COS30018 – Intelligent Systems

Week 8 tutorial

This week we will cover the following items:

- PyGAD: library for genetic algorithms (GAs)
- Applications of GAs using PyGAD:
    - Fitting a linear model
    - 8 Queen Puzzle

1. Introduction to PyGAD:

PyGAD is an open-source Python library for implementing the genetic algorithm and training machine learning algorithms. PyGAD supports 19 parameters for customizing the genetic algorithm for various applications.

In the tutorial this week, we'll discuss 2 applications of the genetic algorithm and implement them using PyGAD.

a. Installation:

PyGAD is available through PyPI (Python Package Index) and thus it can be installed simply using pip. Simply run the following command in your terminal:

```
pip install pygad
```

```
(projectB_COS30018) D:\Envs\projectB_COS30018>pip install pygad
Collecting pygad
  Downloading pygad-2.18.0-py3-none-any.whl (56 kB)
```

Then make sure the library is installed by importing it from the Python shell:

```
import pygad

print(pygad.__version__)
```

```
(projectB_COS30018) D:\Envs\projectB_COS30018>python
Python 3.9.12 (main, Apr  4 2022, 05:22:27) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32

Warning:
This Python interpreter is in a conda environment, but the environment has
not been activated.  Libraries may fail to load.  To activate this environment
please see https://conda.io/activation

Type "help", "copyright", "credits" or "license" for more information.
>>> import pygad
>>>
>>> print(pygad.__version__)
2.18.0
```

The latest PyGAD version is currently 3.1.0. Using the _version__ special variable, the current version of the installation can be returned.

b. Getting started with PyGAD:

The main goal of PyGAD is to provide a simple implementation of the genetic algorithm. It offers a range of parameters that allow the user to customize the genetic algorithm for a wide range of applications.

The full documentation of PyGAD is available at PyGAD Docs. Here we'll cover a more digestible breakdown of the library.

There are 5 modules in PyGAD:

- `pygad`: The main module comes already imported.
- `pygad.nn`: For implementing neural networks.
- `pygad.gann`: For training neural networks using the genetic algorithm.
- `pygad.cnn`: For implementing convolutional neural networks.
- `pygad.gacnn`: For training convolutional neural networks using the genetic algorithm.

The main module of the library is named `pygad`. This module has a single class named `GA`. Just create an instance of the `pygad.GA` class to use the genetic algorithm.

The steps to use the pygad module are:

1. Create the fitness function.
2. Prepare the necessary parameters for the `pygad.GA` class.
3. Create an instance of the `pygad.GA` class.
4. Run the genetic algorithm.

The constructor of the `pygad.GA` class has 19 parameters, of which 16 are optional. The three required parameters are:

1. `num_generations`: Number of generations.
2. `num_parents_mating`: Number of solutions to be selected as parents.
3. `fitness_func`: The fitness function that calculates the fitness value for the solutions.

The `fitness_func` parameter is what allows the genetic algorithm to be customized for different problems. This parameter accepts a user-defined function that calculates the fitness value for a single solution. This takes two additional parameters: the solution, and its index within the population. If you are using pygad version 3.1.0, the `fitness_func` takes three parameters: ga_instance, solution, and index.

For example, assume there is a population with 3 solutions, as given below:

```
[221, 342, 213]
[675, 32, 242]
[452, 23, -212]
```

The assigned function to the `fitness_func` parameter must return a single number representing the fitness of each solution. Here is an example of a fitness function that returns the sum of the solution.

```
def fitness_function(solution, solution_idx):
    return sum(solution)
```

For pygad version 3.1.0, the example fitness function is written as:

```
1 def fitness_func(ga_instance, solution, solution_idx):
2     return sum(solution)
```

Output:

1. 776
2. 949
3. 263

The parents are selected based on such fitness values. The higher the fitness value, the better the solution.

After creating an instance of the `pygad.GA` class, the next step is to call the `run()` method which goes through the generations that evolve the solutions.

```
import pygad

ga_instance = pygad.GA(...)

ga_instance.run()
```

These steps are the minimum needed to run PyGAD for Genetic Algorithms. Of course there are additional steps that can be taken as well.

2. Fitting a linear model:

To get started with PyGAD, simply import it.

```
import pygad
```

Assume we have a linear equation with 6 inputs, 1 output, and 6 parameters, as follows:

$$y = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6$$

Where:

- x_i are the inputs
- w_i are the parameters
- y is the output

Let's assume that the inputs are (4, -2, 3.5, 5, -11, -4.7) and the output is 44. What are the values for the 6 parameters to satisfy the equation? The genetic algorithm can be used to find the answer.

The first step is to prepare the inputs and the outputs of this equation.

```
function_inputs = [4, -2, 3.5, 5, -11, -4.7]  # Function inputs.
desired_output = 44  # Function output.
```

The very important thing to do is to prepare the fitness function as given below. It calculates the sum of products between each input and its corresponding parameter. The absolute difference between the desired output and the sum of products is calculated. Because the fitness function must be a maximization function, the returned fitness is equal to $\frac{1}{difference}$. The solutions with the highest fitness values are selected as parents.

```
def fitness_func(solution, solution_idx):
    output = numpy.sum(solution*function_inputs)
    fitness = 1.0 / numpy.abs(output - desired_output)
    return fitness
```

Now that we've prepared the fitness function, next is to prepare the parameters of PyGAD. Here's an example for a set of parameters.

```
fitness_function = fitness_func

num_generations = 50
num_parents_mating = 4

sol_per_pop = 8
num_genes = len(function_inputs)

init_range_low = -2
init_range_high = 5

parent_selection_type = "sss"
keep_parents = 1

crossover_type = "single_point"

mutation_type = "random"
mutation_percent_genes = 10
```

Where:

- `num_generations`: Number of generations.
- `num_parents_mating`: Number of solutions to be selected as parents.
- `sol_per_pop`: Number of solutions (i.e. chromosomes) within the population.
- `num_genes`: Number of genes in the solution/chromosome.
- `init_range_low`: The lower value of the random range from which the gene values in the initial population are selected. `init_range_low` defaults to -4.
- `init_range_high`: The upper value of the random range from which the gene values in the initial population are selected. `init_range_high` defaults to 4.
- `parent_selection_type="sss"`: The parent selection type. Supported types are sss (for steady-state selection), rws (for roulette wheel selection), sus (for stochastic universal selection), rank (for rank selection), random (for random selection), and tournament (for tournament selection).
- `keep_parents`: Number of parents to keep in the next population. -1 (default) means to keep all parents in the next population. 0 means keep no parents in the next population. A value greater than 0 means keeps the specified number of parents in the next population. Note that the value assigned to `keep_parents` cannot be < - 1 or greater than the number of solutions within the population `sol_per_pop`.
- `crossover_type`: Type of the crossover operation. Supported types are single_point (for single-point crossover), two_points (for two points crossover), uniform (for uniform crossover), and scattered (for scattered crossover). It defaults to single_point.
- `mutation_type`: Type of the mutation operation. Supported types are random (for random mutation), swap (for swap mutation), inversion (for inversion mutation), scramble (for scramble mutation), and adaptive (for adaptive mutation). It defaults to random.
- `mutation_percent_genes`: Percentage of genes to mutate. It defaults to the string "default" which is later translated into the integer 10 which means 10% of the genes will be mutated. It must be >0 and <=100.

For information about each of the parameters, refer to this doc.

After the parameters are prepared, an instance of the `pygad.GA` class is created.

```
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       fitness_func=fitness_function,
                       sol_per_pop=sol_per_pop,
                       num_genes=num_genes,
                       init_range_low=init_range_low,
                       init_range_high=init_range_high,
                       parent_selection_type=parent_selection_type,
                       keep_parents=keep_parents,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       mutation_percent_genes=mutation_percent_genes)
```

The next step is to call the `run()` method which starts the generations.

```
ga_instance.run()
```

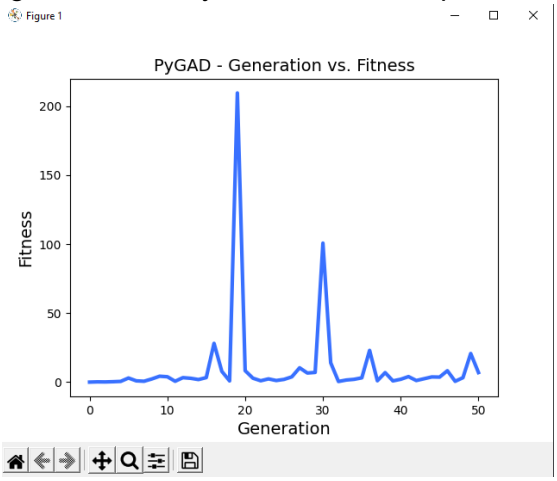
After the `run()` method completes, the `plot_result()` method can be used to show the fitness values over the generations.

```
ga_instance.plot_result()
```

Using the `best_solution()` method we can also retrieve what the best solution was, its fitness, and its index within the population.

```
solution, solution_fitness, solution_idx = ga_instance.best_solution()

print("Parameters of the best solution:
{solution}".format(solution=solution))
print("Fitness value of the best solution =
{solution_fitness}".format(solution_fitness=solution_fitness))
print("Index of the best solution :
{solution_idx}".format(solution_idx=solution_idx))

prediction = numpy.sum(numpy.array(function_inputs)*solution)
print("Predicted output based on the best solution :
{prediction}".format(prediction=prediction))

if ga_instance.best_solution_generation != -1:
    print("Best fitness value reached after {best_solution_generation}
generations.".format(best_solution_generation=ga_instance.best_solution_g
eneration))
```

When you put everything together and run, you will have an output similar to this:



```
Parameters of the best solution: [ 5.03514775  3.96127777  5.24666506
1.92304337 -1.56137984  2.87562317]
Fitness value of the best solution = 6.960364528984452
Index of the best solution : 1
Predicted output based on the best solution : 43.85632936380907
Best fitness value reached after 19 generations.
```

Note: due to the randomness of the genetic algorithm, you will not have the exact similar output. If you can generate the predicted output close to 44 (which is our target output), then we've got it correct.

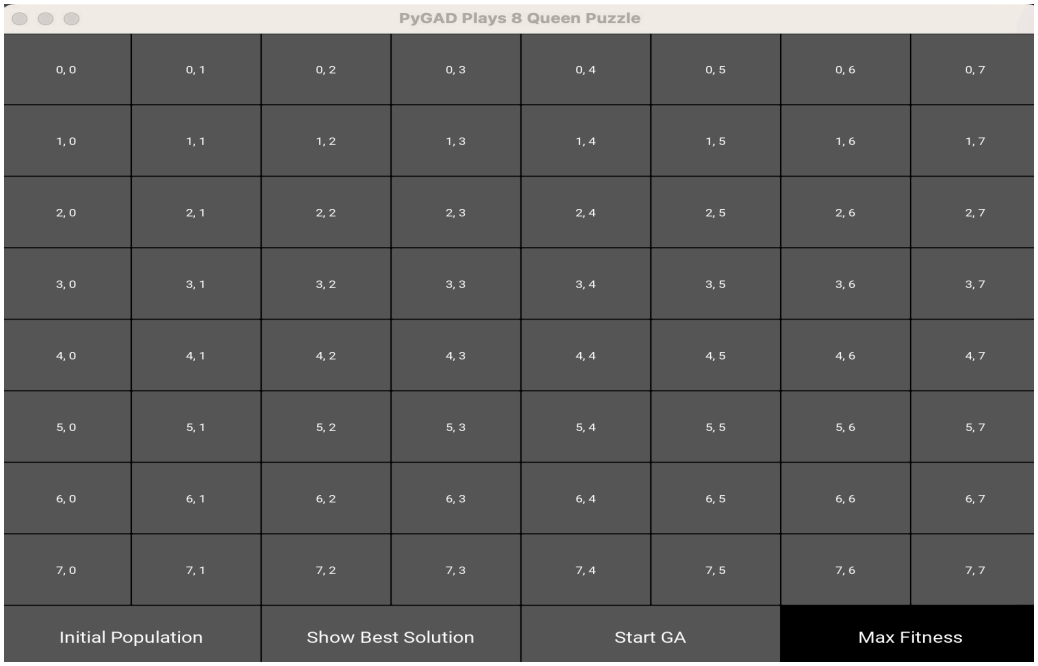
You can also tune the parameters to get a better result.

The full code for this project can be found on Canvas.

3. 8 Queen Puzzle:

The 8 Queen Puzzle involves 8 chess queens distributed across an 8×8 matrix, with one queen per row. The goal is to place these queens such that no queen can attack another one vertically, horizontally, or diagonally. The genetic algorithm can be used to find a solution that satisfies such conditions.

This project is available on this GitHub link. It has a GUI built using Kivy that shows an 8×8 matrix, as shown in the next figure.



The GUI has three buttons at the bottom of the screen. The function of these buttons are as follows:

- The Initial Population button creates the initial population of the GA.
- The Show Best Solution button shows the best solution from the last generation the GA stopped at.
- The Start GA button starts the GA iterations/generations.

To use this project, start by pressing the Initial Population button, followed by the Start GA button. Below is the method called by the Initial Population button which, as you might have guessed, generates the initial population.

```
def initialize_population(self, *args):
    self.num_solutions = 100
    self.reset_board_text()
    self.population_1D_vector = numpy.zeros(shape=(self.num_solutions, 8)) # Each s
    # Creating the initial population RANDOMLY as a set of 1D vectors.
    for solution_idx in range(self.num_solutions):
        initial_queens_y_indices = numpy.random.rand(8) * 8
        initial_queens_y_indices = initial_queens_y_indices.astype(numpy.uint8)
        self.population_1D_vector[solution_idx, :] = initial_queens_y_indices
    self.vector_to_matrix()
    self.pop_created = 1 # indicates that the initial population is created in orde
    self.num_attacks_Label.text = "Initialized"
    self.ga_instance = pygad.GA(num_generations=100,
                                num_parents_mating=30,
                                fitness_func=fitness,
                                num_genes=8,
                                initial_population=self.population_1D_vector,
                                mutation_percent_genes=0.01,
                                mutation_type="random",
                                mutation_num_genes=2,
                                mutation_by_replacement=True,
                                random_mutation_min_val=0.0,
                                random_mutation_max_val=8.0,
                                on_generation=on_gen_callback,
```

Each solution in the population is a vector with 8 elements referring to the column indices of the 8 queens. Using the `vector_to_matrix()` method to convert 1D vector into a 2D matrix which represents the board, where 1 means a queen exists. The matrix form of the solutions makes calculating the fitness value much easier. The next figure shows an example of a 2D matrix converted from a 1D population vector. Get_instance method create an instance of the pygad.GA class.

Now that the GUI is built, we'll build and run the genetic algorithm using PyGAD.

The fitness function used in this project is given below. It simply calculates the number of attacks that can be made by each of the 8 queens and returns this as the fitness value.

```
def fitness(solution_vector, solution_idx):

    # Convert solution to 2D matrix if it is a 1D vector
    if solution_vector.ndim == 2:
        solution = solution_vector
    else:
        solution = numpy.zeros(shape=(8, 8))
        row_idx = 0
        for col_idx in solution_vector:
            solution[row_idx, int(col_idx)] = 1
            row_idx = row_idx + 1

    # Calculate the number of attacks
    total_num_attacks_column = attacks_column(solution)
    total_num_attacks_diagonal = attacks_diagonal(solution)
    total_num_attacks = total_num_attacks_column +
total_num_attacks_diagonal

    if total_num_attacks == 0:
        total_num_attacks = 1.1 # float("inf")
    else:
        total_num_attacks = 1.0/total_num_attacks
```

```
    return total_num_attacks
```

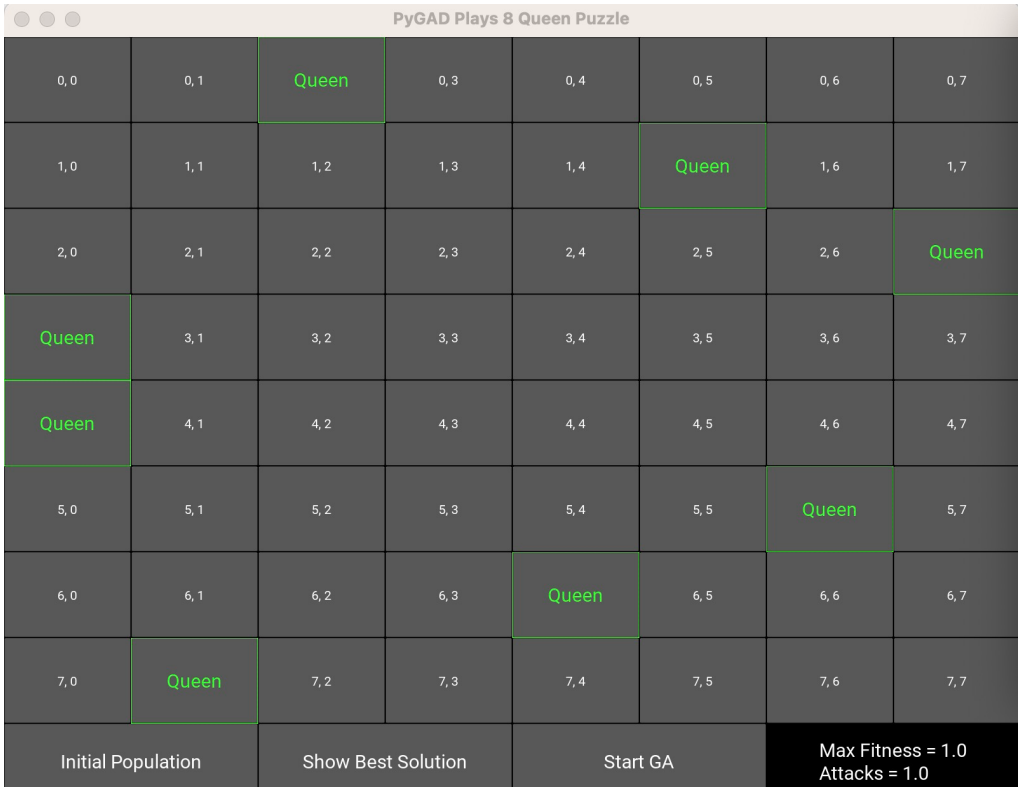
By pressing the Start GA button, the run() method of PygadThread class is called.

```
def start_ga(self, *args):
    if (not ("pop_created" in vars(self)) or (self.pop_created == 0)):
        print("No Population Created Yet. Create the initial Population
        self.num_attacks_Label.text = "Press \"Initial Population\""
        return
    pygadThread = PygadThread(self, self.ga_instance)
    pygadThread.run()
```

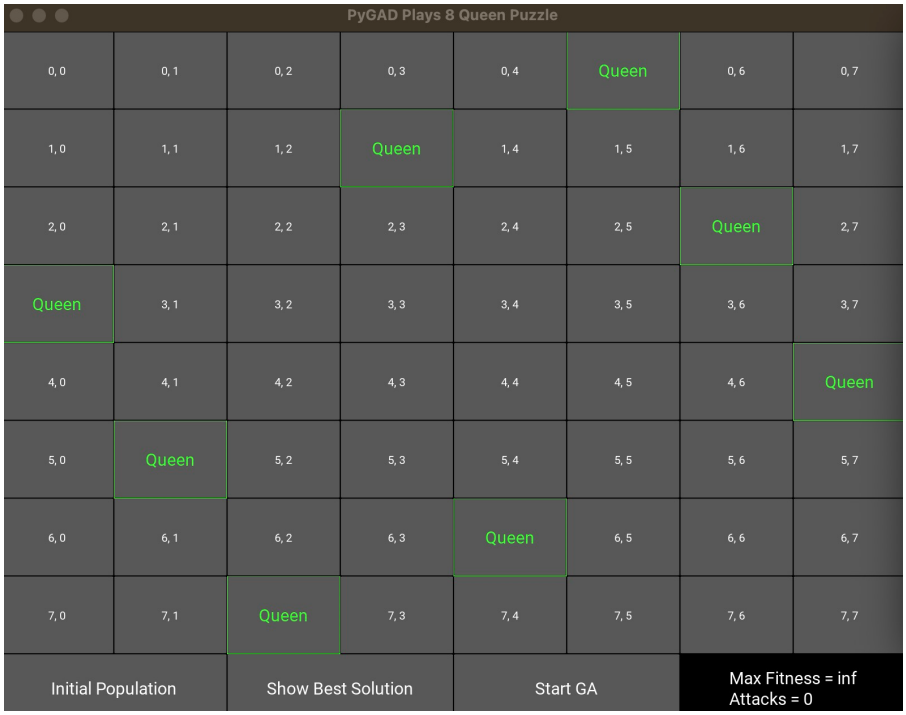
Where the PygadThread class is defined as:

```
class PygadThread(threading.Thread):

    def __init__(self, app, ga_instance):
        super().__init__()
        self.ga_instance = ga_instance
        self.app = app

    def run(self):
        self.ga_instance.run()
```

Here is a possible solution in which the 8 queens are placed on the board where one pair of queens attacks another.



The picture below shows a possible solution in which the 8 queens are placed on the board where no queen attacks another.



It is very important to note that the GA does not guarantee reaching the optimal solution each time it works. You can make changes in the number of solutions per population, the number of generations, or the number of mutations. Other than doing that, the initial population might also be another factor for not reaching the optimal solution for a given trial.

The complete code for this project can be found on Canvas or in the provided GitHub link.

References:

- PyGAD quick start: https://pygad.readthedocs.io/en/latest/
- PyGAD docs: https://pygad.readthedocs.io/en/latest/README_pygad_ReadTheDocs.html#pygad-ga-class
- 8 Queen Puzzle: https://github.com/ahmedfgad/8QueensGenetic