Computational Intelligence Laboratory

B.Tech. VII Semester

Department: Computer Science and Engineering

Faculty of Engineering & Technology

Ramaiah University of Applied Sciences

NAME: SATYAJIT GHANA
REG NO: 17ETCS002159

Ramaiah University of Applied Sciences

Private University Established in Karnataka State by Act No. 15 of 2013

Faculty	Engineering & Technology
Programme	B. Tech. in Computer Science and Engineering
Year/Semester	7th Semester
Name of the Laboratory	Computational Intelligence Laboratory
Laboratory Code	CSC401A

List of Experiments

1. **BFS**
2. **DFS**
3. **Genetic Algorithm**
4. **Hill Climbing**
5. **MinMax – Alpha Beta Pruning**
6. **Fuzzy Logic**
7. **Neural Network**

Laboratory 1

Title of the Laboratory Exercise: **Breadth First Search**

1. Introduction and Purpose of Experiment

   In this laboratory exercises students get to implement BFS

2. Aim and Objectives

   Aim

   - To implement the algorithm in Python

   Objectives

   At the end of this lab, the student will be able to

   - Develop the program using Python

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in Python language
   iv. Test the implemented program
   v. Document the Results
   vi. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm:

```
BFS (G, s) //Where G is the graph and s is the source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are
marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                        Q.enqueue( w ) //Stores w in Q to further visit its neighbour
                        mark w as visited.
```

Source Code:

```
from collections import defaultdict
from queue import Queue
from typing import List, Any

class Graph:
    def __init__(self):
        self._graph = defaultdict(list)

    def add_edge(self, u, v):
        self._graph[u].append(v)

    def neighbours(self, u):
        return self._graph[u]

    def bfs(self, start):
        t_queue: Queue = Queue()
        d_visited: Dict[Any, List] = {n: False for n in self._graph.keys()}

        print(f'BFS, start={start}: [', end="")

        t_queue.put(start)
        d_visited[start] = True

        while not t_queue.empty():
            node = t_queue.get()

            print(f'{node},', end=" ")

            for neighbour in self.neighbours(node):
                if not d_visited[neighbour]:
                    t_queue.put(neighbour)
                    d_visited[neighbour] = True
        print("\b\b]")

graph = Graph()

graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)

graph.bfs(2)
```

5. Presentation of Results

```
[79]: 1 graph._graph

[79]: defaultdict(list, {0: [1, 2], 1: [2], 2: [0, 3], 3: [3]})

[80]: 1 graph.bfs(2)

      BFS, start=2: [2, 0, 3, 1]
```

6. Analysis and Discussions

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

7. Conclusions

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

8. Comments

1. Limitations of Experiments

One disadvantage of BFS is that it is a 'blind' search, when the search space is large the search performance will be poor compared to other heuristic searches.

2. Limitations of Results

BFS will perform well if the search space is small. It performs best if the goal state lies in upper left-hand side of the tree. But it will perform relatively poorly relative to the depth-first search algorithm if the goal state lies in the bottom of the tree. BFS will always find the shortest path if the weight on the links are uniform.So BFS is complete and optimal. As we discussed memory utilization is poor in BFS, so we can say that BFS needs more memory as compared to DFS.

3. Learning happened

We learnt to implement BFS

4. Recommendations

None

NAME: SATYAJIT GHANA ROLL NO: 17ETCS002159

Laboratory 2

Title of the Laboratory Exercise: **Depth First Search**

1. Introduction and Purpose of Experiment

   In this laboratory exercises students get to implement DFS

2. Aim and Objectives

   Aim

   - To implement the algorithm in Python

   Objectives

   At the end of this lab, the student will be able to

   - Develop the program using Python

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in Python language
   iv. Test the implemented program
   v. Document the Results
   vi. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm:

```
DFS-iterative (G, s): //Where G is graph and s is source vertex
  let S be stack
  S.push( s )            //Inserting s in stack
  mark s as visited.
  while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
    for all neighbours w of v in Graph G:
        if w is not visited :
                 S.push( w )
                mark w as visited
```

Source Code:

```
from collections import defaultdict, deque
from typing import List, Any

class Graph:
    def __init__(self):
        self._graph = defaultdict(list)

    def add_edge(self, u, v):
        self._graph[u].append(v)

    def neighbours(self, u):
        return self._graph[u]

    def dfs(self, start):
        t_stack: deque = deque()
        d_visited: Dict[Any, List] = {n: False for n in self._graph.keys()}
        path: List[Any] = []

        t_stack.append(start)
        print(f'DFS, start={start}:', end=" ")

        while not (len(t_stack) == 0):
            node = t_stack.pop()

            if not d_visited[node]:
                path.append(node)
                d_visited[node] = True
                for neighbour in self.neighbours(node):
                    if not d_visited[neighbour]:
                        t_stack.append(neighbour)
        print(path)

graph = Graph()
graph.add_edge(0, 1)
graph.add_edge(0, 2)
graph.add_edge(1, 2)
graph.add_edge(2, 0)
graph.add_edge(2, 3)
graph.add_edge(3, 3)

graph.dfs(2)
```

5. Presentation of Results

```
1 graph.dfs(2)
```

```
DFS, start=2: [2, 3, 0, 1]
```

6. Analysis and Discussions

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

7. Conclusions

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

8. Comments

1. Limitations of Experiments

The disadvantage of Depth-First Search is that there is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree. One solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.

2. Limitations of Results

- Depth-First Search is not guaranteed to find the solution.
- And there is no guarantee to find a minimal solution, if more than one solution exists.

3. Learning happened

We learnt to implement DFS

4. Recommendations

None

Laboratory 3

Title of the Laboratory Exercise: **Genetic Algorithm**

1. Introduction and Purpose of Experiment

   In this laboratory exercises students get to implement GA

2. Aim and Objectives

   Aim

   - To implement the algorithm in Python

   Objectives

   At the end of this lab, the student will be able to

   - Develop the program using Python

3. Experimental Procedure

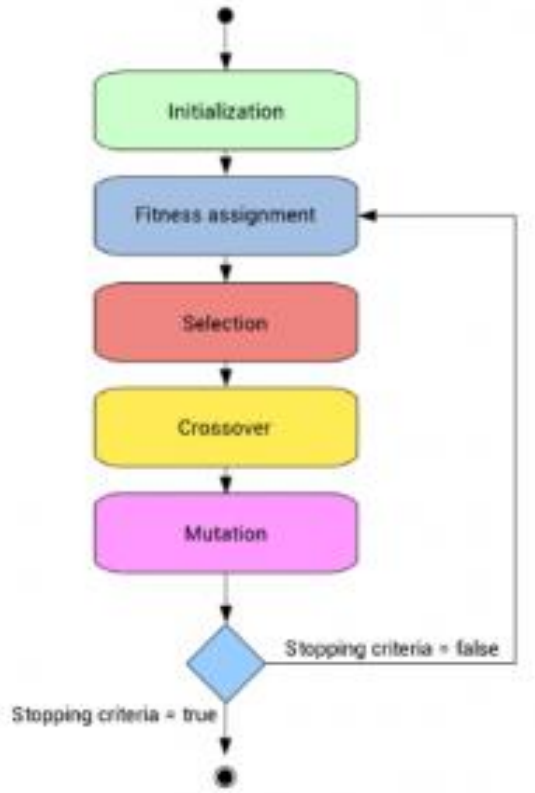
   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in Python language
   iv. Test the implemented program
   v. Document the Results
   vi. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm:

```
Simple Genetic Algorithm ()
{
      Initialization;
      Evaluation;
      while termination criterion has not been reached
      {
            Selection_and_Reproduction;
            Crossover;
            Mutation;
            Evaluation;
      }
}
```

NAME: SATYAJIT GHANA ROLL NO: 17ETCS002159



Source Code:

```
import random
import math

def generate_population(size, x_boundaries, y_boundaries):
    lower_x_boundary, upper_x_boundary = x_boundaries
    lower_y_boundary, upper_y_boundary = y_boundaries

    population = []
    for i in range(size):
        individual = {
            "x": random.uniform(lower_x_boundary, upper_x_boundary),
            "y": random.uniform(lower_y_boundary, upper_y_boundary),
        }
        population.append(individual)

    return population

def apply_function(individual):
    x = individual["x"]
    y = individual["y"]
    return math.sin(math.sqrt(x ** 2 + y ** 2))

def choice_by_roulette(sorted_population, fitness_sum):
    offset = 0
```

```
    normalized_fitness_sum = fitness_sum

    lowest_fitness = apply_function(sorted_population[0])
    if lowest_fitness < 0:
        offset = -lowest_fitness
        normalized_fitness_sum += offset * len(sorted_population)

    draw = random.uniform(0, 1)

    accumulated = 0
    for individual in sorted_population:
        fitness = apply_function(individual) + offset
        probability = fitness / normalized_fitness_sum
        accumulated += probability

        if draw <= accumulated:
            return individual

def sort_population_by_fitness(population):
    return sorted(population, key=apply_function)

def crossover(individual_a, individual_b):
    xa = individual_a["x"]
    ya = individual_a["y"]

    xb = individual_b["x"]
    yb = individual_b["y"]

    return {"x": (xa + xb) / 2, "y": (ya + yb) / 2}

def mutate(individual):
    next_x = individual["x"] + random.uniform(-0.05, 0.05)
    next_y = individual["y"] + random.uniform(-0.05, 0.05)

    lower_boundary, upper_boundary = (-4, 4)

    # Guarantee we keep inside boundaries
    next_x = min(max(next_x, lower_boundary), upper_boundary)
    next_y = min(max(next_y, lower_boundary), upper_boundary)

    return {"x": next_x, "y": next_y}

def make_next_generation(previous_population):
    next_generation = []
    sorted_by_fitness_population = sort_population_by_fitness(previous_population)
    population_size = len(previous_population)
    fitness_sum = sum(apply_function(individual) for individual in population)

    for i in range(population_size):
```

```
        first_choice = choice_by_roulette(sorted_by_fitness_population, fitness_sum)
        second_choice = choice_by_roulette(sorted_by_fitness_population, fitness_sum)

        individual = crossover(first_choice, second_choice)
        individual = mutate(individual)
        next_generation.append(individual)

    return next_generation

generations = 100

population = generate_population(size=10, x_boundaries=(-4, 4), y_boundaries=(-4, 4))

i = 1
while True:
    if i == 1 or i % 20 == 0 or i == generations:
        print(f"🧬 GENERATION {i}")

        for individual in population:
            print(individual, apply_function(individual))

    if i == generations:
        break

    i += 1

    population = make_next_generation(population)

best_individual = sort_population_by_fitness(population)[-1]
print("\n🔬 FINAL RESULT")
print(best_individual, apply_function(best_individual))
```

5. Presentation of Results

```
🧬 GENERATION 1
{'x': -1.3011230969271406, 'y': -3.633537495598837} -0.6577880831654199
{'x': -2.119026984622984, 'y': 2.102465533380462} 0.15588534887550368
{'x': 0.2258877280628413, 'y': -0.852273106392226} 0.7718208579881026
{'x': 3.523665033526653, 'y': -2.3916619172710982} -0.8988228302038925
{'x': 1.0716611510972722, 'y': 3.006856846809704} -0.05050829406013226
{'x': -0.3371910094634556, 'y': 3.3656225947712546} -0.2385561596849194
{'x': 1.233782611074913, 'y': 3.8997458799803164} -0.812640355233504
{'x': 2.0807337670398125, 'y': 1.432221135489133} 0.577436926986806
{'x': 1.284542147162064, 'y': 0.8818566304294659} 0.9999195820297072
{'x': 1.9490102787696877, 'y': -3.0661837176260596} -0.4720412396341789
🧬 GENERATION 20
{'x': -0.25501248268691085, 'y': 1.298605067947228} 0.9695550471145489
{'x': -0.3102677233192787, 'y': 1.3383850186501984} 0.980674139993738
```

{'x': -0.2969637219238426, 'y': 1.2698228993778777} 0.9646428715765631
{'x': -0.2884059712581866, 'y': 1.3048660531354859} 0.972645031881805
{'x': -0.34106324692687945, 'y': 1.2817862310302959} 0.9702802089328333
{'x': -0.3543749463396456, 'y': 1.3141230707384843} 0.9780871251363045
{'x': -0.35403714593777347, 'y': 1.3262495016049205} 0.980441193228268
{'x': -0.31120755312537224, 'y': 1.3095744665671967} 0.9748494153264677
{'x': -0.3171700634599102, 'y': 1.2907312687456638} 0.9709403112098671
{'x': -0.3471483479124146, 'y': 1.2741460491933512} 0.9688615595810348
GENERATION 40
{'x': -0.3204575268642972, 'y': 1.0913054445320722} 0.907537635434244
{'x': -0.3213674462397161, 'y': 1.1668506166257784} 0.9357205984890135
{'x': -0.314921918018518, 'y': 1.2799972977063667} 0.9682590088821852
{'x': -0.26396881748109796, 'y': 1.1918741287436063} 0.939358640251267
{'x': -0.3570451360562707, 'y': 1.1121533444999891} 0.9199923281717582
{'x': -0.35002600333563744, 'y': 1.266502291665258} 0.9672038119773779
{'x': -0.36115151839379267, 'y': 1.1760739784370693} 0.9425812037465775
{'x': -0.3423333235437869, 'y': 1.2789076269853756} 0.969683516430281
{'x': -0.33789703450129904, 'y': 1.1459467448996619} 0.9301147707241133
{'x': -0.3623542472234567, 'y': 1.1803057273039872} 0.9440410388928706
GENERATION 60
{'x': -0.364734012287457, 'y': 1.2080735416817012} 0.9526794277899364
{'x': -0.34625019531068163, 'y': 1.3058858618331266} 0.9759439811726925
{'x': -0.35768189640645104, 'y': 1.2824954516578737} 0.9714906383411136
{'x': -0.2779159116655593, 'y': 1.2855394231524784} 0.967522104405859
{'x': -0.3422394198130492, 'y': 1.25512946387112} 0.9638125802892203
{'x': -0.302355056938097, 'y': 1.226511339473213} 0.9530729958077736
{'x': -0.318109037826522, 'y': 1.2963059974503004} 0.9722741244061192
{'x': -0.24634206847433218, 'y': 1.3015516148137143} 0.9698607908043188
{'x': -0.3880079039347942, 'y': 1.18830880464998} 0.9490007558200467
{'x': -0.3055694526132662, 'y': 1.299060058985006} 0.9722151461816764
GENERATION 80
{'x': -0.3517552480542303, 'y': 1.302665573449483} 0.9755746056177922
{'x': -0.3206454911066755, 'y': 1.231181461818631} 0.9557652150830914
{'x': -0.39665252053874256, 'y': 1.2044979320733122} 0.9545445288509216
{'x': -0.3340772200176675, 'y': 1.2120931249538613} 0.9512579831242151
{'x': -0.35446519164395607, 'y': 1.1871338589928904} 0.9454339309072595
{'x': -0.35324099780557316, 'y': 1.2174264152774843} 0.9543983977474785
{'x': -0.31900861204607867, 'y': 1.2884931831623017} 0.9705241885137081
{'x': -0.35120883934971814, 'y': 1.2634345137213314} 0.966529781509391
{'x': -0.3286180849137487, 'y': 1.3274918756156289} 0.9794187954446535
{'x': -0.3462668631667928, 'y': 1.256706377197603} 0.9644985982620475
GENERATION 100
{'x': -0.33365874741221907, 'y': 1.2199764352280804} 0.9535415276564833
{'x': -0.38070421528797255, 'y': 1.299428722627413} 0.9766022749334101
{'x': -0.26381480303819627, 'y': 1.2425266657944485} 0.955167403106868
{'x': -0.31317120669100784, 'y': 1.2582723134718474} 0.9626591852350879
{'x': -0.31124126616042935, 'y': 1.225980431279141} 0.9535687340997426
{'x': -0.32901087904617515, 'y': 1.2375843672528115} 0.9581795457050271
{'x': -0.32161690681596833, 'y': 1.249702366633856} 0.9609524063375459
{'x': -0.31679939402461466, 'y': 1.2254501004325749} 0.9538290941135198
{'x': -0.290518915565998, 'y': 1.2403488443796566} 0.9562542843573523
{'x': -0.31364326697100264, 'y': 1.351666644784323} 0.9832625615919761

```
🔬 FINAL RESULT
{'x': -0.31364326697100264, 'y': 1.351666644784323} 0.9832625615919761
```

6. Analysis and Discussions

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

7. Conclusions

Genetic algorithms operate on string structures, like biological structures, which are evolving in time according to the rule of survival of the fittest by using a randomized yet structured information exchange. Thus, in every generation, a new set of strings is created, using parts of the fittest members of the old set. The main characteristics of a genetic algorithm are as follows:

a. The genetic algorithm works with a coding of the parameter set, not the parameters themselves.
b. The genetic algorithm initiates its search from a population of points, not a single point.
c. The genetic algorithm uses payoff information, not derivatives.
d. The genetic algorithm uses probabilistic transition rules, not deterministic ones.

8. Comments

1. Limitations of Experiments

- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.

2. Limitations of Results

- One major obstacle of genetic algorithms is the coding of the fitness (evaluation) function so that a higher fitness can be attained and better solutions for the problem at hand are produced. A wrong choice of the fitness function may lead to critical problems such as unable to find the solution to a problem or even worse, returning a wrong solution to the problem.

3. Learning happened

We learnt to implement GA

4. Recommendations

None

NAME: SATYAJIT GHANA ROLL NO: 17ETCS002159

Laboratory 4

Title of the Laboratory Exercise: **Hill Climbing**

1. Introduction and Purpose of Experiment

   In this laboratory exercises students get to implement Hill Climbing

2. Aim and Objectives

   Aim

   - To implement the algorithm in Python

   Objectives

   At the end of this lab, the student will be able to

   - Develop the program using Python

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in Python language
   iv. Test the implemented program
   v. Document the Results
   vi. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm:

```
Pseudocode
algorithm Discrete Space Hill Climbing is
    currentNode := startNode
    loop do
        L := NEIGHBORS(currentNode)
        nextEval := -INF
        nextNode := NULL
        for all x in L do
            if EVAL(x) > nextEval then
                nextNode := x
                nextEval := EVAL(x)
        if nextEval ≤ EVAL(currentNode) then
            // Return current node since no better neighbors exist
            return currentNode
```

```
currentNode := nextNode
```

Source Code:

```
import numpy as np
import matplotlib.pyplot as plt

class HillClimbing:
    def __init__(self, init_point):
        self.init_point = np.array(init_point)
        self.best_point = self.init_point.copy()
        self.history = []

    def eval_function(self, inp):
        # simple sphere function as `eval_function`
        return np.sqrt(np.dot(inp, inp))

    def perform_climb(self, max_iters: int = 100, step_size: int = 2, print_interval = 10):
        for it in range(1, max_iters+1):
            new_point = self.best_point + np.random.randn(*self.best_point.shape) * step_size
            if self.eval_function(new_point) < self.eval_function(self.best_point):
                self.best_point = new_point

            if it % print_interval == 0 or it == 1 or it == max_iters:
                print(f'iter: {it}, current best: {self.best_point}, eval_val: {self.eval_function
(self.best_point)}')
            self.history.append(self.eval_function(self.best_point))

hc = HillClimbing(init_point = np.array([10, 10]))

hc.perform_climb(max_iters=5000, step_size=0.01, print_interval=200)

plt.plot(hc.history)
plt.xlabel('iterations')
plt.ylabel('loss')
```

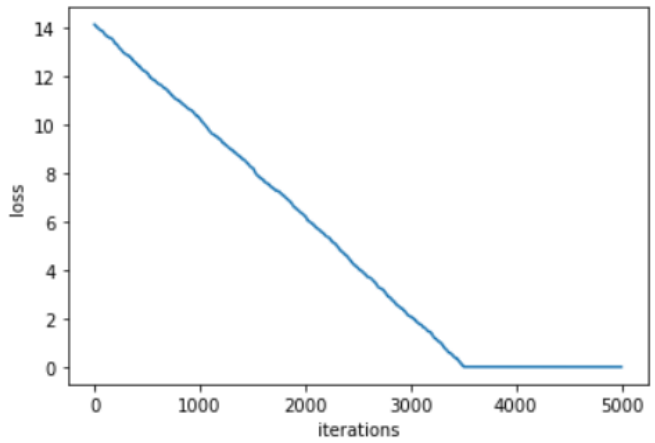
5. Presentation of Results

```
1 hc.perform_climb(max_iters=5000, step_size=0.01, print_interval=200)
```

```
iter: 1, current best: [ 9.98635198 10.00550587], eval_val: 14.136384742915844
iter: 200, current best: [9.55343791 9.30183383], eval_val: 13.333877475867638
iter: 400, current best: [8.96416743 8.70970694], eval_val: 12.498611630062948
iter: 600, current best: [8.57865281 7.98539969], eval_val: 11.720063663596903
iter: 800, current best: [8.11924282 7.42828176], eval_val: 11.00461148524459
iter: 1000, current best: [7.53116891 6.93846841], eval_val: 10.240158642490599
iter: 1200, current best: [6.80079027 6.41440563], eval_val: 9.348547896381062
iter: 1400, current best: [6.32146629 5.84299991], eval_val: 8.608227695997956
iter: 1600, current best: [5.85878055 5.07542102], eval_val: 7.7514648928218195
iter: 1800, current best: [5.3086119 4.6581421], eval_val: 7.062552520560365
iter: 2000, current best: [4.57303044 4.20183023], eval_val: 6.210312765726735
iter: 2200, current best: [3.83323265 3.80406831], eval_val: 5.400426677602964
iter: 2400, current best: [3.25867389 3.19213541], eval_val: 4.561653646390361
iter: 2600, current best: [2.69460381 2.55140574], eval_val: 3.7108706448977657
iter: 2800, current best: [2.10608543 1.96339766], eval_val: 2.879327385440341
iter: 3000, current best: [1.47321448 1.46170393], eval_val: 2.0753166713413242
iter: 3200, current best: [1.0520408  0.89351662], eval_val: 1.3802759840377845
iter: 3400, current best: [0.35712524 0.33189577], eval_val: 0.4875379300573248
iter: 3600, current best: [-0.00130743  0.00077312], eval_val: 0.0015189103937671474
iter: 3800, current best: [0.00012122 0.00043111], eval_val: 0.0004478245099984501
iter: 4000, current best: [0.00012122 0.00043111], eval_val: 0.0004478245099984501
iter: 4200, current best: [0.00012122 0.00043111], eval_val: 0.0004478245099984501
iter: 4400, current best: [0.00012122 0.00043111], eval_val: 0.0004478245099984501
iter: 4600, current best: [0.00012122 0.00043111], eval_val: 0.0004478245099984501
iter: 4800, current best: [0.00012122 0.00043111], eval_val: 0.0004478245099984501
iter: 5000, current best: [0.00012122 0.00043111], eval_val: 0.0004478245099984501
```

```
1 plt.plot(hc.history)
2 plt.xlabel('iterations')
3 plt.ylabel('loss')
```

```
Text(0, 0.5, 'loss')
```



6. Analysis and Discussions

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.

It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.

A node of hill climbing algorithm has two components which are state and value.

Hill Climbing is mostly used when a good heuristic is available.

7. Conclusions

Following are some main features of Hill Climbing Algorithm:

- Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
- No backtracking: It does not backtrack the search space, as it does not remember the previous states.

8. Comments

1. Limitations of Experiments

Local Maxima: Hill-climbing algorithm reaching on the vicinity a local maximum value, gets drawn towards the peak and gets stuck there, having no other place to go.

Ridges: These are sequences of local maxima, making it difficult for the algorithm to navigate.

Plateaux: This is a flat state-space region. As there is no uphill to go, algorithm often gets lost in the plateau.

2. Limitations of Results

None

3. Learning happened

We learnt to implement Hill Climbing

4. Recommendations

None

Laboratory 5

Title of the Laboratory Exercise: **Min Max – Alpha Beta Pruning**

1. Introduction and Purpose of Experiment

   In this laboratory exercises students get to implement Alpha Beta Pruning

2. Aim and Objectives

   Aim

   - To implement the algorithm in Python

   Objectives

   At the end of this lab, the student will be able to

   - Develop the program using Python

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in Python language
   iv. Test the implemented program
   v. Document the Results
   vi. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm:

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

```
    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

Source Code:

```python
import numpy as np


def minmax(maximizing_player, values):
    alpha = -np.inf
    beta = np.inf

    def _minmax(depth, node_idx, maximizing_player, values, alpha, beta):

        if depth == 3:
            return values[node_idx]

        if maximizing_player:

            best = -np.inf

            # left and right children
            for i in range(0, 2):
                val = _minmax(depth + 1, node_idx * 2 + i, False, values, alpha, beta)

                best = max(best, val)
                alpha = max(alpha, best)

                if beta <= alpha:
                    break

            return best

        else:

            best = np.inf

            # left and right children
            for i in range(0, 2):
                val = _minmax(depth + 1, node_idx * 2 + i, True, values, alpha, beta)
```

```
                best = min(best, val)
                beta = min(beta, best)

                if beta <= alpha:
                    break

            return best
    return _minmax(depth=0, node_idx=0, maximizing_player=maximizing_player, values=val
ues, alpha=alpha, beta=beta)

values = [3, 5, 6, 9, 1, 2, 0, -1]

optimal_val = minmax(maximizing_player=True, values=values)

print(f'values: {values}')
print(f'optimal value: {optimal_val}')
```

5. Presentation of Results

```
1 values = [3, 5, 6, 9, 1, 2, 0, -1]
2
3 optimal_val = minmax(maximizing_player=True, values=values)
4
5 print(f'values: {values}')
6 print(f'optimal value: {optimal_val}')
```

```
values: [3, 5, 6, 9, 1, 2, 0, -1]
optimal value: 5
```

6. Analysis and Discussions

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.

7. Conclusions

Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

The two-parameter can be defined as:

- Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.
- Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

8. Comments

1. Limitations of Experiments

- It does not solve all the problems associated with the original minimax algorithm.
- Requires a set depth limit, as in most cases, it is not feasible to search the entire game tree.
- Though designed to calculate the good move, it also calculates the values of all the legal moves.

2. Limitations of Results

None

3. Learning happened

We learnt to implement Alpha-Beta Pruning

4. Recommendations

None

Laboratory 6

Title of the Laboratory Exercise: **Fuzzy Logic**

1. Introduction and Purpose of Experiment

   In this laboratory exercises students get to implement Fuzzy Logic

2. Aim and Objectives

   Aim

   - To implement the algorithm in Python

   Objectives

   At the end of this lab, the student will be able to

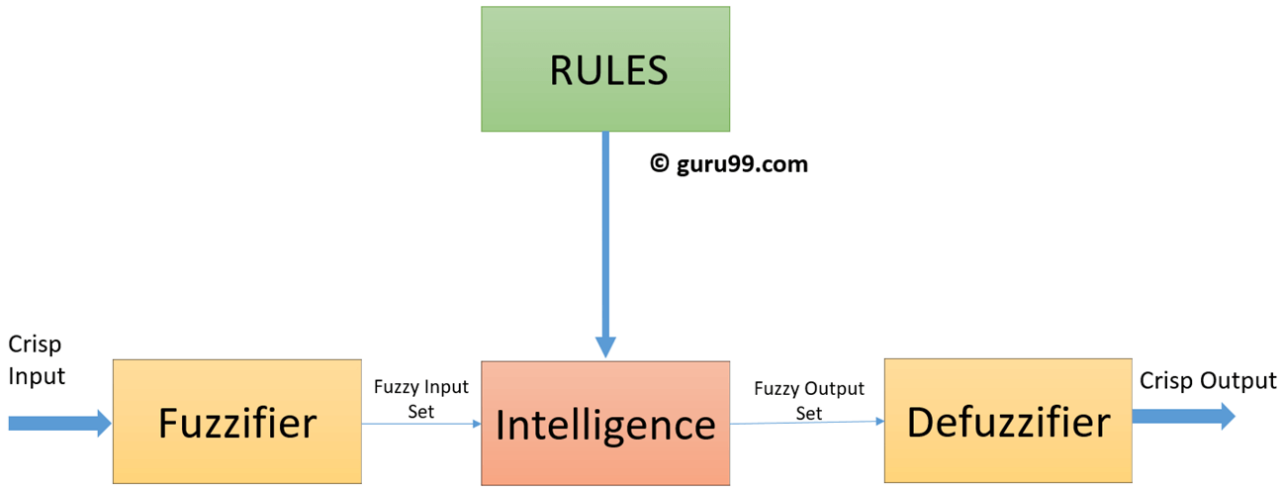
   - Develop the program using Python

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in Python language
   iv. Test the implemented program
   v. Document the Results
   vi. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm:

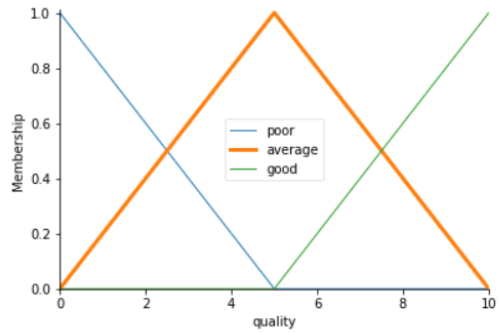


Source Code:

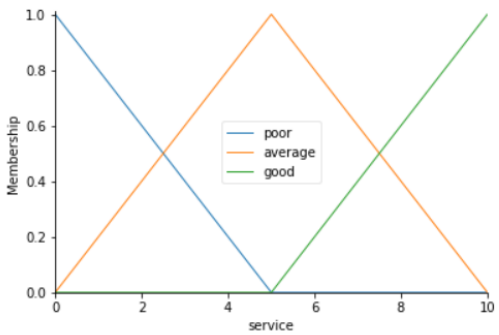
```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# New Antecedent/Consequent objects hold universe variables and membership
# functions
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip     = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)
quality.automf(3)
service.automf(3)

# Custom membership functions can be built interactively with a familiar,
# Pythonic API
tip['low']    = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high']   = fuzz.trimf(tip.universe, [13, 25, 25])

quality['average'].view()
service.view()
tip.view()

rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()

tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
# Note: if you like passing many inputs all at once, use .inputs(dict_of_data)
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers
tipping.compute()

print(tipping.output['tip'])
tip.view(sim=tipping)
```

5. Presentation of Results

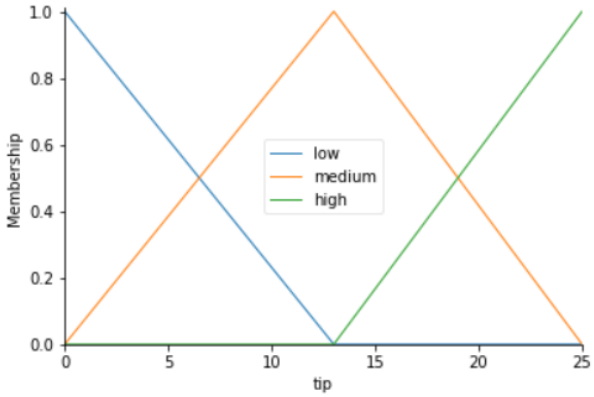
```
quality['average'].view()
```




```
service.view()
```

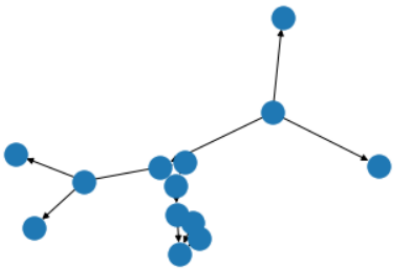



```
tip.view()
```



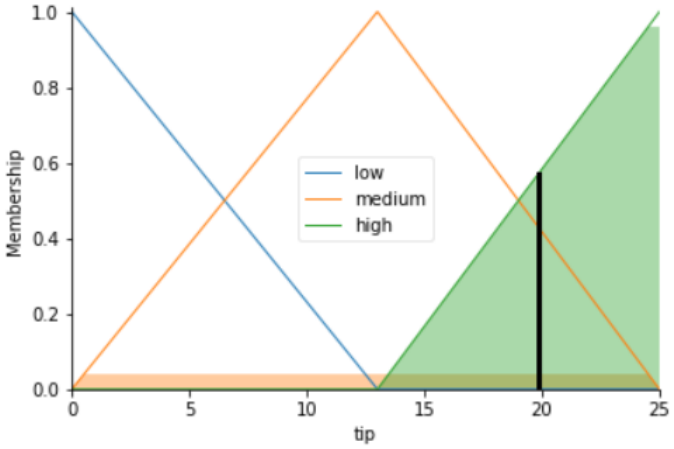

```
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()
```

```
(<Figure size 432x288 with 1 Axes>, <AxesSubplot:>)
```



```
print(tipping.output['tip'])
tip.view(sim=tipping)
```

```
19.847607361963192
```



6. Analysis and Discussions

Fuzzy Logic (FL) is a method of reasoning that resembles human reasoning. The approach of FL imitates the way of decision making in humans that involves all intermediate possibilities between digital values YES and NO.

The conventional logic block that a computer can understand takes precise input and produces a definite output as TRUE or FALSE, which is equivalent to human's YES or NO.

7. Conclusions

Fuzzy logic is useful for commercial and practical purposes.

- It can control machines and consumer products.
- It may not give accurate reasoning, but acceptable reasoning.
- Fuzzy logic helps to deal with the uncertainty in engineering.

Algorithm

- Define linguistic Variables and terms (start)
- Construct membership functions for them. (start)
- Construct knowledge base of rules (start)
- Convert crisp data into fuzzy data sets using membership functions. (fuzzification)
- Evaluate rules in the rule base. (Inference Engine)
- Combine results from each rule. (Inference Engine)
- Convert output data into non-fuzzy values. (defuzzification)

8. Comments

1. Limitations of Experiments

- A major drawback of Fuzzy Logic control systems is that they are completely dependent on human knowledge and expertise
- You have to regularly update the rules of a Fuzzy Logic control system
- These systems cannot recognize machine learning or neural networks
- The systems require a lot of testing for validation and verification

2. Limitations of Results

None

3. Learning happened

We learnt to implement Fuzzy Logic

4. Recommendations

None

NAME: SATYAJIT GHANA ROLL NO: 17ETCS002159

Laboratory 7

Title of the Laboratory Exercise: **Neural Network**

1. Introduction and Purpose of Experiment

   In this laboratory exercises students get to implement Neural Network

2. Aim and Objectives

   Aim

   - To implement the algorithm in Python

   Objectives

   At the end of this lab, the student will be able to

   - Develop the program using Python

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in Python language
   iv. Test the implemented program
   v. Document the Results
   vi. Analyse and discuss the outcomes of your experiment

4. Calculations/Computations/Algorithms

Algorithm:

Hypothesis: $h_\theta(x) = \theta_0 + \theta_1 x$

Parameters: θ_0, θ_1

Cost Function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

NAME: SATYAJIT GHANA ROLL NO: 17ETCS002159

Gradient descent algorithm

$$\text{repeat until convergence } \{$$
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$
$$(\text{for } j = 1 \text{ and } j = 0)$$
$$\}$$

Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

class NeuralNet:
    def __init__(self, w_init, b_init):
        self.w = w_init
        self.b = b_init
        self.w_h = []
        self.b_h = []
        self.e_h = []

    def sigmoid(self, x, w=None, b=None):
        w = self.w if w is None else w
        b = self.b if b is None else b

        return 1. / (1. + np.exp(-(w * x + b)))

    def error(self, X, Y, w=None, b=None):
        w = self.w if w is None else w
        b = self.b if b is None else b

        err = 0
        for x, y in zip(X, Y):
            err += 0.5 * (self.sigmoid(x, w, b) - y) ** 2

        return err

    def grad_w(self, x, y, w=None, b=None):
        w = self.w if w is None else w
        b = self.b if b is None else b
```

```
        y_pred = self.sigmoid(x)

        return (y_pred - y) * y_pred * (1 - y_pred) * x

    def grad_b(self, x, y, w=None, b=None):
        w = self.w if w is None else w
        b = self.b if b is None else b

        y_pred = self.sigmoid(x, w, b)

        return (y_pred - y) * y_pred * (1 - y_pred)

    def fit(self, X, Y, epochs=100, eta=0.01):
        self.w_h = []
        self.b_h = []
        self.e_h = []
        self.X = X
        self.Y = Y

        for i in range(epochs):
            dw, db = 0, 0
            for x, y in zip(X, Y):
                dw += self.grad_w(x, y)
                db += self.grad_b(x, y)

            self.w -= eta * dw / X.shape[0]
            self.b -= eta * db / X.shape[0]

            # log the error
            self.e_h.append(self.error(self.X, self.Y))
            self.w_h.append(self.w)
            self.b_h.append(self.b)

X = np.asarray([3.5, 0.35, 3.2, -2.0, 1.5, -0.5])
Y = np.asarray([0.5, 0.50, 0.5, 0.5, 0.1, 0.3])

w_init = -6
b_init = 4.0

nn = NeuralNet(w_init=w_init, b_init=b_init)

nn.fit(X, Y, epochs=4000, eta=1)

plt.plot(nn.e_h, 'r')
plt.plot(nn.w_h, 'b')
plt.plot(nn.b_h, 'g')

plt.show()
```

```
w_min = -7
w_max = 5

b_min = -7
b_max = 5

W = np.linspace(w_min, w_max, 256)
b = np.linspace(b_min, b_max, 256)
WW, BB = np.meshgrid(W, b)
Z = nn.error(X, Y, WW, BB)

fig = plt.figure(dpi=100)
ax = plt.subplot(111)
ax.set_xlabel('w')
ax.set_xlim(w_min - 1, w_max + 1)
ax.set_ylabel('b')
ax.set_ylim(b_min - 1, b_max + 1)
title = ax.set_title('Epoch 0')
cset = plt.contourf(WW, BB, Z, 25, alpha=0.6, cmap=cm.bwr)
```

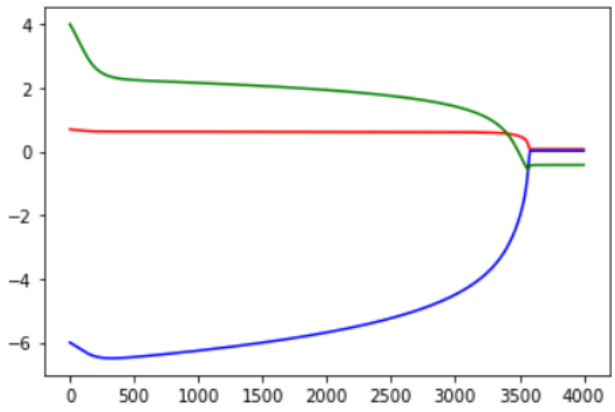
5. Presentation of Results

```
w_init = -6
b_init = 4.0
```

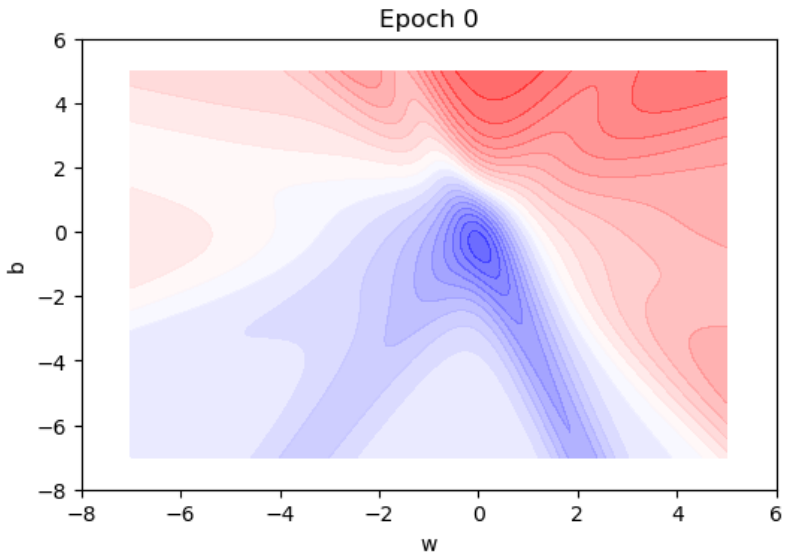
```
nn = NeuralNet(w_init=w_init, b_init=b_init)
```

```
nn.fit(X, Y, epochs=4000, eta=1)
```

```
plt.plot(nn.e_h, 'r')
plt.plot(nn.w_h, 'b')
plt.plot(nn.b_h, 'g')

plt.show()
```

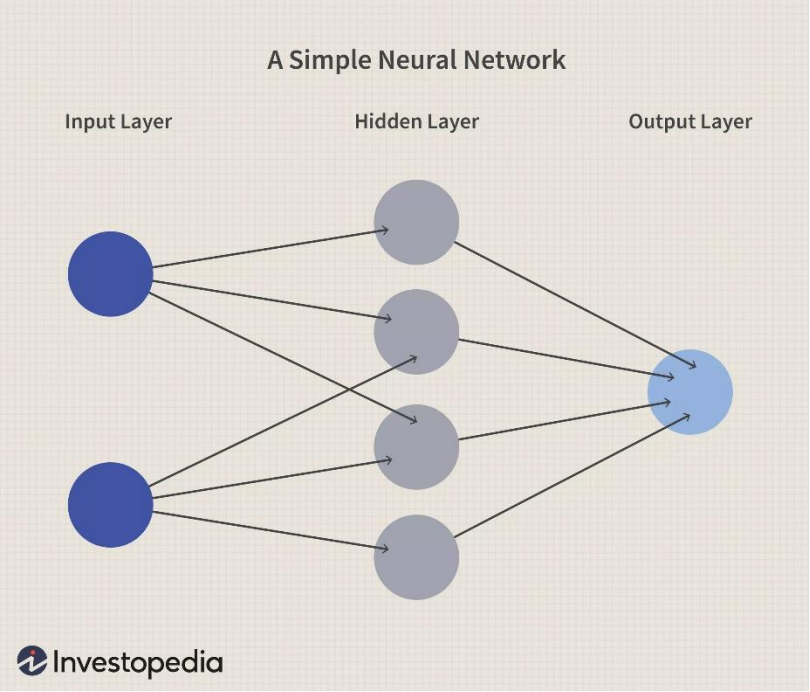


```
w_min = -7
w_max = 5

b_min = -7
b_max = 5

W = np.linspace(w_min, w_max, 256)
b = np.linspace(b_min, b_max, 256)
WW, BB = np.meshgrid(W, b)
Z = nn.error(X, Y, WW, BB)

fig = plt.figure(dpi=100)
ax = plt.subplot(111)
ax.set_xlabel('w')
ax.set_xlim(w_min - 1, w_max + 1)
ax.set_ylabel('b')
ax.set_ylim(b_min - 1, b_max + 1)
title = ax.set_title('Epoch 0')
cset = plt.contourf(WW, BB, Z, 25, alpha=0.6, cmap=cm.bwr)
```



6. Analysis and Discussions

A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria. The concept of neural networks, which has its roots in artificial intelligence, is swiftly gaining popularity in the development of trading systems.



7. Conclusions

A neural network works similarly to the human brain's neural network. A "neuron" in a neural network is a mathematical function that collects and classifies information according to a specific architecture. The network bears a strong resemblance to statistical methods such as curve fitting and regression analysis.

A neural network contains layers of interconnected nodes. Each node is a perceptron and is similar to a multiple linear regression. The perceptron feeds the signal produced by a multiple linear regression into an activation function that may be nonlinear.

In a multi-layered perceptron (MLP), perceptrons are arranged in interconnected layers. The input layer collects input patterns. The output layer has classifications or output signals to which input patterns may map. For instance, the patterns may comprise a list of quantities for technical indicators about a security; potential outputs could be "buy," "hold" or "sell."

8. Comments

1. Limitations of Experiments

- Hardware Dependence:

  - Artificial Neural Networks require processors with parallel processing power, by their structure.
  - For this reason, the realization of the equipment is dependent.
- Unexplained functioning of the network:
  - This the most important problem of ANN.
  - When ANN gives a probing solution, it does not give a clue as to why and how.
  - This reduces trust in the network.

2. Limitations of Results

None

3. Learning happened

We learnt to implement Neural Network

4. Recommendations

None