<u>AI Assignment - 2: Task 1:</u> Reproducing Images using Genetic Algorithms

By

Aaryak Prasad
2105171

Submitted to: Mr. Sohail Khan

This project has been implemented using the PyGAD Python library. Check it out at: https://github.com/AaryakPrasad/2105171_AI

Introduction:

The task was to implement a program that takes an image as its input and generates the same image using N number of squares. This was to be done using the Genetic Algorithm.

The genetic algorithm is a classical evolutionary algorithm that relies on randomness. When we say "random," we mean that it introduces random modifications to the current solutions in order to generate new ones. It's worth mentioning that the genetic algorithm is sometimes referred to as the Simple Genetic Algorithm (SGA) due to its straightforward nature compared to other Evolutionary Algorithms.

The genetic algorithm is founded on Darwin's theory of evolution. It operates as a gradual and methodical process, making incremental adjustments to the solutions, progressively moving towards the optimal solution.

Implementation:

The project takes an image as its input, and this image can consist of one or more channels, meaning it can be binary, grayscale, or color (e.g., in the RGB format). RGB, short for Red, Green, and Blue, is the most widely used color model that combines these three channels to create a wide spectrum of colors.

In the genetic algorithm (GA), the process starts with the creation of a random image that matches the shape of the input image. This randomly generated image is then subject to evolutionary processes involving crossover and mutation using GA until it produces an image that closely resembles the original. While the exact replica of the original image may not be achieved, the goal is to generate an image that bears a strong resemblance.

The project comprises two Python files. The first file, "converter.py," contains the implementation of the GA functions responsible for image reproduction. The second file, "main.py," serves as a driver program that invokes the functions defined in the former file. It reads a colored image file named "pepsi.jpg," which is included in the project, and processes it through the GA functions.

GA has a number of generic steps that are generally followed to solve any optimization problem. They are as follows:

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

Data Representation:

In the context of optimization problems using Genetic Algorithms (GA), the initial step involves determining the most suitable way to represent the data. GA operates with chromosomes, which represent potential solutions, in the form of 1D row vectors. However, the input image may not be 1D; for instance, it can be 2D in the case of binary or grayscale images. To facilitate this conversion, the ImageIO Python library is utilized.

For images with more than one dimension, such as color images, there are multiple dimensions. In the case of RGB images, there are three dimensions, one for each color channel. To work with multi-dimensional data in a GA, it must be transformed into a 1D vector. To illustrate, when starting with a 2D image (a 2D matrix), it must be condensed into a single dimension. This involves merging the multiple rows of the matrix into a single row, typically achieved by stacking these rows together.

After converting a 2D image into a 1D vector, it is crucial to understand how to reverse this process to reconstruct the image, a key aspect in the project. This necessitates knowledge of the original image's dimensions. For instance, if the original image was 3x3, the first three elements of the vector will form the first row of the image, followed by the next three elements forming the subsequent row, and so on.

Converting a 3D RGB color image is relatively straightforward because it can be visualized as three separate 2D images, with each image representing one of the RGB color channels. Therefore, rather than converting a single 2D image, you repeat the process three times.

Given that the resulting vector contains all elements from all three 2x2 images, its length will be 3x2x2=12. The first image contains 2x2=4 elements, and these initial four elements are placed at the beginning of the vector. The next four elements correspond to the second image, and the final four elements represent the last image, positioned at the end of the vector. It's worth noting that the project is not limited to 3D images and can handle images in color spaces with more than three channels. This transformation is carried out using the 'imgtochromosome' function in the 'converter.py' file.

Additionally, it's important to extract information from the 1D vector to reconstruct the multi-dimensional image. In the case of a 3-channel image, the vector is divided into three equal parts, with each part's length corresponding to the number of elements within each channel. If, for example, each channel is 2x2 in size, the first four elements in the vector form the first channel, with the first two elements creating the initial row of this channel, and the remaining two elements forming the second row of the same channel. This process can be performed using the 'chromosometoimg' function found in the 'converter.py' file.

Initial Population

GA starts an initial population, which is a group of solutions (chromosomes) to the given problem. These solutions are randomly generated.
The PyGAD library creates an empty NumPy array according to the number of chromosomes and their length. It 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

In this project, the fitness function, named 'fitness_fun()', is used to assess the similarity between the target image and the current solution. This function takes two arguments: the target image and the current image, and it returns a numerical value.

To calculate the fitness value, the function first computes the mean of the absolute differences between the elements of the target image and the current image. This value serves as an initial measure of similarity. However, in the context of Genetic Algorithms (GA), it's more desirable to have a higher fitness value for better solutions. To achieve this, the initial value is subtracted from the sum of all elements in the target image. As a result, the fitness value increases as the solution improves, with higher values indicating better solutions.

The 'fitness_fun()' function processes a single solution and returns its corresponding fitness value. To calculate the fitness values for all solutions within a population, a separate function called 'cal_pop_fitness()' is implemented in the PyGAD library. This function iterates through the solutions within the population to compute their fitness values.

Parent Selection

The ParentSelection class in the pygad.utils.parent_selection module has several methods for selecting the parents that will mate to produce the offspring. All of such methods accept the same parameters which are:

- fitness: The fitness values of the solutions in the current population.
- num_parents: The number of parents to be selected.

All of such methods return an array of the selected parents.

A variety of methods are provided by the PyGAD library but upon testing various algorithms, **steady_state_selection()** algorithm works best.

Crossover

Mating 2 organisms means creating a new offspring that shares the genes inside both of them. The crossover operation selects a number of genes from each parent and places them into their offspring. Consider a scenario where all solutions in a given population share a particular gene that signifies a disadvantageous trait. When crossover is applied during the reproduction process, this gene will inevitably be present in the offspring. If the offspring inherits genes from two parents, both of whom possess this unfavorable gene, then the offspring will now possess two copies of this undesirable gene.

Consequently, the offspring is expected to be less favorable than its parents due to the presence of these two bad genes. To avoid moving the Genetic Algorithm (GA) toward solutions that have not evolved or improved, it's preferable to retain the previous generation's parents in addition to the newly generated offspring. Even if the offspring may be inferior to the parents, maintaining the parents in the population prevents the GA from deviating from previously evolved solutions.

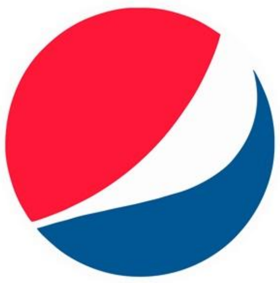
This is precisely why the 'crossover()' function returns the new population with a **single point crossover**, which combines both the current parents and their offspring. It selects a point randomly at which crossover takes place between the pairs of parents. This approach ensures that the GA maintains a mix of existing solutions, even if the offspring is not superior, in order to preserve the progress made during the optimization process.

Mutation

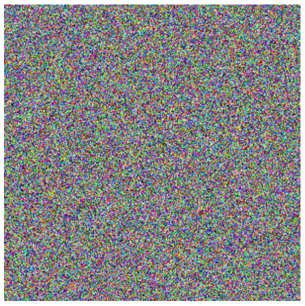
The mutation operation selects some genes within the chromosome and then randomly changes their values.
It’s implemented according to the mutation() function in the pygad.utils.mutation module. It accepts the population returned by the crossover() function, number of parents, and the percent of the genes to be changed. The number of parents is passed in order to simply apply the mutation over the offspring and skip the parents. While a variety of mutation algorithms maybe applied, testing revealed **random mutation** worked best.

Result

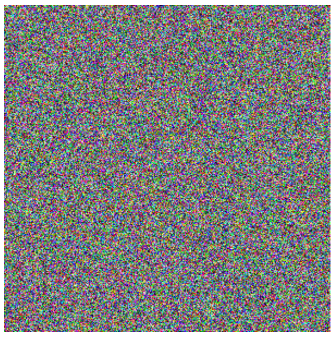
The input image is:



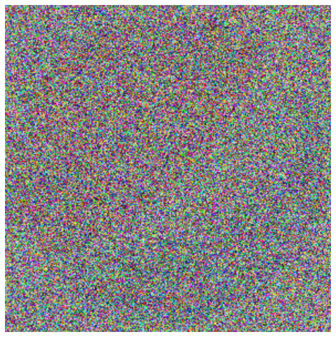
A randomly generated initial population converted to an image looks like this:



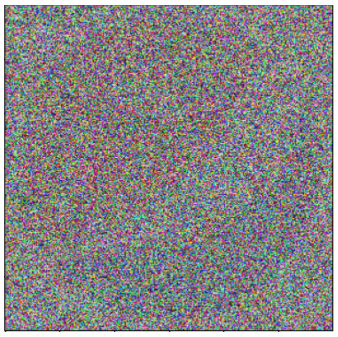
Result after 5000, 10000 and 20000 iterations respectively:



5000 Generations



10000 Generations



20000 Generations