KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)
Deemed to be University U/S 3 of the UGC Act, 1956
School of Computer Engineering
KIIT (Deemed to be University) Bhubaneswar, Odisha, India

AI Assignment 2
Report

Task 1:
You are required to implement a program that takes an image as its input and generates the same image using N number of squares. You need to implement this using Genetic Algorithm.

Name: ABENEZER GOLDA
Roll No: 21053260
Section: AI 38

Github link: *Click Here*

Submission Date: Nov 06, 2023
Submitted to: Sohail Khan

AI Assignment 2 Report

In this report, we will the detailed implementation of a program that takes an image as its input and generates the same image using N number of squares. GA is a random-based optimization technique that has several generic steps that are generally followed to solve any optimization problem. These steps are then customized to the problem being solved. This tutorial discusses these steps briefly but concentrates on how to customize them according to this project. The summary of these steps is as follows:

- Data Representation
- Initial Population
- Fitness Calculation
- Parent Selection
- Crossover
- Mutation

Now let's see in-depth into each of the points:

Data Representation

The first task for an optimization problem using GA is to think about the best way to represent the data. GA accepts the chromosome (i.e. solution) as a 1D row vector. The input image will not be 1D. The image may be 2D if it's a binary or a gray image.

Starting with the simplest case in which the input is a 2D image (i.e. 2D matrix), converting it into a 1D vector requires us to merge the 2 dimensions into a single dimension. The matrix has multiple rows, and it's necessary to merge all of these rows into a single row. This can be achieved by stacking the different rows together. This is illustrated in the next figure. The figure shows an image/matrix of 3 rows and 3 columns. That's a total of 3x3=9 elements.

After converting the 2D image into a 1D vector, it's important to know how to restore the image back from the vector. This will be helpful in this project. In order to do that, it's essential to know what the size of the image was. Knowing it was 3x3, the first 3 elements of the vector will form the first row in the image, the next 3 elements in the vector will form the next row, and so on.

So there is no single 2D image to convert but 3. The previous work of converting the 2D image into a 1D vector will just be repeated 3 times.

Because the resulting vector will hold all elements in all 3 2x2 images, then its length will be 3x2x2=12. The first image has 2x2=4 elements. These 4 elements are placed at the beginning of the vector. The 4 elements of the next image will be placed after these elements. Finally, the 4 elements of the last image will be placed at the end of the vector. Note that the project is not just limited to 3D images but can accept images in color spaces with more than 3 channels.

Converting the MD data into a 1D vector is the end for representing the data of this project, but it might not be the end of other types of problems. In this project, value encoding is used. The same values in the image are used in the chromosome. In other problems, it might be preferred to encode the values in different ways. In this case, there is an encoding step between the original form of the data and the chromosome. The encoding might be binary, octal, or hexadecimal, for example.

It's also important to extract the MD image from the 1D vector. Given that the image has 3 channels, the vector will be divided into 3 parts of equal length. This length is equal to the number of elements within each channel. If the channel size is 2x2, then the first 4 elements of the vector will create the first channel in the image. The first 2 elements of these 4 will create the first row in this channel, and the last 2 elements of these 4 will be the second row for the same channel. The next 4 elements in the vector will create the next channel, and so on.

Python Code for Converting the Image into a Chromosome and Vice Versa

After understanding the concept well, we can build a Python function that accepts an image and returns its chromosome representation as a 1D row vector. The function is named img2chromosome() which is shown below. The function accepts the image to be converted as an argument named img_arr.

Using the reshape() function of the NumPy library, we can reshape any array from its current shape into a new shape. This function accepts the array and its new shape and returns an array of that shape:

```
def img2chromosome(img_arr):
        chromosome=numpy.reshape(a=img_arr,newshape=(functools.reduce(operator.mul, img_arr.shape)))
    return chromosome
```

In order to make the code independent on a specific image shape and able to work the same regardless of an image's number of dimensions, the reduce() function from the functools library is used.

It accepts an operation and a list of numbers and applies this operation between every 2 numbers until returning just a single number. The operation it accepted is mul, which is short for multiplication. This operation is defined in the operator library. The list of numbers the reduce() function accepts is the shape of the image returned by the img_arr.shape property.

The reverse of this process is also important. That is, converting that row vector into the original matrix. This can be implemented into another function named chromosome2img(), as listed below. This function accepts the chromosome and the image shape. By simply passing these arguments to the numpy.reshape() function, the original image can be restored:

```
def chromosome2img(chromosome, img_shape):
    img_arr = numpy.reshape(a=chromosome, newshape=img_shape)
    return img_arr
```

At this time, the input image can be converted into a chromosome. and the image can be restored back from that chromosome. After creating the chromosome, the next step is to build the initial population, which is simply a group of chromosomes.

Initial Population

GA starts an initial population, which is a group of solutions (chromosomes) to the given problem. These solutions are randomly generated. A Python function named initial_population() is created (as shown below) to return such a population.

Without looking at the function arguments, let's think about what arguments are expected to be passed to it. At first, a chromosome (1D vector) is to be created. The vector length is equal to the number of elements in the image. Thus, there must be an argument to help in calculating this length. This is why the function accepts the shape of the image as the first argument named img_shape.

The function not only returns a single chromosome but a group of chromosomes. How many chromosomes should it return? It's specified as the second argument n_individuals, which defaults to 8 if not specified:

```
def initial_population(img_shape, n_individuals=8):
    init_population = numpy.empty(shape=(n_individuals,
                                         functools.reduce(operator.mul, img_shape)),dtype=numpy.uint8)
```

```
    for indv_num in range(n_individuals):
        # Randomly generating initial population chromosomes genes values.
        init_population[indv_num, :] = numpy.random.random(
                            functools.reduce(operator.mul, img_shape))*256
        return init_population
```

Let's assume that the image shape is (100, 50, 3). In other words, it's an RGB image with 100 rows and 50 columns. Thus, the chromosome length is 15,000. If the number of chromosomes to return is 5, then there will be 5 chromosomes of length 15,000 each.

The first line in the function creates an empty NumPy array according to the number of chromosomes and their length. The remaining part fills these chromosomes by randomly generated numbers using the random() function inside the NumPy.random module.

Because these chromosomes/solutions are randomly generated, there is no guarantee that they will solve the problem correctly. This is why the GA evolves them using the following steps.

Fitness Calculation

GA starts with several bad solutions that are randomly generated. GA is based on the idea that evolving bad solutions might return better solutions. For every generation, the GA selects the best of the solutions in the current population and evolves them, hoping to return better solutions.

The fitness function used in this project is named fitness_fun() and is listed below. It accepts 2 arguments—the target image and the current solution—and returns a number to measure the similarity between them:

```
def fitness_fun(target_chrom, indiv_chrom):
    quality = numpy.mean(numpy.abs(target_chrom-indiv_chrom))
    quality = numpy.sum(target_chrom) - quality
    return quality
```

At first, the mean of absolute differences between the elements in the target image and the current image are calculated. Based on this value, the lower the value, the better the solution. But the target is to return a value that is better when it's higher. It's better to do that to meet the specifications of GA. This is why it's subtracted from the summation of all elements in the target image. In this way, the higher the value returned, the better the solution.

For each solution, the previous fitness_fun() is called to return its fitness value. The fitness values for all solutions are saved in an array named qualities, which is finally returned by this function:

```
def cal_pop_fitness(target_chrom, pop):
    qualities = numpy.zeros(pop.shape[0])
    for indv_num in range(pop.shape[0]):
        qualities[indv_num] = fitness_fun(target_chrom, pop[indv_num, :])
    return qualities
```

After calculating the fitness values for all solutions, the next step is to select the best of them for creating the next population. The best of these solutions are called parents. By mating these parents, the expected return is better solutions (offspring).

Parent Selection

The parents selected from a given population are the best solutions within it. When we say “best solutions”, we’re referring to the solutions with the highest fitness values.

The parents are returned in this project according to a function named select_mating_pool() (shown below). It accepts 3 arguments: population (pop), the fitness values (qualities), and the number of parents (num_parents). It loops through the parents to select the ones with the highest fitness values and return them into an array named parents.

The function works by searching for the solution with the maximum fitness value and returning it into the parents array. In order to avoid selecting this parent again in the next iteration of the loop, its fitness value is set to -1. This guarantees not selecting it again. Then the next solution with the second maximum fitness value is selected, and the process repeats until returning all parents required:

```
def select_mating_pool(pop, qualities, num_parents):
    parents = numpy.empty((num_parents, pop.shape[1]), dtype=numpy.uint8)
    for parent_num in range(num_parents):
        max_qual_idx = numpy.where(qualities == numpy.max(qualities))
        max_qual_idx = max_qual_idx[0][0]
        parents[parent_num, :] = pop[max_qual_idx, :]
        qualities[max_qual_idx] = -1
    return parents
```

After selecting the parents, our next step is to mate them for creating the new generation. Mating is applied using 2 operations: crossover and mutation.

Crossover

Mating 2 organisms means creating a new offspring that shares the genes inside both of them. The crossover operation selects several genes from each parent and places them into their offspring. The crossover operation is applied in the project using a function named crossover(). It accepts 3 arguments: the parents selected previously using the select_mating_pool() function, the input image shape (img_shape), and the number of offspring to return (n_individuals), which defaults to 8.

```
def crossover(parents, img_shape, n_individuals=8):
        new_population= numpy.empty(shape=(n_individuals, functools.reduce(operator.mul, img_shape)), dtype=numpy.uint8)
    #Previous parents (best elements).
    new_population[0:parents.shape[0], :] = parents
      # Getting how many offspring to be generated. If the population size is 8 and number of parents mating is 4, then number of offspring to be generated is 4.
    num_newly_generated = n_individuals-parents.shape[0]
    # Getting all possible permutations of the selected parents.
            parents_permutations = list(itertools.permutations(iterable=numpy.arange(0, parents.shape[0]), r=2))
    # Randomly selecting the parents permutations to generate the offspring.
    selected_permutations = random.sample(range(len(parents_permutations)),
                                          num_newly_generated)
    comb_idx = parents.shape[0]
    for comb in range(len(selected_permutations)):
        # Generating the offspring using the permutations previously selected randmly.
        selected_comb_idx = selected_permutations[comb]
        selected_comb = parents_permutations[selected_comb_idx]
        # Applying crossover by exchanging half of the genes between two parents.
        half_size = numpy.int32(new_population.shape[1]/2)
        new_population[comb_idx+comb, 0:half_size] = parents[selected_comb[0],
                                                             0:half_size]
        new_population[comb_idx+comb, half_size:] =  parents[selected_comb[1],
                                                             half_size:]
    return new_population
```

Let's assume that all solutions in a given population share a gene that represents a bad property. After applying crossover, this gene will definitely be available in the offspring. If the offspring takes its genes from 2 parents where each parent has a bad gene, then the offspring will now have 2 bad genes.

As a result, it will be worse than its parents. This is why it's preferable that the next generation keep the previous parents in addition to the generated offspring. Even if the offspring are worse than the parents, the parents will be kept to avoid moving GA toward solutions that aren't evolved. This is why the crossover() function returns the new population, which consists of both the current parents and their offspring.

These modifications might solve a problem in the parents by replacing a bad gene with a better one. It might also introduce a problem by replacing a good gene and making it worse. If that happens, then the parents kept previously will prevent the selection of these bad offspring.

Mutation

The mutation operation selects some genes within the chromosome and then randomly changes their values.

It's implemented according to the mutation() function listed below. It accepts the population returned by the crossover() function, number of parents, and the percent of the genes to be changed. The number of parents is passed in order to simply apply the mutation over the offspring and skip the parents.

Avoid setting the percentage to a high value because it will introduce many random changes, which will definitely lead to bad solutions. Slight changes using small percentages might (but not certainly) introduce good changes. This function returns the offspring after introducing such random changes:

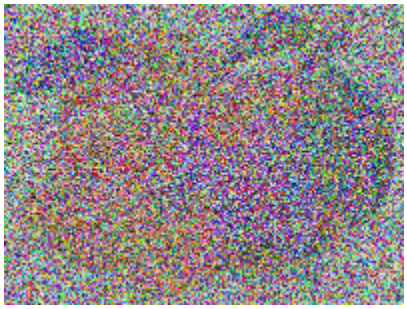
```
def mutation(population, num_parents_mating, mut_percent):
    for idx in range(num_parents_mating, population.shape[0]):
        # A predefined percent of genes are selected randomly.
            rand_idx=numpy.uint32(numpy.random.random(size=numpy.uint32(mut_percent/100*po
pulation.shape[1]))*population.shape[1])
        # Changing the values of the selected genes randomly.
        new_values = numpy.uint8(numpy.random.random(size=rand_idx.shape[0])*256)
        # Updating population after mutation.
        population[idx, rand_idx] = new_values
    return population
```

After the mutation operation is applied, the end of the current generation is reached and a new generation starts. The population used in this new generation is the combination

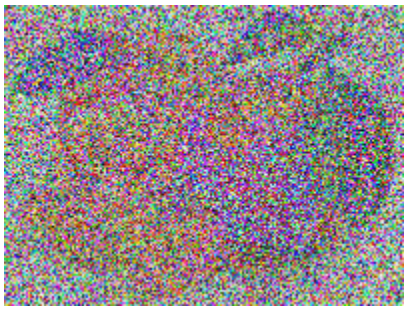
of the parents selected from the previous generation in addition to the offspring returned after the mutation operation is applied.

The read image:

After 7000:



After 8500:



After 20000 generations: