SCT Experiment No: 6

Name: Abhishek S Waghchaure

PRN: 1032221714

Dept: FY M Tech DSA(2022-24)

Aim:

Implementing a genetic algorithm for an optimization problem compare the results with classical approaches.

Introduction:

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

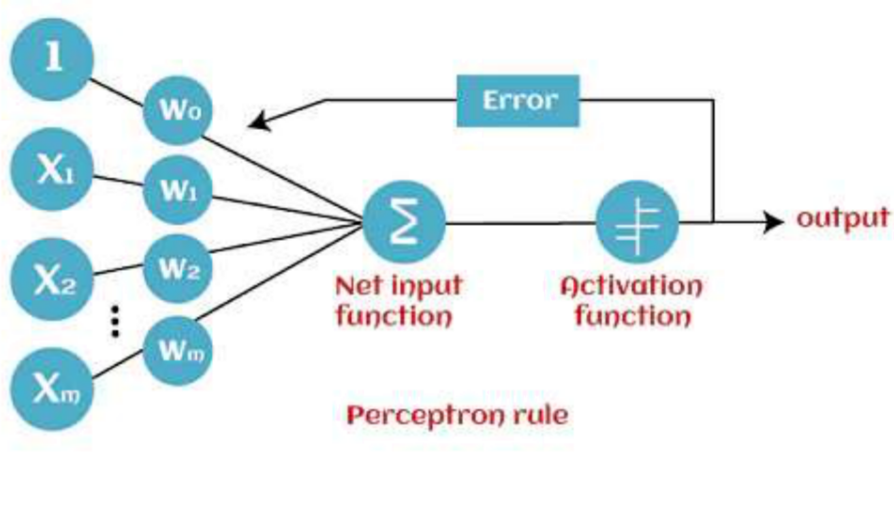
Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate "survival of the fittest" among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Fitness Score:

A Fitness Score is given to each individual which shows the ability of an individual to "compete". The individual having optimal fitness score (or near optimal) are sought.

The GAs maintains the population of n individuals (chromosome/solutions) along with their fitness scores.The individuals having better fitness scores are given more chance to reproduce than others. The individuals with better fitness scores are selected who mate and produce better offspring by combining chromosomes of parents. The population size is static so the room has to be created for new arrivals. So, some individuals die and get replaced by new arrivals eventually creating new generation when all the mating opportunity of the old population is exhausted. It is hoped that over successive generations better solutions will arrive while least fit die.

Each new generation has on average more "better genes" than the individual (solution) of previous generations. Thus each new generations have better "partial solutions" than previous generations. Once the offspring produced having no significant difference from offspring produced by previous populations, the population is converged. The algorithm is said to be converged to a set of solutions for the problem.


1
W_0
Error
X_1
W_1
output
X_2
W_2
Net input
function
Activation
function
W_m
X_m
Perceptron rule

```
        '''
        global GENES
        gene = random.choice(GENES)
        return gene
@classmethod
def create_gnome(self):
        '''
        create chromosome or string of genes
        '''
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in
range(gnome_len)] def mate(self, par2):
        '''
        Perform mating and produce new offspring
        '''
        # chromosome for offspring
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
                # random probability
                prob = random.random()

                # if prob is less than 0.45, insert gene
                # from parent 1
                if prob < 0.45:
                        child_chromosome.append(gp1)
                # if prob is between 0.45 and 0.90, insert
                # gene from parent 2
                elif prob < 0.90:
                        child_chromosome.append(gp2)
```

```
                # otherwise insert random gene(mutate),
                # for maintaining diversity
                else:
            child_chromosome.append(self.mutated_genes())
        # create new Individual(offspring) using
        # generated chromosome for offspring
        return Individual(child_chromosome)

    def cal_fitness(self):
        '''
        Calculate fitness score, it is the number of
        characters in string which differ from target
        string.
        '''
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt: fitness+= 1
        return fitness

# Driver code
def main():
    global POPULATION_SIZE
    #current generation
    generation = 1
    found = False
    population = []
    # create initial population
    for _ in range(POPULATION_SIZE):
                gnome = Individual.create_gnome()
```

```
                    population.append(Individual(gnome))
while not found:
    #  sort the population in increasing order of fitness score
    population = sorted(population, key = lambda x:x.fitness)
    # if the individual having lowest fitness score ie.
    # 0 then we know that we have reached to the target
    # and break the loop
    if population[0].fitness <= 0:
        found = True
        break

    # Otherwise generate new offsprings for new
    generation new_generation = []

    # Perform Elitism, that mean 10% of fittest population
    # goes to the next generation
    s = int((10*POPULATION_SIZE)/100)
    new_generation.extend(population[:s])

    # From 50% of fittest population, Individuals
    # will mate to produce offspring
    s = int((90*POPULATION_SIZE)/100)
    for _ in range(s):
        parent1 = random.choice(population[:50])
        parent2 = random.choice(population[:50])
        child = parent1.mate(parent2)
        new_generation.append(child)
    population = new_generation
    print("Generation: {}\tString: {}\tFitness: {}".\
        format(generation,
```

```
                    "".join(population[0].chromosome),
                    population[0].fitness))
            generation += 1
        print("Generation: {}\tString: {}\tFitness: {}".\
            format(generation,
            "".join(population[0].chromosome),
            population[0].fitness))
if __name__ == '__main__':
    main()
```

<u>Output</u>:

Generation: 1	String: E6KqYeGbWm3d S?F$I,ZVDP	Fitness: 21
Generation: 2	String: E6KqYeGbWm3d S?F$I,ZVDP	Fitness: 21
Generation: 3	String: do7)zyhv(LCocb!]xt8a7ID	Fitness: 20
Generation: 4	String: do7)zyhv(LCocb!]xt8a7ID	Fitness: 20
Generation: 5	String: E,m1TetbNb9=r%aqcn/c8h	Fitness: 18
Generation: 6	String: E,myTetbQeC=#haqcn(cth	Fitness: 17
Generation: 7	String: E,myTetbQeC=#haqcn(cth	Fitness: 17
Generation: 8	String: E,myTetbQeC=#haqcn(cth	Fitness: 17
Generation: 9	String: Cos8metz=LCVP7!#cy,c;j	Fitness: 16

Generation: 10 String:
Comy

&teQ8CTshaqc,8cVh Fitness: 14

Generation: 11 String:
Comp

etbR8CT} @qc,wc:hQ   Fitness: 13

Generation: 12 String:
Comp

etbR8CT} @qc,wc:hQ   Fitness: 13

Generation: 13 String:
Comp

etbR8CT} @qc,wc:hQ   Fitness: 13

Generation: 14 String: CompTeteQSCT:Ew7Nt,cVc Fitness: 11

Generation: 15 String: CompTeteQSCT:Ew7Nt,cVc Fitness: 11

Generation: 16 String: CompTeteQSCT:Ew7Nt,cVc Fitness: 11

Generation: 17 String: nompleteNdCT7Ew7ct,cac	Fitness: 10
Generation: 18 String: nompleteNdCT7Ew7ct,cac	Fitness: 10
Generation: 19 String: nompleteNdCT7Ew7ct,cac	Fitness: 10
Generation: 20 String: nompleteNdCT7Ew7ct,cac	Fitness: 10
Generation: 21 String: Complete/SCTsPaocn1clcQ	Fitness: 9
Generation: 22 String: Complete SCT Eaqct38oJ-	Fitness: 8
Generation: 23 String: Complete SCT Eaqct38oJ-	Fitness: 8
Generation: 24 String: Complete SCT Eaqct38oJ-	Fitness: 8
Generation: 25 String: Cocplete SCT P@qct,c.NQ	Fitness: 7

Generation: 26 String: Complete S9T P

acnaca
: Fitness: 6

Generation: 27 String: Complete S9T P

acnaca
: Fitness: 6

Generation: 28 String: Complete SCT P9aDt3caL)	Fitness: 5
Generation: 29 String: Complete SCT P9aDt3caL)	Fitness: 5
Generation: 30 String: Complete SCT P9aDt3caL)	Fitness: 5
Generation: 31 String: Complete SCT P9aDt3caL)	Fitness: 5
Generation: 32 String: Complete SCTeP@actBcal5	Fitness: 4
Generation: 33 String: Complete SCTeP@actBcal5	Fitness: 4
Generation: 34 String: Complete SCTeP@actBcal5	Fitness: 4
Generation: 35 String: Complete SCTeP@actBcal5	Fitness: 4

Generation: 36 String: Complete SCT Pract?c$IY	**Fitness: 3**
Generation: 37 String: Complete SCT Pract?c$IY	**Fitness: 3**
Generation: 38 String: Complete SCT Pract?c$IY	**Fitness: 3**
Generation: 39 String: Complete SCT Pract?c$IY	**Fitness: 3**
Generation: 40 String: Complete SCT PractBcal5	**Fitness: 2**
Generation: 41 String: Complete SCT PractBcal5	**Fitness: 2**
Generation: 42 String: Complete SCT PractBcal5	**Fitness: 2**
Generation: 43 String: Complete SCT PractBcal5	**Fitness: 2**
Generation: 44 String: Complete SCT PractBcal5	**Fitness: 2**

Generation: 45 String: Complete SCT PractBcal5	Fitness: 2
Generation: 46 String: Complete SCT PractBcal5	Fitness: 2
Generation: 47 String: Complete SCT PractBcal5	Fitness: 2
Generation: 48 String: Complete SCT PractBcal5	Fitness: 2
Generation: 49 String: Complete SCT PractBcal5	Fitness: 2
Generation: 50 String: Complete SCT PractBcal5	Fitness: 2
Generation: 51 String: Lomplete SCT Practicals	Fitness: 1
Generation: 52 String: Lomplete SCT Practicals	Fitness: 1
Generation: 53 String: Lomplete SCT Practicals	Fitness: 1
Generation: 54 String: Lomplete SCT Practicals	Fitness: 1
Generation: 55 String: Lomplete SCT Practicals	Fitness: 1
Generation: 56 String: Lomplete SCT Practicals	Fitness: 1
Generation: 57 String: Lomplete SCT Practicals	Fitness: 1
Generation: 58 String: Lomplete SCT Practicals	Fitness: 1
Generation: 59 String: Lomplete SCT Practicals	Fitness: 1
Generation: 60 String: Complete SCT Practicals	Fitness: 0

Every-time algorithm start with random strings, so output may differ. As we can see from the output, our algorithm sometimes stuck at a local optimum solution, this can be further improved by updating fitness score calculation algorithm or by tweaking mutation and crossover operators.

Why use Genetic Algorithms

They are Robust

Provide optimisation over large space state.
Unlike traditional AI, they do not break on slight change in input or presence of noise

Application of Genetic Algorithms

Genetic algorithms have many applications, some of them are –

- Recurrent Neural
- Network Mutation testing
- Code breaking
- Filtering and signal processing

Learning fuzzy rule base etc

Reference:

https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications

https://link.springer.com/chapter/10.1007/978-3-540-31880-4_22

https://towardsdatascience.com/how-to-validate-the-correctness-of-an-evolutionary-optimization-algorithm-570c8b71b6d7

https://www.geeksforgeeks.org/genetic-algorithms/