example usage:

graph = {

'WA': ['SA', 'NT'],

'NT': ['WA', 'SA', 'Q'],

'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],

'Q': ['NT', 'SA', 'NSW'],

'NSW': ['SA', 'Q', 'V'],

'V': ['SA', 'NSW']

}

colors = ['Red', 'Green', 'Blue', 'Yellow']

color\_assignment = {}

if map\_coloring(graph, colors, next\_node(list(graph.keys())[0], graph), color\_assignment):

print("Map Coloring Solution:")

for node, color in color\_assignment.items():

print(f"{node}: {color}")

else:

print("No solution found.")

**A\* search**

import heapq

class Graph:

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

self.graph = {}

def add\_edge(self, node1, node2, cost):

if node1 not in self.graph:

self.graph[node1] = []

self.graph[node1].append((node2, cost))

def astar\_search(graph, start, goal):

open\_set = [(0, start)]

came\_from = {}

g\_score = {node: float('inf') for node in graph.graph}

g\_score[start] = 0

f\_score = {node: float('inf') for node in graph.graph}

f\_score[start] = heuristic(start, goal)

while open\_set:

\_, current = heapq.heappop(open\_set)

if current == goal:

return reconstruct\_path(came\_from, current)

for neighbor, cost in graph.graph[current]:

tentative\_g\_score = g\_score[current] + cost

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = g\_score[neighbor] + heuristic(neighbor, goal)

heapq.heappush(open\_set, (f\_score[neighbor], neighbor))

return None

def heuristic(node, goal):

# In this example, we use the Manhattan distance as the heuristic.

x1, y1 = node

x2, y2 = goal

return abs(x1 - x2) + abs(y1 - y2)

def reconstruct\_path(came\_from, current):

path = [current]

while current in came\_from:

current = came\_from[current]

path.append(current)

return list(reversed(path))

# Example usage:

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

g = Graph()

g.add\_edge((0, 0), (1, 0), 1)

g.add\_edge((1, 0), (1, 1), 1)

g.add\_edge((1, 1), (2, 1), 1)

g.add\_edge((0, 1), (0, 2), 1)

g.add\_edge((0, 2), (1, 2), 1)

g.add\_edge((1, 2), (2, 2), 1)

g.add\_edge((2, 0), (2, 1), 1)

g.add\_edge((2, 1), (2, 2), 1)

start\_node = (0, 0)

goal\_node = (2, 2)

path = astar\_search(g, start\_node, goal\_node)

if path:

print("Shortest Path:")

for node in path:

print(node)

else:

print("No path found.")

**decision tree**

from sklearn.datasets import load\_iris

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score, classification\_report

# Load the Iris dataset (a sample dataset included in scikit-learn)

data = load\_iris()

X = data.data # Features

y = data.target # Target variable (class labels)

# Split the dataset into a training set and a test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a Decision Tree classifier

clf = DecisionTreeClassifier()

# Fit the classifier to the training data

clf.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = clf.predict(X\_test)

# Evaluate the classifier

accuracy = accuracy\_score(y\_test, y\_pred)

report = classification\_report(y\_test, y\_pred, target\_names=data.target\_names)

print("Accuracy:", accuracy)

print("Classification Report:\n", report)

**two layer feed forward**

import numpy as np

import tensorflow as tf

from tensorflow import keras

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import make\_classification

from sklearn.preprocessing import StandardScaler

# Generate a synthetic dataset for binary classification

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_classes=2, random\_state=42)

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize the input features (optional but often recommended)

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Create a two-layer feed-forward neural network using Keras

model = keras.Sequential([

keras.layers.Dense(units=64, activation='relu', input\_shape=(X\_train.shape[1],)),

keras.layers.Dense(units=1, activation='sigmoid')

])

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.1)

# Evaluate the model on the test data

loss, accuracy = model.evaluate(X\_test, y\_test)

print("Test Loss:", loss)

print("Test Accuracy:", accuracy)