GENETIC ALGORITHM CONCEPT IN CREATING NEUROID NETWORK

STEP 1 : CREATING POPULATION BASED ON THIRD MATRIX

```
motherpopulation = dict()
numpop = int(input("Enter the number of population "))
for i in range(1,(numpop//2)+1):
    key = 'FG1-'+str(i)
    print(key)
    print(' ')
    ratio = np.random.randint(4,11)
    print(ratio)
    value = np.random.uniform(a.T)
    print(value)
    motherpopulation[key] = [value, ratio]
    print(' ')


fatherpopulation = dict()
for i in range(1,(numpop//2)+1):
    key = 'MG1-'+str(i)
    print(key)
    print(' ')
    ratio = np.random.randint(4,11)
    print(ratio)
    value = np.random.uniform(a.T)
    print(value)
    fatherpopulation[key] = [value, ratio]
    print(' ')
```

Here we are generating population at equal sex ratio where we will be getting an output of equal number of mothers and fathers if there is an even number of input . This step works for the condition of giving an odd number as input too but there will be either mother or father more or less in the total population.

STEP 2 : CREATING BABIES BASED ON PARENT POPULATION

```
print(fatherpopulation, motherpopulation, sep='\n')
generation=1
def assignbaby():
    # This method picks a gender for the baby randomly
    let = [ 'M', 'F']
    import random
    pos = random.randint(0,1)
    letter = let[pos]
    return letter
```

In the above lines of the code we are able to randomly pick the gender of the baby.

```
def avg_fitness_per_gen(fatherpopulation, motherpopulation):
  generation_count = dict()
  for key in fatherpopulation.keys():
    hyphen_ind = key.find('-')
    gen_int = int(key[2:hyphen_ind])
    gen_list = generation_count.get(gen_int, [])
    gen_list.append(fatherpopulation[key][1])
    generation_count[gen_int] = gen_list
```

This method calculates the average of the fitness ratio for each generation in the current population

```
   for key in motherpopulation.keys():
    hyphen_ind = key.find('-')
    gen_int = int(key[2:hyphen_ind])
    gen_list = generation_count.get(gen_int, [])
    gen_list.append(motherpopulation[key][1])
    generation_count[gen_int] = gen_list

  gen_avg_fitness = dict()
  for key in generation_count.keys():
    avg = sum(generation_count[key]) / len(generation_count[key])
    gen_avg_fitness[key] = gen_avg_fitness.get(key, 0.0) + avg
  return gen_avg_fitness
```

Here we can acquire the average fitness ratio of the each generation

```
def create_baby(father, mother, number, generation):
    If both father and mother are unfit from this condition they are  not creating a baby
    if father[1] < 4 and mother[1] < 4: return ('', '')
```

choosing a number at random to represent the choice of rows from mother and father matrices (if a bit is 0 that index is filled with mother's row else filled with father's row at that index) to get the baby matrix

```
    r=((2**n)-2)
    comb = np.random.randint(1,r,size=1)[0]

    to ensure we get 4 bits (removing zfill strips the leading zeros if any)
    bin_comb = bin(comb)[2:].zfill(n)

    debug print, can be removed
    print("bin_comb: ", bin_comb)
    baby = []
    for i in range(len(bin_comb)):
        if bin_comb[i] == '0': baby.append(mother[0][i])
        else: baby.append(father[0][i])
    baby = np.asarray(baby)
    baby_fr = np.random.randint(5,11)
    baby_gender=assignbaby()
    key = baby_gender+'G' + str(generation + 1) + '-' + str(number)
    if baby_gender == 'F':
        motherpopulation[key] = [baby, baby_fr]
    else:
        fatherpopulation[key] = [baby, baby_fr]
    return (baby_gender, key)

for i in range(1, (numpop//2)+1):
    print("generating baby ", i)
    father_key = 'MG1-'+str(i)
    mother_key = 'FG1-'+str(i)
    baby_gender, baby_title = create_baby(fatherpopulation[father_key],
                                          motherpopulation[mother_key],
                                          i, generation)
    if baby_title != '':
        print(baby_title)
```

```
    else:
        print("This couple cannot give birth to a fit baby")
    if baby_gender == 'F':
        print(motherpopulation[baby_title])
    elif baby_gender == 'M':
        print(fatherpopulation[baby_title])
generation += 1  Manual Gen-2 creation
print(avg_fitness_per_gen(fatherpopulation, motherpopulation))
```

By the end of this part we are able to create babies randomly by choosing gender and fitness ratio automatically. And always the baby population is half of the parent population.

Outputs :

```
generating baby  1
bin_comb:  011010
MG2-1
[array([[2.6974262 , 9.89612155, 8.7290229 , 3.97521493],
       [4.45246826, 3.43121835, 2.10950489, 1.25933132],
       [8.36534822, 5.19096647, 8.39699492, 1.82391669],
       [1.00051287, 4.63783623, 1.38959603, 7.70989135],
       [2.84865323, 1.38219898, 3.7404347 , 1.36601665],
       [6.12150961, 6.29645964, 3.15409891, 8.64911152]]), 6]

generating baby  2
bin_comb:  111010
MG2-2
[array([[ 3.43131467,  3.52551355, 11.26430086,  2.99185703],
       [ 5.71568717,  2.31944948,  2.09654316,  4.31693377],
       [ 4.01783016,  8.17014323,  4.12600276,  3.02462242],
       [ 1.00073798,  4.85128825,  1.60565228,  8.19204768],
       [ 1.35790499,  3.35888527,  3.25650224,  3.59771614],
       [ 2.69492944,  9.78302788,  6.24261648,  3.21433603]]), 7]
{1: 6.5, 2: 6.5}
```

Highlighted part in the baby population is the automatically generated fitness value for the baby.

{1: 6.5, 2: 6.5}

These set of values represent the average fitness ratio of each generation

STEP 3 : CREATING GENERATION MATRIX

Here the generation matrix is a matrix which holds the values of all the matrices generated and clubbed together in a single matrix.

```
def create_pop_matrix():
  pop_matrix = []
  for key in fatherpopulation.keys():
    pop_matrix.append(key)
  for key in motherpopulation.keys():
    pop_matrix.append(key)
  return pop_matrix
pop_matrix = create_pop_matrix()
print(pop_matrix)
```

This method creates a list of all the titles currently "alive" in the father / mother populationThis is needed to update the population after every death / birth cycle later in this document

```
def delete_min(pop_matrix):
   min_fr = 11
  min_title = ""

  for title in pop_matrix:
    if title[0]=='M' and fatherpopulation[title][1] < min_fr:
      min_fr = fatherpopulation[title][1]
      min_title = title
    if title[0]=='F' and motherpopulation[title][1] < min_fr:
      min_fr = motherpopulation[title][1]
      min_title = title

  if min_title[0]=='M':
    del fatherpopulation[min_title]
  else:
    del motherpopulation[min_title]
```

```
  return min_title
```

This method is used to simulate the "death of the least fit" in the population. Simply picking minimum in the population

def createnextgen(pop_matrix, death_ratio, generation):

This method simulates one cycle of the death / birth cycle by killing floor(population * death_ratio / 100) titles and replacing them with the next generation

```
   import random
  #death_ratio = int(input("Enter the death ratio (0 - 30 %): "))
  for baby_no in range(len(pop_matrix)*death_ratio//100):
    deleted_title = delete_min(pop_matrix)
    print("Death of ->", deleted_title)
    father_title = str(random.choice(list(fatherpopulation.keys())))
    mother_title = str(random.choice(list(motherpopulation.keys())))
    baby_gender, baby_title = create_baby(fatherpopulation[father_title],
                                          motherpopulation[mother_title],
                                          baby_no+1, generation)
    pop_matrix = create_pop_matrix()
    #print(motherpopulation,fatherpopulation)
    #print(pop_matrix)
  return pop_matrix
```

By the end of this particular process we are able to generate a set of array elements which consists of titles of the parent and child population.

Output :

['MG1-1', 'MG1-2', 'MG2-1', 'MG2-2', 'FG1-1', 'FG1-2']

where MG1-1: Male Generation 1 - Matrix 1
FG1-1 : Female Generation 1 - Matrix 1

G2 , G3 …… will be the generations of the population

-1 , - 2 , - 3 , …… will the matrix number of individual generation

STEP 4 : DEATH RATIO AND RANDOM POPULATION GENERATION

```
gc = int(input("Enter No. of Generations: "))
generation = 2
dr=int(input("Enter the death ratio (0 - 30 %): "))
for _ in range(gc):
  pop_matrix = createnextgen(pop_matrix, dr, generation)
  generation += 1
  print(pop_matrix)
  print(avg_fitness_per_gen(fatherpopulation, motherpopulation))
```

Here we will be entering the count of generations and death ratio which would we ranging from 0-30 % where the death is occured to the particular element of the population which has the least fitness value and the population count which has been deceased will be compensated by the birth of latest generated population from the random selection of the parents and till the count of the input generation given we will be able to create the generation matrix till that particular range with individual average fitness ratio of that particular generation.

OUTPUT:

```
Enter No. of Generations: 10
Enter the death ratio (0 - 30 %): 20
Death of -> MG1-2
bin_comb:  011011
['MG1-1', 'MG2-1', 'MG2-2', 'FG1-1', 'FG1-2', 'FG3-1']
{1: 7.0, 2: 6.5, 3: 7.0}
Death of -> FG1-2
bin_comb:  000010
['MG1-1', 'MG2-1', 'MG2-2', 'FG1-1', 'FG3-1', 'FG4-1']
{1: 8.0, 2: 6.5, 3: 7.0, 4: 6.0}
Death of -> MG2-1
bin_comb:  101001
['MG1-1', 'MG2-2', 'FG1-1', 'FG3-1', 'FG4-1', 'FG5-1']
{1: 8.0, 2: 7.0, 3: 7.0, 4: 6.0, 5: 6.0}
```

Death of -> FG4-1
bin_comb: 111000
['MG1-1', 'MG2-2', 'FG1-1', 'FG3-1', 'FG5-1', 'FG6-1']
{1: 8.0, 2: 7.0, 3: 7.0, 5: 6.0, 6: 9.0}
Death of -> FG5-1
bin_comb: 111100
['MG1-1', 'MG2-2', 'MG7-1', 'FG1-1', 'FG3-1', 'FG6-1']
{1: 8.0, 2: 7.0, 7: 10.0, 3: 7.0, 6: 9.0}
Death of -> MG2-2
bin_comb: 001100
['MG1-1', 'MG7-1', 'MG8-1', 'FG1-1', 'FG3-1', 'FG6-1']
{1: 8.0, 7: 10.0, 8: 6.0, 3: 7.0, 6: 9.0}
Death of -> MG8-1
bin_comb: 111101
['MG1-1', 'MG7-1', 'FG1-1', 'FG3-1', 'FG6-1', 'FG9-1']
{1: 8.0, 7: 10.0, 3: 7.0, 6: 9.0, 9: 5.0}
Death of -> FG9-1
bin_comb: 100011
['MG1-1', 'MG7-1', 'FG1-1', 'FG3-1', 'FG6-1', 'FG10-1']
{1: 8.0, 7: 10.0, 3: 7.0, 6: 9.0, 10: 9.0}
Death of -> FG1-1
bin_comb: 010111
['MG1-1', 'MG7-1', 'FG3-1', 'FG6-1', 'FG10-1', 'FG11-1']
{1: 9.0, 7: 10.0, 3: 7.0, 6: 9.0, 10: 9.0, 11: 7.0}
Death of -> FG3-1
bin_comb: 010001
['MG1-1', 'MG7-1', 'FG6-1', 'FG10-1', 'FG11-1', 'FG12-1']
{1: 9.0, 7: 10.0, 6: 9.0, 10: 9.0, 11: 7.0, 12: 7.0}