PyGAD

()

().

```
pip3 install pygad
```

```
import pygad
```

```
y = f(w1:w6) = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + w6x6
where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7) and y=44
```

```
function_inputs = [4,-2,3.5,5,-11,-4.7]
desired_output = 44
```

listtuplenumpy.ndarray().

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

```
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
```

```python
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)
```

run()

```python
ga_instance.run()
```

run()

```python
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))

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

```
Parameters of the best solution : [3.92692328 -0.11554946 2.39873381 3.29579039 -0.
↪74091476 1.05468517]
Fitness value of the best solution = 157.37320042925006
Predicted output based on the best solution : 44.00635432206546
```

,

pygad

nn

gann()

cnn

gacnn

kerasga

torchga

visualize

utils()

helper

```
@article{gad2023pygad,
  title={Pygad: An intuitive genetic algorithm python library},
  author={Gad, Ahmed Fawzy},
  journal={Multimedia Tools and Applications},
  pages={1--14},
  year={2023},
  publisher={Springer}
}
```

pygad

' pygad

pygad

pygad.GA

pygadGA

__init__()

pygad.GA

pygad.GA

- num_generations
- num_parents_mating
- fitness_func() (pygad.GA). ' listtuplenumpy.ndarray
- fitness_batch_size=Nonefitness_batch_size1None(), fitness_batch_size1 < fitness_batch_size <= sol_per_popfitness_batch_size
- initial_populationNonesol_per_popnum_genesinitial_populationNone (sol_per_pop num_genes) None
- sol_per_pop() initial_population
- num_genesinitial_population
- gene_type=floatfloatfloatgene_typeintfloatnumpy.int/uint/float(8-64) listtuplenumpy.ndarray(gene_type=[int, float, numpy.int8]). float(gene_type=[float, 2]
- init_range_low=-4init_range_low-4initial_population
- init_range_high=4init_range_high+4initial_population
- parent_selection_type="sss"sss(), rws (), sus (), rank (), random (), tournament().
- keep_parents=-1-1() 0greater than 0keep_parents< - 1sol_per_popkeep_elitism0 ' keep_parents=0

keep_elitism=10(0 <= keep_elitism <= sol_per_pop). 10KKsol_per_pop0 keep_parents

K_tournament=3tournamentK_tournament3

crossover_type="single_point"single_point(), two_points (), uniform (), scattered (). single_pointcrossover_type=None

crossover_probability=Nonecrossover_probability

mutation_type="random"random(), swap (), inversion (), scramble (), adaptive(). random mutation_type=NoneAdaptive

mutation_probability=Nonemutation_probabilitymutation_percent_genesmutation_num_genes

mutation_by_replacement=False(mutation_type="random"). mutation_by_replacement=True

mutation_percent_genes="default""default"10>0<=100mutation_num_genesmutation_percent_genesmutation_probabilitymutation_num_genesmutation_type None

mutation_num_genes=NoneNonemutation_num_genesmutation_probabilitymutation_typeNone

random_mutation_min_val=-1.0randomrandom_mutation_min_val-1mutation_type None

random_mutation_max_val=1.0randomrandom_mutation_max_val+1mutation_type None

gene_space=Nonelistrangenumpy.ndarraylisttuplerangenumpy.ndarraygene_space = [0.3, 5.2, -4, 8]gene_space[[0.4, -5], [0.5, -3.2, 8.2, -9], ...]None' init_range_lowinit_range_highrandom_mutation_min_valrandom_mutation_max_valgene_spacegene_spacegene_space{'low': 2, 'high': 4}"step""low""high"

on_start=None'

on_fitness=None)')'

on_parents=None))'

on_crossover=None

on_mutation=None

on_generation=Nonestoprun()

on_stop=None'

delay_after_gen=0.00.0

save_best_solutions=FalseTruebest_solutionsFalse(), best_solutions

save_solutions=FalseTruesolutions

suppress_warnings=FalseFalse

allow_duplicate_genes=TrueTrueFalse

stop_criteria=Nonestrreachsaturatereachrun()reach"reach_40">saturatesaturate"saturate_7"run()

parallel_processing=NoneNone(), [] 'process''thread')]parallel_processing=['process', 10]parallel_processing=5parallel_processing=["thread", 5]

random_seed=None(). (random_seed=2). None

logger=Nonelogging.Loggerprint()logger=NoneStreamHandler

' fitness_func

init_range_lowinit_range_high(init_range_lowinit_range_high). random_mutation_min_valrandom_mutation_max_val

mutation_typecrossover_typeNone

pygad.GA

plot_fitness()

plot_genes()

plot_new_solution_rate()

supported_int_types

supported_float_types

supported_int_float_types

pygad.GApygad.GApygad.GA

generations_completed

population

valid_parametersTrueGA

run_completedTruerun()

pop_size

best_solutions_fitness

best_solution_generationrun()

best_solutionssave_best_solutionspygad.GATrue

last_generation_fitness

- previous_generation_fitnesslast_generation_fitnesslast_generation_fitness previous_generation_fitness
- last_generation_parents
- last_generation_offspring_crossover
- last_generation_offspring_mutation
- gene_type_singleTruegene_typegene_typelisttuplenumpy.ndarray gene_type_singleFalse
- last_generation_parents_indices
- last_generation_elitismkeep_elitism
- last_generation_elitism_indiceskeep_elitism
- loggerlogging
- gene_space_unpackedgene_spacerange(1, 5)[1, 2, 3, 4]{'low': 2, 'high': 4}( ).
- pareto_frontspareto_frontspygad.GA

last_generation_

- cal_pop_fitness()fitness_func
- crossover()crossover_type
- mutation()mutation_type
- select_parents()parent_selection_type
- adaptive_mutation_population_fitness()
- summary()
- run_run()run_run()
  - run_select_parents(call_on_parents=True)on_parents()call_on_parents Trueon_parents()Falserun_select_parents()run()
  - run_crossover()on_crossover()
  - run_mutation()on_mutation()
  - run_update_population()population

pygad.GA

initialize_population()

population

low

high

pop_size

population

initial_population

cal_pop_fitness()

cal_pop_fitness()

save_solutionsTruesolutionssolutions_fitness

save_solutionsFalseTruecal_pop_fitness()keep_elitismlast_generation_elitism previous_generation_fitness

(save_solutionsFalseTruekeep_elitism), cal_pop_fitness()keep_parents-1 last_generation_parentsprevious_generation_fitness

fitness_func

parallel_processing

fitness_batch_size

,

run()

cal_pop_fitness()fitness_funcpygad.GA

select_parents()parent_selection_typepygad.GA

crossover()mutation()crossover_typemutation_typepygad.GA

population

generations_completed

on_generation

run()

best_solution_generation

run_completedTrue

pygad.GA

ParentSelectionpygad.utils.parent_selection

fitness

num_parents

steady_state_selection()

rank_selection()

random_selection()

tournament_selection()

roulette_wheel_selection()

stochastic_universal_selection()

nsga2_selection()

tournament_selection_nsga2()

Crossoverpygad.utils.crossover

parents

offspring_size

single_point_crossover()

two_points_crossover()

uniform_crossover()

scattered_crossover()

Mutationpygad.utils.mutation

offspring

random_mutation()

mutation_num_genesmutation_percent_genes

random_mutation_min_valrandom_mutation_max_val

swap_mutation()

inversion_mutation()

scramble_mutation()

adaptive_mutation()

' (),

best_solution()

pop_fitness=NoneNonecal_pop_fitness()

best_solution

best_solution_fitness

best_match_idx

plot_fitness()

plot_result()

(),

plot_new_solution_rate()

plot_new_solution_rate()save_solutions=Truepygad.GA

(),

plot_genes()

plot_genes()

graph_type

(),

save()

filename

pygad

pygad.GApygadload()

pygad.load()

pygadpygad.load(filename)

filename

pygad

pygad

fitness_func

pygad

pygad.GA

,

fitness_func

fitness_func

supported_int_float_typespygad.GA

listtuplenumpy.ndarray

,

()

()()

()?

W1W6y=44' y=44

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

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

pygad.GA

() (). fitness_batch_size

fitness_batch_size

fitness_func'

__code__

```python
num_generations = 50
num_parents_mating = 4

fitness_function = fitness_func

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
```

on_generation

on_generation() generations_completed

```
def on_gen(ga_instance):
    print("Generation : ", ga_instance.generations_completed)
    print("Fitness of the best solution :", ga_instance.best_solution()[1])
```

on_generationon_gen()

```
ga_instance = pygad.GA(...,
                       on_generation=on_gen,
                       ...)
```

pygad.GA

pygad

```
import pygad
```

pygad.GA

pygad.GA

pygad.GA

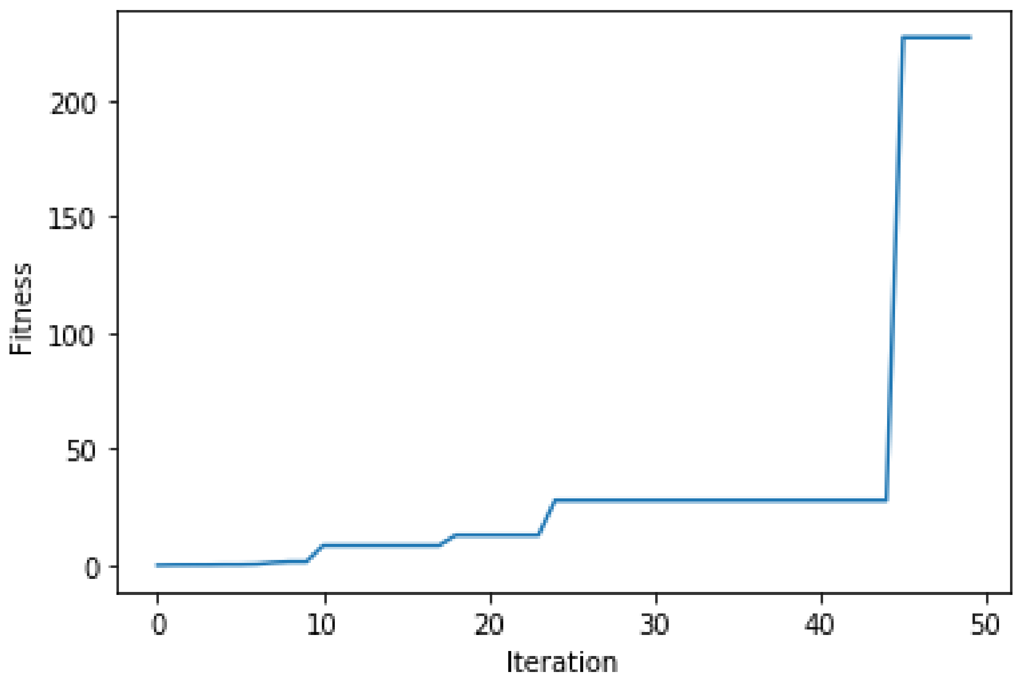
```
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       fitness_func=fitness_function,
                       sol_per_pop=sol_per_pop,
                       num_genes=num_genes,
                       init_range_low=init_range_low,
                       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)
```

pygad.GArun()

```
ga_instance.run()
```

plot_fitness()

```
ga_instance.plot_fitness()
```



best_solution()

```python
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")
```

best_solution_generationpygad.GAbest fitness

```python
if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")
```

run()save()genetic.pkl

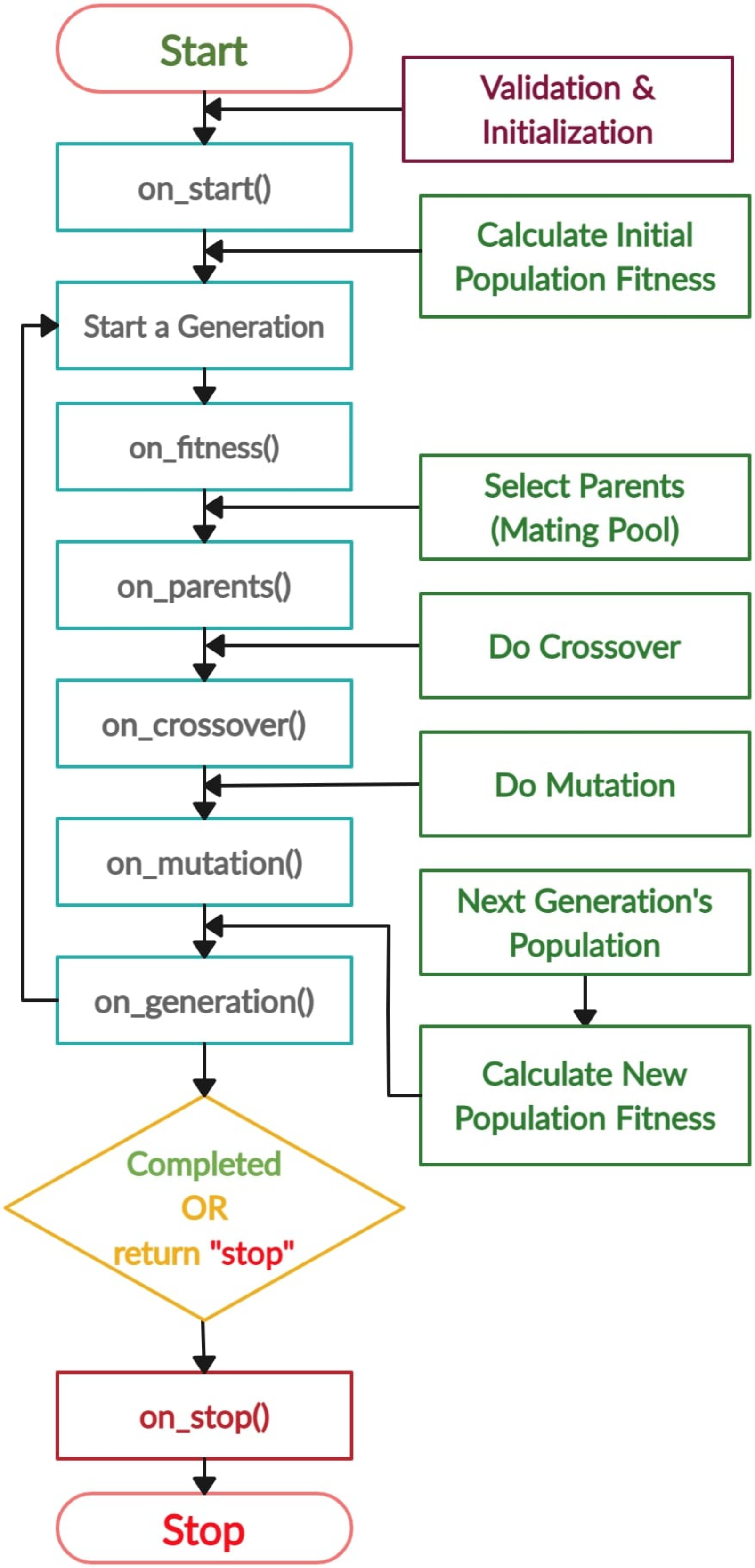
```python
filename = 'genetic'
ga_instance.save(filename=filename)
```

load()save()load()run()

```python
loaded_ga_instance = pygad.load(filename=filename)
```

```python
print(loaded_ga_instance.best_solution())
```

pygad.GAon_generationstop


Start
Validation & Initialization
on_start()
Calculate Initial Population Fitness
Start a Generation
on_fitness()
Select Parents (Mating Pool)
on_parents()
Do Crossover
on_crossover()
Do Mutation
on_mutation()
Next Generation's Population
on_generation()
Calculate New Population Fitness
Completed
OR
return "stop"
on_stop()
Stop

```python
import pygad
import numpy

function_inputs = [4,-2,3.5,5,-11,-4.7]
desired_output = 44

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution*function_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

fitness_function = fitness_func

def on_start(ga_instance):
    print("on_start()")

def on_fitness(ga_instance, population_fitness):
    print("on_fitness()")

def on_parents(ga_instance, selected_parents):
    print("on_parents()")

def on_crossover(ga_instance, offspring_crossover):
    print("on_crossover()")

def on_mutation(ga_instance, offspring_mutation):
    print("on_mutation()")

def on_generation(ga_instance):
    print("on_generation()")

def on_stop(ga_instance, last_population_fitness):
    print("on_stop()")

ga_instance = pygad.GA(num_generations=3,
                       num_parents_mating=5,
                       fitness_func=fitness_function,
                       sol_per_pop=10,
                       num_genes=len(function_inputs),
                       on_start=on_start,
                       on_fitness=on_fitness,
                       on_parents=on_parents,
                       on_crossover=on_crossover,
                       on_mutation=on_mutation,
                       on_generation=on_generation,
                       on_stop=on_stop)

ga_instance.run()
```

num_generations

```
on_start()

on_fitness()
on_parents()
on_crossover()
```

()

()

```
on_mutation()
on_generation()

on_fitness()
on_parents()
on_crossover()
on_mutation()
on_generation()

on_fitness()
on_parents()
on_crossover()
on_mutation()
on_generation()

on_stop()
```

pygad

```python
import pygad
import numpy

"""
Given the following function:
    y = f(w1:w6) = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + 6wx6
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7) and y=44
What are the best values for the 6 weights (w1 to w6)? We are going to use the
↪genetic algorithm to optimize this function.
"""

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

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution*function_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

num_generations = 100 # Number of generations.
num_parents_mating = 10 # Number of solutions to be selected as parents in the mating
↪pool.

sol_per_pop = 20 # Number of solutions in the population.
num_genes = len(function_inputs)

last_fitness = 0
def on_generation(ga_instance):
```

()

```
    global last_fitness
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution(pop_fitness=ga_instance.last_
↪generation_fitness)[1]}")
    print(f"Change     = {ga_instance.best_solution(pop_fitness=ga_instance.last_
↪generation_fitness)[1] - last_fitness}")
    last_fitness = ga_instance.best_solution(pop_fitness=ga_instance.last_generation_
↪fitness)[1]

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       sol_per_pop=sol_per_pop,
                       num_genes=num_genes,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Running the GA to optimize the parameters of the function.
ga_instance.run()

ga_instance.plot_fitness()

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution(ga_instance.last_
↪generation_fitness)
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

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

if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")

# Saving the GA instance.
filename = 'genetic' # The filename to which the instance is saved. The name is␣
↪without extension.
ga_instance.save(filename=filename)

# Loading the saved GA instance.
loaded_ga_instance = pygad.load(filename=filename)
loaded_ga_instance.plot_fitness()
```

```
y1 = f(w1:w6) = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + 6wx6

y2 = f(w1:w6) = w1x7 + w2x8 + w3x9 + w4x10 + w5x11 + 6wx12
```

```
(x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7)y=50

(x7,x8,x9,x10,x11,x12)=(-2,0.7,-9,1.4,3,5)y=30
```

() w1 w6

y1y2

listtuplenumpy.ndarray

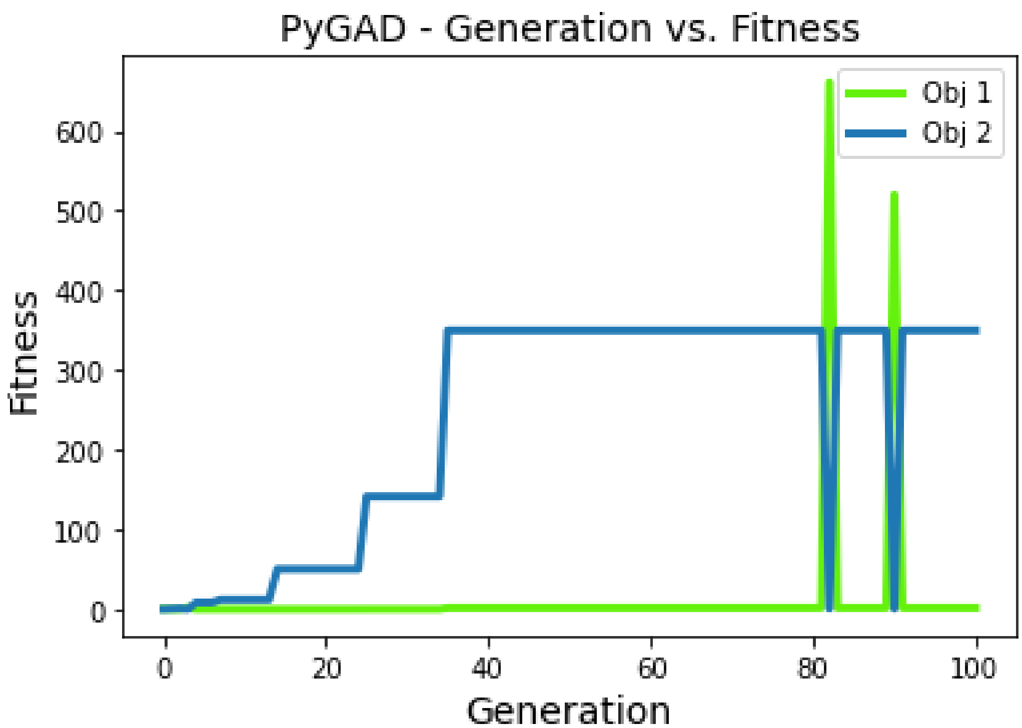
```python
import pygad
import numpy

"""
Given these 2 functions:
    y1 = f(w1:w6) = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + 6wx6
    y2 = f(w1:w6) = w1x7 + w2x8 + w3x9 + w4x10 + w5x11 + 6wx12
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7) and y=50
    and   (x7,x8,x9,x10,x11,x12)=(-2,0.7,-9,1.4,3,5) and y=30
What are the best values for the 6 weights (w1 to w6)? We are going to use the␣
↪genetic algorithm to optimize these 2 functions.
This is a multi-objective optimization problem.

PyGAD considers the problem as multi-objective if the fitness function returns:
    1) List.
    2) Or tuple.
    3) Or numpy.ndarray.
"""

function_inputs1 = [4,-2,3.5,5,-11,-4.7] # Function 1 inputs.
function_inputs2 = [-2,0.7,-9,1.4,3,5] # Function 2 inputs.
desired_output1 = 50 # Function 1 output.
desired_output2 = 30 # Function 2 output.

def fitness_func(ga_instance, solution, solution_idx):
    output1 = numpy.sum(solution*function_inputs1)
    output2 = numpy.sum(solution*function_inputs2)
    fitness1 = 1.0 / (numpy.abs(output1 - desired_output1) + 0.000001)
    fitness2 = 1.0 / (numpy.abs(output2 - desired_output2) + 0.000001)
    return [fitness1, fitness2]

num_generations = 100
num_parents_mating = 10

sol_per_pop = 20
num_genes = len(function_inputs1)

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       sol_per_pop=sol_per_pop,
                       num_genes=num_genes,
                       fitness_func=fitness_func,
                       parent_selection_type='nsga2')

ga_instance.run()

ga_instance.plot_fitness(label=['Obj 1', 'Obj 2'])

solution, solution_fitness, solution_idx = ga_instance.best_solution(ga_instance.last_
↪generation_fitness)
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
```

()

```
prediction = numpy.sum(numpy.array(function_inputs1)*solution)
print(f"Predicted output 1 based on the best solution : {prediction}")
prediction = numpy.sum(numpy.array(function_inputs2)*solution)
print(f"Predicted output 2 based on the best solution : {prediction}")
```

```
Parameters of the best solution : [ 0.79676439 -2.98823386 -4.12677662  5.70539445 -2.
↪02797016 -1.07243922]
Fitness value of the best solution = [  1.68090829 349.8591915 ]
Predicted output 1 based on the best solution : 50.59491545442283
Predicted output 2 based on the best solution : 29.99714270722312
```

```
plot_fitness()
```



fruit.jpg

```
import imageio
import numpy

target_im = imageio.imread('fruit.jpg')
target_im = numpy.asarray(target_im/255, dtype=float)
```

pygad.GA

```
import gari

target_chromosome = gari.img2chromosome(target_im)

def fitness_fun(ga_instance, solution, solution_idx):
    fitness = numpy.sum(numpy.abs(target_chromosome-solution))

    # Negating the fitness value to make it increasing rather than decreasing.
    fitness = numpy.sum(target_chromosome) - fitness
    return fitness
```

gari.img2chromosome()

gari

```python
import numpy
import functools
import operator

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

def chromosome2img(vector, shape):
    if len(vector) != functools.reduce(operator.mul, shape):
        raise ValueError(f"A vector of length {len(vector)} into an array of shape
↪{shape}.")

    return numpy.reshape(a=vector, newshape=shape)
```

pygad.GA

mutation_by_replacementTrueinit_range_lowinit_range_highrandom_mutation_min_valrandom_mutation_max_val

init_range_lowrandom_mutation_min_valinit_range_highrandom_mutation_max_val
,

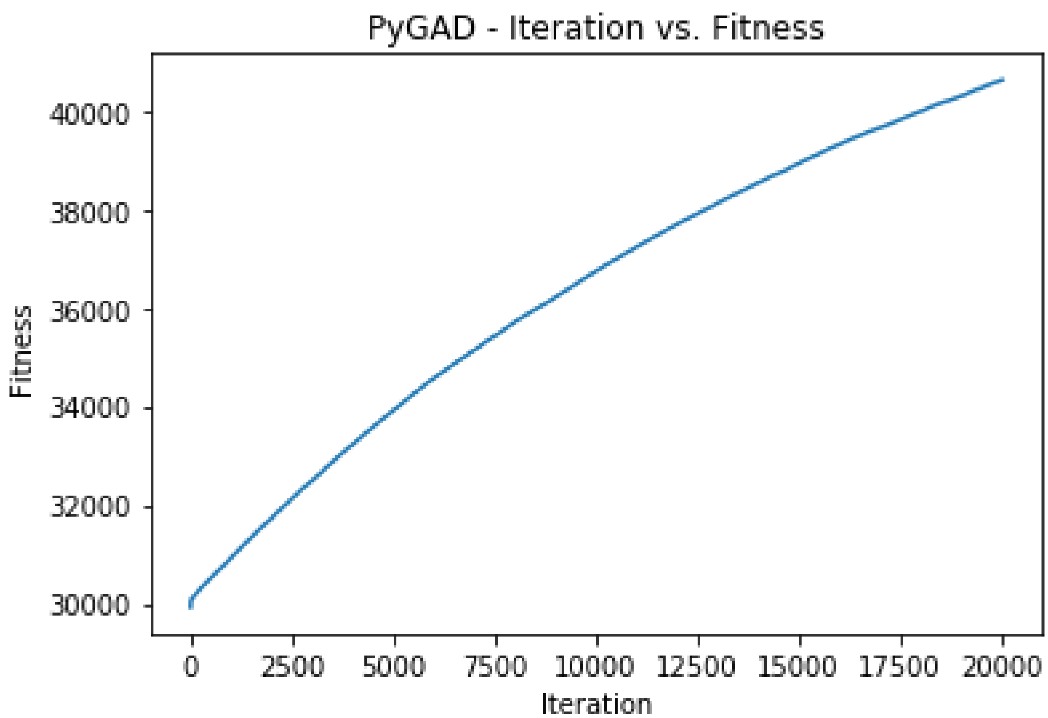
```python
import pygad

ga_instance = pygad.GA(num_generations=20000,
                       num_parents_mating=10,
                       fitness_func=fitness_fun,
                       sol_per_pop=20,
                       num_genes=target_im.size,
                       init_range_low=0.0,
                       init_range_high=1.0,
                       mutation_percent_genes=0.01,
                       mutation_type="random",
                       mutation_by_replacement=True,
                       random_mutation_min_val=0.0,
                       random_mutation_max_val=1.0)
```

run()

```
ga_instance.run()
```

run()plot_fitness()

```
ga_instance.plot_fitness()
```



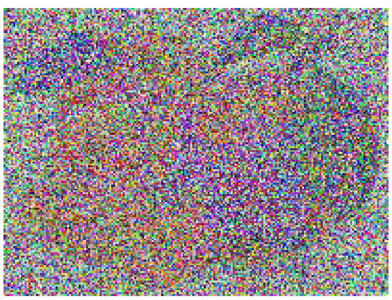

```
# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
→generations.")

result = gari.chromosome2img(solution, target_im.shape)
matplotlib.pyplot.imshow(result)
```

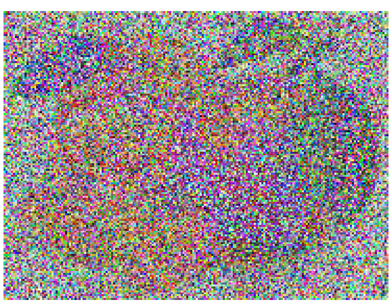
()

()

```
matplotlib.pyplot.title("PyGAD & GARI for Reproducing Images")
matplotlib.pyplot.show()
```

pygad.GA





().

fitness

```python
def fitness_func(ga_instance, solution, solution_idx):
    ...
    return fitness
```

list

tuple

numpy.ndarray

```python
def fitness_func(ga_instance, solution, solution_idx):
    ...
    return [fitness1, fitness2, ..., fitnessN]
```

nsga2

tournament_nsga2

```
y1 = f(w1:w6) = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + 6wx6
y2 = f(w1:w6) = w1x7 + w2x8 + w3x9 + w4x10 + w5x11 + 6wx12
```

```
(x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7)y=50
(x7,x8,x9,x10,x11,x12)=(-2,0.7,-9,1.4,3,5)y=30
```

() w1 w6

y1y2

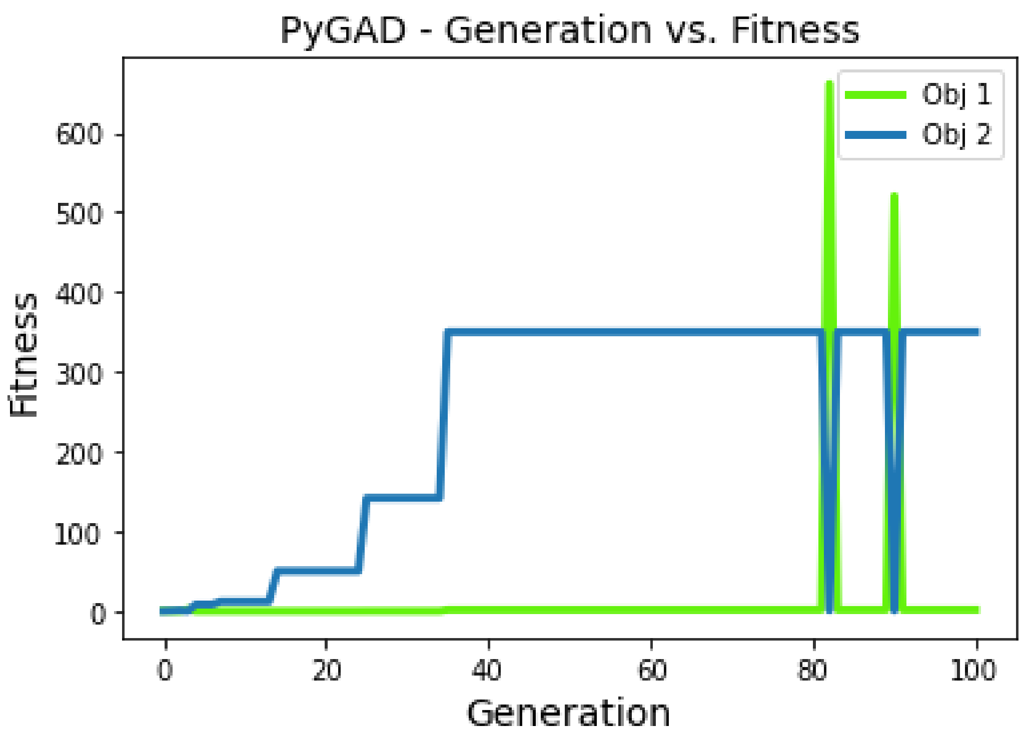
```
import pygad
import numpy

"""
Given these 2 functions:
    y1 = f(w1:w6) = w1x1 + w2x2 + w3x3 + w4x4 + w5x5 + 6wx6
    y2 = f(w1:w6) = w1x7 + w2x8 + w3x9 + w4x10 + w5x11 + 6wx12
    where (x1,x2,x3,x4,x5,x6)=(4,-2,3.5,5,-11,-4.7) and y=50
    and   (x7,x8,x9,x10,x11,x12)=(-2,0.7,-9,1.4,3,5) and y=30
What are the best values for the 6 weights (w1 to w6)? We are going to use the␣
↪genetic algorithm to optimize these 2 functions.
This is a multi-objective optimization problem.

PyGAD considers the problem as multi-objective if the fitness function returns:
    1) List.
    2) Or tuple.
    3) Or numpy.ndarray.
"""

function_inputs1 = [4,-2,3.5,5,-11,-4.7] # Function 1 inputs.
function_inputs2 = [-2,0.7,-9,1.4,3,5] # Function 2 inputs.
desired_output1 = 50 # Function 1 output.
desired_output2 = 30 # Function 2 output.

def fitness_func(ga_instance, solution, solution_idx):
    output1 = numpy.sum(solution*function_inputs1)
    output2 = numpy.sum(solution*function_inputs2)
    fitness1 = 1.0 / (numpy.abs(output1 - desired_output1) + 0.000001)
    fitness2 = 1.0 / (numpy.abs(output2 - desired_output2) + 0.000001)
    return [fitness1, fitness2]

num_generations = 100
num_parents_mating = 10

sol_per_pop = 20
num_genes = len(function_inputs1)

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       sol_per_pop=sol_per_pop,
                       num_genes=num_genes,
                       fitness_func=fitness_func,
                       parent_selection_type='nsga2')

ga_instance.run()

ga_instance.plot_fitness(label=['Obj 1', 'Obj 2'])

solution, solution_fitness, solution_idx = ga_instance.best_solution(ga_instance.last_
↪generation_fitness)
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")

prediction = numpy.sum(numpy.array(function_inputs1)*solution)
```

()

```
print(f"Predicted output 1 based on the best solution : {prediction}")
prediction = numpy.sum(numpy.array(function_inputs2)*solution)
print(f"Predicted output 2 based on the best solution : {prediction}")
```

```
Parameters of the best solution : [ 0.79676439 -2.98823386 -4.12677662  5.70539445 -2.
↪02797016 -1.07243922]
Fitness value of the best solution = [  1.68090829 349.8591915 ]
Predicted output 1 based on the best solution : 50.59491545442283
Predicted output 2 based on the best solution : 29.99714270722312
```

plot_fitness()



gene_space

gene_space' gene_space

gene_space

[0.4, 12, -5, 21.2]

[-2, 0.3]

[1.2, 63.2, 7.4]

gene_space

```
gene_space = [[0.4, 12, -5, 21.2],
              [-2, 0.3],
              [1.2, 63.2, 7.4]]
```

gene_space

```
gene_space = [33, 7, 0.5, 95. 6.3, 0.74]
```

range()range(1, 7)1, 2, 3, 4, 5, and 6numpy.arange()numpy.linspace()

gene_space

[)

'low'

'high'

```
{'low': 1,
 'high': 5}
```

'low''high'

() ().

```
gene_space = [{'low': 1, 'high': 5}, {'low': 0.3, 'high': 1.4}, {'low': -0.2, 'high':␣
↪4.5}]
```

gene_space

gene_space

gene_spacegene_space

```
gene_space = [0.3, 5.2, -4, 8]
```

gene_space

(intfloatNumPy):

listtuplenumpy.ndarrayrangenumpy.arange()numpy.linspace

dict"low""high""step""low""high""step"().

None: Noneinit_range_lowinit_range_highrandom_mutation_min_valrandom_mutation_max_valgene_spaceNone

gene_space

[0.4, -5][0.5, -3.2, 8.8, -9]

```
gene_space = [[0.4, -5], [0.5, -3.2, 8.2, -9]]
```

```
gene_space = [range(5), range(10, 20)]
```

gene_space

```
gene_space = numpy.arange(15)
```

gene_space

```
gene_space = {"low": 4, "high": 30}
```

```
gene_space = {"low": 4, "high": 30, "step": 2.5}
```

dict{"low": 0, "high": 10}gene_space[) 010109.9999[float, 2]

Noneinit_range_lowinit_range_highpygad.GA' random_mutation_min_valrandom_mutation_max_valNone

```
gene_space = [range(5), None, numpy.linspace(10, 20, 300)]
```

initial_populationgene_space

gene_space

gene_space

gene_spaceintfloat

gene_space[1, 2, 3]

```
Gene space: [[1, 2, 3],
             None]
Solution: [1, 5]
```

[1, 5]().

None

random_mutation_min_valrandom_mutation_max_val
,

-0.5

gene_space

gene_space{'low': 1, 'high': 5}

```
Gene space: {'low': 1, 'high': 5}
Solution: [1.5, 3.4]
```

random_mutation_min_val=-1random_mutation_max_val=10.3() 1.51.5+0.3=1.8

```
Gene space: {'low': 1, 'high': 5, 'step': 0.5}
```

"stop"on_generationon_generationpygad.GA'

num_generationspygad.GA

"stop"

```
def func_generation(ga_instance):
    if ga_instance.best_solution()[1] >= 70:
        return "stop"
```

stop_criteriapygad.GA

str

```
"word_num"
```

reachsaturate

reachrun()reach"reach_40">

saturatesaturate"saturate_7"run()

127.415

```
import pygad
import numpy

equation_inputs = [4, -2, 3.5, 8, 9, 4]
desired_output = 44

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)

    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)

    return fitness

ga_instance = pygad.GA(num_generations=200,
                       sol_per_pop=10,
                       num_parents_mating=4,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       stop_criteria=["reach_127.4", "saturate_15"])

ga_instance.run()
print(f"Number of generations passed is {ga_instance.generations_completed}")
```

keep_elitism() 1

keep_elitismpygad.GA

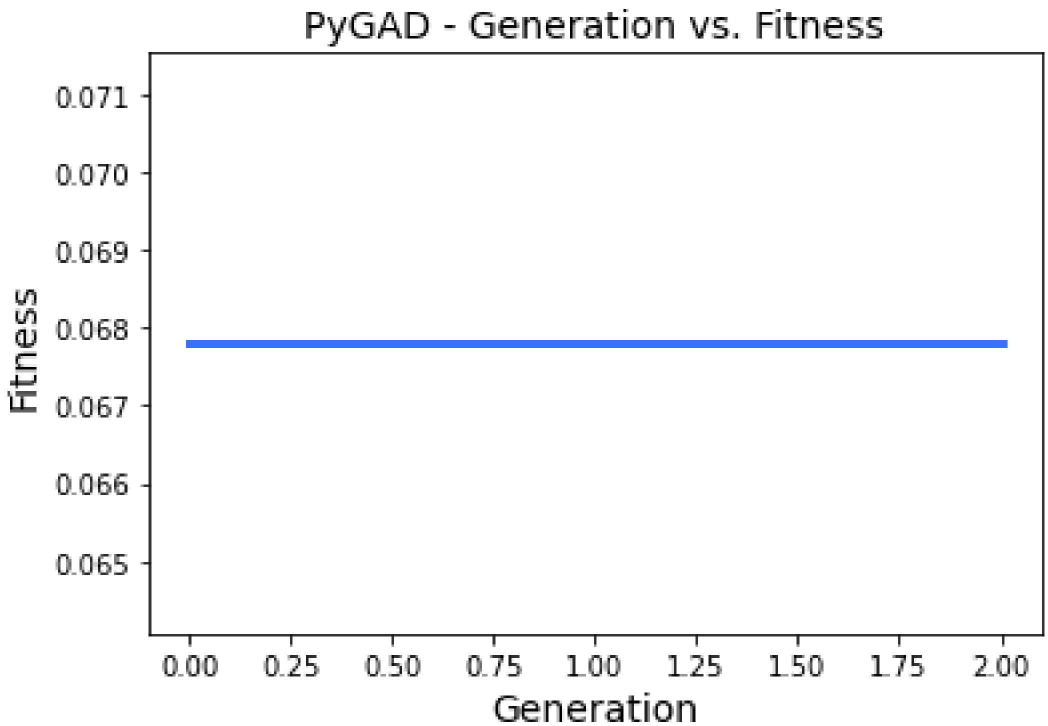
```python
import numpy
import pygad

function_inputs = [4,-2,3.5,5,-11,-4.7]
desired_output = 44

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

ga_instance = pygad.GA(num_generations=2,
                       num_parents_mating=3,
                       fitness_func=fitness_func,
                       num_genes=6,
                       sol_per_pop=5,
                       keep_elitism=2)

ga_instance.run()
```

keep_elitism

- >= 0
- <= sol_per_pop

keep_elitismsol_per_pop

```python
...

ga_instance = pygad.GA(...,
                       sol_per_pop=5,
                       keep_elitism=5)

ga_instance.run()
```



keep_elitism(), keep_parentskeep_elitismkeep_parentskeep_parentskeep_elitism=0

random_seed

random_seedNone

```
import numpy
import pygad

function_inputs = [4,-2,3.5,5,-11,-4.7]
desired_output = 44

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

ga_instance = pygad.GA(num_generations=2,
                       num_parents_mating=3,
                       fitness_func=fitness_func,
                       sol_per_pop=5,
```

()

()

```
                       num_genes=6,
                       random_seed=2)

ga_instance.run()
best_solution, best_solution_fitness, best_match_idx = ga_instance.best_solution()
print(best_solution)
print(best_solution_fitness)
```

```
[ 2.77249188 -4.06570662  0.04196872 -3.47770796 -0.57502138 -3.22775267]
0.04872203136549972
```

```
[ 2.77249188 -4.06570662  0.04196872 -3.47770796 -0.57502138 -3.22775267]
0.04872203136549972
```

run()

- self.best_solutions
- self.best_solutions_fitness
- self.solutions
- self.solutions_fitness

save()

```python
import pygad

def fitness_func(ga_instance, solution, solution_idx):
    ...
    return fitness

ga_instance = pygad.GA(...)

ga_instance.run()

ga_instance.plot_fitness()

ga_instance.save("pygad_GA")
```

load() run()

```python
import pygad

def fitness_func(ga_instance, solution, solution_idx):
    ...
    return fitness

loaded_ga_instance = pygad.load("pygad_GA")

loaded_ga_instance.run()
```

()

()

```
loaded_ga_instance.plot_fitness()
```

plot_fitness()

(self.best_solutions self.best_solutions_fitness) save_best_solutionsTrue(self.solutions self.solutions_fitness) save_solutionsTrue

- population
- num_offspring
- num_parents_mating
- fitness_func
- sol_per_poppopulation
- last_generation_*
  - last_generation_fitness
  - last_generation_parentslast_generation_parents_indices
  - last_generation_elitismlast_generation_elitism_indiceskeep_elitism != 0 keep_elitism
- pop_size

allow_duplicate_genes

allow_duplicate_genes=True(), allow_duplicate_genes=False

allow_duplicate_genes

```
import pygad

def fitness_func(ga_instance, solution, solution_idx):
    return 0

def on_generation(ga):
    print("Generation", ga.generations_completed)
    print(ga.population)
```

()

()

```
ga_instance = pygad.GA(num_generations=5,
                       sol_per_pop=5,
                       num_genes=4,
                       mutation_num_genes=3,
                       random_mutation_min_val=-5,
                       random_mutation_max_val=5,
                       num_parents_mating=2,
                       fitness_func=fitness_func,
                       gene_type=int,
                       on_generation=on_generation,
                       allow_duplicate_genes=False)
ga_instance.run()
```

```
Generation 1
[[ 2 -2 -3  3]
 [ 0  1  2  3]
 [ 5 -3  6  3]
 [-3  1 -2  4]
 [-1  0 -2  3]]
Generation 2
[[-1  0 -2  3]
 [-3  1 -2  4]
 [ 0 -3 -2  6]
 [-3  0 -2  3]
 [ 1 -4  2  4]]
Generation 3
[[ 1 -4  2  4]
 [-3  0 -2  3]
 [ 4  0 -2  1]
 [-4  0 -2 -3]
 [-4  2  0  3]]
Generation 4
[[-4  2  0  3]
 [-4  0 -2 -3]
 [-2  5  4 -3]
 [-1  2 -4  4]
 [-4  2  0 -3]]
Generation 5
[[-4  2  0 -3]
 [-1  2 -4  4]
 [ 3  4 -4  0]
 [-1  0  2 -2]
 [-4  2 -1  1]]
```

allow_duplicate_genesgene_space().

```
import pygad

def fitness_func(ga_instance, solution, solution_idx):
    return 0

def on_generation(ga):
    print("Generation", ga.generations_completed)
    print(ga.population)
```

()

()

```
ga_instance = pygad.GA(num_generations=1,
                       sol_per_pop=5,
                       num_genes=4,
                       num_parents_mating=2,
                       fitness_func=fitness_func,
                       gene_type=int,
                       gene_space=[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3,
↪ 4]],
                       on_generation=on_generation,
                       allow_duplicate_genes=False)
ga_instance.run()
```

```
Generation 1
[[2 3 1 4]
 [2 3 1 4]
 [2 4 1 3]
 [2 3 1 4]
 [1 3 2 4]]
Generation 2
[[1 3 2 4]
 [2 3 1 4]
 [1 3 2 4]
 [2 3 4 1]
 [1 3 4 2]]
Generation 3
[[1 3 4 2]
 [2 3 4 1]
 [1 3 4 2]
 [3 1 4 2]
 [3 2 4 1]]
Generation 4
[[3 2 4 1]
 [3 1 4 2]
 [3 2 4 1]
 [1 2 4 3]
 [1 3 4 2]]
Generation 5
[[1 3 4 2]
 [1 2 4 3]
 [2 1 4 3]
 [1 2 4 3]
 [1 2 4 3]]
```

`gene_space=[[3, 0, 1], [4, 1, 2], [0, 2], [3, 2, 0]]` `[3 2 0 0]`

`[3 4 2 0]`

allow_duplicate_genes=Falsegene_space

,

```
Gene space: [[2, 3],
             [3, 4],
             [4, 5],
             [5, 6]]
Solution: [3, 4, 4, 5]
```

[3, 4][4, 5]

[3, 4, 4, 5]

'[3, 4]

[]

[]

```
Gene space: [[0, 1],
             [1, 2],
             [2, 3],
             [3, 4]]
Solution: [1, 2, 2, 3]
```

[]

[]

gene_type

gene_typegene_typefloatgene_type

,

gene_typeint

```
gene_type=int
```

' intfloatNumPygene_type

float

int

```
import pygad
import numpy

equation_inputs = [4, -2, 3.5, 8, -2]
desired_output = 2671.1234

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       gene_type=int)

print("Initial Population")
print(ga_instance.initial_population)

ga_instance.run()
```

()

```
print("Final Population")
print(ga_instance.population)
```

```
Initial Population
[[ 1 -1  2  0 -3]
 [ 0 -2  0 -3 -1]
 [ 0 -1 -1  2  0]
 [-2  3 -2  3  3]
 [ 0  0  2 -2 -2]]

Final Population
[[ 1 -1  2  2  0]
 [ 1 -1  2  2  0]
 [ 1 -1  2  2  0]
 [ 1 -1  2  2  0]
 [ 1 -1  2  2  0]]
```

floatfloatfloat

```
gene_type=[float, 3]
```

float

```
import pygad
import numpy

equation_inputs = [4, -2, 3.5, 8, -2]
desired_output = 2671.1234

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)

    return fitness

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       gene_type=[float, 3])

print("Initial Population")
print(ga_instance.initial_population)

ga_instance.run()

print("Final Population")
print(ga_instance.population)
```

gene_type

```
Initial Population
[[-2.417 -0.487  3.623  2.457 -2.362]
 [-1.231  0.079 -1.63   1.629 -2.637]
 [ 0.692 -2.098  0.705  0.914 -3.633]
 [ 2.637 -1.339 -1.107 -0.781 -3.896]
 [-1.495  1.378 -1.026  3.522  2.379]]

Final Population
[[ 1.714 -1.024  3.623  3.185 -2.362]
 [ 0.692 -1.024  3.623  3.185 -2.362]
 [ 0.692 -1.024  3.623  3.375 -2.362]
 [ 0.692 -1.024  4.041  3.185 -2.362]
 [ 1.714 -0.644  3.623  3.185 -2.362]]
```

gene_typelisttuplenumpy.ndarray

```
gene_type=[int, float, numpy.float16, numpy.int8, float]
```

```
import pygad
import numpy

equation_inputs = [4, -2, 3.5, 8, -2]
desired_output = 2671.1234

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       gene_type=[int, float, numpy.float16, numpy.int8, float])

print("Initial Population")
print(ga_instance.initial_population)

ga_instance.run()

print("Final Population")
print(ga_instance.population)
```

```
Initial Population
[[0 0.8615522360026828 0.7021484375 -2 3.5301821368185866]
 [-3 2.648189378595294 -3.830078125 1 -0.9586271572917742]
 [3 3.7729827570110714 1.2529296875 -3 1.395741994211889]
 [0 1.0490687178053282 1.51953125 -2 0.7243617940450235]
 [0 -0.6550158436937226 -2.861328125 -2 1.8212734549263097]]
```

()

```
Final Population
[[3 3.7729827570110714 2.055 0 0.7243617940450235]
 [3 3.7729827570110714 1.458 0 -0.14638754050305036]
 [3 3.7729827570110714 1.458 0 0.0869406120516778]
 [3 3.7729827570110714 1.458 0 0.7243617940450235]
 [3 3.7729827570110714 1.458 0 -0.14638754050305036]]
```

float

```
gene_type=[int, [float, 2], numpy.float16, numpy.int8, [float, 1]]
```

```python
import pygad
import numpy

equation_inputs = [4, -2, 3.5, 8, -2]
desired_output = 2671.1234

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       gene_type=[int, [float, 2], numpy.float16, numpy.int8, [float,␣
↪1]])

print("Initial Population")
print(ga_instance.initial_population)

ga_instance.run()

print("Final Population")
print(ga_instance.population)
```

```
Initial Population
[[-2 -1.22 1.716796875 -1 0.2]
 [-1 -1.58 -3.091796875 0 -1.3]
 [3 3.35 -0.107421875 1 -3.3]
 [-2 -3.58 -1.779296875 0 0.6]
 [2 -3.73 2.65234375 3 -0.5]]

Final Population
[[2 -4.22 3.47 3 -1.3]
 [2 -3.73 3.47 3 -1.3]
 [2 -4.22 3.47 2 -1.3]
```

()

gene_type

()

```
[2 -4.58 3.47 3 -1.3]
[2 -3.73 3.47 3 -1.3]]
```

()

parallel_processingpygad.GA

```
import pygad
...
ga_instance = pygad.GA(...,
                       parallel_processing=...)
...
```

parallel_processing

None()

(

listtuple

'process''thread'

0parallel_processing=None

Noneconcurrent.futures module

parallel_processing

parallel_processing=4

parallel_processing=["thread", 5]parallel_processing=5

parallel_processing=["process", 8]

parallel_processing=["process", 0]parallel_processing=None

forpygad.GAparallel_processing=None

```
import pygad
import time

def fitness_func(ga_instance, solution, solution_idx):
    for _ in range(99):
        pass
    return 0

ga_instance = pygad.GA(num_generations=9999,
                       num_parents_mating=3,
                       sol_per_pop=5,
                       num_genes=10,
                       fitness_func=fitness_func,
                       suppress_warnings=True,
                       parallel_processing=None)

if __name__ == '__main__':
    t1 = time.time()

    ga_instance.run()

    t2 = time.time()
    print("Time is", t2-t1)
```

1.5

' 5

```
...
ga_instance = pygad.GA(...,
                       parallel_processing=5)
...
```

99

```
...
ga_instance = pygad.GA(num_generations=99,
                       ...,
                       parallel_processing=["process", 5])
...
```

```python
import pygad
import time

def fitness_func(ga_instance, solution, solution_idx):
    for _ in range(99999999):
        pass
    return 0

ga_instance = pygad.GA(num_generations=5,
                       num_parents_mating=3,
                       sol_per_pop=5,
                       num_genes=10,
                       fitness_func=fitness_func,
                       suppress_warnings=True,
                       parallel_processing=None)

if __name__ == '__main__':
    t1 = time.time()
    ga_instance.run()
    t2 = time.time()
    print("Time is", t2-t1)
```

```python
...
ga_instance = pygad.GA(...,
                       parallel_processing=["process", 10])
...
```

```python
...
ga_instance = pygad.GA(...,
                       parallel_processing=["thread", 10])
...
```

`summary()`

- `line_length=70`
- `fill_character=" "`
- `line_character="-"`
- `line_character2="="`
- `columns_equal_len=False`

print_step_parameters=Trueprint_step_parameters=False
print_parameters_summary=True

print_parameters_summary=Trueprint_step_parameters=False

```python
import pygad
import numpy

function_inputs = [4,-2,3.5,5,-11,-4.7]
desired_output = 44

def genetic_fitness(solution, solution_idx):
    output = numpy.sum(solution*function_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

def on_gen(ga):
    pass

def on_crossover_callback(a, b):
    pass

ga_instance = pygad.GA(num_generations=100,
                       num_parents_mating=10,
                       sol_per_pop=20,
                       num_genes=len(function_inputs),
                       on_crossover=on_crossover_callback,
                       on_generation=on_gen,
                       parallel_processing=2,
                       stop_criteria="reach_10",
                       fitness_batch_size=4,
                       crossover_probability=0.4,
                       fitness_func=genetic_fitness)
```

summary()on_crossover_callback()on_gen()

```python
ga_instance.summary()
```

```
----------------------------------------------------------------------
                           PyGAD Lifecycle
======================================================================
Step                    Handler                             Output Shape
======================================================================
Fitness Function        genetic_fitness()                   (1)
Fitness batch size: 4
----------------------------------------------------------------------
Parent Selection        steady_state_selection()            (10, 6)
Number of Parents: 10
----------------------------------------------------------------------
Crossover               single_point_crossover()            (10, 6)
Crossover probability: 0.4
----------------------------------------------------------------------
On Crossover            on_crossover_callback()             None
----------------------------------------------------------------------
Mutation                random_mutation()                   (10, 6)
Mutation Genes: 1
```

()

()

```
Random Mutation Range: (-1.0, 1.0)
Mutation by Replacement: False
Allow Duplicated Genes: True
----------------------------------------------------------------------
On Generation           on_gen()                           None
Stop Criteria: [['reach', 10.0]]
----------------------------------------------------------------------
======================================================================
Population Size: (20, 6)
Number of Generations: 100
Initial Population Range: (-4, 4)
Keep Elitism: 1
Gene DType: [<class 'float'>, None]
Parallel Processing: ['thread', 2]
Save Best Solutions: False
Save Solutions: False
======================================================================
```

print_step_parametersprint_parameters_summaryFalse

```
ga_instance.summary(print_step_parameters=False,
                    print_parameters_summary=False)
```

```
----------------------------------------------------------------------
                           PyGAD Lifecycle
======================================================================
Step                    Handler                            Output Shape
======================================================================
Fitness Function        genetic_fitness()                  (1)
----------------------------------------------------------------------
Parent Selection        steady_state_selection()           (10, 6)
----------------------------------------------------------------------
Crossover               single_point_crossover()           (10, 6)
----------------------------------------------------------------------
On Crossover            on_crossover_callback()            None
----------------------------------------------------------------------
Mutation                random_mutation()                  (10, 6)
----------------------------------------------------------------------
On Generation           on_gen()                           None
----------------------------------------------------------------------
======================================================================
```

print()logger

```
import logging

logger = ...

ga_instance = pygad.GA(...,
                       logger=logger,
                       ...)
```

None(logger=None), print()

print()

Handler

Formatter

logging

```
import logging

# Create a logger
logger = logging.getLogger(__name__)

# Set the logger level to debug so that all the messages are printed.
logger.setLevel(logging.DEBUG)

# Create a stream handler to log the messages to the console.
stream_handler = logging.StreamHandler()

# Set the handler level to debug.
stream_handler.setLevel(logging.DEBUG)

# Create a formatter
formatter = logging.Formatter('%(message)s')

# Add the formatter to handler.
stream_handler.setFormatter(formatter)

# Add the stream handler to the logger
logger.addHandler(stream_handler)
```

Formatter

```
logger.debug('Debug message.')
logger.info('Info message.')
logger.warning('Warn message.')
logger.error('Error message.')
logger.critical('Critical message.')
```

print()

```
Debug message.
Info message.
Warn message.
Error message.
Critical message.
```

```
formatter = logging.Formatter('%(asctime)s %(levelname)s: %(message)s', datefmt='%Y-
↪%m-%d %H:%M:%S')
```

```
2023-04-03 18:46:27 DEBUG: Debug message.
2023-04-03 18:46:27 INFO: Info message.
2023-04-03 18:46:27 WARNING: Warn message.
2023-04-03 18:46:27 ERROR: Error message.
2023-04-03 18:46:27 CRITICAL: Critical message.
```

```
logger.handlers.clear()
```

logfile.txt

```
import logging

level = logging.DEBUG
name = 'logfile.txt'

logger = logging.getLogger(name)
logger.setLevel(level)

file_handler = logging.FileHandler(name, 'a+', 'utf-8')
file_handler.setLevel(logging.DEBUG)
file_format = logging.Formatter('%(asctime)s %(levelname)s: %(message)s -
↪%(pathname)s:%(lineno)d', datefmt='%Y-%m-%d %H:%M:%S')
file_handler.setFormatter(file_format)
logger.addHandler(file_handler)
```

```
2023-04-03 18:54:03 DEBUG: Debug message. - c:\users\agad069\desktop\logger\example2.
↪py:46
2023-04-03 18:54:03 INFO: Info message. - c:\users\agad069\desktop\logger\example2.
↪py:47
2023-04-03 18:54:03 WARNING: Warn message. - c:\users\agad069\desktop\logger\example2.
↪py:48
2023-04-03 18:54:03 ERROR: Error message. - c:\users\agad069\desktop\logger\example2.
↪py:49
2023-04-03 18:54:03 CRITICAL: Critical message. - c:\users\agad069\desktop\logger\
↪example2.py:50
```

```
logger.handlers.clear()
```

```
import logging

level = logging.DEBUG
name = 'logfile.txt'

logger = logging.getLogger(name)
logger.setLevel(level)

file_handler = logging.FileHandler(name,'a+','utf-8')
file_handler.setLevel(logging.DEBUG)
file_format = logging.Formatter('%(asctime)s %(levelname)s: %(message)s -
↪%(pathname)s:%(lineno)d', datefmt='%Y-%m-%d %H:%M:%S')
file_handler.setFormatter(file_format)
logger.addHandler(file_handler)

console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)
console_format = logging.Formatter('%(message)s')
console_handler.setFormatter(console_format)
logger.addHandler(console_handler)
```

logfile.txt

```
logger.handlers.clear()
```

logger

```
import logging
import pygad
import numpy

level = logging.DEBUG
name = 'logfile.txt'

logger = logging.getLogger(name)
logger.setLevel(level)

file_handler = logging.FileHandler(name,'a+','utf-8')
file_handler.setLevel(logging.DEBUG)
file_format = logging.Formatter('%(asctime)s %(levelname)s: %(message)s', datefmt='%Y-
↪%m-%d %H:%M:%S')
file_handler.setFormatter(file_format)
logger.addHandler(file_handler)

console_handler = logging.StreamHandler()
```

()

()

```
console_handler.setLevel(logging.INFO)
console_format = logging.Formatter('%(message)s')
console_handler.setFormatter(console_format)
logger.addHandler(console_handler)

equation_inputs = [4, -2, 8]
desired_output = 2671.1234

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

def on_generation(ga_instance):
    ga_instance.logger.info(f"Generation = {ga_instance.generations_completed}")
    ga_instance.logger.info(f"Fitness    = {ga_instance.best_solution(pop_fitness=ga_
↪instance.last_generation_fitness)[1]}")

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=40,
                       num_parents_mating=2,
                       keep_parents=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       on_generation=on_generation,
                       logger=logger)
ga_instance.run()

logger.handlers.clear()
```

```
2023-04-03 19:04:27 INFO: Generation = 1
2023-04-03 19:04:27 INFO: Fitness    = 0.00038086960368076276
2023-04-03 19:04:27 INFO: Generation = 2
2023-04-03 19:04:27 INFO: Fitness    = 0.00038214871408010853
2023-04-03 19:04:27 INFO: Generation = 3
2023-04-03 19:04:27 INFO: Fitness    = 0.0003832795907974678
2023-04-03 19:04:27 INFO: Generation = 4
2023-04-03 19:04:27 INFO: Fitness    = 0.00038398612055017196
2023-04-03 19:04:27 INFO: Generation = 5
2023-04-03 19:04:27 INFO: Fitness    = 0.00038442348890867516
2023-04-03 19:04:27 INFO: Generation = 6
2023-04-03 19:04:27 INFO: Fitness    = 0.0003854406039137763
2023-04-03 19:04:27 INFO: Generation = 7
2023-04-03 19:04:27 INFO: Fitness    = 0.00038646083174063284
2023-04-03 19:04:27 INFO: Generation = 8
2023-04-03 19:04:27 INFO: Fitness    = 0.0003875169193024936
2023-04-03 19:04:27 INFO: Generation = 9
2023-04-03 19:04:27 INFO: Fitness    = 0.0003888816727311021
2023-04-03 19:04:27 INFO: Generation = 10
2023-04-03 19:04:27 INFO: Fitness    = 0.000389832593101348
```

solutionssave_solutions=True

best_solutionssave_best_solutions=True

last_generation_elitismkeep_elitism>

last_generation_parentskeep_parents> keep_parents=-1

keep_elisitm=0

keep_parents=0

keep_solutions=False

keep_best_solutions=False

```python
import pygad
...
ga_instance = pygad.GA(...,
                       keep_elitism=0,
                       keep_parents=0,
                       save_solutions=False,
                       save_best_solutions=False,
                       ...)
```

(>).

pygad.GA

cal_pop_fitness()pygad.GA

save_solutions

FalseTruesolutionspygad.GAsolutions

save_best_solutions

FalseTrue

keep_elitism

keep_parents

-1

keep_elitism

```
ga_instance = pygad.GA(...,
                       keep_elitism=1,
                       ...)
```

keep_elitism=1keep_elitism=2

().

keep_elitismkeep_parentssave_solutionssave_best_solutionsFalse

```
ga_instance = pygad.GA(...,
                       keep_elitism=0,
                       keep_parents=0,
                       save_solutions=False,
                       save_best_solutions=False,
                       ...)
```

fitness_batch_sizefitness_batch_size

- 1Nonefitness_batch_size1None(),
- 1 < fitness_batch_size <= sol_per_popfitness_batch_size1 < fitness_batch_size <= sol_per_popfitness_batch_size

fitness_batch_size

fitness_batch_sizeNone(). 1fitness_func

```
solution: [ 2.52860734, -0.94178795, 2.97545704, 0.84131987, -3.78447118, 2.41008358]
solution_idx: 3
```

2020*5 = 10020*5 + 20 = 120

keep_elitismkeep_parents0

```python
import pygad
import numpy

function_inputs = [4,-2,3.5,5,-11,-4.7]
desired_output = 44

number_of_calls = 0

def fitness_func(ga_instance, solution, solution_idx):
    global number_of_calls
    number_of_calls = number_of_calls + 1
    output = numpy.sum(solution*function_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

ga_instance = pygad.GA(num_generations=5,
                       num_parents_mating=10,
                       sol_per_pop=20,
                       fitness_func=fitness_func,
                       fitness_batch_size=None,
                       # fitness_batch_size=1,
                       num_genes=len(function_inputs),
                       keep_elitism=0,
                       keep_parents=0)

ga_instance.run()
print(number_of_calls)
```

```
120
```

fitness_batch_size

fitness_batch_size44().

```
solutions:
    [[ 3.1129432  -0.69123589  1.93792414  2.23772968 -1.54616001 -0.53930799]
     [ 3.38508121  0.19890812  1.93792414  2.23095014 -3.08955597  3.10194128]
     [ 2.37079504 -0.88819803  2.97545704  1.41742256 -3.95594055  2.45028256]
     [ 2.52860734 -0.94178795  2.97545704  0.84131987 -3.78447118  2.41008358]]
solutions_indices:
    [16, 17, 18, 19]
```

20/4 = 5

5*5 = 255*5 + 5 = 30

```
import pygad
import numpy

function_inputs = [4,-2,3.5,5,-11,-4.7]
desired_output = 44

number_of_calls = 0

def fitness_func_batch(ga_instance, solutions, solutions_indices):
    global number_of_calls
    number_of_calls = number_of_calls + 1
    batch_fitness = []
    for solution in solutions:
        output = numpy.sum(solution*function_inputs)
        fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
        batch_fitness.append(fitness)
    return batch_fitness

ga_instance = pygad.GA(num_generations=5,
                       num_parents_mating=10,
                       sol_per_pop=20,
                       fitness_func=fitness_func_batch,
                       fitness_batch_size=4,
                       num_genes=len(function_inputs),
                       keep_elitism=0,
                       keep_parents=0)

ga_instance.run()
print(number_of_calls)
```

```
30
```

120 - 30 = 90

- fitness_func
- on_start
- on_fitness
- on_parents
- on_crossover
- on_mutation
- on_generation
- on_stop

pygad.GA

```python
import pygad
import numpy

def fitness_func(ga_instanse, solution, solution_idx):
    return numpy.random.rand()

def on_start(ga_instanse):
    print("on_start")

def on_fitness(ga_instanse, last_gen_fitness):
    print("on_fitness")

def on_parents(ga_instanse, last_gen_parents):
    print("on_parents")

def on_crossover(ga_instanse, last_gen_offspring):
    print("on_crossover")

def on_mutation(ga_instanse, last_gen_offspring):
    print("on_mutation")

def on_generation(ga_instanse):
    print("on_generation\n")

def on_stop(ga_instanse, last_gen_fitness):
    print("on_stop")

ga_instance = pygad.GA(num_generations=5,
                       num_parents_mating=4,
                       sol_per_pop=10,
                       num_genes=2,
                       on_start=on_start,
                       on_fitness=on_fitness,
                       on_parents=on_parents,
                       on_crossover=on_crossover,
                       on_mutation=on_mutation,
                       on_generation=on_generation,
                       on_stop=on_stop,
                       fitness_func=fitness_func)

ga_instance.run()
```

Test' Test

selfpygad.GA

```
import pygad
import numpy

class Test:
    def fitness_func(self, ga_instanse, solution, solution_idx):
        return numpy.random.rand()

    def on_start(self, ga_instanse):
        print("on_start")

    def on_fitness(self, ga_instanse, last_gen_fitness):
        print("on_fitness")

    def on_parents(self, ga_instanse, last_gen_parents):
        print("on_parents")

    def on_crossover(self, ga_instanse, last_gen_offspring):
        print("on_crossover")

    def on_mutation(self, ga_instanse, last_gen_offspring):
        print("on_mutation")

    def on_generation(self, ga_instanse):
        print("on_generation\n")

    def on_stop(self, ga_instanse, last_gen_fitness):
        print("on_stop")

ga_instance = pygad.GA(num_generations=5,
                       num_parents_mating=4,
                       sol_per_pop=10,
                       num_genes=2,
                       on_start=Test().on_start,
                       on_fitness=Test().on_fitness,
                       on_parents=Test().on_parents,
                       on_crossover=Test().on_crossover,
                       on_mutation=Test().on_mutation,
                       on_generation=Test().on_generation,
                       on_stop=Test().on_stop,
                       fitness_func=Test().fitness_func)

ga_instance.run()
```

pygad.torchga

,

pygad.utils

crossoverCrossover

mutationMutation

parent_selectionParentSelection

nsga2NSGA2().

pygad.GApygad.GA

pygad.utils.crossover

pygad.utils.crossoverCrossover

single_point_crossover()

two_points_crossover()

uniform_crossover()

scattered_crossover()

parents

offspring_size

pygad.utils.mutation

pygad.utils.mutationMutation

random_mutation()

swap_mutation()

inversion_mutation()

scramble_mutation()

adaptive_mutation()

offspring

“” ():

“”

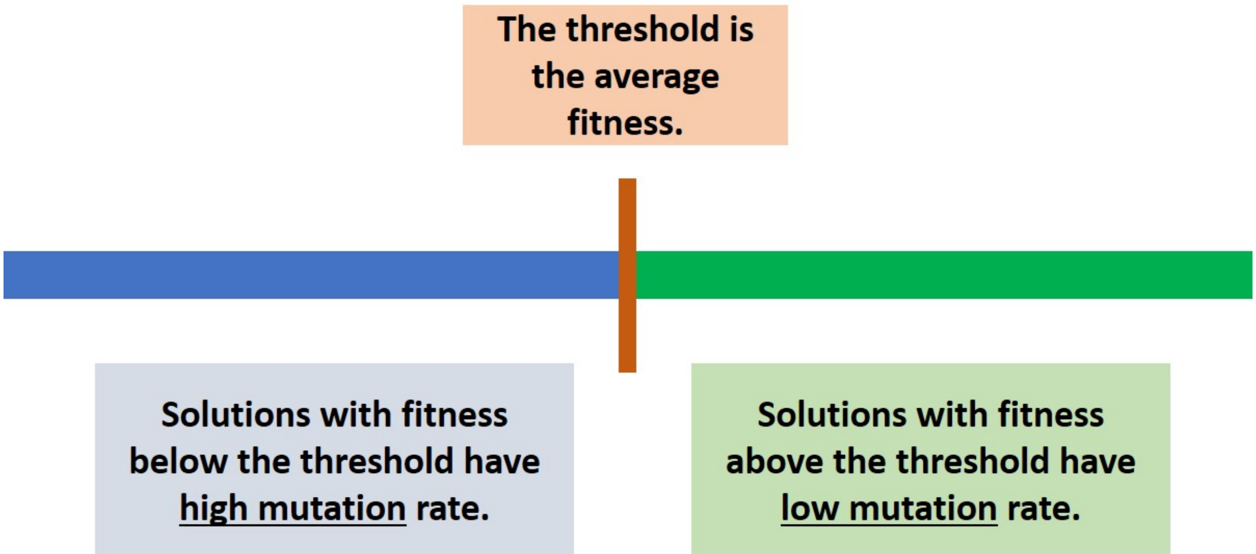
(f_avg).

(f).

f<f_avg

f>f_avg

f=f_avg



pygad.GAmutation_type="adaptive"

mutation_probabilitymutation_num_genesmutation_percent_genes

list

tuple

numpy.ndarray

listtuplenumpy.ndarray

```
# mutation_probability
mutation_probability = [0.25, 0.1]
mutation_probability = (0.35, 0.17)
mutation_probability = numpy.array([0.15, 0.05])

# mutation_num_genes
mutation_num_genes = [4, 2]
mutation_num_genes = (3, 1)
mutation_num_genes = numpy.array([7, 2])

# mutation_percent_genes
mutation_percent_genes = [25, 12]
```

()

()

```
mutation_percent_genes = (15, 8)
mutation_percent_genes = numpy.array([21, 13])
```

```
mutation_probability = [0.25, 0.1]
```

```
import pygad
import numpy

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

def fitness_func(ga_instance, solution, solution_idx):
    # The fitness function calulates the sum of products between each input and its␣
↪corresponding weight.
    output = numpy.sum(solution*function_inputs)
    # The value 0.000001 is used to avoid the Inf value when the denominator numpy.
↪abs(output - desired_output) is 0.0.
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

# Creating an instance of the GA class inside the ga module. Some parameters are␣
↪initialized within the constructor.
ga_instance = pygad.GA(num_generations=200,
                       fitness_func=fitness_func,
                       num_parents_mating=10,
                       sol_per_pop=20,
                       num_genes=len(function_inputs),
                       mutation_type="adaptive",
                       mutation_num_genes=(3, 1))

# Running the GA to optimize the parameters of the function.
ga_instance.run()

ga_instance.plot_fitness(title="PyGAD with Adaptive Mutation", linewidth=5)
```

pygad.utils.parent_selection

pygad.utils.parent_selectionParentSelection

steady_state_selection()

roulette_wheel_selection()

stochastic_universal_selection()

rank_selection()

random_selection()

tournament_selection()

nsga2_selection()

tournament_nsga2_selection()

fitness

num_parents

pygad.utils.nsga2

pygad.utils.nsga2NSGA2

non_dominated_sorting()

get_non_dominated_set()

crowding_distance()

sort_solutions_nsga2()

pygad.GA'

crossover_type

mutation_type

parent_selection_type

```
import pygad
import numpy

equation_inputs = [4,-2,3.5]
desired_output = 44

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func)

ga_instance.run()
ga_instance.plot_fitness()
```

pygad.GA

().

pygad.GApopulationgene_typegene_space

```
def crossover_func(parents, offspring_size, ga_instance):
    offspring = ...
    ...
    return numpy.array(offspring)
```

(),

```
def crossover_func(parents, offspring_size, ga_instance):
    offspring = []
    idx = 0
    while len(offspring) != offspring_size[0]:
        parent1 = parents[idx % parents.shape[0], :].copy()
        parent2 = parents[(idx + 1) % parents.shape[0], :].copy()

        random_split_point = numpy.random.choice(range(offspring_size[1]))

        parent1[random_split_point:] = parent2[random_split_point:]

        offspring.append(parent1)

        idx += 1

    return numpy.array(offspring)
```

crossover_typepygad.GA

```
ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       crossover_type=crossover_func)
```

pygad.GApopulationgene_typegene_space

```python
def mutation_func(offspring, ga_instance):
    ...
    return offspring
```

,

```python
def mutation_func(offspring, ga_instance):

    for chromosome_idx in range(offspring.shape[0]):
        random_gene_idx = numpy.random.choice(range(offspring.shape[1]))

        offspring[chromosome_idx, random_gene_idx] += numpy.random.random()

    return offspring
```

mutation_type

```python
ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       crossover_type=crossover_func,
                       mutation_type=mutation_func)
```

() gene_type

gene_space

mutation_percent_genesmutation_probabilitymutation_num_genes

mutation_by_replacement

random_mutation_min_valrandom_mutation_max_val

allow_duplicate_genes

pygad.GApopulationgene_typegene_space

(num_genes).

numpy.ndarray

```
def parent_selection_func(fitness, num_parents, ga_instance):
    ...
    return parents, fitness_sorted[:num_parents]
```

num_parents

```
def parent_selection_func(fitness, num_parents, ga_instance):

    fitness_sorted = sorted(range(len(fitness)), key=lambda k: fitness[k])
    fitness_sorted.reverse()

    parents = numpy.empty((num_parents, ga_instance.population.shape[1]))

    for parent_num in range(num_parents):
        parents[parent_num, :] = ga_instance.population[fitness_sorted[parent_num],␣
↪:].copy()

    return parents, numpy.array(fitness_sorted[:num_parents])
```

parent_selection_type

```
ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       crossover_type=crossover_func,
                       mutation_type=mutation_func,
                       parent_selection_type=parent_selection_func)
```

```
import pygad
import numpy

equation_inputs = [4,-2,3.5]
desired_output = 44

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)

    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)

    return fitness

def parent_selection_func(fitness, num_parents, ga_instance):

    fitness_sorted = sorted(range(len(fitness)), key=lambda k: fitness[k])
    fitness_sorted.reverse()

    parents = numpy.empty((num_parents, ga_instance.population.shape[1]))

    for parent_num in range(num_parents):
        parents[parent_num, :] = ga_instance.population[fitness_sorted[parent_num],␣
```

()

()

```
↪:].copy()

    return parents, numpy.array(fitness_sorted[:num_parents])

def crossover_func(parents, offspring_size, ga_instance):

    offspring = []
    idx = 0
    while len(offspring) != offspring_size[0]:
        parent1 = parents[idx % parents.shape[0], :].copy()
        parent2 = parents[(idx + 1) % parents.shape[0], :].copy()

        random_split_point = numpy.random.choice(range(offspring_size[1]))

        parent1[random_split_point:] = parent2[random_split_point:]

        offspring.append(parent1)

        idx += 1

    return numpy.array(offspring)

def mutation_func(offspring, ga_instance):

    for chromosome_idx in range(offspring.shape[0]):
        random_gene_idx = numpy.random.choice(range(offspring.shape[0]))

        offspring[chromosome_idx, random_gene_idx] += numpy.random.random()

    return offspring

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       crossover_type=crossover_func,
                       mutation_type=mutation_func,
                       parent_selection_type=parent_selection_func)

ga_instance.run()
ga_instance.plot_fitness()
```

```
import pygad
import numpy

equation_inputs = [4,-2,3.5]
desired_output = 44

class Test:
    def fitness_func(self, ga_instance, solution, solution_idx):
        output = numpy.sum(solution * equation_inputs)

        fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
```

()

```
        return fitness

    def parent_selection_func(self, fitness, num_parents, ga_instance):

        fitness_sorted = sorted(range(len(fitness)), key=lambda k: fitness[k])
        fitness_sorted.reverse()

        parents = numpy.empty((num_parents, ga_instance.population.shape[1]))

        for parent_num in range(num_parents):
            parents[parent_num, :] = ga_instance.population[fitness_sorted[parent_
↪num], :].copy()

        return parents, numpy.array(fitness_sorted[:num_parents])

    def crossover_func(self, parents, offspring_size, ga_instance):

        offspring = []
        idx = 0
        while len(offspring) != offspring_size[0]:
            parent1 = parents[idx % parents.shape[0], :].copy()
            parent2 = parents[(idx + 1) % parents.shape[0], :].copy()

            random_split_point = numpy.random.choice(range(offspring_size[0]))

            parent1[random_split_point:] = parent2[random_split_point:]

            offspring.append(parent1)

            idx += 1

        return numpy.array(offspring)

    def mutation_func(self, offspring, ga_instance):

        for chromosome_idx in range(offspring.shape[0]):
            random_gene_idx = numpy.random.choice(range(offspring.shape[1]))

            offspring[chromosome_idx, random_gene_idx] += numpy.random.random()

        return offspring

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=5,
                       num_parents_mating=2,
                       num_genes=len(equation_inputs),
                       fitness_func=Test().fitness_func,
                       parent_selection_type=Test().parent_selection_func,
                       crossover_type=Test().crossover_func,
                       mutation_type=Test().mutation_func)

ga_instance.run()
ga_instance.plot_fitness()
```

pygad.visualize

,

    plot_fitness()

    plot_genes()

    plot_new_solution_rate()

save_solutionsTruesolutions

```
import pygad
import numpy

equation_inputs = [4, -2, 3.5, 8, -2, 3.5, 8]
desired_output = 2671.1234

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

ga_instance = pygad.GA(num_generations=10,
                       sol_per_pop=10,
                       num_parents_mating=5,
                       num_genes=len(equation_inputs),
                       fitness_func=fitness_func,
                       gene_space=[range(1, 10), range(10, 20), range(15, 30),
 ↪range(20, 40), range(25, 50), range(10, 30), range(20, 50)],
                       gene_type=int,
                       save_solutions=True)

ga_instance.run()
```

,

plot_fitness()

plot_fitness()()()

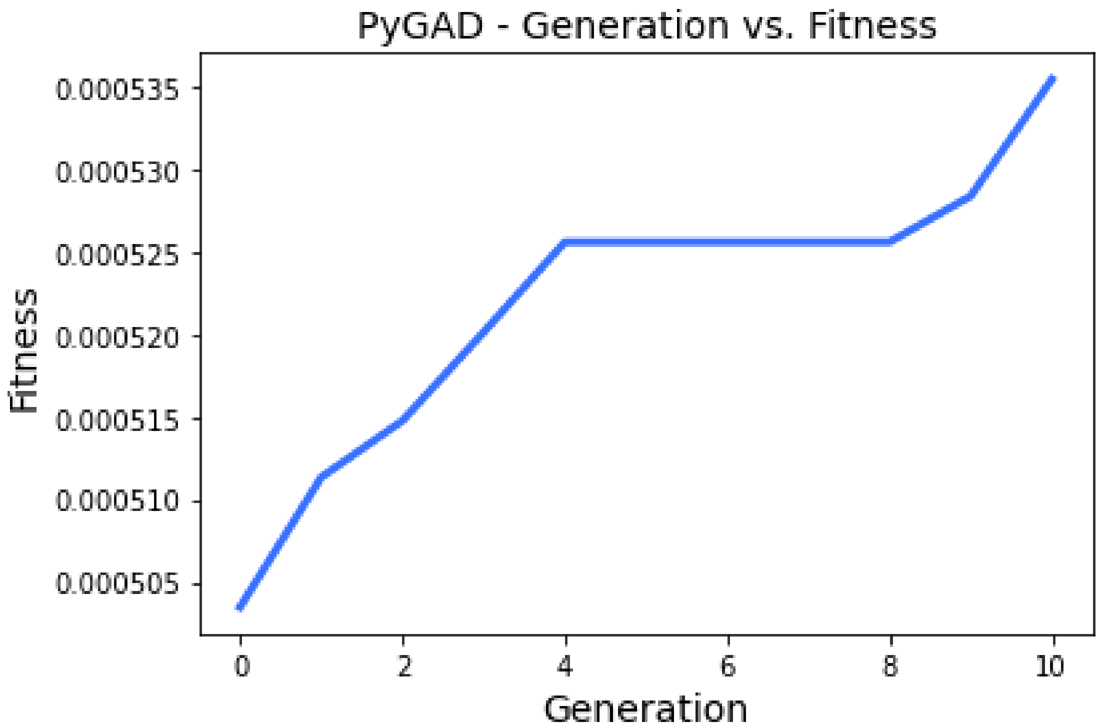
(),

- title
- xlabel
- ylabel
- linewidth3
- font_size14
- plot_type"plot"(), "scatter", "bar"
- color"#64f20c"
- labelNone
- save_dir

plot_type="plot"

plot_type"plot"

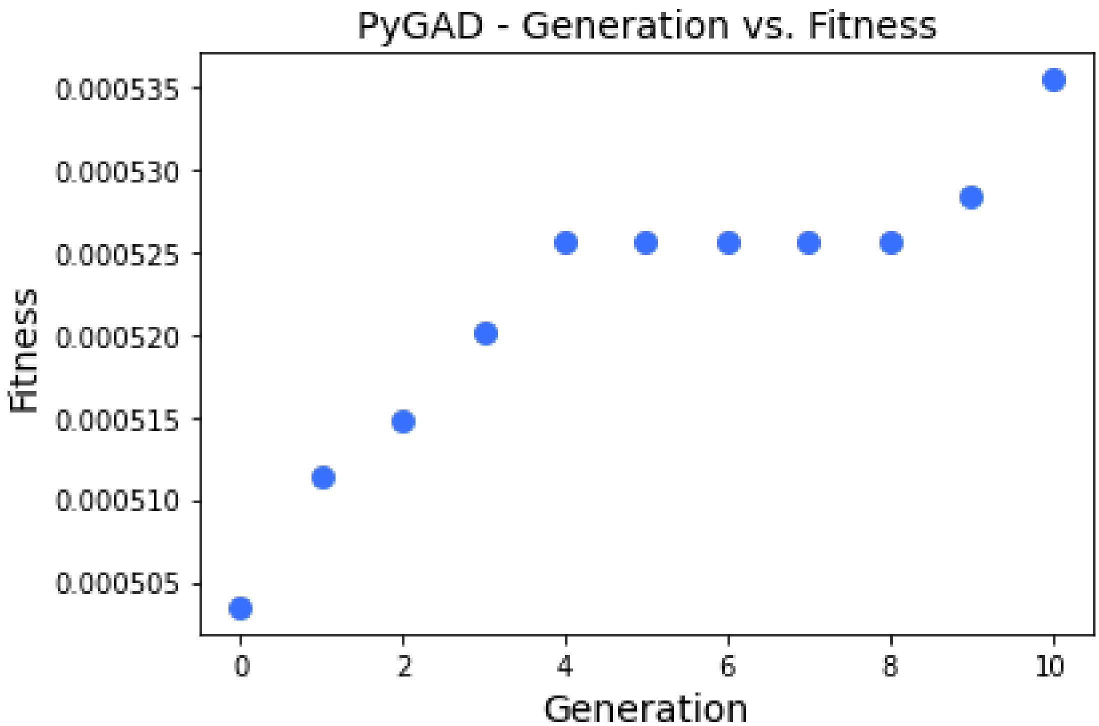
```
ga_instance.plot_fitness()
# ga_instance.plot_fitness(plot_type="plot")
```



plot_type="scatter"

plot_type"scatter"linewidth

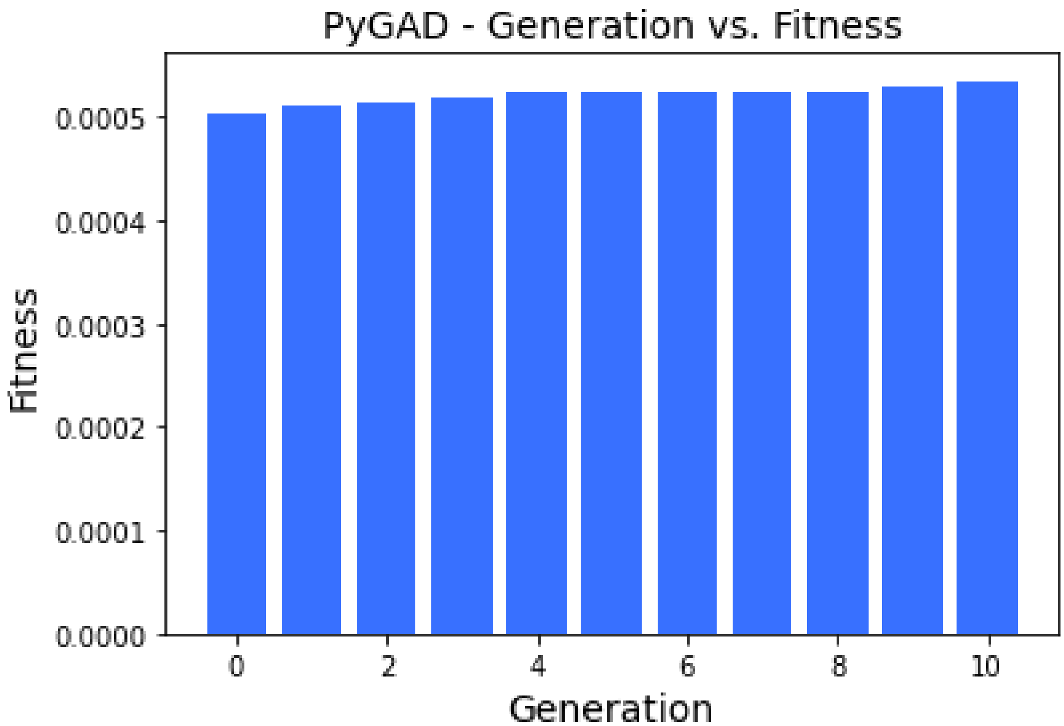
```
ga_instance.plot_fitness(plot_type="scatter")
```



plot_type="bar"

plot_type"bar"

```
ga_instance.plot_fitness(plot_type="bar")
```



plot_new_solution_rate()

plot_new_solution_rate()

(),

plot_new_solution_rate()plot_fitness()(plot_type).

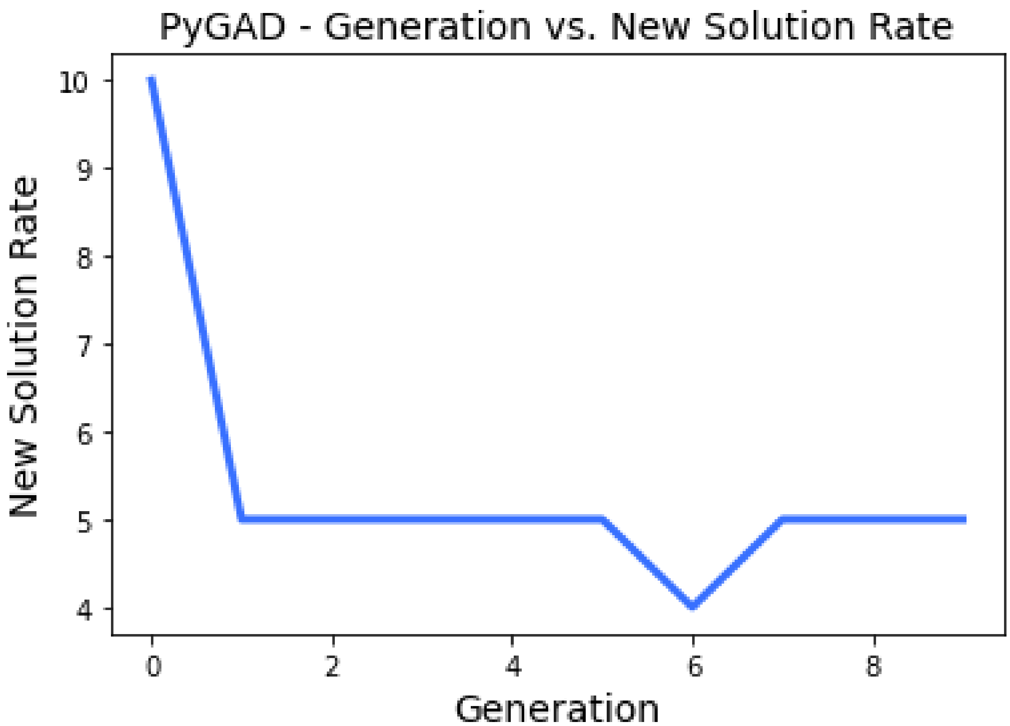
- title
- xlabel
- ylabel
- linewidth3
- font_size14
- plot_type"plot"(), "scatter", "bar"
- color"#3870FF"
- save_dir

plot_type="plot"

plot_type"plot"

```
ga_instance.plot_new_solution_rate()
# ga_instance.plot_new_solution_rate(plot_type="plot")
```

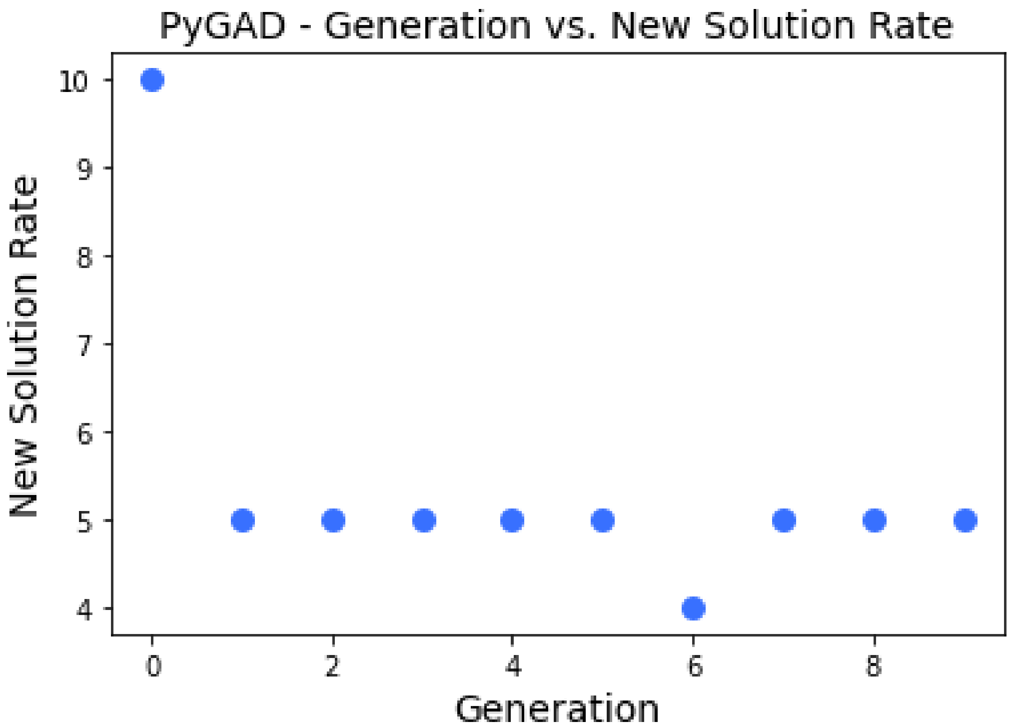
(sol_per_poppygad.GA)



plot_type="scatter"

plot_type="scatter"

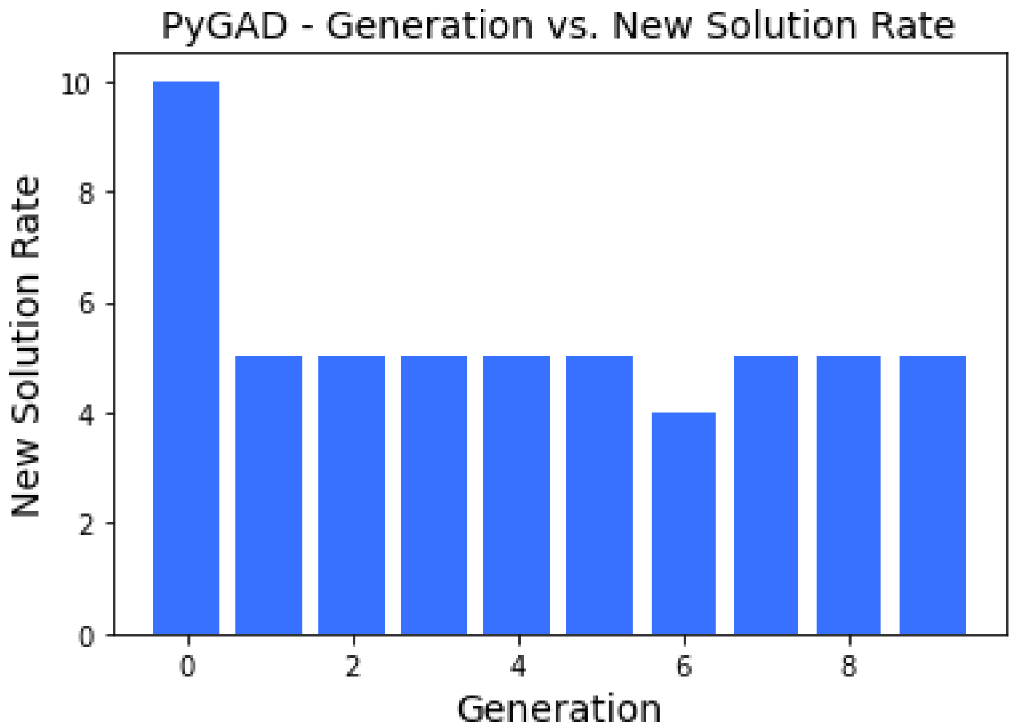
```
ga_instance.plot_new_solution_rate(plot_type="scatter")
```



plot_type="bar"

plot_type="scatter"

```
ga_instance.plot_new_solution_rate(plot_type="bar")
```



plot_genes()

plot_genes()plot_genes()

(),

- title
- xlabel
- ylabel
- linewidth3
- font_size14
- plot_type"plot"(), "scatter", "bar"
- graph_type"plot"(), "boxplot", "histogram"
- fill_color"#3870FF"graph_type="plot"
- color"#3870FF"

solutions"all""best"

save_dir

graph_type="plot""plot"(), "boxplot", "histogram"

plot_type="plot"plot_typeplot_fitness()plot_new_solution_rate()

solutions="all""all"() "best"

graph_type

plot_type"plot"

solutions

solutions="all"save_solutions=Falsepygad.GA

solutions="best"save_best_solutions=Falsepygad.GA

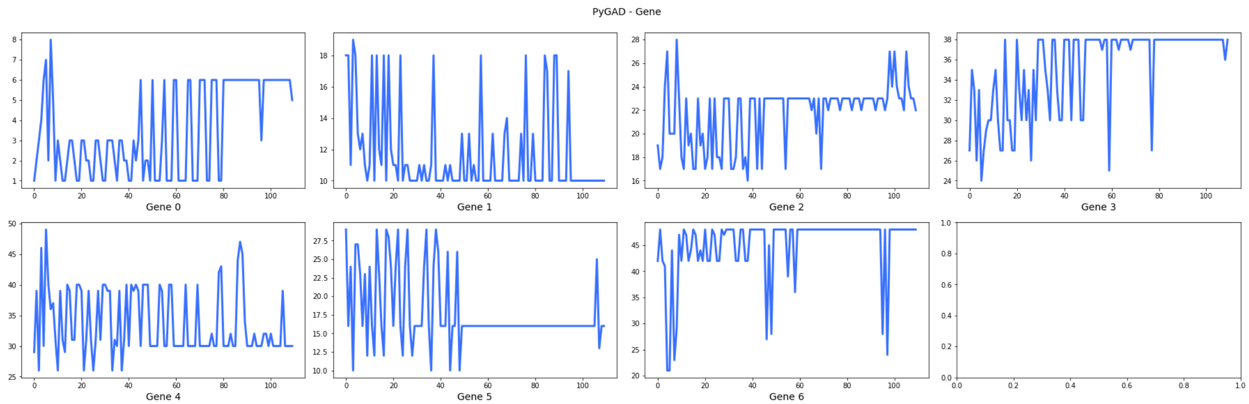
graph_type="plot"

graph_type="plot"

plot_type="plot"

graph_typeplot_type"plot"()

```
ga_instance.plot_genes()

ga_instance.plot_genes(graph_type="plot")

ga_instance.plot_genes(plot_type="plot")

ga_instance.plot_genes(graph_type="plot",
                       plot_type="plot")
```



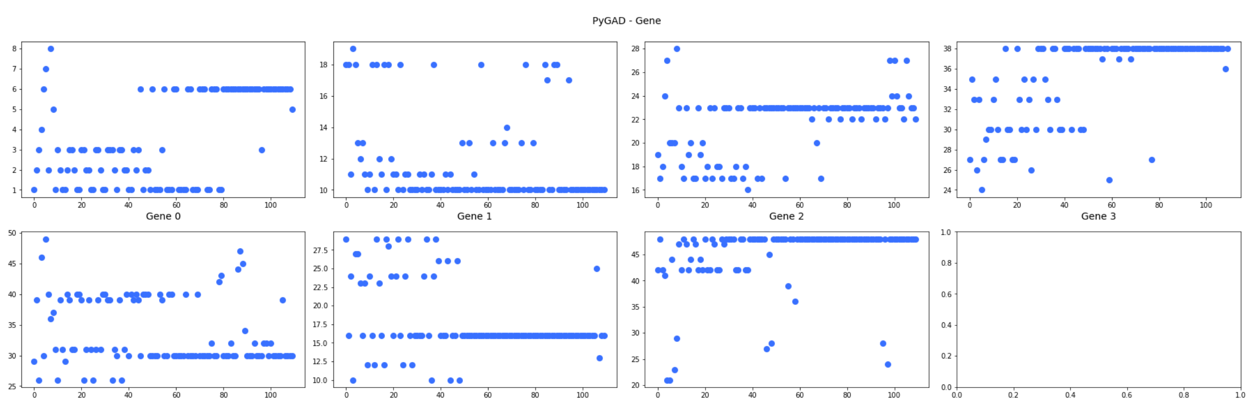
solutions"all"

```
ga_instance.plot_genes(solutions="all")

ga_instance.plot_genes(graph_type="plot",
                       solutions="all")

ga_instance.plot_genes(plot_type="plot",
                       solutions="all")

ga_instance.plot_genes(graph_type="plot",
                       plot_type="plot",
                       solutions="all")
```

plot_type="scatter"

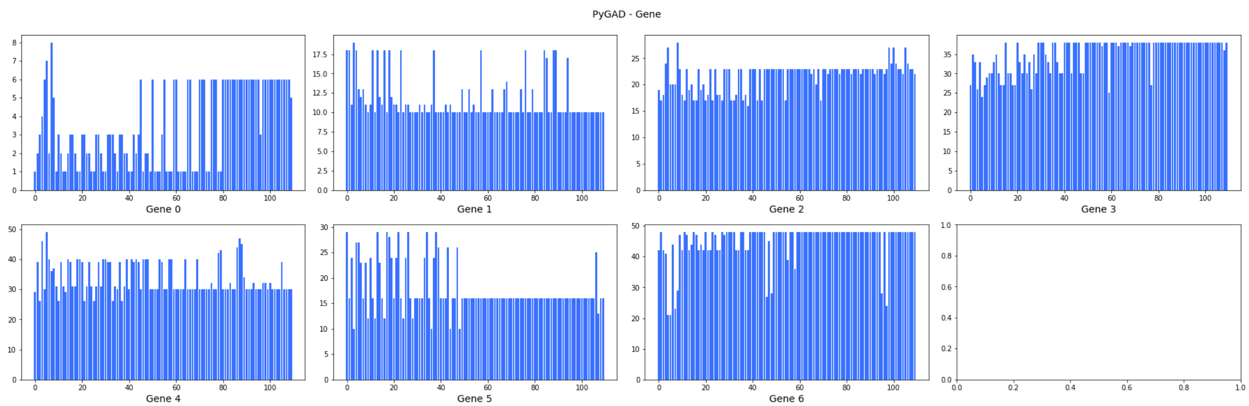
plot_genes()

```
ga_instance.plot_genes(plot_type="scatter")

ga_instance.plot_genes(graph_type="plot",
                       plot_type="scatter",
                       solutions='all')
```



plot_type="bar"

```
ga_instance.plot_genes(plot_type="bar")

ga_instance.plot_genes(graph_type="plot",
                       plot_type="bar",
                       solutions='all')
```

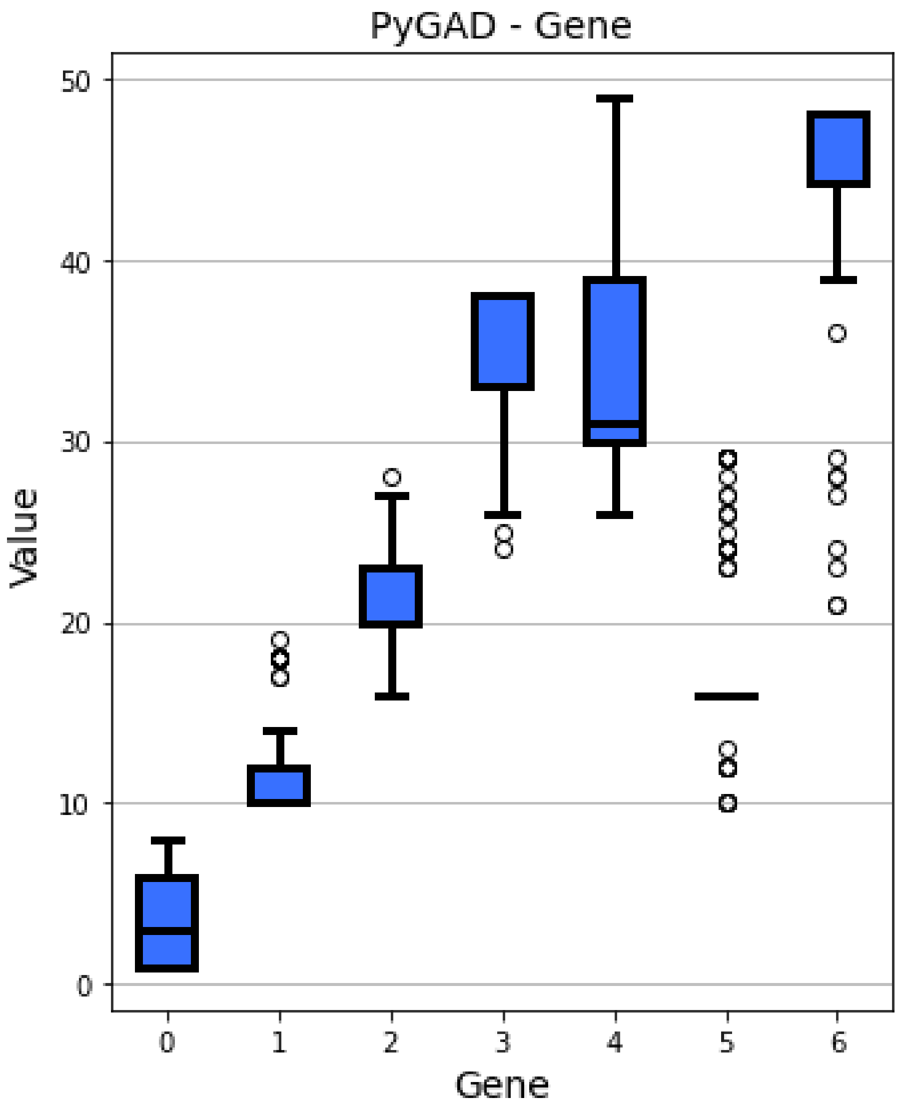


graph_type="boxplot"

graph_type"boxplot"plot_type

plot_genes()solutions"all"

```
ga_instance.plot_genes(graph_type="boxplot")

ga_instance.plot_genes(graph_type="boxplot",
                       solutions='all')
```

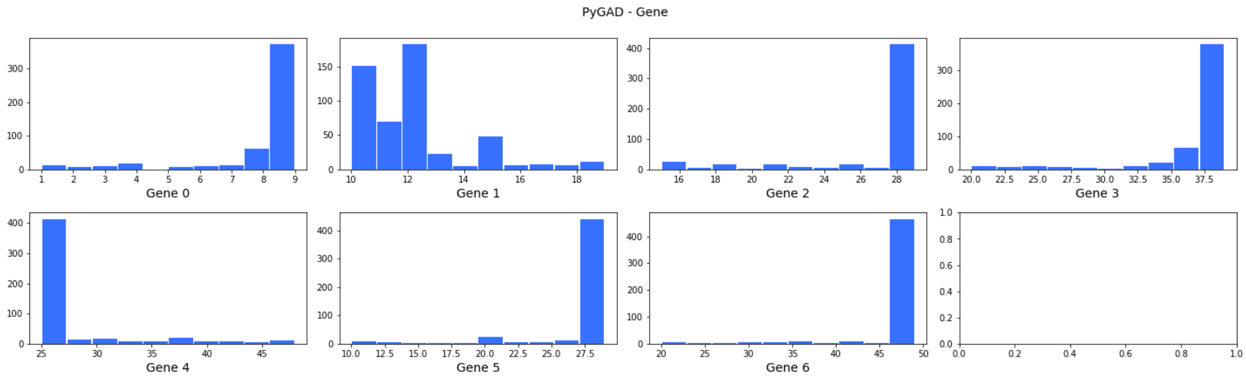


graph_type="histogram"

graph_type="boxplot"graph_type="boxplot"plot_type

plot_genes()solutions"all"

```
ga_instance.plot_genes(graph_type="histogram")

ga_instance.plot_genes(graph_type="histogram",
                       solutions='all')
```




```
solutions="best"
```

pygad.helper

,

uniqueUnique

- solve_duplicate_genes_randomly()
- solve_duplicate_genes_by_space()
- unique_int_gene_from_range()
- unique_genes_by_space()unique_gene_by_space()
- unique_gene_by_space()

pygad.nn

,

problem_typepygad.nn.train()pygad.nn.predict()

- pygad.nn.InputLayer
- (): pygad.nn.DenseLayer

pygad.nn.InputLayer

pygad.nn.InputLayer

num_neurons

num_neurons

```
input_layer = pygad.nn.InputLayer(num_neurons=20)
```

num_neuronspygad.nn.InputLayer

```
num_input_neurons = input_layer.num_neurons

print("Number of input neurons =", num_input_neurons)
```

pygad.nn.DenseLayer

pygad.nn.DenseLayer()

num_neurons

previous_layerprevious_layer

activation_function"sigmoid""sigmoid""relu""softmax"(), "None"(). "None"() "None"

initial_weights

trained_weightsinitial_weights

previous_layerinput_layer

```
dense_layer = pygad.nn.DenseLayer(num_neurons=12,
                                  previous_layer=input_layer,
                                  activation_function="relu")
```

```
num_dense_neurons = dense_layer.num_neurons
dense_initail_weights = dense_layer.initial_weights

print("Number of dense layer attributes =", num_dense_neurons)
print("Initial weights of the dense layer :", dense_initail_weights)
```

dense_layer

```
input_layer = dense_layer.previous_layer
num_input_neurons = input_layer.num_neurons

print("Number of input neurons =", num_input_neurons)
```

'previous_layer

```
dense_layer2 = pygad.nn.DenseLayer(num_neurons=5,
                                   previous_layer=dense_layer,
                                   activation_function="relu")
```

dense_layer2dense_layerprevious_layerdense_layer

```
dense_layer = dense_layer2.previous_layer
dense_layer_neurons = dense_layer.num_neurons

print("Number of dense neurons =", num_input_neurons)
```

dense_layer

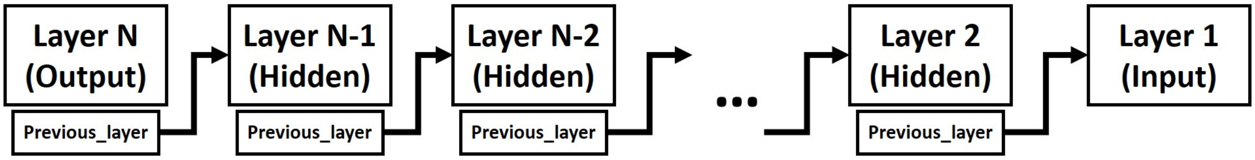
```
dense_layer = dense_layer2.previous_layer
input_layer = dense_layer.previous_layer
num_input_neurons = input_layer.num_neurons

print("Number of input neurons =", num_input_neurons)
```

dense_layer2

previous_layer

previous_layerpygad.nn.DenseLayer

() ().



previous_layerprevious_layerprevious_layer().

(), whilewhileprevious_layer

```
layer = dense_layer2

while "previous_layer" in layer.__init__.__code__.co_varnames:
    print("Number of neurons =", layer.num_neurons)

    # Go to the previous layer.
    layer = layer.previous_layer
```

pygad.nn

pygad.nn.layers_weights()

- last_layer()
- initialTrue(), ' initial_weights False' trained_weights

whileprevious_layerinitialTrueFalse

pygad.nn.layers_weights_as_vector()

layers_weights()

- last_layer()
- initialTrue(), ' initial_weights False' trained_weights

whileprevious_layerinitialTrueFalse

pygad.nn.layers_weights_as_matrix()

layers_weights_as_vectors()

last_layer()

vector_weights

whileprevious_layer

pygad.nn.layers_activations()

last_layer()

whileprevious_layer' activation_function

pygad.nn.sigmoid()

sop

pygad.nn.relu()

()

sop

pygad.nn.softmax()

sop

pygad.nn.train()

num_epochs

last_layer()

data_inputs

data_outputs

problem_type"classification""regression"

learning_rate

pygad.nn.update_weights()

weights

network_error

learning_rate

pygad.nn.update_layers_trained_weights()

trained_weights(final_weights)

() trained_weights

last_layer()

final_weights

whileprevious_layertrained_weightsfinal_weights

pygad.nn.predict()

,

last_layer()

data_inputs

problem_type"classification""regression" pygad.nn

pygad.nn.to_vector()

() array

array

pygad.nn.to_array()

vector

vector

shape

pygad.nn.sigmoid()

(): pygad.nn.relu()

pygad.nn.softmax()

pygad.nn

,

,

,

,

,

```
import numpy
import skimage.io, skimage.color, skimage.feature
import os

fruits = ["apple", "raspberry", "mango", "lemon"]
# Number of samples in the datset used = 492+490+490+490=1,962
# 360 is the length of the feature vector.
dataset_features = numpy.zeros(shape=(1962, 360))
outputs = numpy.zeros(shape=(1962))

idx = 0
class_label = 0
for fruit_dir in fruits:
    curr_dir = os.path.join(os.path.sep, fruit_dir)
    all_imgs = os.listdir(os.getcwd()+curr_dir)
    for img_file in all_imgs:
        if img_file.endswith(".jpg"): # Ensures reading only JPG files.
            fruit_data = skimage.io.imread(fname=os.path.sep.join([os.getcwd(), curr_
↪dir, img_file]), as_gray=False)
            fruit_data_hsv = skimage.color.rgb2hsv(rgb=fruit_data)
            hist = numpy.histogram(a=fruit_data_hsv[:, :, 0], bins=360)
            dataset_features[idx, :] = hist[0]
            outputs[idx] = class_label
            idx = idx + 1
    class_label = class_label + 1

# Saving the extracted features and the outputs as NumPy files.
numpy.save("dataset_features.npy", dataset_features)
numpy.save("outputs.npy", outputs)
```

```
data_inputs = numpy.load("dataset_features.npy")
data_outputs = numpy.load("outputs.npy")
```

pygad.nn.InputLayer

```
import pygad.nn
num_inputs = data_inputs.shape[1]

input_layer = pygad.nn.InputLayer(num_inputs)
```

```
hidden_layer = pygad.nn.DenseLayer(num_neurons=HL2_neurons, previous_layer=input_
↪layer, activation_function="relu")
output_layer = pygad.nn.DenseLayer(num_neurons=4, previous_layer=hidden_layer2,␣
↪activation_function="softmax")
```

pygad.nn.train()

```
pygad.nn.train(num_epochs=10,
               last_layer=output_layer,
               data_inputs=data_inputs,
               data_outputs=data_outputs,
               learning_rate=0.01)
```

pygad.nn.predict()

```
predictions = pygad.nn.predict(last_layer=output_layer, data_inputs=data_inputs)
```

```
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

pygad.gann

pygad.nn

(),

```
import numpy
import pygad.nn

# Preparing the NumPy array of the inputs.
data_inputs = numpy.array([[1, 1],
                           [1, 0],
                           [0, 1],
                           [0, 0]])

# Preparing the NumPy array of the outputs.
data_outputs = numpy.array([0,
                            1,
                            1,
                            0])

# The number of inputs (i.e. feature vector length) per sample
num_inputs = data_inputs.shape[1]
# Number of outputs per sample
num_outputs = 2

HL1_neurons = 2

# Building the network architecture.
input_layer = pygad.nn.InputLayer(num_inputs)
hidden_layer1 = pygad.nn.DenseLayer(num_neurons=HL1_neurons, previous_layer=input_
↪layer, activation_function="relu")
output_layer = pygad.nn.DenseLayer(num_neurons=num_outputs, previous_layer=hidden_
↪layer1, activation_function="softmax")

# Training the network.
pygad.nn.train(num_epochs=10,
               last_layer=output_layer,
               data_inputs=data_inputs,
```

()

()

```
                data_outputs=data_outputs,
                learning_rate=0.01)

# Using the trained network for predictions.
predictions = pygad.nn.predict(last_layer=output_layer, data_inputs=data_inputs)

# Calculating some statistics
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

```
import numpy
import pygad.nn

# Reading the data features. Check the 'extract_features.py' script for extracting␣
↪the features & preparing the outputs of the dataset.
data_inputs = numpy.load("dataset_features.npy") # Download from https://github.com/
↪ahmedfgad/NumPyANN/blob/master/dataset_features.npy

# Optional step for filtering the features using the standard deviation.
features_STDs = numpy.std(a=data_inputs, axis=0)
data_inputs = data_inputs[:, features_STDs > 50]

# Reading the data outputs. Check the 'extract_features.py' script for extracting the␣
↪features & preparing the outputs of the dataset.
data_outputs = numpy.load("outputs.npy") # Download from https://github.com/ahmedfgad/
↪NumPyANN/blob/master/outputs.npy

# The number of inputs (i.e. feature vector length) per sample
num_inputs = data_inputs.shape[1]
# Number of outputs per sample
num_outputs = 4

HL1_neurons = 150
HL2_neurons = 60

# Building the network architecture.
input_layer = pygad.nn.InputLayer(num_inputs)
hidden_layer1 = pygad.nn.DenseLayer(num_neurons=HL1_neurons, previous_layer=input_
↪layer, activation_function="relu")
hidden_layer2 = pygad.nn.DenseLayer(num_neurons=HL2_neurons, previous_layer=hidden_
↪layer1, activation_function="relu")
output_layer = pygad.nn.DenseLayer(num_neurons=num_outputs, previous_layer=hidden_
↪layer2, activation_function="softmax")

# Training the network.
```

()

()

```
pygad.nn.train(num_epochs=10,
               last_layer=output_layer,
               data_inputs=data_inputs,
               data_outputs=data_outputs,
               learning_rate=0.01)

# Using the trained network for predictions.
predictions = pygad.nn.predict(last_layer=output_layer, data_inputs=data_inputs)

# Calculating some statistics
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

problem_typepygad.nn.train()pygad.nn.predict()"regression"

```
pygad.nn.train(...,
               problem_type="regression")

predictions = pygad.nn.predict(...,
                               problem_type="regression")
```

"None"

```
output_layer = pygad.nn.DenseLayer(num_neurons=num_outputs, previous_layer=hidden_
↪layer1, activation_function="None")
```

```
abs_error = numpy.mean(numpy.abs(predictions - data_outputs))
print(f"Absolute error : {abs_error}.")
```

pygad.gann

```
import numpy
import pygad.nn

# Preparing the NumPy array of the inputs.
data_inputs = numpy.array([[2, 5, -3, 0.1],
                           [8, 15, 20, 13]])

# Preparing the NumPy array of the outputs.
data_outputs = numpy.array([0.1,
                            1.5])

# The number of inputs (i.e. feature vector length) per sample
num_inputs = data_inputs.shape[1]
# Number of outputs per sample
```

()

()

```
num_outputs = 1

HL1_neurons = 2

# Building the network architecture.
input_layer = pygad.nn.InputLayer(num_inputs)
hidden_layer1 = pygad.nn.DenseLayer(num_neurons=HL1_neurons, previous_layer=input_
↪layer, activation_function="relu")
output_layer = pygad.nn.DenseLayer(num_neurons=num_outputs, previous_layer=hidden_
↪layer1, activation_function="None")

# Training the network.
pygad.nn.train(num_epochs=100,
               last_layer=output_layer,
               data_inputs=data_inputs,
               data_outputs=data_outputs,
               learning_rate=0.01,
               problem_type="regression")

# Using the trained network for predictions.
predictions = pygad.nn.predict(last_layer=output_layer,
                         data_inputs=data_inputs,
                         problem_type="regression")

# Calculating some statistics
abs_error = numpy.mean(numpy.abs(predictions - data_outputs))
print(f"Absolute error : {abs_error}.")
```

(https://www.kaggle.com/aungpyaeap/fish-market). (https://www.kaggle.com/aungpyaeap/fish-market/download).

`read_csv()`

```
data = numpy.array(pandas.read_csv("Fish.csv"))
```

```
# Preparing the NumPy array of the inputs.
data_inputs = numpy.asarray(data[:, 2:], dtype=numpy.float32)

# Preparing the NumPy array of the outputs.
data_outputs = numpy.asarray(data[:, 1], dtype=numpy.float32) # Fish Weight
```

`"None"problem_typepygad.nn.train()pygad.nn.predict()"regression"`

`pygad.nn.train()`

```
abs_error = numpy.mean(numpy.abs(predictions - data_outputs))
print(f"Absolute error : {abs_error}.")
```

```python
import numpy
import pygad.nn
import pandas

data = numpy.array(pandas.read_csv("Fish.csv"))

# Preparing the NumPy array of the inputs.
data_inputs = numpy.asarray(data[:, 2:], dtype=numpy.float32)

# Preparing the NumPy array of the outputs.
data_outputs = numpy.asarray(data[:, 1], dtype=numpy.float32) # Fish Weight

# The number of inputs (i.e. feature vector length) per sample
num_inputs = data_inputs.shape[1]
# Number of outputs per sample
num_outputs = 1

HL1_neurons = 2

# Building the network architecture.
input_layer = pygad.nn.InputLayer(num_inputs)
hidden_layer1 = pygad.nn.DenseLayer(num_neurons=HL1_neurons, previous_layer=input_
↪layer, activation_function="relu")
output_layer = pygad.nn.DenseLayer(num_neurons=num_outputs, previous_layer=hidden_
↪layer1, activation_function="None")

# Training the network.
pygad.nn.train(num_epochs=100,
               last_layer=output_layer,
               data_inputs=data_inputs,
               data_outputs=data_outputs,
               learning_rate=0.01,
               problem_type="regression")

# Using the trained network for predictions.
predictions = pygad.nn.predict(last_layer=output_layer,
                               data_inputs=data_inputs,
                               problem_type="regression")

# Calculating some statistics
abs_error = numpy.mean(numpy.abs(predictions - data_outputs))
print(f"Absolute error : {abs_error}.")
```

pygad.gann

,

pygad.gann() pygadpygad.nn

pygad.gann.GANN

pygad.gannpygad.gann.GANN

__init__()

pygad.gann.GANN

pygad.gann.GANN

- num_solutions()
- num_neurons_input
- num_neurons_output
- num_neurons_hidden_layers=[](). []intintnum_neurons_hidden_layers=[10] num_neurons_hidden_layers=[10, 5]
- output_activation="softmax""softmax"
- hidden_activations="relu"() () (). "relu"num_neurons_hidden_layershidden_activationsnum_neurons_hidden_layershidden_activations

pygad.gann.GANNpygad.gann.validate_network_parameters()

pygad.gann.GANNpygad.gann.GANN

parameters_validatedTrueFalse

population_networks()

pygad.gann.GANN

create_population()

create_population()(). pygad.gann.create_network()

() pygad.gann.GANN

population_networks

update_population_trained_weights()

update_population_trained_weights()trained_weights() population_trained_weights

population_trained_weightstrained_weights

pygad.gann

pygad.gann

pygad.gann.validate_network_parameters()

pygad.gann.GANN

pygad.gann.GANN

num_solutionsNoneNone

hidden_activations(num_neurons_hidden_layers).

() () ().

pygad.gann.create_network()

() ()

parameters_validatedpygad.gann.GANNnum_solutionscreate_network()

parameters_validatedFalsevalidate_network_parameters()

pygad.gann.population_as_vectors()

()

(), ().

population_networks()

().

pygad.gann.population_as_matrices()

()

(),

population_networks()

population_vectors

().

pygad.gann.GANN

pygad.GA

pygad.GA

,

()

() (200, 50) num_inputs

```
data_inputs = numpy.array([[1, 1],
                           [1, 0],
                           [0, 1],
                           [0, 0]])

data_outputs = numpy.array([0,
                            1,
                            1,
                            0])

num_inputs = data_inputs.shape[1]
```

0 (200) 0 N-1 N

num_classes

pygad.gann.GANN

pygad.gann.GANN

num_solutions().

().

```
import pygad.gann
import pygad.nn

num_solutions = 6
GANN_instance = pygad.gann.GANN(num_solutions=num_solutions,
                                num_neurons_input=num_inputs,
                                num_neurons_hidden_layers=[2],
                                num_neurons_output=2,
                                hidden_activations=["relu"],
                                output_activation="softmax")
```

()

().

(number inputs x number of hidden neurons) = (2x2)

(number of hidden neurons x number of outputs) = (2x2) softmaxrelu

pygad.gann.GANN

()

pygad.gann.population_as_vectors()

```
population_vectors = pygad.gann.population_as_vectors(population_networks=GANN_
↪instance.population_networks)
```

pygad.nn.predict()' pygad.nn.predict()trained_weights

```
def fitness_func(ga_instance, solution, sol_idx):
    global GANN_instance, data_inputs, data_outputs

    predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[sol_
↪idx],
                                   data_inputs=data_inputs)
    correct_predictions = numpy.where(predictions == data_outputs)[0].size
    solution_fitness = (correct_predictions/data_outputs.size)*100

    return solution_fitness
```

pygad.nn.predict()' trained_weights

pygad.GAon_generationpygad.GA

trained_weights

trained_weights

pygad.gann.population_as_matrices()

update_population_trained_weights()pygad.ganntrained_weights

```python
def callback_generation(ga_instance):
    global GANN_instance

    population_matrices = pygad.gann.population_as_matrices(population_networks=GANN_
↪instance.population_networks, population_vectors=ga_instance.population)
    GANN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)

    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")
```

pygad.GA

pygad.GA

pygad.GA

```python
initial_population = population_vectors.copy()

num_parents_mating = 4

num_generations = 500

mutation_percent_genes = 5

parent_selection_type = "sss"

crossover_type = "single_point"

mutation_type = "random"

keep_parents = 1

init_range_low = -2
init_range_high = 5

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       mutation_percent_genes=mutation_percent_genes,
                       init_range_low=init_range_low,
                       init_range_high=init_range_high,
                       parent_selection_type=parent_selection_type,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       keep_parents=keep_parents,
                       on_generation=callback_generation)
```

run()

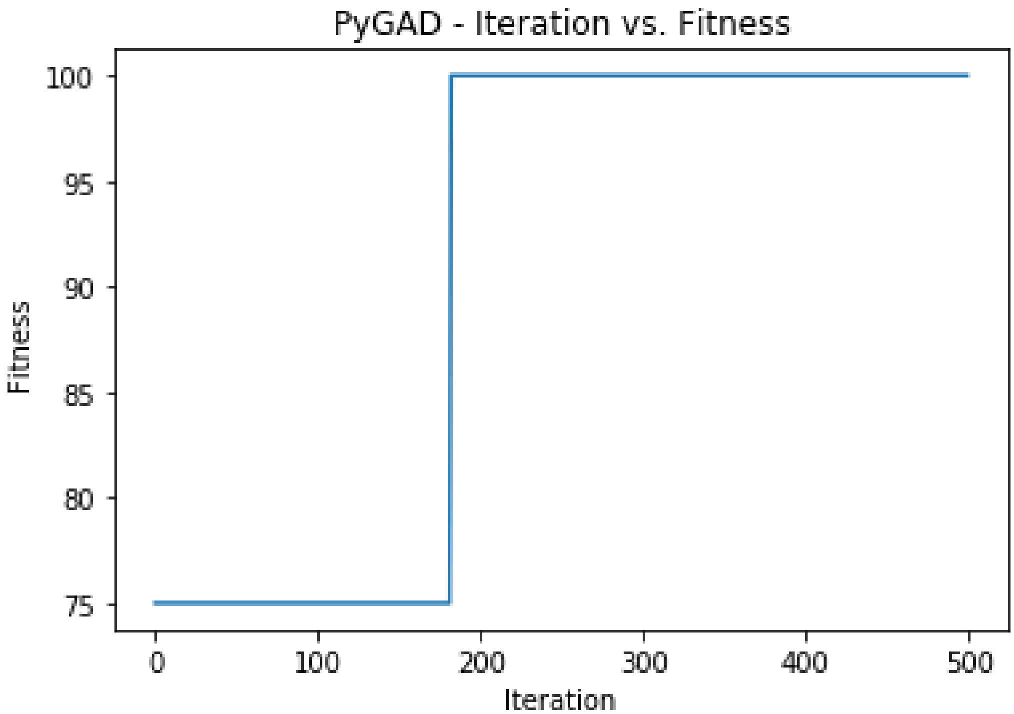
pygad.GA

run()pygad.GAnum_generations

```
ga_instance.run()
```

run()plot_fitness()()

```
ga_instance.plot_fitness()
```



best_solution()pygad.GA

()

```
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")
```

```
Parameters of the best solution : [3.55081391 -3.21562011 -14.2617784 0.68044231 -1.
↪41258145 -3.2979315 1.58136006 -7.83726169]
Fitness value of the best solution = 100.0
Index of the best solution : 0
```

best_solution_generationpygad.GA

```
if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")
```

```
Best solution reached after 182 generations.
```

pygad.nn.predict()

```
predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[solution_
↪idx], data_inputs=data_inputs)
print(f"Predictions of the trained network : {predictions}")
```

```
Predictions of the trained network : [0. 1. 1. 0.]
```

```
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

```
Number of correct classifications : 4
print("Number of wrong classifications : 0
Classification accuracy : 100
```

```
import numpy
import pygad
import pygad.nn
import pygad.gann

def fitness_func(ga_instance, solution, sol_idx):
    global GANN_instance, data_inputs, data_outputs

    # If adaptive mutation is used, sometimes sol_idx is None.
    if sol_idx == None:
        sol_idx = 1

    predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[sol_
↪idx],
                                   data_inputs=data_inputs)
    correct_predictions = numpy.where(predictions == data_outputs)[0].size
    solution_fitness = (correct_predictions/data_outputs.size)*100

    return solution_fitness

def callback_generation(ga_instance):
    global GANN_instance, last_fitness

    population_matrices = pygad.gann.population_as_matrices(population_networks=GANN_
↪instance.population_networks,
                                                            population_vectors=ga_
↪instance.population)

    GANN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)

    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")
    print(f"Change     = {ga_instance.best_solution()[1] - last_fitness}")

    last_fitness = ga_instance.best_solution()[1].copy()

# Holds the fitness value of the previous generation.
last_fitness = 0

# Preparing the NumPy array of the inputs.
data_inputs = numpy.array([[1, 1],
                           [1, 0],
                           [0, 1],
                           [0, 0]])

# Preparing the NumPy array of the outputs.
data_outputs = numpy.array([0,
                            1,
                            1,
                            0])

# The length of the input vector for each sample (i.e. number of neurons in the input␣
↪layer).
```

()

()

```
num_inputs = data_inputs.shape[1]
# The number of neurons in the output layer (i.e. number of classes).
num_classes = 2

# Creating an initial population of neural networks. The return of the initial_
↪population() function holds references to the networks, not their weights. Using␣
↪such references, the weights of all networks can be fetched.
num_solutions = 6 # A solution or a network can be used interchangeably.
GANN_instance = pygad.gann.GANN(num_solutions=num_solutions,
                                num_neurons_input=num_inputs,
                                num_neurons_hidden_layers=[2],
                                num_neurons_output=num_classes,
                                hidden_activations=["relu"],
                                output_activation="softmax")

# population does not hold the numerical weights of the network instead it holds a␣
↪list of references to each last layer of each network (i.e. solution) in the␣
↪population. A solution or a network can be used interchangeably.
# If there is a population with 3 solutions (i.e. networks), then the population is a␣
↪list with 3 elements. Each element is a reference to the last layer of each network.
↪ Using such a reference, all details of the network can be accessed.
population_vectors = pygad.gann.population_as_vectors(population_networks=GANN_
↪instance.population_networks)

# To prepare the initial population, there are 2 ways:
# 1) Prepare it yourself and pass it to the initial_population parameter. This way is␣
↪useful when the user wants to start the genetic algorithm with a custom initial␣
↪population.
# 2) Assign valid integer values to the sol_per_pop and num_genes parameters. If the␣
↪initial_population parameter exists, then the sol_per_pop and num_genes parameters␣
↪are useless.
initial_population = population_vectors.copy()

num_parents_mating = 4 # Number of solutions to be selected as parents in the mating␣
↪pool.

num_generations = 500 # Number of generations.

mutation_percent_genes = [5, 10] # Percentage of genes to mutate. This parameter has␣
↪no action if the parameter mutation_num_genes exists.

parent_selection_type = "sss" # Type of parent selection.

crossover_type = "single_point" # Type of the crossover operator.

mutation_type = "adaptive" # Type of the mutation operator.

keep_parents = 1 # Number of parents to keep in the next population. -1 means keep␣
↪all parents and 0 means keep nothing.

init_range_low = -2
init_range_high = 5

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
```

()

()

```
                       mutation_percent_genes=mutation_percent_genes,
                       init_range_low=init_range_low,
                       init_range_high=init_range_high,
                       parent_selection_type=parent_selection_type,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       keep_parents=keep_parents,
                       suppress_warnings=True,
                       on_generation=callback_generation)

ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness()

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")

# Predicting the outputs of the data using the best solution.
predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[solution_
↪idx],
                               data_inputs=data_inputs)
print(f"Predictions of the trained network : {predictions}")

# Calculating some statistics
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

pygad.nnpygad.gann

num_neurons_outputpygad.gann.GANN

```
import numpy
import pygad
import pygad.nn
```

()

()

```
import pygad.gann

def fitness_func(ga_instance, solution, sol_idx):
    global GANN_instance, data_inputs, data_outputs

    predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[sol_
↪idx],
                                   data_inputs=data_inputs)
    correct_predictions = numpy.where(predictions == data_outputs)[0].size
    solution_fitness = (correct_predictions/data_outputs.size)*100

    return solution_fitness

def callback_generation(ga_instance):
    global GANN_instance, last_fitness

    population_matrices = pygad.gann.population_as_matrices(population_networks=GANN_
↪instance.population_networks,
                                                            population_vectors=ga_
↪instance.population)

    GANN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)

    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")
    print(f"Change     = {ga_instance.best_solution()[1] - last_fitness}")

    last_fitness = ga_instance.best_solution()[1].copy()

# Holds the fitness value of the previous generation.
last_fitness = 0

# Reading the input data.
data_inputs = numpy.load("dataset_features.npy") # Download from https://github.com/
↪ahmedfgad/NumPyANN/blob/master/dataset_features.npy

# Optional step of filtering the input data using the standard deviation.
features_STDs = numpy.std(a=data_inputs, axis=0)
data_inputs = data_inputs[:, features_STDs>50]

# Reading the output data.
data_outputs = numpy.load("outputs.npy") # Download from https://github.com/ahmedfgad/
↪NumPyANN/blob/master/outputs.npy

# The length of the input vector for each sample (i.e. number of neurons in the input␣
↪layer).
num_inputs = data_inputs.shape[1]
# The number of neurons in the output layer (i.e. number of classes).
num_classes = 4

# Creating an initial population of neural networks. The return of the initial_
↪population() function holds references to the networks, not their weights. Using␣
↪such references, the weights of all networks can be fetched.
num_solutions = 8 # A solution or a network can be used interchangeably.
GANN_instance = pygad.gann.GANN(num_solutions=num_solutions,
                                num_neurons_input=num_inputs,
```

()

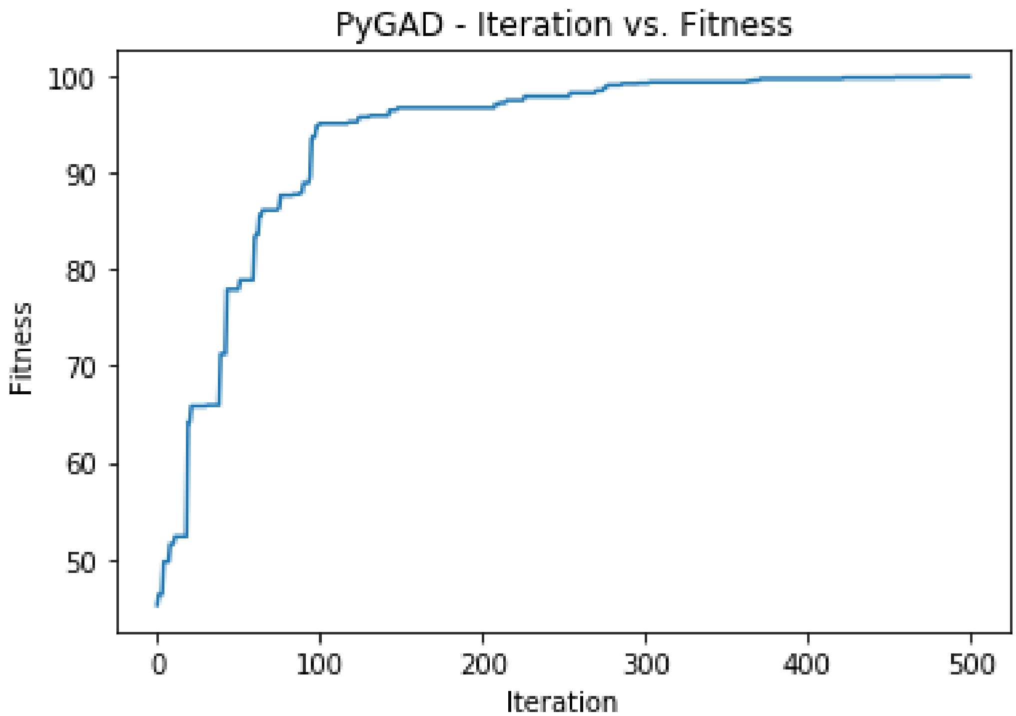
()

```
                                  num_neurons_hidden_layers=[150, 50],
                                  num_neurons_output=num_classes,
                                  hidden_activations=["relu", "relu"],
                                  output_activation="softmax")

# population does not hold the numerical weights of the network instead it holds a␣
↪list of references to each last layer of each network (i.e. solution) in the␣
↪population. A solution or a network can be used interchangeably.
# If there is a population with 3 solutions (i.e. networks), then the population is a␣
↪list with 3 elements. Each element is a reference to the last layer of each network.
↪ Using such a reference, all details of the network can be accessed.
population_vectors = pygad.gann.population_as_vectors(population_networks=GANN_
↪instance.population_networks)

# To prepare the initial population, there are 2 ways:
# 1) Prepare it yourself and pass it to the initial_population parameter. This way is␣
↪useful when the user wants to start the genetic algorithm with a custom initial␣
↪population.
# 2) Assign valid integer values to the sol_per_pop and num_genes parameters. If the␣
↪initial_population parameter exists, then the sol_per_pop and num_genes parameters␣
↪are useless.
initial_population = population_vectors.copy()

num_parents_mating = 4 # Number of solutions to be selected as parents in the mating␣
↪pool.

num_generations = 500 # Number of generations.

mutation_percent_genes = 10 # Percentage of genes to mutate. This parameter has no␣
↪action if the parameter mutation_num_genes exists.

parent_selection_type = "sss" # Type of parent selection.

crossover_type = "single_point" # Type of the crossover operator.

mutation_type = "random" # Type of the mutation operator.

keep_parents = -1 # Number of parents to keep in the next population. -1 means keep␣
↪all parents and 0 means keep nothing.

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       mutation_percent_genes=mutation_percent_genes,
                       parent_selection_type=parent_selection_type,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       keep_parents=keep_parents,
                       on_generation=callback_generation)

ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness()
```

()

```
# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")

# Predicting the outputs of the data using the best solution.
predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[solution_
↪idx],
                               data_inputs=data_inputs)
print(f"Predictions of the trained network : {predictions}")

# Calculating some statistics
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

```
Fitness value of the best solution = 99.94903160040775
Index of the best solution : 0
Best fitness value reached after 482 generations.
Number of correct classifications : 1961.
Number of wrong classifications : 1.
Classification accuracy : 99.94903160040775.
```



output_activationpygad.gann.GANN"None"

```
GANN_instance = pygad.gann.GANN(...
                                output_activation="None")
```

pygad.nn.predict()problem_type"regression"

```
predictions = pygad.nn.predict(...,
                               problem_type="regression")
```

().

```
def fitness_func(ga_instance, solution, sol_idx):
    ...

    predictions = pygad.nn.predict(...,
                                   problem_type="regression")

    solution_fitness = 1.0/numpy.mean(numpy.abs(predictions - data_outputs))

    return solution_fitness
```

```
import numpy
import pygad
```

()

()

```python
import pygad.nn
import pygad.gann

def fitness_func(ga_instance, solution, sol_idx):
    global GANN_instance, data_inputs, data_outputs

    predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[sol_
↪idx],
                                   data_inputs=data_inputs, problem_type="regression")
    solution_fitness = 1.0/numpy.mean(numpy.abs(predictions - data_outputs))

    return solution_fitness

def callback_generation(ga_instance):
    global GANN_instance, last_fitness

    population_matrices = pygad.gann.population_as_matrices(population_networks=GANN_
↪instance.population_networks,
                                                             population_vectors=ga_
↪instance.population)

    GANN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)

    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution(pop_fitness=ga_instance.last_
↪generation_fitness)[1]}")
    print(f"Change     = {ga_instance.best_solution(pop_fitness=ga_instance.last_
↪generation_fitness)[1] - last_fitness}")

    last_fitness = ga_instance.best_solution(pop_fitness=ga_instance.last_generation_
↪fitness)[1].copy()

# Holds the fitness value of the previous generation.
last_fitness = 0

# Preparing the NumPy array of the inputs.
data_inputs = numpy.array([[2, 5, -3, 0.1],
                           [8, 15, 20, 13]])

# Preparing the NumPy array of the outputs.
data_outputs = numpy.array([[0.1, 0.2],
                            [1.8, 1.5]])

# The length of the input vector for each sample (i.e. number of neurons in the input␣
↪layer).
num_inputs = data_inputs.shape[1]

# Creating an initial population of neural networks. The return of the initial_
↪population() function holds references to the networks, not their weights. Using␣
↪such references, the weights of all networks can be fetched.
num_solutions = 6 # A solution or a network can be used interchangeably.
GANN_instance = pygad.gann.GANN(num_solutions=num_solutions,
                                num_neurons_input=num_inputs,
                                num_neurons_hidden_layers=[2],
                                num_neurons_output=2,
                                hidden_activations=["relu"],
```

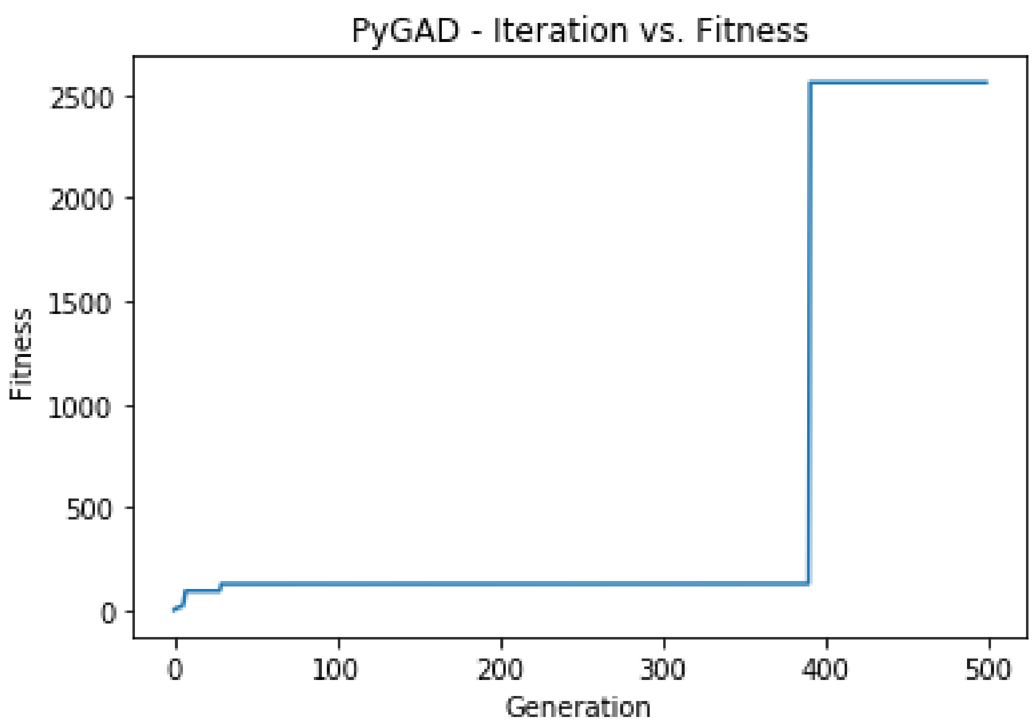
()

()

```
                                     output_activation="None")

# population does not hold the numerical weights of the network instead it holds a␣
↪list of references to each last layer of each network (i.e. solution) in the␣
↪population. A solution or a network can be used interchangeably.
# If there is a population with 3 solutions (i.e. networks), then the population is a␣
↪list with 3 elements. Each element is a reference to the last layer of each network.
↪ Using such a reference, all details of the network can be accessed.
population_vectors = pygad.gann.population_as_vectors(population_networks=GANN_
↪instance.population_networks)

# To prepare the initial population, there are 2 ways:
# 1) Prepare it yourself and pass it to the initial_population parameter. This way is␣
↪useful when the user wants to start the genetic algorithm with a custom initial␣
↪population.
# 2) Assign valid integer values to the sol_per_pop and num_genes parameters. If the␣
↪initial_population parameter exists, then the sol_per_pop and num_genes parameters␣
↪are useless.
initial_population = population_vectors.copy()

num_parents_mating = 4 # Number of solutions to be selected as parents in the mating␣
↪pool.

num_generations = 500 # Number of generations.

mutation_percent_genes = 5 # Percentage of genes to mutate. This parameter has no␣
↪action if the parameter mutation_num_genes exists.

parent_selection_type = "sss" # Type of parent selection.

crossover_type = "single_point" # Type of the crossover operator.

mutation_type = "random" # Type of the mutation operator.

keep_parents = 1 # Number of parents to keep in the next population. -1 means keep␣
↪all parents and 0 means keep nothing.

init_range_low = -1
init_range_high = 1

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       mutation_percent_genes=mutation_percent_genes,
                       init_range_low=init_range_low,
                       init_range_high=init_range_high,
                       parent_selection_type=parent_selection_type,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       keep_parents=keep_parents,
                       on_generation=callback_generation)

ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
```

()

```
ga_instance.plot_fitness()

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution(pop_fitness=ga_
↪instance.last_generation_fitness)
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")

# Predicting the outputs of the data using the best solution.
predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[solution_
↪idx],
                               data_inputs=data_inputs,
                               problem_type="regression")
print(f"Predictions of the trained network : {predictions}")

# Calculating some statistics
abs_error = numpy.mean(numpy.abs(predictions - data_outputs))
print(f"Absolute error : {abs_error}.")
```



(https://www.kaggle.com/aungpyaeap/fish-market). (https://www.kaggle.com/aungpyaeap/fish-market/download).

read_csv()

```
data = numpy.array(pandas.read_csv("Fish.csv"))
```

```
# Preparing the NumPy array of the inputs.
data_inputs = numpy.asarray(data[:, 2:], dtype=numpy.float32)

# Preparing the NumPy array of the outputs.
data_outputs = numpy.asarray(data[:, 1], dtype=numpy.float32) # Fish Weight
```

"None"problem_typepygad.nn.train()pygad.nn.predict()"regression"

```
solution_fitness = 1.0/numpy.mean(numpy.abs(predictions - data_outputs))
```

```
import numpy
import pygad
import pygad.nn
import pygad.gann
import pandas

def fitness_func(ga_instance, solution, sol_idx):
    global GANN_instance, data_inputs, data_outputs

    predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[sol_
↪idx],
                                   data_inputs=data_inputs, problem_type="regression")
    solution_fitness = 1.0/numpy.mean(numpy.abs(predictions - data_outputs))

    return solution_fitness

def callback_generation(ga_instance):
    global GANN_instance, last_fitness

    population_matrices = pygad.gann.population_as_matrices(population_networks=GANN_
↪instance.population_networks,
                                                            population_vectors=ga_
↪instance.population)

    GANN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)

    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution(pop_fitness=ga_instance.last_
↪generation_fitness)[1]}")
    print(f"Change     = {ga_instance.best_solution(pop_fitness=ga_instance.last_
↪generation_fitness)[1] - last_fitness}")

    last_fitness = ga_instance.best_solution(pop_fitness=ga_instance.last_generation_
↪fitness)[1].copy()
```

()

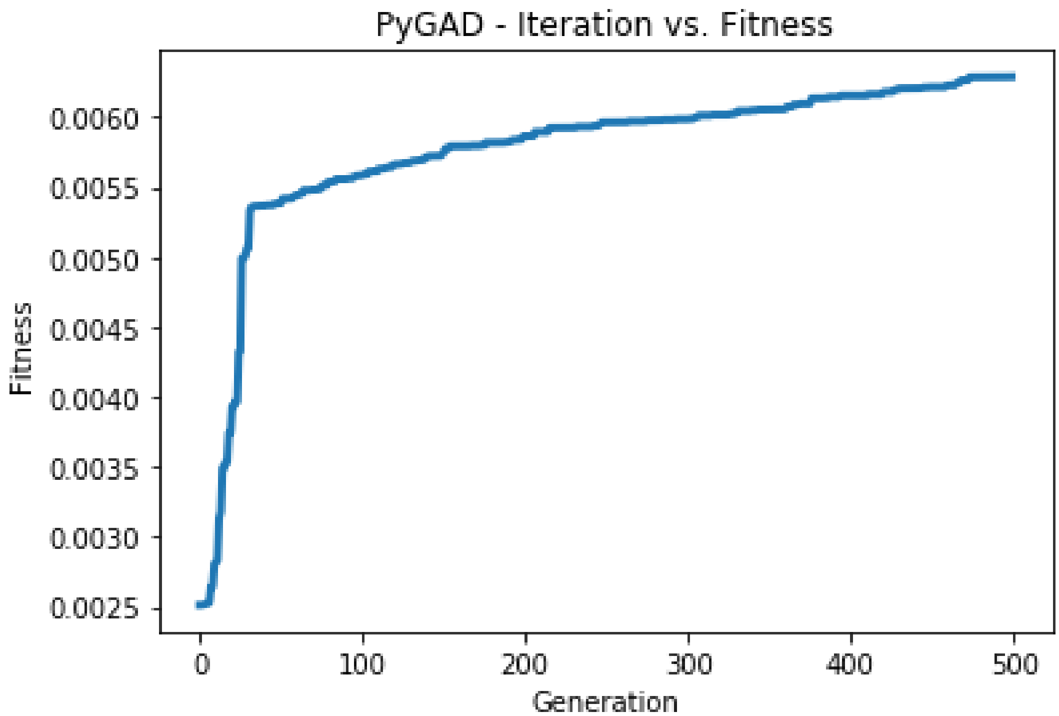
()

```
# Holds the fitness value of the previous generation.
last_fitness = 0

data = numpy.array(pandas.read_csv("../data/Fish.csv"))

# Preparing the NumPy array of the inputs.
data_inputs = numpy.asarray(data[:, 2:], dtype=numpy.float32)

# Preparing the NumPy array of the outputs.
data_outputs = numpy.asarray(data[:, 1], dtype=numpy.float32)

# The length of the input vector for each sample (i.e. number of neurons in the input␣
↪layer).
num_inputs = data_inputs.shape[1]

# Creating an initial population of neural networks. The return of the initial_
↪population() function holds references to the networks, not their weights. Using␣
↪such references, the weights of all networks can be fetched.
num_solutions = 6 # A solution or a network can be used interchangeably.
GANN_instance = pygad.gann.GANN(num_solutions=num_solutions,
                                num_neurons_input=num_inputs,
                                num_neurons_hidden_layers=[2],
                                num_neurons_output=1,
                                hidden_activations=["relu"],
                                output_activation="None")

# population does not hold the numerical weights of the network instead it holds a␣
↪list of references to each last layer of each network (i.e. solution) in the␣
↪population. A solution or a network can be used interchangeably.
# If there is a population with 3 solutions (i.e. networks), then the population is a␣
↪list with 3 elements. Each element is a reference to the last layer of each network.
↪ Using such a reference, all details of the network can be accessed.
population_vectors = pygad.gann.population_as_vectors(population_networks=GANN_
↪instance.population_networks)

# To prepare the initial population, there are 2 ways:
# 1) Prepare it yourself and pass it to the initial_population parameter. This way is␣
↪useful when the user wants to start the genetic algorithm with a custom initial␣
↪population.
# 2) Assign valid integer values to the sol_per_pop and num_genes parameters. If the␣
↪initial_population parameter exists, then the sol_per_pop and num_genes parameters␣
↪are useless.
initial_population = population_vectors.copy()

num_parents_mating = 4 # Number of solutions to be selected as parents in the mating␣
↪pool.

num_generations = 500 # Number of generations.

mutation_percent_genes = 5 # Percentage of genes to mutate. This parameter has no␣
↪action if the parameter mutation_num_genes exists.

parent_selection_type = "sss" # Type of parent selection.

crossover_type = "single_point" # Type of the crossover operator.
```

()

()

```
mutation_type = "random" # Type of the mutation operator.

keep_parents = 1 # Number of parents to keep in the next population. -1 means keep␣
↪all parents and 0 means keep nothing.

init_range_low = -1
init_range_high = 1

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       mutation_percent_genes=mutation_percent_genes,
                       init_range_low=init_range_low,
                       init_range_high=init_range_high,
                       parent_selection_type=parent_selection_type,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       keep_parents=keep_parents,
                       on_generation=callback_generation)

ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness()

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution(pop_fitness=ga_
↪instance.last_generation_fitness)
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")

# Predicting the outputs of the data using the best solution.
predictions = pygad.nn.predict(last_layer=GANN_instance.population_networks[solution_
↪idx],
                               data_inputs=data_inputs,
                               problem_type="regression")
print(f"Predictions of the trained network : {predictions}")

# Calculating some statistics
abs_error = numpy.mean(numpy.abs(predictions - data_outputs))
print(f"Absolute error : {abs_error}.")
```


PyGAD - Iteration vs. Fitness
Fitness
0.0025
0.0030
0.0035
0.0040
0.0045
0.0050
0.0055
0.0060
0
100
200
300
400
500
Generation

pygad.cnn

,

()

pygad.cnn.Input2D

pygad.cnn.Conv2D

pygad.cnn.MaxPooling2D

pygad.cnn.AveragePooling2D

pygad.cnn.Flatten

pygad.cnn.ReLU

pygad.cnn.Sigmoid

(): pygad.cnn.Dense

previous_layer

layer_input_size

layer_output_size

layer_outputNone

pygad.cnn.Input2D

pygad.cnn.Input2D

input_shape

Input2D

- input_shape
- layer_output_size

(50, 50, 3)

```
input_layer = pygad.cnn.Input2D(input_shape=(50, 50, 3))
```

pygad.cnn.Input2D

```
input_shape = input_layer.input_shape
layer_output_size = input_layer.layer_output_size

print("Input2D Input shape =", input_shape)
print("Input2D Output shape =", layer_output_size)
```

pygad.cnn.Conv2D

pygad.cnn.Conv2D()

- num_filters
- kernel_size
- previous_layerprevious_layerpygad.nn
- activation_function=NoneNonerelusigmoid

- filter_bank_size
- initial_weights
- trained_weightsinitial_weights
- layer_input_size
- layer_output_size
- layer_output

previous_layerinput_layer

```
conv_layer = pygad.cnn.Conv2D(num_filters=2,
                              kernel_size=3,
                              previous_layer=input_layer,
                              activation_function=None)
```

```python
filter_bank_size = conv_layer.filter_bank_size
conv_initail_weights = conv_layer.initial_weights

print("Filter bank size attributes =", filter_bank_size)
print("Initial weights of the conv layer :", conv_initail_weights)
```

conv_layer

```python
input_layer = conv_layer.previous_layer
input_shape = input_layer.num_neurons

print("Input shape =", input_shape)
```

previous_layerReLU

```python
conv_layer2 = pygad.cnn.Conv2D(num_filters=2,
                               kernel_size=3,
                               previous_layer=conv_layer,
                               activation_function="relu")
```

conv_layer2conv_layerprevious_layerconv_layer

```python
conv_layer = conv_layer2.previous_layer
filter_bank_size = conv_layer.filter_bank_size

print("Filter bank size attributes =", filter_bank_size)
```

conv_layer

```python
conv_layer = conv_layer2.previous_layer
input_layer = conv_layer.previous_layer
input_shape = input_layer.num_neurons

print("Input shape =", input_shape)
```

pygad.cnn.MaxPooling2D

pygad.cnn.MaxPooling2D

- pool_size
- previous_layer
- stride=2

- layer_input_size
- layer_output_size
- layer_output

pygad.cnn.AveragePooling2D

pygad.cnn.AveragePooling2Dpygad.cnn.MaxPooling2D

pygad.cnn.Flatten

pygad.cnn.Flattenprevious_layer

- previous_layer
- layer_input_size
- layer_output_size
- layer_output

pygad.cnn.ReLU

pygad.cnn.ReLU

previous_layer

- previous_layer
- layer_input_size
- layer_output_size
- layer_output

pygad.cnn.Sigmoid

pygad.cnn.Sigmoidpygad.cnn.ReLU

pygad.cnn.Dense

pygad.cnn.Dense

- num_neurons
- previous_layer
- activation_function"sigmoid""sigmoid""relu"softmax

- initial_weights
- trained_weightsinitial_weights
- layer_input_size
- layer_output_size
- layer_output

pygad.cnn.Model

pygad.cnn.Model

    last_layer().

    epochs=10:

    learning_rate=0.01

network_layersget_layers()pygad.cnn.Model

pygad.cnn.Model

get_layers()

train()

    train_inputs

    train_outputs

pygad.cnn.Model

feed_sample()

update_weights()

predict()

    data_inputs

summary()

pygad.cnn.sigmoid()

(): pygad.cnn.relu()

softmaxpygad.cnn.softmax()

pygad.cnn

,

,

,

,

(80, 100, 100, 3)(100, 100, 3)

(https://github.com/ahmedfgad/NumPyCNN/blob/master/dataset_outputs.npy)

```python
train_inputs = numpy.load("dataset_inputs.npy")
train_outputs = numpy.load("dataset_outputs.npy")
```

pygad.cnn.Input2D

```python
import pygad.cnn
sample_shape = train_inputs.shape[1:]

input_layer = pygad.cnn.Input2D(input_shape=sample_shape)
```

```python
conv_layer1 = pygad.cnn.Conv2D(num_filters=2,
                               kernel_size=3,
                               previous_layer=input_layer,
                               activation_function=None)
relu_layer1 = pygad.cnn.Sigmoid(previous_layer=conv_layer1)
average_pooling_layer = pygad.cnn.AveragePooling2D(pool_size=2,
                                                   previous_layer=relu_layer1,
                                                   stride=2)

conv_layer2 = pygad.cnn.Conv2D(num_filters=3,
                               kernel_size=3,
                               previous_layer=average_pooling_layer,
                               activation_function=None)
relu_layer2 = pygad.cnn.ReLU(previous_layer=conv_layer2)
max_pooling_layer = pygad.cnn.MaxPooling2D(pool_size=2,
                                           previous_layer=relu_layer2,
                                           stride=2)

conv_layer3 = pygad.cnn.Conv2D(num_filters=1,
                               kernel_size=3,
                               previous_layer=max_pooling_layer,
                               activation_function=None)
relu_layer3 = pygad.cnn.ReLU(previous_layer=conv_layer3)
pooling_layer = pygad.cnn.AveragePooling2D(pool_size=2,
                                           previous_layer=relu_layer3,
                                           stride=2)

flatten_layer = pygad.cnn.Flatten(previous_layer=pooling_layer)
dense_layer1 = pygad.cnn.Dense(num_neurons=100,
                               previous_layer=flatten_layer,
                               activation_function="relu")
dense_layer2 = pygad.cnn.Dense(num_neurons=4,
                               previous_layer=dense_layer1,
                               activation_function="softmax")
```

pygad.cnn.Model

```
model = pygad.cnn.Model(last_layer=dense_layer2,
                        epochs=5,
                        learning_rate=0.01)
```

summary()pygad.cnn.Model

```
model.summary()
```

```
----------Network Architecture----------
<class 'pygad.cnn.Conv2D'>
<class 'pygad.cnn.Sigmoid'>
<class 'pygad.cnn.AveragePooling2D'>
<class 'pygad.cnn.Conv2D'>
<class 'pygad.cnn.ReLU'>
<class 'pygad.cnn.MaxPooling2D'>
<class 'pygad.cnn.Conv2D'>
<class 'pygad.cnn.ReLU'>
<class 'pygad.cnn.AveragePooling2D'>
<class 'pygad.cnn.Flatten'>
<class 'pygad.cnn.Dense'>
<class 'pygad.cnn.Dense'>
----------------------------------------
```

pygad.cnn.train()

```
model.train(train_inputs=train_inputs,
            train_outputs=train_outputs)
```

pygad.cnn.predict()

```
predictions = model.predict(data_inputs=train_inputs)
```

```python
num_wrong = numpy.where(predictions != train_outputs)[0]
num_correct = train_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/train_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

pygad.gacnn

pygad.cnn

```python
import numpy
import pygad.cnn

"""
Convolutional neural network implementation using NumPy
A tutorial that helps to get started (Building Convolutional Neural Network using NumPy from Scratch) available in these links:
    https://www.linkedin.com/pulse/building-convolutional-neural-network-using-numpy-from-ahmed-gad
    https://towardsdatascience.com/building-convolutional-neural-network-using-numpy-from-scratch-b30aac50e50a
    https://www.kdnuggets.com/2018/04/building-convolutional-neural-network-numpy-scratch.html
It is also translated into Chinese: http://m.aliyun.com/yunqi/articles/585741
"""

train_inputs = numpy.load("dataset_inputs.npy")
train_outputs = numpy.load("dataset_outputs.npy")

sample_shape = train_inputs.shape[1:]
num_classes = 4

input_layer = pygad.cnn.Input2D(input_shape=sample_shape)
conv_layer1 = pygad.cnn.Conv2D(num_filters=2,
                               kernel_size=3,
                               previous_layer=input_layer,
                               activation_function=None)
relu_layer1 = pygad.cnn.Sigmoid(previous_layer=conv_layer1)
average_pooling_layer = pygad.cnn.AveragePooling2D(pool_size=2,
                                                   previous_layer=relu_layer1,
                                                   stride=2)
```

()

```
conv_layer2 = pygad.cnn.Conv2D(num_filters=3,
                               kernel_size=3,
                               previous_layer=average_pooling_layer,
                               activation_function=None)
relu_layer2 = pygad.cnn.ReLU(previous_layer=conv_layer2)
max_pooling_layer = pygad.cnn.MaxPooling2D(pool_size=2,
                                           previous_layer=relu_layer2,
                                           stride=2)

conv_layer3 = pygad.cnn.Conv2D(num_filters=1,
                               kernel_size=3,
                               previous_layer=max_pooling_layer,
                               activation_function=None)
relu_layer3 = pygad.cnn.ReLU(previous_layer=conv_layer3)
pooling_layer = pygad.cnn.AveragePooling2D(pool_size=2,
                                           previous_layer=relu_layer3,
                                           stride=2)

flatten_layer = pygad.cnn.Flatten(previous_layer=pooling_layer)
dense_layer1 = pygad.cnn.Dense(num_neurons=100,
                               previous_layer=flatten_layer,
                               activation_function="relu")
dense_layer2 = pygad.cnn.Dense(num_neurons=num_classes,
                               previous_layer=dense_layer1,
                               activation_function="softmax")

model = pygad.cnn.Model(last_layer=dense_layer2,
                        epochs=1,
                        learning_rate=0.01)

model.summary()

model.train(train_inputs=train_inputs,
            train_outputs=train_outputs)

predictions = model.predict(data_inputs=train_inputs)
print(predictions)

num_wrong = numpy.where(predictions != train_outputs)[0]
num_correct = train_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/train_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

pygad.gacnn

,

pygad.gacnnpygadpygad.cnn

pygad.gacnn.GACNN

pygad.gacnnpygad.gacnn.GACNN()

__init__()

pygad.gacnn.GACNN

pygad.gacnn.GACNN

model

num_solutions()

pygad.gacnn.GACNNpygad.gacnn.GACNN

population_networks()

pygad.gacnn.GACNN

create_population()

create_population()().

population_networks

update_population_trained_weights()

update_population_trained_weights()trained_weights(pygad.cnn)) population_trained_weights

population_trained_weightstrained_weights

pygad.gacnn

pygad.gacnn

pygad.gacnn.population_as_vectors()

pygad.cnn.Model()

(), ().

population_networkspygad.cnn.Model

().

pygad.gacnn.population_as_matrices()

()

(),

population_networkspygad.cnn.Model

population_vectors

().

pygad.gacnn.GACNN

pygad.GA

pygad.GA

,

()

(100, 100, 3)pygad.cnn

```
import numpy

train_inputs = numpy.load("dataset_inputs.npy")
train_outputs = numpy.load("dataset_outputs.npy")
```

0(80)0N-1N

```
import pygad.cnn

input_layer = pygad.cnn.Input2D(input_shape=(80, 80, 3))
conv_layer = pygad.cnn.Conv2D(num_filters=2,
                              kernel_size=3,
                              previous_layer=input_layer,
                              activation_function="relu")
average_pooling_layer = pygad.cnn.AveragePooling2D(pool_size=5,
                                                   previous_layer=conv_layer,
                                                   stride=3)

flatten_layer = pygad.cnn.Flatten(previous_layer=average_pooling_layer)
dense_layer = pygad.cnn.Dense(num_neurons=4,
                              previous_layer=flatten_layer,
                              activation_function="softmax")
```

pygad.cnn.Model

```
model = pygad.cnn.Model(last_layer=dense_layer,
                        epochs=5,
                        learning_rate=0.01)
```

summary() pygad.cnn.Model

```
model.summary()
```

```
----------Network Architecture----------
<class 'cnn.Conv2D'>
<class 'cnn.AveragePooling2D'>
<class 'cnn.Flatten'>
<class 'cnn.Dense'>
----------------------------------------
```

pygad.gacnn.GACNN

pygad.gacnn.GACNN

pygad.gacnn.GACNN

num_solutions(). model

```
import pygad.gacnn

GACNN_instance = pygad.gacnn.GACNN(model=model,
                                   num_solutions=4)
```

pygad.gacnn.GACNN

()

pygad.gacnn.population_as_vectors()

```
population_vectors = gacnn.population_as_vectors(population_networks=GACNN_instance.
↪population_networks)
```

```
initial_population = population_vectors.copy()
```

pygad.cnn.predict()' pygad.cnn.predict()trained_weights

```
def fitness_func(ga_instance, solution, sol_idx):
    global GACNN_instance, data_inputs, data_outputs

    predictions = GACNN_instance.population_networks[sol_idx].predict(data_
↪inputs=data_inputs)
    correct_predictions = numpy.where(predictions == data_outputs)[0].size
    solution_fitness = (correct_predictions/data_outputs.size)*100

    return solution_fitness
```

pygad.cnn.predict()' trained_weights

pygad.GAon_generationpygad.GA

trained_weights

trained_weights

pygad.gacnn.population_as_matrices()

update_population_trained_weights()pygad.gacnntrained_weights

```
def callback_generation(ga_instance):
    global GACNN_instance, last_fitness

    population_matrices = gacnn.population_as_matrices(population_networks=GACNN_
↪instance.population_networks, population_vectors=ga_instance.population)
    GACNN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)

    print(f"Generation = {ga_instance.generations_completed}")
```

pygad.GA

pygad.GA

pygad.GA

```python
import pygad

num_parents_mating = 4

num_generations = 10

mutation_percent_genes = 5

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       mutation_percent_genes=mutation_percent_genes,
                       on_generation=callback_generation)
```

run()

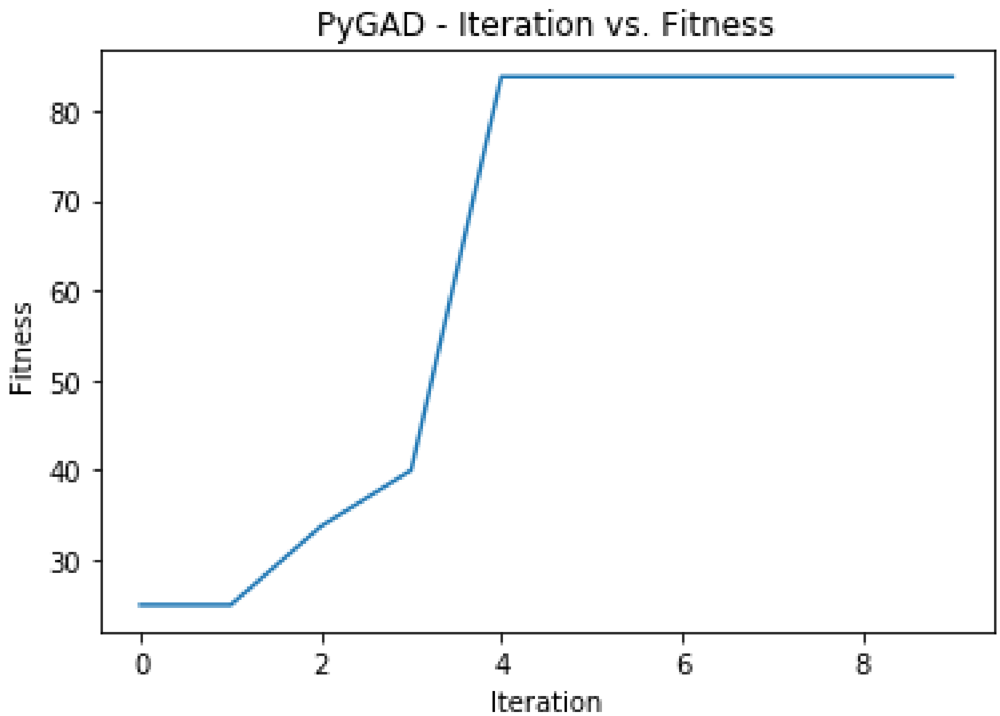
pygad.GA

run()pygad.GAnum_generations

```python
ga_instance.run()
```

run()plot_fitness()

```python
ga_instance.plot_fitness()
```



best_solution()pygad.GA

```
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")
```

```
...
Fitness value of the best solution = 83.75
Index of the best solution : 0
Best fitness value reached after 4 generations.
```

pygad.cnn.predict()

```
predictions = pygad.cnn.predict(last_layer=GANN_instance.population_networks[solution_
↪idx], data_inputs=data_inputs)
print(f"Predictions of the trained network : {predictions}")
```

```
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

```
Number of correct classifications : 67.
Number of wrong classifications : 13.
Classification accuracy : 83.75.
```

```
import numpy
import pygad.cnn
import pygad.gacnn
import pygad

"""
Convolutional neural network implementation using NumPy
A tutorial that helps to get started (Building Convolutional Neural Network using␣
↪NumPy from Scratch) available in these links:
    https://www.linkedin.com/pulse/building-convolutional-neural-network-using-numpy-
↪from-ahmed-gad
    https://towardsdatascience.com/building-convolutional-neural-network-using-numpy-
↪from-scratch-b30aac50e50a
    https://www.kdnuggets.com/2018/04/building-convolutional-neural-network-numpy-
↪scratch.html
It is also translated into Chinese: http://m.aliyun.com/yunqi/articles/585741
"""

def fitness_func(ga_instance, solution, sol_idx):
    global GACNN_instance, data_inputs, data_outputs
```

()

()

```
    predictions = GACNN_instance.population_networks[sol_idx].predict(data_
↪inputs=data_inputs)
    correct_predictions = numpy.where(predictions == data_outputs)[0].size
    solution_fitness = (correct_predictions/data_outputs.size)*100

    return solution_fitness

def callback_generation(ga_instance):
    global GACNN_instance, last_fitness

    population_matrices = pygad.gacnn.population_as_matrices(population_
↪networks=GACNN_instance.population_networks,
                                                    population_vectors=ga_instance.
↪population)

    GACNN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)

    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solutions_fitness}")

data_inputs = numpy.load("dataset_inputs.npy")
data_outputs = numpy.load("dataset_outputs.npy")

sample_shape = data_inputs.shape[1:]
num_classes = 4

data_inputs = data_inputs
data_outputs = data_outputs

input_layer = pygad.cnn.Input2D(input_shape=sample_shape)
conv_layer1 = pygad.cnn.Conv2D(num_filters=2,
                               kernel_size=3,
                               previous_layer=input_layer,
                               activation_function="relu")
average_pooling_layer = pygad.cnn.AveragePooling2D(pool_size=5,
                                                   previous_layer=conv_layer1,
                                                   stride=3)

flatten_layer = pygad.cnn.Flatten(previous_layer=average_pooling_layer)
dense_layer2 = pygad.cnn.Dense(num_neurons=num_classes,
                               previous_layer=flatten_layer,
                               activation_function="softmax")

model = pygad.cnn.Model(last_layer=dense_layer2,
                        epochs=1,
                        learning_rate=0.01)

model.summary()


GACNN_instance = pygad.gacnn.GACNN(model=model,
                              num_solutions=4)

# GACNN_instance.update_population_trained_weights(population_trained_
↪weights=population_matrices)
```

()

()

```
# population does not hold the numerical weights of the network instead it holds a␣
↪list of references to each last layer of each network (i.e. solution) in the␣
↪population. A solution or a network can be used interchangeably.
# If there is a population with 3 solutions (i.e. networks), then the population is a␣
↪list with 3 elements. Each element is a reference to the last layer of each network.
↪ Using such a reference, all details of the network can be accessed.
population_vectors = pygad.gacnn.population_as_vectors(population_networks=GACNN_
↪instance.population_networks)

# To prepare the initial population, there are 2 ways:
# 1) Prepare it yourself and pass it to the initial_population parameter. This way is␣
↪useful when the user wants to start the genetic algorithm with a custom initial␣
↪population.
# 2) Assign valid integer values to the sol_per_pop and num_genes parameters. If the␣
↪initial_population parameter exists, then the sol_per_pop and num_genes parameters␣
↪are useless.
initial_population = population_vectors.copy()

num_parents_mating = 2 # Number of solutions to be selected as parents in the mating␣
↪pool.

num_generations = 10 # Number of generations.

mutation_percent_genes = 0.1 # Percentage of genes to mutate. This parameter has no␣
↪action if the parameter mutation_num_genes exists.

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       mutation_percent_genes=mutation_percent_genes,
                       on_generation=callback_generation)

ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness()

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Parameters of the best solution : {solution}")
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

if ga_instance.best_solution_generation != -1:
    print(f"Best fitness value reached after {ga_instance.best_solution_generation}␣
↪generations.")

# Predicting the outputs of the data using the best solution.
predictions = GACNN_instance.population_networks[solution_idx].predict(data_
↪inputs=data_inputs)
print(f"Predictions of the trained network : {predictions}")

# Calculating some statistics
num_wrong = numpy.where(predictions != data_outputs)[0]
num_correct = data_outputs.size - num_wrong.size
```

()

()

```python
accuracy = 100 * (num_correct/data_outputs.size)
print(f"Number of correct classifications : {num_correct}.")
print(f"Number of wrong classifications : {num_wrong.size}.")
print(f"Classification accuracy : {accuracy}.")
```

pygad.kerasga

,

pygad.kerarsga().

KerasGA

model_weights_as_vector()

model_weights_as_matrix()

predict()

pygad.kerasga.KerasGA

pygad.GA

```python
import tensorflow.keras

input_layer  = tensorflow.keras.layers.Input(3)
dense_layer1 = tensorflow.keras.layers.Dense(5, activation="relu")
output_layer = tensorflow.keras.layers.Dense(1, activation="linear")

model = tensorflow.keras.Sequential()
model.add(input_layer)
model.add(dense_layer1)
model.add(output_layer)
```

```python
input_layer  = tensorflow.keras.layers.Input(3)
dense_layer1 = tensorflow.keras.layers.Dense(5, activation="relu")(input_layer)
output_layer = tensorflow.keras.layers.Dense(1, activation="linear")(dense_layer1)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)
```

pygad.kerasga.KerasGA

pygad.kerasgaKerasGA

__init__()

pygad.kerasga.KerasGA

model

num_solutions

pygad.kerasga.KerasGApopulation_weights

model

num_solutions

population_weights

KerasGA

pygad.kerasga.KerasGA

create_population()

create_population()population_weights

pygad.kerasga

pygad.kerasga

pygad.kerasga.model_weights_as_vector()

model_weights_as_vector()model

trainabletrainable=False

    model

pygad.kerasga.model_weights_as_matrix()

model_weights_as_matrix()

    model
    weights_vector

pygad.kerasga.predict()

predict()

    model
    solution
    data
    batch_size=None().
    verbose=None:
    steps=None().

()batch_sizeverbosesteps

```
import tensorflow.keras
import pygad.kerasga
import numpy
import pygad

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, keras_ga, model

    predictions = pygad.kerasga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    mae = tensorflow.keras.losses.MeanAbsoluteError()
    abs_error = mae(data_outputs, predictions).numpy() + 0.00000001
    solution_fitness = 1.0/abs_error

    return solution_fitness

def on_generation(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

input_layer  = tensorflow.keras.layers.Input(3)
dense_layer1 = tensorflow.keras.layers.Dense(5, activation="relu")(input_layer)
output_layer = tensorflow.keras.layers.Dense(1, activation="linear")(dense_layer1)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)

keras_ga = pygad.kerasga.KerasGA(model=model,
                                 num_solutions=10)

# Data inputs
data_inputs = numpy.array([[0.02, 0.1, 0.15],
                           [0.7, 0.6, 0.8],
                           [1.5, 1.2, 1.7],
                           [3.2, 2.9, 3.1]])

# Data outputs
data_outputs = numpy.array([[0.1],
                            [0.6],
                            [1.3],
                            [2.5]])

# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 250 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
```

()

```
initial_population = keras_ga.population_weights # Initial population of network␣
↪weights

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

# Make prediction based on the best solution.
predictions = pygad.kerasga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
print(f"Predictions : \n{predictions}")

mae = tensorflow.keras.losses.MeanAbsoluteError()
abs_error = mae(data_outputs, predictions).numpy()
print(f"Absolute Error : {abs_error}")
```

```
import tensorflow.keras

input_layer  = tensorflow.keras.layers.Input(3)
dense_layer1 = tensorflow.keras.layers.Dense(5, activation="relu")(input_layer)
output_layer = tensorflow.keras.layers.Dense(1, activation="linear")(dense_layer1)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)
```

```
input_layer  = tensorflow.keras.layers.Input(3)
dense_layer1 = tensorflow.keras.layers.Dense(5, activation="relu")
output_layer = tensorflow.keras.layers.Dense(1, activation="linear")

model = tensorflow.keras.Sequential()
model.add(input_layer)
model.add(dense_layer1)
model.add(output_layer)
```

pygad.kerasga.KerasGA

pygad.kerasga.KerasGA

```
import pygad.kerasga

keras_ga = pygad.kerasga.KerasGA(model=model,
                                 num_solutions=10)
```

```
import numpy

# Data inputs
data_inputs = numpy.array([[0.02, 0.1, 0.15],
                           [0.7, 0.6, 0.8],
                           [1.5, 1.2, 1.7],
                           [3.2, 2.9, 3.1]])

# Data outputs
data_outputs = numpy.array([[0.1],
                            [0.6],
                            [1.3],
                            [2.5]])
```

predict()()

```
def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, keras_ga, model

    predictions = pygad.kerasga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    mae = tensorflow.keras.losses.MeanAbsoluteError()
    abs_error = mae(data_outputs, predictions).numpy() + 0.00000001
    solution_fitness = 1.0/abs_error

    return solution_fitness
```

pygad.GA

pygad.GAinitial_population

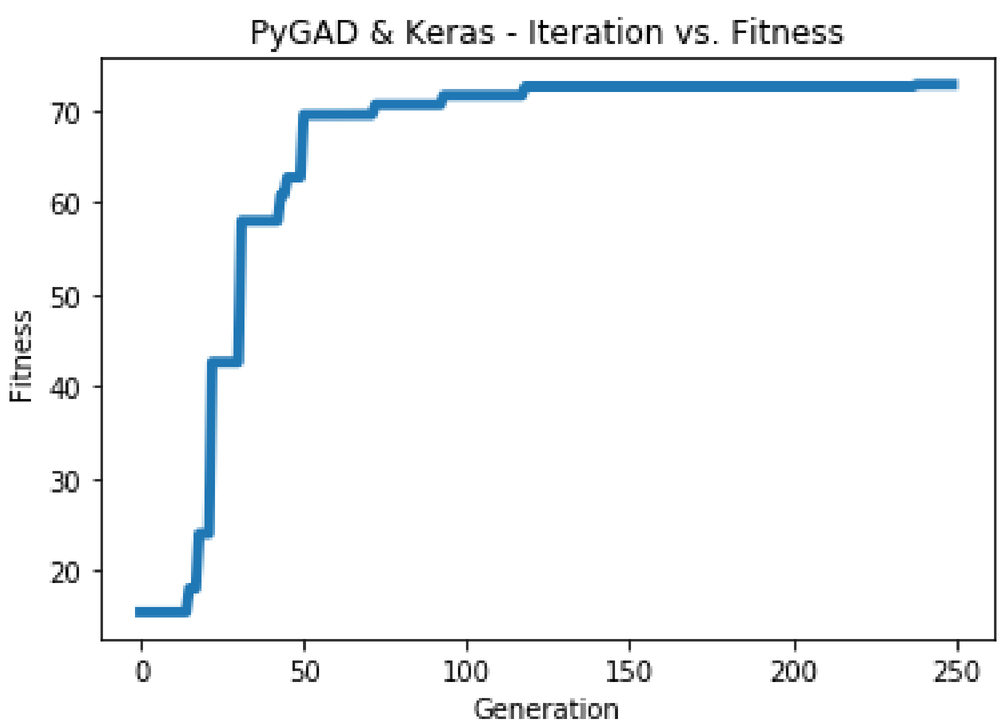
```
# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 250 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
initial_population = keras_ga.population_weights # Initial population of network␣
↪weights

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)
```

run()

```
ga_instance.run()
```

plot_fitness()

```
ga_instance.plot_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)
```



best_solution()

```python
# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")
```

```
Fitness value of the best solution = 72.77768757825352
Index of the best solution : 0
```

predict()

```python
# Fetch the parameters of the best solution.
predictions = pygad.kerasga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
print(f"Predictions : \n{predictions}")
```

```
Predictions :
[[0.09935353]
 [0.63082725]
 [1.2765523 ]
 [2.4999595 ]]
```

```python
mae = tensorflow.keras.losses.MeanAbsoluteError()
abs_error = mae(data_outputs, predictions).numpy()
print(f"Absolute Error : {abs_error}")
```

```
Absolute Error :  0.013740465
```

,

```python
import tensorflow.keras
import pygad.kerasga
import numpy
import pygad

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, keras_ga, model

    predictions = pygad.kerasga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    bce = tensorflow.keras.losses.BinaryCrossentropy()
    solution_fitness = 1.0 / (bce(data_outputs, predictions).numpy() + 0.00000001)

    return solution_fitness

def on_generation(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
```

()

()

```
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

# Build the keras model using the functional API.
input_layer  = tensorflow.keras.layers.Input(2)
dense_layer = tensorflow.keras.layers.Dense(4, activation="relu")(input_layer)
output_layer = tensorflow.keras.layers.Dense(2, activation="softmax")(dense_layer)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)

# Create an instance of the pygad.kerasga.KerasGA class to build the initial␣
↪population.
keras_ga = pygad.kerasga.KerasGA(model=model,
                                 num_solutions=10)

# XOR problem inputs
data_inputs = numpy.array([[0, 0],
                           [0, 1],
                           [1, 0],
                           [1, 1]])

# XOR problem outputs
data_outputs = numpy.array([[1, 0],
                            [0, 1],
                            [0, 1],
                            [1, 0]])

# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 250 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
initial_population = keras_ga.population_weights # Initial population of network␣
↪weights.

# Create an instance of the pygad.GA class
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Start the genetic algorithm evolution.
ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

# Make predictions based on the best solution.
predictions = pygad.kerasga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
```

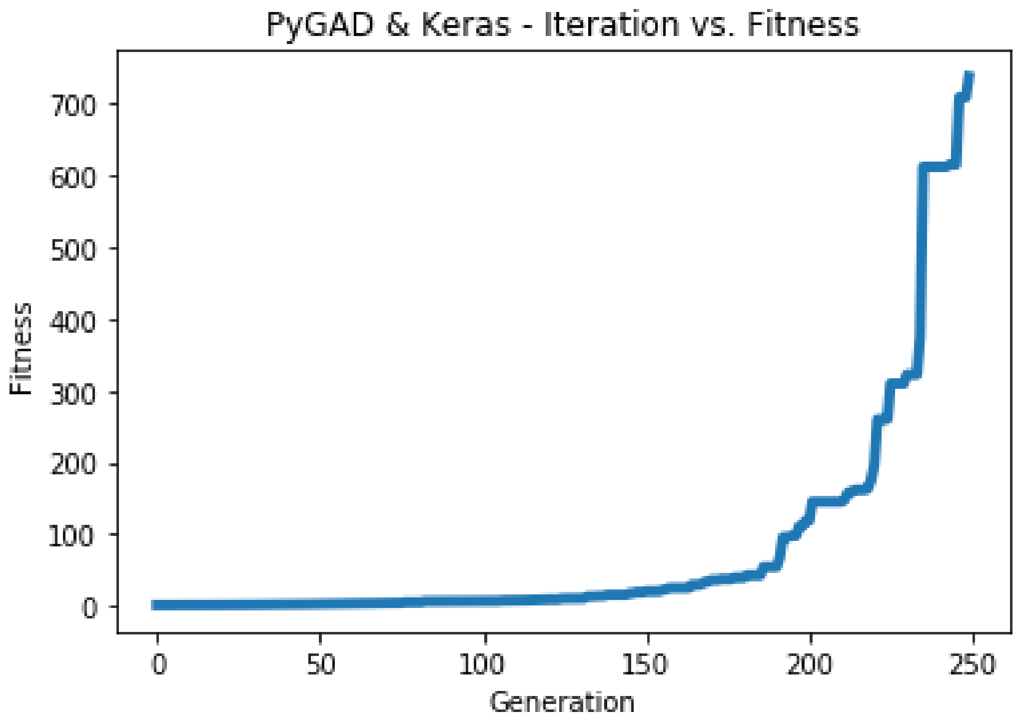
()

```
print(f"Predictions : \n{predictions}")

# Calculate the binary crossentropy for the trained model.
bce = tensorflow.keras.losses.BinaryCrossentropy()
print("Binary Crossentropy : ", bce(data_outputs, predictions).numpy())

# Calculate the classification accuracy for the trained model.
ba = tensorflow.keras.metrics.BinaryAccuracy()
ba.update_state(data_outputs, predictions)
accuracy = ba.result().numpy()
print(f"Accuracy : {accuracy}")
```

```
# Build the keras model using the functional API.
input_layer  = tensorflow.keras.layers.Input(2)
dense_layer = tensorflow.keras.layers.Dense(4, activation="relu")(input_layer)
output_layer = tensorflow.keras.layers.Dense(2, activation="softmax")(dense_layer)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)
```

```
# XOR problem inputs
data_inputs = numpy.array([[0, 0],
                           [0, 1],
                           [1, 0],
                           [1, 1]])

# XOR problem outputs
data_outputs = numpy.array([[1, 0],
                            [0, 1],
                            [0, 1],
                            [1, 0]])
```

```
bce = tensorflow.keras.losses.BinaryCrossentropy()
solution_fitness = 1.0 / (bce(data_outputs, predictions).numpy() + 0.00000001)
```




```
739.240.0013527311
```

```
Fitness value of the best solution = 739.2397344644013
Index of the best solution : 7

Predictions :
[[9.9694413e-01 3.0558957e-03]
 [5.0176249e-04 9.9949825e-01]
 [1.8470541e-03 9.9815291e-01]
 [9.9999976e-01 2.0538971e-07]]

Binary Crossentropy :  0.0013527311

Accuracy :  1.0
```

()

```python
import tensorflow.keras
import pygad.kerasga
import numpy
import pygad

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, keras_ga, model

    predictions = pygad.kerasga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)
```

()

()

```
    cce = tensorflow.keras.losses.CategoricalCrossentropy()
    solution_fitness = 1.0 / (cce(data_outputs, predictions).numpy() + 0.00000001)

    return solution_fitness

def on_generation(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

# Build the keras model using the functional API.
input_layer  = tensorflow.keras.layers.Input(360)
dense_layer = tensorflow.keras.layers.Dense(50, activation="relu")(input_layer)
output_layer = tensorflow.keras.layers.Dense(4, activation="softmax")(dense_layer)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)

# Create an instance of the pygad.kerasga.KerasGA class to build the initial␣
↪population.
keras_ga = pygad.kerasga.KerasGA(model=model,
                                 num_solutions=10)

# Data inputs
data_inputs = numpy.load("../data/dataset_features.npy")

# Data outputs
data_outputs = numpy.load("../data/outputs.npy")
data_outputs = tensorflow.keras.utils.to_categorical(data_outputs)

# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 100 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
initial_population = keras_ga.population_weights # Initial population of network␣
↪weights.

# Create an instance of the pygad.GA class
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Start the genetic algorithm evolution.
ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

# Make predictions based on the best solution.
```

()

()

```
predictions = pygad.kerasga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
# print(f"Predictions : \n{predictions}")

# Calculate the categorical crossentropy for the trained model.
cce = tensorflow.keras.losses.CategoricalCrossentropy()
print(f"Categorical Crossentropy : {cce(data_outputs, predictions).numpy()}")

# Calculate the classification accuracy for the trained model.
ca = tensorflow.keras.metrics.CategoricalAccuracy()
ca.update_state(data_outputs, predictions)
accuracy = ca.result().numpy()
print(f"Accuracy : {accuracy}")
```

()

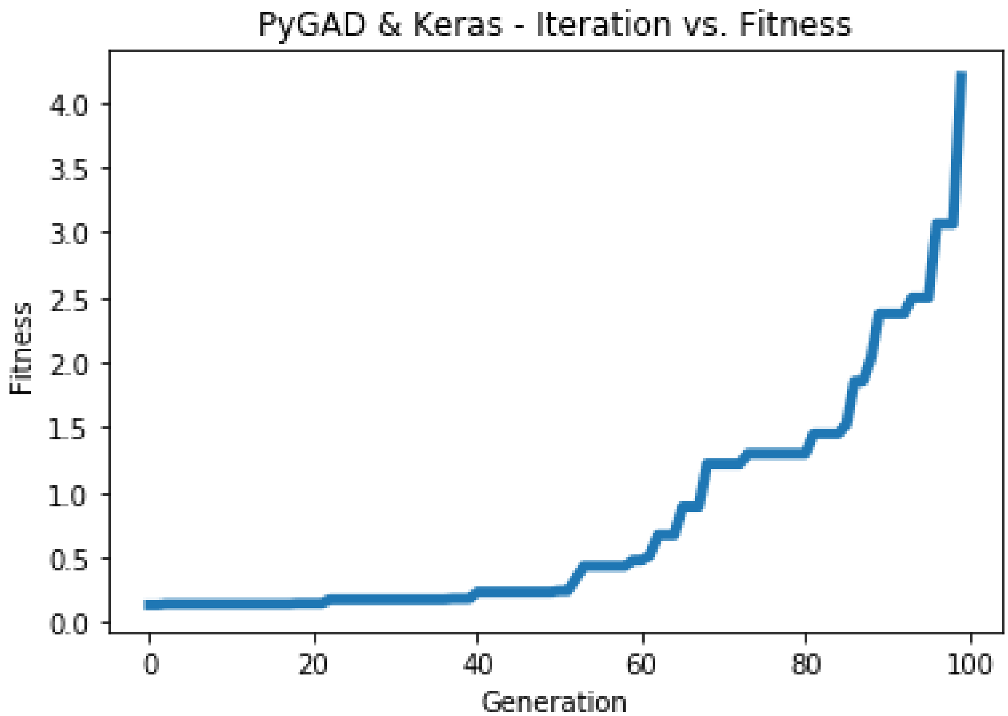
```
cce = tensorflow.keras.losses.CategoricalCrossentropy()
solution_fitness = 1.0 / (cce(data_outputs, predictions).numpy() + 0.00000001)
```

()

`(100, 100, 3)`

`tensorflow.keras.utils.to_categorical()`

```
import numpy

data_inputs = numpy.load("../data/dataset_features.npy")

data_outputs = numpy.load("../data/outputs.npy")
data_outputs = tensorflow.keras.utils.to_categorical(data_outputs)
```




```
Fitness value of the best solution = 4.197464252185969
Index of the best solution : 0
Categorical Crossentropy :  0.23823906
Accuracy :  0.9852192
```

()

```python
import tensorflow.keras
import pygad.kerasga
import numpy
import pygad

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, keras_ga, model

    predictions = pygad.kerasga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    cce = tensorflow.keras.losses.CategoricalCrossentropy()
    solution_fitness = 1.0 / (cce(data_outputs, predictions).numpy() + 0.00000001)

    return solution_fitness

def on_generation(ga_instance):
```

()

()

```
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

# Build the keras model using the functional API.
input_layer = tensorflow.keras.layers.Input(shape=(100, 100, 3))
conv_layer1 = tensorflow.keras.layers.Conv2D(filters=5,
                                             kernel_size=7,
                                             activation="relu")(input_layer)
max_pool1 = tensorflow.keras.layers.MaxPooling2D(pool_size=(5,5),
                                                 strides=5)(conv_layer1)
conv_layer2 = tensorflow.keras.layers.Conv2D(filters=3,
                                             kernel_size=3,
                                             activation="relu")(max_pool1)
flatten_layer  = tensorflow.keras.layers.Flatten()(conv_layer2)
dense_layer = tensorflow.keras.layers.Dense(15, activation="relu")(flatten_layer)
output_layer = tensorflow.keras.layers.Dense(4, activation="softmax")(dense_layer)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)

# Create an instance of the pygad.kerasga.KerasGA class to build the initial␣
↪population.
keras_ga = pygad.kerasga.KerasGA(model=model,
                                 num_solutions=10)

# Data inputs
data_inputs = numpy.load("../data/dataset_inputs.npy")

# Data outputs
data_outputs = numpy.load("../data/dataset_outputs.npy")
data_outputs = tensorflow.keras.utils.to_categorical(data_outputs)

# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 200 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
initial_population = keras_ga.population_weights # Initial population of network␣
↪weights.

# Create an instance of the pygad.GA class
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Start the genetic algorithm evolution.
ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")
```

()

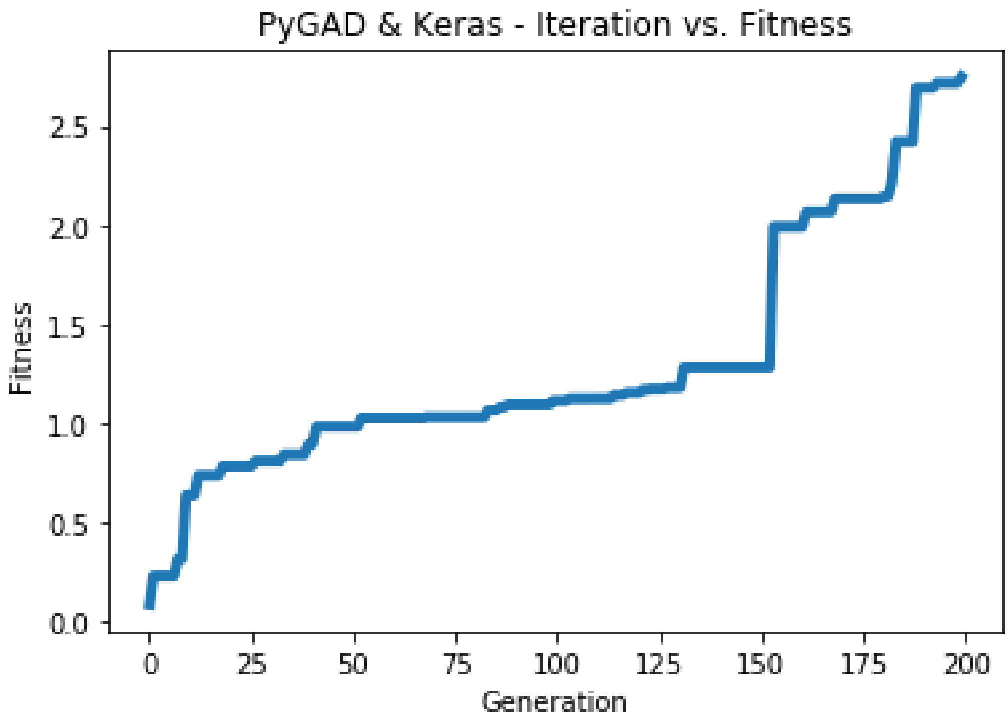
```
# Make predictions based on the best solution.
predictions = pygad.kerasga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
# print(f"Predictions : \n{predictions}")

# Calculate the categorical crossentropy for the trained model.
cce = tensorflow.keras.losses.CategoricalCrossentropy()
print(f"Categorical Crossentropy : {cce(data_outputs, predictions).numpy()}")

# Calculate the classification accuracy for the trained model.
ca = tensorflow.keras.metrics.CategoricalAccuracy()
ca.update_state(data_outputs, predictions)
accuracy = ca.result().numpy()
print(f"Accuracy : {accuracy}")
```

```
# Build the keras model using the functional API.
input_layer = tensorflow.keras.layers.Input(shape=(100, 100, 3))
conv_layer1 = tensorflow.keras.layers.Conv2D(filters=5,
                                             kernel_size=7,
                                             activation="relu")(input_layer)
max_pool1 = tensorflow.keras.layers.MaxPooling2D(pool_size=(5,5),
                                                 strides=5)(conv_layer1)
conv_layer2 = tensorflow.keras.layers.Conv2D(filters=3,
                                             kernel_size=3,
                                             activation="relu")(max_pool1)
flatten_layer  = tensorflow.keras.layers.Flatten()(conv_layer2)
dense_layer = tensorflow.keras.layers.Dense(15, activation="relu")(flatten_layer)
output_layer = tensorflow.keras.layers.Dense(4, activation="softmax")(dense_layer)

model = tensorflow.keras.Model(inputs=input_layer, outputs=output_layer)
```

`(100, 100, 3)` `pygad.cnn`

`tensorflow.keras.utils.to_categorical()`

```
import numpy

data_inputs = numpy.load("../data/dataset_inputs.npy")

data_outputs = numpy.load("../data/dataset_outputs.npy")
data_outputs = tensorflow.keras.utils.to_categorical(data_outputs)
```




```
Fitness value of the best solution = 2.7462310258668805
Index of the best solution : 0
Categorical Crossentropy :  0.3641354
Accuracy :  0.75
```

()

tensorflow.keras.preprocessing.image.ImageDataGenerator

```
import tensorflow as tf
import tensorflow.keras
import pygad.kerasga
import pygad

def fitness_func(ga_instanse, solution, sol_idx):
    global train_generator, data_outputs, keras_ga, model

    predictions = pygad.kerasga.predict(model=model,
                                        solution=solution,
                                        data=train_generator)
```

()

()

```
    cce = tensorflow.keras.losses.CategoricalCrossentropy()
    solution_fitness = 1.0 / (cce(data_outputs, predictions).numpy() + 0.00000001)

    return solution_fitness

def on_generation(ga_instance):
    print("Generation = {ga_instance.generations_completed}")
    print("Fitness    = {ga_instance.best_solution(ga_instance.last_generation_
↪fitness)[1]}")

# The dataset path.
dataset_path = r'../data/Skin_Cancer_Dataset'

num_classes = 2
img_size = 224

# Create a simple CNN. This does not gurantee high classification accuracy.
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Input(shape=(img_size, img_size, 3)))
model.add(tf.keras.layers.Conv2D(32, (3,3), activation="relu", padding="same"))
model.add(tf.keras.layers.MaxPooling2D((2, 2)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dropout(rate=0.2))
model.add(tf.keras.layers.Dense(num_classes, activation="softmax"))

# Create an instance of the pygad.kerasga.KerasGA class to build the initial␣
↪population.
keras_ga = pygad.kerasga.KerasGA(model=model,
                                 num_solutions=10)

data_generator = tf.keras.preprocessing.image.ImageDataGenerator()
train_generator = data_generator.flow_from_directory(dataset_path,
                                                     class_mode='categorical',
                                                     target_size=(224, 224),
                                                     batch_size=32,
                                                     shuffle=False)
# train_generator.class_indices
data_outputs = tf.keras.utils.to_categorical(train_generator.labels)

# Check the documentation for more information about the parameters: https://pygad.
↪readthedocs.io/en/latest/pygad.html#pygad-ga-class
initial_population = keras_ga.population_weights # Initial population of network␣
↪weights.

# Create an instance of the pygad.GA class
ga_instance = pygad.GA(num_generations=10,
                       num_parents_mating=5,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Start the genetic algorithm evolution.
ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
```

()

()

```
ga_instance.plot_fitness(title="PyGAD & Keras - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution(ga_instance.last_
↪generation_fitness)
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

predictions = pygad.kerasga.predict(model=model,
                                    solution=solution,
                                    data=train_generator)
# print(f"Predictions : \n{predictions}")

# Calculate the categorical crossentropy for the trained model.
cce = tensorflow.keras.losses.CategoricalCrossentropy()
print(f"Categorical Crossentropy : {cce(data_outputs, predictions).numpy()}")

# Calculate the classification accuracy for the trained model.
ca = tensorflow.keras.metrics.CategoricalAccuracy()
ca.update_state(data_outputs, predictions)
accuracy = ca.result().numpy()
print(f"Accuracy : {accuracy}")
```

pygad.torchga

,

pygad.torchga().

TorchGA

model_weights_as_vector()

model_weights_as_dict()

predict()

pygad.torchga.TorchGA

pygad.GA

```python
import torch

input_layer = torch.nn.Linear(3, 5)
relu_layer = torch.nn.ReLU()
output_layer = torch.nn.Linear(5, 1)

model = torch.nn.Sequential(input_layer,
                            relu_layer,
                            output_layer)
```

pygad.torchga.TorchGA

pygad.torchgaTorchGA

__init__()

pygad.torchga.TorchGA

- model
- num_solutions

pygad.torchga.TorchGApopulation_weights

- model
- num_solutions
- population_weights

TorchGA

pygad.torchga.TorchGA

create_population()

create_population()population_weights

pygad.torchga

pygad.torchga

pygad.torchga.model_weights_as_vector()

model_weights_as_vector()model

model

pygad.torch.model_weights_as_dict()

model_weights_as_dict()

model

weights_vector

state_dict()load_state_dict()'

pygad.torchga.predict()

predict()

model

solution

data

```
import torch
import torchga
import pygad

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, torch_ga, model, loss_function

    predictions = pygad.torchga.predict(model=model,
                                        solution=solution,
```

()

()

```
                                          data=data_inputs)

    abs_error = loss_function(predictions, data_outputs).detach().numpy() + 0.00000001

    solution_fitness = 1.0 / abs_error

    return solution_fitness

def on_generation(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

# Create the PyTorch model.
input_layer = torch.nn.Linear(3, 5)
relu_layer = torch.nn.ReLU()
output_layer = torch.nn.Linear(5, 1)

model = torch.nn.Sequential(input_layer,
                            relu_layer,
                            output_layer)
# print(model)

# Create an instance of the pygad.torchga.TorchGA class to build the initial␣
↪population.
torch_ga = torchga.TorchGA(model=model,
                           num_solutions=10)

loss_function = torch.nn.L1Loss()

# Data inputs
data_inputs = torch.tensor([[0.02, 0.1, 0.15],
                            [0.7, 0.6, 0.8],
                            [1.5, 1.2, 1.7],
                            [3.2, 2.9, 3.1]])

# Data outputs
data_outputs = torch.tensor([[0.1],
                             [0.6],
                             [1.3],
                             [2.5]])

# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 250 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
initial_population = torch_ga.population_weights # Initial population of network␣
↪weights

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

ga_instance.run()
```

()

()

```
# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & PyTorch - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

# Make predictions based on the best solution.
predictions = pygad.torchga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
print("Predictions : \n", predictions.detach().numpy())

abs_error = loss_function(predictions, data_outputs)
print("Absolute Error : ", abs_error.detach().numpy())
```

```
import torch

input_layer = torch.nn.Linear(3, 5)
relu_layer = torch.nn.ReLU()
output_layer = torch.nn.Linear(5, 1)

model = torch.nn.Sequential(input_layer,
                            relu_layer,
                            output_layer)
```

pygad.torchga.TorchGA

pygad.torchga.TorchGA

```
import pygad.torchga

torch_ga = torchga.TorchGA(model=model,
                           num_solutions=10)
```

```
import numpy

# Data inputs
data_inputs = numpy.array([[0.02, 0.1, 0.15],
                           [0.7, 0.6, 0.8],
                           [1.5, 1.2, 1.7],
                           [3.2, 2.9, 3.1]])
```

()

()

```
# Data outputs
data_outputs = numpy.array([[0.1],
                            [0.6],
                            [1.3],
                            [2.5]])
```

()

```
loss_function = torch.nn.L1Loss()

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, torch_ga, model, loss_function

    predictions = pygad.torchga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    abs_error = loss_function(predictions, data_outputs).detach().numpy() + 0.00000001

    solution_fitness = 1.0 / abs_error

    return solution_fitness
```

pygad.GA

pygad.GAinitial_population

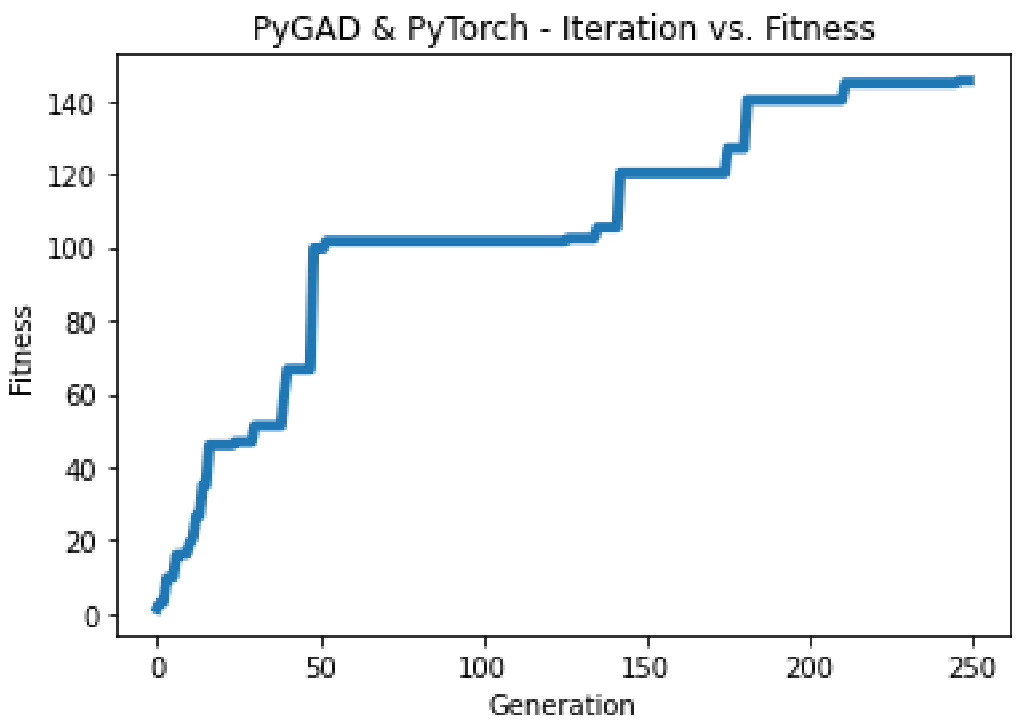
```
# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 250 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
initial_population = torch_ga.population_weights # Initial population of network␣
↪weights

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)
```

run()

```
ga_instance.run()
```

plot_fitness()

```
ga_instance.plot_fitness(title="PyGAD & PyTorch - Iteration vs. Fitness", linewidth=4)
```



best_solution()

```
# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")
```

```
Fitness value of the best solution = 145.42425295191546
Index of the best solution : 0
```

model_weights_as_dict()

```
predictions = pygad.torchga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
print("Predictions : \n", predictions.detach().numpy())
```

```
Predictions :
[[0.08401088]
```

()

()

```
 [0.60939324]
 [1.3010881 ]
 [2.5010352 ]]
```

```
abs_error = loss_function(predictions, data_outputs)
print("Absolute Error : ", abs_error.detach().numpy())
```

```
Absolute Error :  0.006876422
```

,

```
import torch
import torchga
import pygad

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, torch_ga, model, loss_function

    predictions = pygad.torchga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    solution_fitness = 1.0 / (loss_function(predictions, data_outputs).detach().
↪numpy() + 0.00000001)

    return solution_fitness

def on_generation(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

# Create the PyTorch model.
input_layer  = torch.nn.Linear(2, 4)
relu_layer = torch.nn.ReLU()
dense_layer = torch.nn.Linear(4, 2)
output_layer = torch.nn.Softmax(1)

model = torch.nn.Sequential(input_layer,
                            relu_layer,
                            dense_layer,
                            output_layer)
# print(model)

# Create an instance of the pygad.torchga.TorchGA class to build the initial␣
↪population.
torch_ga = torchga.TorchGA(model=model,
                           num_solutions=10)

loss_function = torch.nn.BCELoss()

# XOR problem inputs
```

()

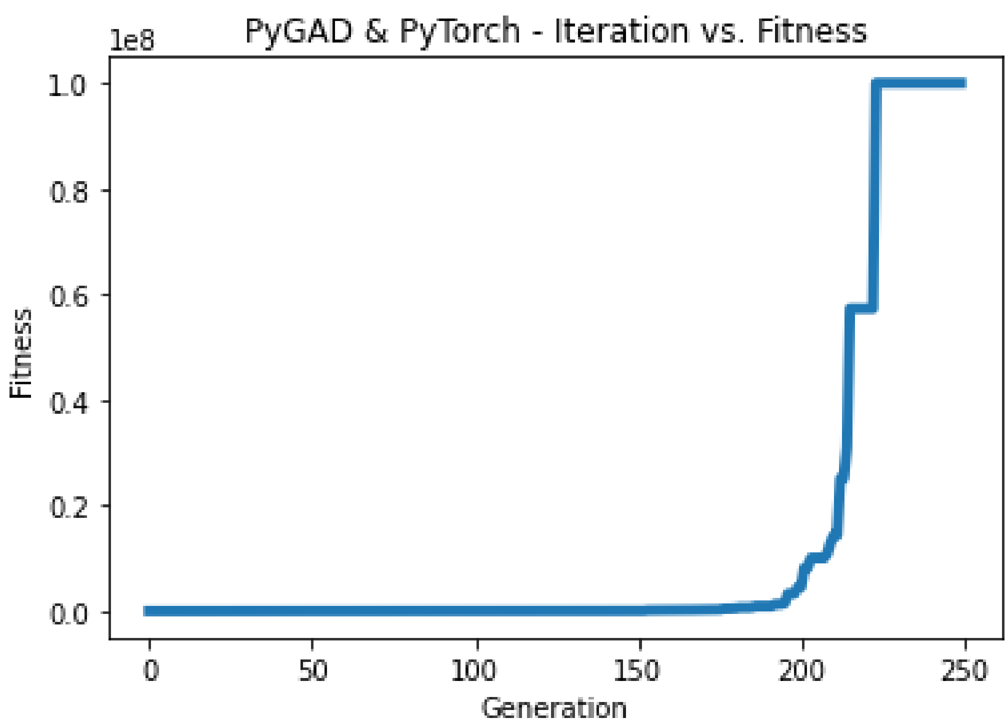
()

```
data_inputs = torch.tensor([[0.0, 0.0],
                            [0.0, 1.0],
                            [1.0, 0.0],
                            [1.0, 1.0]])

# XOR problem outputs
data_outputs = torch.tensor([[1.0, 0.0],
                             [0.0, 1.0],
                             [0.0, 1.0],
                             [1.0, 0.0]])

# Prepare the PyGAD parameters. Check the documentation for more information: https://pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 250 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating pool.
initial_population = torch_ga.population_weights # Initial population of network weights.

# Create an instance of the pygad.GA class
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Start the genetic algorithm evolution.
ga_instance.run()

# After the generations complete, some plots are showed that summarize how the outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & PyTorch - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

# Make predictions based on the best solution.
predictions = pygad.torchga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
print("Predictions : \n", predictions.detach().numpy())

# Calculate the binary crossentropy for the trained model.
print("Binary Crossentropy : ", loss_function(predictions, data_outputs).detach().numpy())

# Calculate the classification accuracy of the trained model.
a = torch.max(predictions, axis=1)
b = torch.max(data_outputs, axis=1)
accuracy = torch.sum(a.indices == b.indices) / len(data_outputs)
print("Accuracy : ", accuracy.detach().numpy())
```

```
input_layer  = torch.nn.Linear(2, 4)
relu_layer = torch.nn.ReLU()
dense_layer = torch.nn.Linear(4, 2)
output_layer = torch.nn.Softmax(1)

model = torch.nn.Sequential(input_layer,
                            relu_layer,
                            dense_layer,
                            output_layer)
```

```
# XOR problem inputs
data_inputs = torch.tensor([[0.0, 0.0],
                            [0.0, 1.0],
                            [1.0, 0.0],
                            [1.0, 1.0]])

# XOR problem outputs
data_outputs = torch.tensor([[1.0, 0.0],
                             [0.0, 1.0],
                             [0.0, 1.0],
                             [1.0, 0.0]])
```

```
loss_function = torch.nn.BCELoss()
```




```
100000000.00.0
```

```
Fitness value of the best solution = 100000000.0

Index of the best solution : 0

Predictions :
[[1.0000000e+00 1.3627675e-10]
 [3.8521746e-09 1.0000000e+00]
 [4.2789325e-10 1.0000000e+00]
 [1.0000000e+00 3.3668417e-09]]

Binary Crossentropy :  0.0

Accuracy :  1.0
```

()

```
import torch
import torchga
import pygad
import numpy

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, torch_ga, model, loss_function

    predictions = pygad.torchga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    solution_fitness = 1.0 / (loss_function(predictions, data_outputs).detach().
↪numpy() + 0.00000001)

    return solution_fitness

def on_generation(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

# Build the PyTorch model using the functional API.
input_layer = torch.nn.Linear(360, 50)
relu_layer = torch.nn.ReLU()
dense_layer = torch.nn.Linear(50, 4)
output_layer = torch.nn.Softmax(1)

model = torch.nn.Sequential(input_layer,
                            relu_layer,
                            dense_layer,
                            output_layer)

# Create an instance of the pygad.torchga.TorchGA class to build the initial␣
↪population.
torch_ga = torchga.TorchGA(model=model,
                           num_solutions=10)

loss_function = torch.nn.CrossEntropyLoss()
```

()

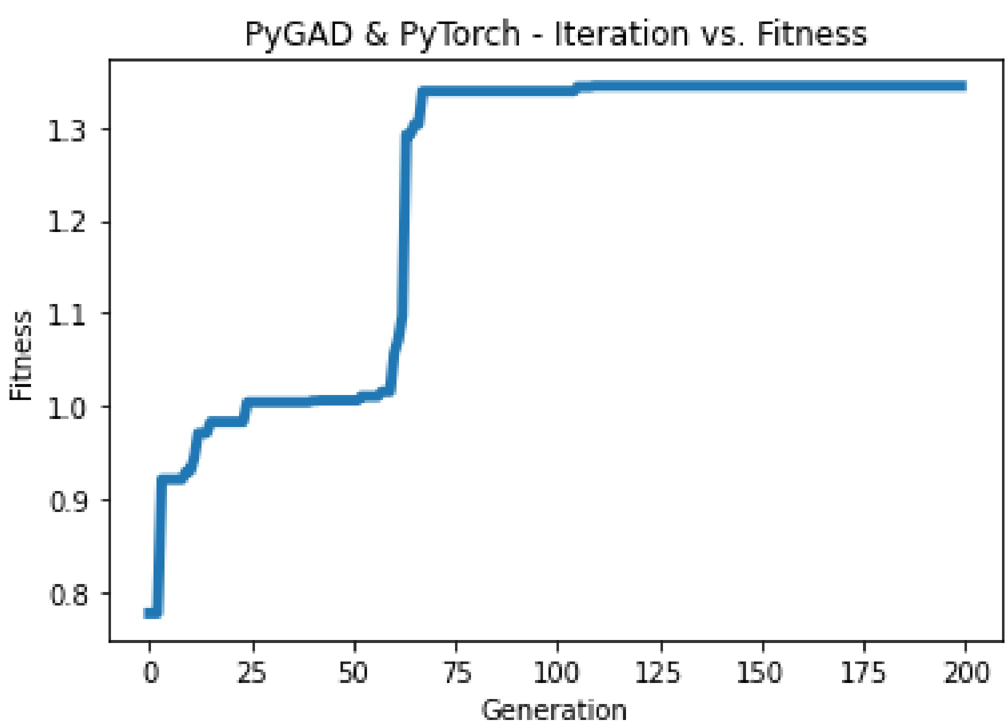
```python
# Data inputs
data_inputs = torch.from_numpy(numpy.load("dataset_features.npy")).float()

# Data outputs
data_outputs = torch.from_numpy(numpy.load("outputs.npy")).long()
# The next 2 lines are equivelant to this Keras function to perform 1-hot encoding: tensorflow.keras.utils.to_categorical(data_outputs)
# temp_outs = numpy.zeros((data_outputs.shape[0], numpy.unique(data_outputs).size), dtype=numpy.uint8)
# temp_outs[numpy.arange(data_outputs.shape[0]), numpy.uint8(data_outputs)] = 1

# Prepare the PyGAD parameters. Check the documentation for more information: https://pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 200 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating pool.
initial_population = torch_ga.population_weights # Initial population of network weights.

# Create an instance of the pygad.GA class
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Start the genetic algorithm evolution.
ga_instance.run()

# After the generations complete, some plots are showed that summarize how the outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & PyTorch - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

# Fetch the parameters of the best solution.
best_solution_weights = torchga.model_weights_as_dict(model=model,
                                                      weights_vector=solution)
model.load_state_dict(best_solution_weights)
predictions = model(data_inputs)
# print("Predictions : \n", predictions)

# Calculate the crossentropy loss of the trained model.
print("Crossentropy : ", loss_function(predictions, data_outputs).detach().numpy())

# Calculate the classification accuracy for the trained model.
accuracy = torch.sum(torch.max(predictions, axis=1).indices == data_outputs) / len(data_outputs)
print("Accuracy : ", accuracy.detach().numpy())
```

```
loss_function = torch.nn.CrossEntropyLoss()
```

()

(100, 100, 3)

```
import numpy

data_inputs = numpy.load("dataset_features.npy")

data_outputs = numpy.load("outputs.npy")
```




```
Fitness value of the best solution = 1.3446997034434534
Index of the best solution : 0
Crossentropy :  0.74366045
Accuracy :  1.0
```

()

```python
import torch
import torchga
import pygad
import numpy

def fitness_func(ga_instance, solution, sol_idx):
    global data_inputs, data_outputs, torch_ga, model, loss_function

    predictions = pygad.torchga.predict(model=model,
                                        solution=solution,
                                        data=data_inputs)

    solution_fitness = 1.0 / (loss_function(predictions, data_outputs).detach().numpy() + 0.00000001)

    return solution_fitness

def on_generation(ga_instance):
    print(f"Generation = {ga_instance.generations_completed}")
    print(f"Fitness    = {ga_instance.best_solution()[1]}")

# Build the PyTorch model.
input_layer = torch.nn.Conv2d(in_channels=3, out_channels=5, kernel_size=7)
relu_layer1 = torch.nn.ReLU()
max_pool1 = torch.nn.MaxPool2d(kernel_size=5, stride=5)

conv_layer2 = torch.nn.Conv2d(in_channels=5, out_channels=3, kernel_size=3)
relu_layer2 = torch.nn.ReLU()

flatten_layer1 = torch.nn.Flatten()
# The value 768 is pre-computed by tracing the sizes of the layers' outputs.
dense_layer1 = torch.nn.Linear(in_features=768, out_features=15)
relu_layer3 = torch.nn.ReLU()

dense_layer2 = torch.nn.Linear(in_features=15, out_features=4)
output_layer = torch.nn.Softmax(1)

model = torch.nn.Sequential(input_layer,
                            relu_layer1,
                            max_pool1,
                            conv_layer2,
                            relu_layer2,
                            flatten_layer1,
                            dense_layer1,
                            relu_layer3,
                            dense_layer2,
                            output_layer)

# Create an instance of the pygad.torchga.TorchGA class to build the initial population.
torch_ga = torchga.TorchGA(model=model,
                           num_solutions=10)
```

()

()

```
loss_function = torch.nn.CrossEntropyLoss()

# Data inputs
data_inputs = torch.from_numpy(numpy.load("dataset_inputs.npy")).float()
data_inputs = data_inputs.reshape((data_inputs.shape[0], data_inputs.shape[3], data_
↪inputs.shape[1], data_inputs.shape[2]))

# Data outputs
data_outputs = torch.from_numpy(numpy.load("dataset_outputs.npy")).long()

# Prepare the PyGAD parameters. Check the documentation for more information: https://
↪pygad.readthedocs.io/en/latest/pygad.html#pygad-ga-class
num_generations = 200 # Number of generations.
num_parents_mating = 5 # Number of solutions to be selected as parents in the mating␣
↪pool.
initial_population = torch_ga.population_weights # Initial population of network␣
↪weights.

# Create an instance of the pygad.GA class
ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       initial_population=initial_population,
                       fitness_func=fitness_func,
                       on_generation=on_generation)

# Start the genetic algorithm evolution.
ga_instance.run()

# After the generations complete, some plots are showed that summarize how the␣
↪outputs/fitness values evolve over generations.
ga_instance.plot_fitness(title="PyGAD & PyTorch - Iteration vs. Fitness", linewidth=4)

# Returning the details of the best solution.
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print(f"Fitness value of the best solution = {solution_fitness}")
print(f"Index of the best solution : {solution_idx}")

# Make predictions based on the best solution.
predictions = pygad.torchga.predict(model=model,
                                    solution=solution,
                                    data=data_inputs)
# print("Predictions : \n", predictions)

# Calculate the crossentropy for the trained model.
print("Crossentropy : ", loss_function(predictions, data_outputs).detach().numpy())

# Calculate the classification accuracy for the trained model.
accuracy = torch.sum(torch.max(predictions, axis=1).indices == data_outputs) /␣
↪len(data_outputs)
print("Accuracy : ", accuracy.detach().numpy())
```

```
input_layer = torch.nn.Conv2d(in_channels=3, out_channels=5, kernel_size=7)
relu_layer1 = torch.nn.ReLU()
max_pool1 = torch.nn.MaxPool2d(kernel_size=5, stride=5)
```

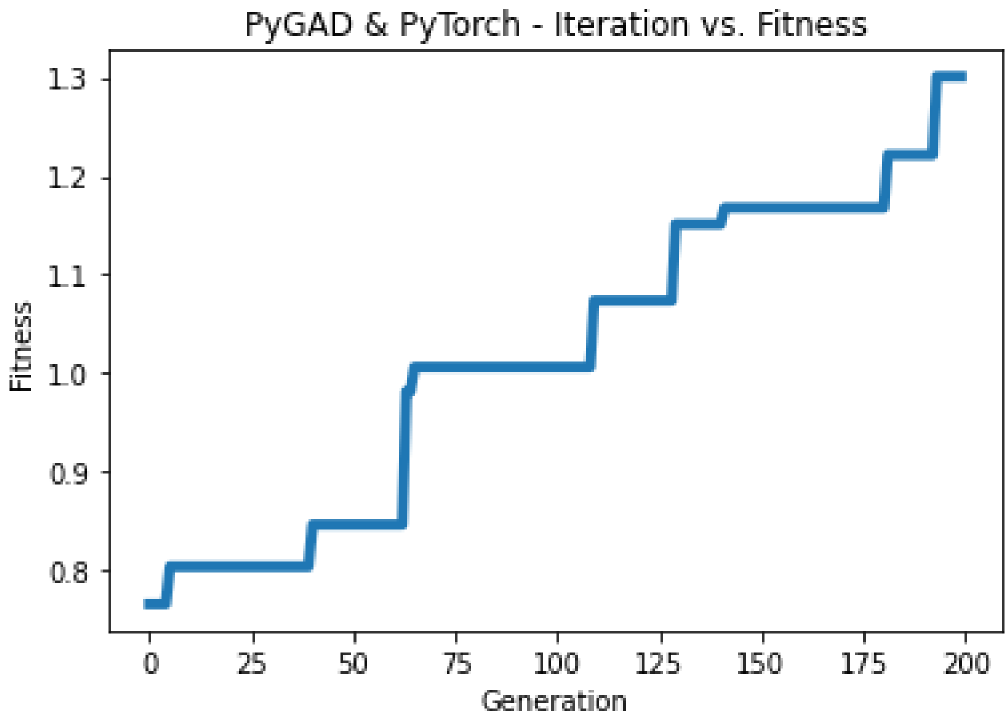
()

()

```
conv_layer2 = torch.nn.Conv2d(in_channels=5, out_channels=3, kernel_size=3)
relu_layer2 = torch.nn.ReLU()

flatten_layer1 = torch.nn.Flatten()
# The value 768 is pre-computed by tracing the sizes of the layers' outputs.
dense_layer1 = torch.nn.Linear(in_features=768, out_features=15)
relu_layer3 = torch.nn.ReLU()

dense_layer2 = torch.nn.Linear(in_features=15, out_features=4)
output_layer = torch.nn.Softmax(1)

model = torch.nn.Sequential(input_layer,
                            relu_layer1,
                            max_pool1,
                            conv_layer2,
                            relu_layer2,
                            flatten_layer1,
                            dense_layer1,
                            relu_layer3,
                            dense_layer2,
                            output_layer)
```

`(100, 100, 3)` `pygad.cnn`

```
import numpy

data_inputs = numpy.load("dataset_inputs.npy")

data_outputs = numpy.load("dataset_outputs.npy")
```




```
Fitness value of the best solution = 1.3009520689219258
Index of the best solution : 0
Crossentropy :  0.7686678
Accuracy :  0.975
```

PyGAD

fitness_func

(init_range_low init_range_high)

__code__

sol_idx

initial_populationNonesol_per_popnum_genes

sol_per_popnum_genesNone

callback_generation

best_solution()

```
solution, solution_fitness, solution_idx = ga_instance.best_solution()
print("Parameters of the best solution :", solution)
print("Fitness value of the best solution :", solution_fitness, "\n")
print("Index of the best solution :", solution_idx, "\n")
```

best_solution_generationrun()

```
print("Best solution reached after {best_solution_generation} generations.".
↪format(best_solution_generation=ga_instance.best_solution_generation))
```

best_solution_fitnessbest_solutions_fitness().

()

generations_completed0None

mutation_by_replacement(mutation_type="random"). muta-tion_by_replacement=TrueFalse

mutation_type="random"mutation_by_replacement=False()

mutation_type="random"mutation_by_replacement=True()

Nonemutation_typecrossover_typeNone

pygad.cnn

pygad.gacnn

pygad.plot_result()titlexlabelylabel

pygad.nn

pygad.nn.predict_outputs()pygad.nn.predict()

pygad.nn.train_network()pygad.nn.train()

delay_after_gen0.0

callback_generationstoprun()

num_generationscallback_generationstop

callback_generation

```
def func_generation(ga_instance):
 if ga_instance.best_solution()[1] >= 70:
     return "stop"
```

pygad.GAcrossover_probabilitymutation_probability
crossover_probability
mutation_probability

linewidthplot_result()
,

() () () () () ()

gene_spacepygad.GA"'<https://pygad.readthedocs.io/en/latest/pygad_more.html#more-about-the-gene-space-parameter>'

initial_population

gene_typeintfloatgene_spaceNone

on_starton_fitnesson_parentson_crossoveron_mutationon_generationon_stop

learning_ratepygad.nn.train()

problem_typepygad.nn.train()pygad.nn.predict()

"None""sigmoid""relu""softmax""None"

pygad.nn

problem_typepygad.nn.train()pygad.nn.predict()"regression"

"None"-infinity+infinity

pygad.nn

pygad.gann

problem_typepygad.nn.train()pygad.nn.predict()"regression"

output_activationpygad.gann.GANN"None"

pygad.gann

problem_type"classification"(). ().

problem_typeregression

kerasga

crossover_probability

best_solutions_fitness

save_best_solutionsFalseTruebest_solutionsFalsebest_solutions

crossover_type"scattered"

gene_space

(gene_type, crossover_probability, mutation_probability, delay_after_gen) intfloatnumpy.intnumpy.int8numpy.int16numpy.int32numpy.int64numpy.float numpy.float16numpy.float32numpy.float64

pygad.torchga

“” ():

run()best_solution_fitness

parent_selection_typesss(), keep_parents

mutation_percent_genes"default"mutation_percent_genes"default"

mutation_percent_genes>0<=100

warningsprint()

boolsuppress_warningspygad.GAFalse

adaptive_mutation_population_fitness()

best_solution()pop_fitnessNonecal_pop_fitness()

gene_spaceNone(), init_range_lowinit_range_highgene_typeNone[..., None, ...] [..., [..., None, ...], ...]

gene_spacegene_type

numpy.uint

pygad.kerasgamodel_weights_as_vector()trainable' trainableTrue(trainable=True), trainableFalse(trainable=False)

save_best_solutions=True

gene_spacelowhighgene_space=[{'low': 1, 'high': 5}, {'low': 0.2, 'high': 0.81}]() () () ().

plot_result()

() gene_space[0, 1]

last_generation_fitnesslast_generation_parentslast_generation_offspring_crossover last_generation_offspring_mutationon_generation()

initial_populationinitial_populationgene_typeinitial_population((1, 1), (3, 3), (5, 5), (7, 7))intgene_typefloatintintinitial_populationgene_type

[]

(),

boolallow_duplicate_genesTrueFalse

last_generation_fitnesslast_generation_fitness

Nonecrossover_typemutation_type

gene_typelist/tuple/numpy.ndarraygene_type"'<https://pygad.readthedocs.io/en/latest/pygad_more.html#more-about-the-gene-type-parameter>'

boolgene_type_singlepygad.GATruegene_typegene_typelist/tuple/numpy.ndarray gene_type_singleFalse

mutation_by_replacementgene_spaceNonegene_space=[None, [5, 6]]mutation_by_replacementNone

Nonegene_space(gene_space=[None, [5, 6]]), Nonegene_space

gene_type

save_best_solutionsTrueibest_solutionsi+1best_solutions

last_generation_parents_indices

last_generation_fitnesslast_generation_parents_indices

Nonegene_space(gene_space=[[1, 2, 3], [5, 6, None]]), Nonegene_space

gene_space"step""low""high"{"low": 0, "high": 30, "step": 2}() "'<https://pygad.readthedocs.io/en/latest/pygad_more.html#more-about-the-gene-space-parameter>'

predict()pygad.kerasgapygad.torchga

stop_criteriastrreachsaturatereachrun()reach"reach_40">saturatesaturate "saturate_7"run()

Falsesave_solutionspygad.GATruesolutions

plot_result()plot_fitness()

plot_fitness()pygad.GAfont_size=14save_dir=Nonecolor="#3870FF" plot_type="plot"font_sizesave_dirNonecolorplot_type"plot"(), "scatter", "bar"

titleplot_fitness()"PyGAD - Generation vs. Fitness""PyGAD - Iteration vs. Fitness"

plot_new_solution_rate()plot_fitness()save_solutions=Truepygad.GA'

plot_genes()plot_fitness()graph_typefill_colorsolutionsgraph_type"plot" (), "boxplot", "histogram"fill_colorgraph_type"boxplot""histogram"solutions "all""best"

gene_typefloatfloatlisttuplenumpy.ndarray[float, 2]0.12340.12"'<https://pygad.readthedocs.io/en/latest/pygad_more.html#more-about-the-gene-type-parameter>'

keep_parents

kerasgatorchga

mutation_typecrossover_typeparent_selection_typepygad.GA

tqdm

```
import pygad
import numpy
import tqdm

equation_inputs = [4,-2,3.5]
desired_output = 44

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
```

()

()

```
    return fitness

num_generations = 10000
with tqdm.tqdm(total=num_generations) as pbar:
    ga_instance = pygad.GA(num_generations=num_generations,
                           sol_per_pop=5,
                           num_parents_mating=2,
                           num_genes=len(equation_inputs),
                           fitness_func=fitness_func,
                           on_generation=lambda _: pbar.update(1))

    ga_instance.run()

ga_instance.plot_result()
```

ga_instance(save()

```
ga_instance.save("test")
```

on_generationon_generation_progress()

```
import pygad
import numpy
import tqdm

equation_inputs = [4,-2,3.5]
desired_output = 44

def fitness_func(ga_instance, solution, solution_idx):
    output = numpy.sum(solution * equation_inputs)
    fitness = 1.0 / (numpy.abs(output - desired_output) + 0.000001)
    return fitness

def on_generation_progress(ga):
    pbar.update(1)

num_generations = 100
with tqdm.tqdm(total=num_generations) as pbar:
    ga_instance = pygad.GA(num_generations=num_generations,
                           sol_per_pop=5,
                           num_parents_mating=2,
                           num_genes=len(equation_inputs),
                           fitness_func=fitness_func,
                           on_generation=on_generation_progress)

    ga_instance.run()

ga_instance.plot_result()

ga_instance.save("test")
```

solutionssolutions_fitnesssave_solutionsTruesolutions_fitness

(solutions, solutions_fitness, best_solutions, best_solutions_fitness) run() run()

(mutation_type="adaptive"). https://github.com/ahmedfgad/GeneticAlgorithmPython/issues/65 previous_generation_fitnesspygad.GAlast_generation_fitness

cal_pop_fitness()' previous_generation_fitness' ()

gene_space[(),]' ()

allow_duplicate_genes(mutation_type=None).

tournament_selection()

save_solutions=True

parallel_processingpygad.GA

run_completedFalse

run()self.best_solutions, self.best_solutions_fitness, self.solutions, self.solutions_fitnessrun()run()

()

crossover_type=None

keep_elitism

last_generation_elitism

random_seed

pygad.TorchGA