به نام خدا

مهدی فقهی

۴۰۱۷۲۲۱۳۶

گزارش کار پروژه چهارم PG

در ابتدا منبعی که به کمک آن توانستم یک پروژه را پیاده سازی کردم را معرفی می‌کنم .
من از این لینک گیت‌هاب استفاده کردم و در ادامه قسمت‌های مختلف کد را شرح خواهم داد و در انتها نیز نتایج بدست آمده را بررسی خواهیم کرد .
خوب برای اجرای این پروژه ما یک فایل genetic algorithm داریم که توابع و کلاس‌های موردنیاز خود را آنجا پیاده سازی کرده‌ایم و با اجرای فایل experiment.ipynb می‌توانید نتایج گزارش شده در گزارش را مشاهده نمایید .

سه قسمت در فایل genetic algorithm داریم .

1. algorithm.py :

در این فایل ، موارد مختلف الگوریتم یادگیری ژنتیک را همانند selection،crossover,mutation... را خواهیم داشت که در قسمت train به کمک تعداد تکرارهای انجام شده، الگوریتم به سمت پیدا کردن یک فرمول مناسب برای تخمین حرکت خواهد کرد.

2.chromosome.py:

در این فایل یک کلاس کروموزم تعریف می‌کنیم که یک حداکثر عمق دارد، یک لیست از توابعی مانند sin,cos,exp, ... و یک لیست از اعمال ریاضی مانند +,-,/,... را می‌گیرد و همچنین متغیرهایی مانند x1, ,x2,x3 ... تحت اسم terminalها که به کمک آنان یک کروموزم تصادفی خواهد ساخت و یک fitness که میزان کارایی که این کروموزم بر اساس ژن‌هایی که دارد که همان فرمول ریاضی ساخته‌شده توسط این کروموزم هست، چه مقدار دقتی بدست خواهد آورد این معیار دقت بین صفر تا یک است .

3.population.py:

در قسمت population ما یک لیست از کروموزم‌ها با توجه به شرایطی که می‌خواهیم می‌سازیم .
در ادامه به صورت دقیق‌تر کد را بررسی خواهیم کرد .

Algorithm:

```python
class Algorithm:
    """
    Class representing the algorithm
    """

    def __init__(self, population, iterations, inputs, outputs, epoch_feedback=500):
        """
        Constructor for Alrogithm class
        @param: population - population for the current algorithm
        @param: iterations - number of iterations for the algorithm
        @param: inputs - inputs
        @param: outputs - outputs
        @param: epoch_feedback - number of epochs to show feedback
        """
        self.population = population
        self.iterations = iterations
        self.inputs = inputs
        self.outputs = outputs
        self.epoch_feedback = epoch_feedback
        self.fitness = []
```

در قسمت population ما یک جمعیت ساخته شده توسط کلاس population را می‌سازیم و به این قسمت پاس می‌دهیم .

در قسمت iterations مشخص می کنیم که در هر بار انجام train حداکثر چه مقدار نسل تولید شود تا در نهایت به جواب ۱ fitness برسد .

در قسمت input ما ورودی و در قسمت outputs خروجی مربوط به این ورودی را خواهیم داشت که به کمک آن میزان fitness را برای هر کدام از کروموزم‌ها بدست آوریم .

در قسمت epoch feedback، می گوید بعد از چه مقدار گذر نسل یک گزارش از حاصل بدهد .

در قسمت fitness بهترین کارایی بدست آمده در هر نسل را نگه می‌داریم .

```
def __one_step(self):
    """
    Function to do one step of the algorithm
    """
    mother = self.__selection()
    father = self.__selection()
    # mother = self.__roulette_selection(self.population)
    # father = self.__roulette_selection(self.population)
    child = self.__cross_over(mother, father)
    child = self.__mutate(child)
    child.calculate_fitness(self.inputs, self.outputs)
    self.population = self.__replace_worst(child)

def train(self):
    for i in range(len(self.population.list)):
        self.population.list[i].calculate_fitness(self.inputs, self.outputs)
    for i in range(self.iterations):
        best_so_far = self.__get_best()
        self.fitness.append(best_so_far.fitness)
        if i % self.epoch_feedback == 0:
            print(f"Best function: {best_so_far}")
            print(f"Best fitness: {best_so_far.fitness}")
        self.__one_step()

        if best_so_far.fitness == 1:
            break
    return self.__get_best()
```

در قسمت train ابتدا به ازای تمام کروموزم‌های موجود در لیست population میزان fitness را به ازای ورودی و خروجی حاصل بدست خواهیم آورد .

سپس به تعدادی که می‌خواهیم نسل بوجود آوریم، عملیات یادگیری را شروع می کنیم .

سپس بهترین کروموزم را پیدا خواهیم کرد در هر نسل و میزان کارایی آن را ذخیره می کنیم .

اگر میزان کارایی به یک برسد روند را متوقف خواهیم کرد .

در هر دوره موارد زیر را انجام می‌دهیم .

یک کروموزم به عنوان مادر و پدر انتخاب می کنیم سپس به کمک عملیات crossover از کروموزم‌های مربوط به مادر و پدر فرزند را تولید می کنیم و در نهایت از فرزندان بوجود آمده به کمک عملیات mutation تصمیم می گیریم که فرزند را با جهش ژنی، تغییر دهیم یا نه .

سپس فرزند را به جای بدترین کروموزم که کارایی دارد جایگزین می‌نماییم . در پایین شیوه پیدا کردن بدترین کروموزم و شیوه جایگزینی کروموزم ساخته شده فرزند با این کروموزم را می‌بینیم .

```
def __get_worst(self):

    population = self.population
    """
    Function to get the worst chromosome of the population
    @param: population -
    @return: worst chromosome from the population
    """
    worst = population.list[0]
    for i in range(1, len(population.list)):
        if population.list[i].fitness < worst.fitness:
            worst = population.list[i]

    return worst

def __replace_worst(self, chromosome):

    population = self.population
    """
    Function to change the worst chromosome of the population with a new one
    @param: population - population
    @param: chromosome - chromosome to be added
    """
    worst = self.__get_worst()
    if chromosome.fitness > worst.fitness:
        for i in range(len(population.list)):
            if population.list[i].fitness == worst.fitness:
                population.list[i] = chromosome
                break
    return population
```

در قسمت selection یک نمونه گیری رندوم از نمونه‌ها خواهیم داشت که از بین این نمونه‌ها بهترین نمونه از لحاظ کارایی را انتخاب می کنیم .

```
def __selection(self):

    population = self.population
    num_sel = self.population.num_selected
    """
    Function to select a member of the population for crossing over
    @param: population - population of chromosomes
    @param: num_sel - number of chromosome selected from the population
    @return: the selected chromosome
    """

    sample = random.sample(population.list, num_sel)
    best = sample[0]
    for i in range(1, len(sample)):
        if population.list[i].fitness > best.fitness:
            best = population.list[i]

    return best
```

در قسمت crossover ابتدا terminal , func و میزان حداکثر عمق فرزند را متناسب با مادر قرار خواهیم داد .
سپس به صورت رندوم از کروموزم مادر و پدر موقعیت یک ژن را برمی گردانیم .

```
def __cross_over(self, mother, father):

    max_depth = self.population.max_depth
    """
    Function to cross over two chromosomes in order to obtain a child
    @param mother: - chromosome
    @param father: - chromosome
    @param max_depth - maximum_depth of a tree
    """
    child = Chromosome(mother.terminal_set, mother.func_set, mother.depth, None)
    start_m = np.random.randint(len(mother.gen))
    start_f = np.random.randint(len(father.gen))
    end_m = self.__traversal(start_m, mother)
    end_f = self.__traversal(start_f, father)
    child.gen = mother.gen[:start_m] + father.gen[start_f: end_f] + mother.gen[end_m:]
    if child.get_depth() > max_depth and random.random() > 0.2:
        child = Chromosome(mother.terminal_set, mother.func_set, mother.depth)
    return child
```

در قسمت traversal اگر ژن انتخابی یک terminal مانند x1, x2 ,... باشد تنها همان متغیر را برمی گردانیم اگر یک تابع مانند sin , cos یا exp باشد که تنها یک ورودی می گیرد درخت را تا جایی که به یک متغیر terminal برسیم به صورت بازگشتی طی میکنیم و موقعیت آن را برمی گردانیم و این گونه آن بخش از فرمول را گویی خواهیم داشت .
اگر عملگرهایی مانند +، * ,... را داشته باشیم که دو طرف دارند ابتدا طرف سمت چپ درخت را تا جایی که به یک متغیر terminal برسیم را طی می کنیم و سپس به سراغ سمت راست می‌رویم و همین گونه و سپس جایگاه آخرین ژن که یک terminal در سمت راست هست را برمی گردانیم .

```
def __traversal(self, poz, chromosome: Chromosome):
    """
    Function to traverse the tree from the given poz
    @param: poz - start position
    @chromosome: chromosome to be traversed
    """
    if chromosome.gen[poz] in chromosome.terminal_set:
        return poz + 1
    elif chromosome.gen[poz] in chromosome.func_set[1]:
        return self.__traversal(poz + 1, chromosome)
    else:
        new_poz = self.__traversal(poz + 1, chromosome)
        return self.__traversal(new_poz, chromosome)
```

در نهایت همانطور که که تابع crossover مشاهده می کنید براساس آن بخش از ژن‌های پدر و مادر که به صورت رندوم بدست آمده یک، ژن‌های فرزند را از پیوند قسمت‌های مختلف درخت پدر و مادر می‌سازیم .
اگر درختی که برای فرزند ساختیم مقدار عمق آن بیشتر از عمق مدنظر ما بود این شانس را دارد که به احتمال ۲۰ درصد آن را نگه داریم در غیر این صورت یک ژن با عمق دلخواه را به عنوان فرزند خواهیم ساخت .

```python
def __mutate(self, chromosome: Chromosome):
    """
    Function to mutate a chromosome
    @param: chromsome - chromosome to be mutated
    @return: the mutated chromosome
    """
    poz = np.random.randint(len(chromosome.gen))
    if chromosome.gen[poz] in chromosome.func_set[1] + chromosome.func_set[2]:
        if chromosome.gen[poz] in chromosome.func_set[1]:
            chromosome.gen[poz] = random.choice(chromosome.func_set[1])
        else:
            chromosome.gen[poz] = random.choice(chromosome.func_set[2])
    else:
        chromosome.gen[poz] = random.choice(chromosome.terminal_set)
    return chromosome
```

در قسمت جهش نیز همانطور که مشاهده می‌کنید یک ژن را به صورت رندوم از کروموزم انتخاب می‌کنیم و سپس آن را با یک ژن که ویژگی مشابه به آن را داشته باشد مثلا اگر جمع باشد با ضرب، تقسیم و اگر sin باشد با توابعی که تنها یک ورودی می‌گیرد عوض می‌کنیم .

chromosome:

```python
class Chromosome:
    """
    Class for representing a chromosome
    """

    def __init__(self, terminal_set, funct_set, depth, method='full'):
        """
        Constructor for Chromosome class
        @param: depth - tree depth
        @param: method - method to generate the tree, default is full
        @param: terminal_set - set of terminals
        @param: funct_set - set of functions
        """
        self.depth = depth
        self.gen = []
        self.terminal_set = terminal_set
        self.func_set = funct_set
        self.fitness = None
        if method == 'grow':
            self.grow()
        elif method == 'full':
            self.full()
```

در این قسمت ساختار یک کروموزوم را بررسی می‌کنیم هر کروموزوم یک حداکثر عمق دارد .
یک لیست که به صورت PreOrder ساختار درختی کروموزم را در خود ذخیره کرده است که در اینجا هر node یک gen است .

سپس ترمینال‌های که در ساخت درخت استفاده شده اند (x1, x2, x3 , ...) را در درخت داریم و همچنین در قسمت func_set توابع و اعلام ریاضی که اجازه ساخت و استفاده از آن‌ها را در ساختار ژن‌های این کروموزم را داریم مشخص می‌کنیم و در نهایت شیوه ساخت ژن را مشخص می‌کنیم .

مهمترین قسمت یک کروموزم در این قسمت نحوه محاسبه fitness آن است که براساس کد داریم .

```
def calculate_fitness(self, inputs, outputs):
    """
    Function to calculate the fitness of a chromosome
    @param inputs: inputs of the function we want to predict
    @param outputs: outputs of the function we want to predict
    @return: the chromosome's fitness (calculated based on MSE)
    """
    diff = 0
    for i in range(len(inputs)):
        try:
            diff += (self.eval(inputs[i])[0] - outputs[i][0]) ** 2
        except RuntimeWarning:
            self.gen = []
            if random.random() > 0.5:
                self.grow()
            else:
                self.full()
            self.calculate_fitness(inputs, outputs)

    if len(inputs) == 0:
        return 1e9
    error = diff / (len(inputs))
    self.fitness = 1.0 / (1.0 + error)

    return self.fitness
```

در اینجا براساس ورودی‌های مختلف، خروجی‌های حاصل از این کروموزوم را حساب می‌کنیم و مجذور خطا نسبت به مقدار واقعی را برای تمامی ورودی‌ها حساب کرده و از آن میانگین می‌گیریم در نهایت برای بدست آوردن fitness حاصل بدست آمده برای خطا به صورت تجمعی را با یک جمع کرده و حاصل بدست آمده را ذخیره می‌کنیم .
سپس یک را تقسیم بر این مقدار می‌کنیم و به عنوان fitness که عددی بین صفر تا یک هست برمی‌گردانیم . اگر خطا برابر با صفر باشد یک و در غیر اینصورت عددی بین صفر تا یک است .
برای بدست آوردن حاصل یک مقدار ورودی بر اساس یک کروموزوم تابع eval را داریم که با پیمایش inorder درخت حاصل را بدست می‌آورد .

```python
def eval(self, input, poz=0):
    """
    Function to evaluate the current chromosome with a given input
    @param: input - function input (x0, x1... xn)
    @poz: current_position in genotype
    @return:
    """
    if self.gen[poz] in self.terminal_set:
        return input[int(self.gen[poz][1:])], poz
    elif self.gen[poz] in self.func_set[2]:
        poz_op = poz
        left, poz = self.eval(input, poz + 1)
        right, poz = self.eval(input, poz + 1)
        if self.gen[poz_op] == '+':
            return left + right, poz
        elif self.gen[poz_op] == '-':
            return left - right, poz
        elif self.gen[poz_op] == '*':
            return left * right, poz
        elif self.gen[poz_op] == '^':
            return left ** right, poz
        elif self.gen[poz_op] == '/':
            return left / right, poz
    else:
        poz_op = poz
        left, poz = self.eval(input, poz + 1)
        if self.gen[poz_op] == 'sin':
            return np.sin(left), poz
        elif self.gen[poz_op] == 'cos':
            return np.cos(left), poz
        elif self.gen[poz_op] == 'ln':
            return np.log(left), poz
        elif self.gen[poz_op] == 'sqrt':
            return np.sqrt(left), poz
        elif self.gen[poz_op] == 'tg':
            return np.tan(left), poz
```

همانطور که مشاهده می‌کنید تمامی توابع و حالت‌هایی که ممکن است ببینیم به عنوان یک ژن را مشخص کرده‌ایم و حاصل را به ازای هر کدام از این موارد حساب می‌کنیم .

حال با دانستن این موارد از پیاده سازی صرفا نیاز است که به سراغ پیاده سازی موارد خواسته شده در تمرین .

ابتدا به عنوان یک حالت ساده تابع **x^3 - 2*x** را خواستم که به کمک این الگوریتم پیاده سازی کنم .

```
"""
ALGORITHM PARAMETERS
"""

SIZE = 1 # number of variables
MAX_DEPTH = 10 #maximum depth of a tree
FUNCTIONS = {1: ['sin','cos'], 2:['+', '-', '*', '/','^']} #unary and binary functions
TERMINAL_SET = ['x'+str(i) for i in range(SIZE)] #create parameters name
DEPTH = 4 #initial depth of trees
POP_SIZE = 100 #number of chromosomes in population
NUM_FOR_SELECTION = 100//3 #number of chromosomes to be chosen for selection
NUMBER_ITERS = 100000 #number of iterations
```

```
"""
DEFINE A FUNCTION TO PREDICT
"""
def f(x):
    return x**3 - 2*x

X = [[x] for x in np.arange(0, 10, 0.01)] #function inputs
y = [[f(x[0])] for x in X] #function outputs

pop = Population(POP_SIZE, NUM_FOR_SELECTION, FUNCTIONS, TERMINAL_SET, 6, MAX_DEPTH) #create the population
alg = Algorithm(pop, NUMBER_ITERS, X, y, epoch_feedback=500)  #create the algorithm
start_time = time.time()
best = alg.train() #train the algorithm
end_time = time.time()

elapsed_time = end_time - start_time

print(f"Elapsed time: {elapsed_time:.2f} seconds")
```

تعداد متغیرهای ورودی ما یک دانه است در این آزمایش و همچنین حداکثر عمق را برابر با ۱۰ و عملیات‌های مجاز را نیز برابر با ^,/,+,-,* و sin , cos است .

و تعداد حداکثر نسلی که می‌خواهیم تا انتهای آن جلو برویم و فرزند بوجود آوریم را مشخص می‌کنیم به همراه تعداد افراد جامعه که در اینجا ۱۰۰ تا نمونه کروموزم را برای انجام این کار می‌سازیم .

سپس تابع مورد نظر خود را می‌سازیم و برحسب xهای تصادفی y های مختلف را بدست می‌آورم .

سپس به کمک الگوریتم و همانطور که در قسمت قبل توضیح دادم سعی می‌کنیم فرمولی که بیشترین شباهت را به تابع دارد را بسازم .

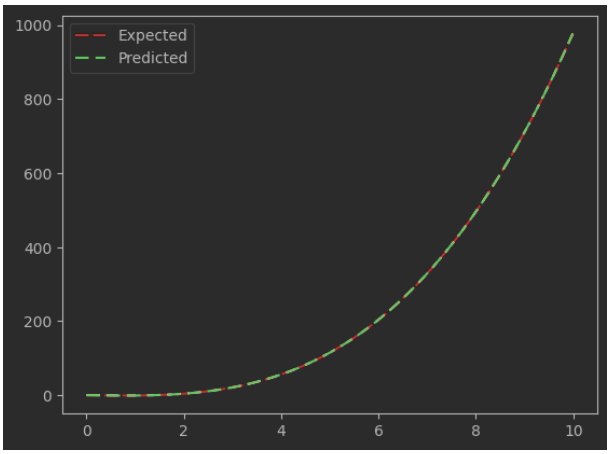
نتایج :

```
((((x0 * x0) * x0) - x0) - x0)
1.0
```

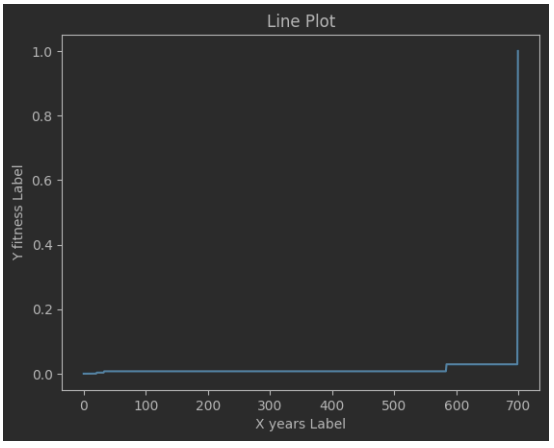
در شکل بالا تابعی که توانسته‌ایم به کمک الگوریتم پیدا کنیم را مشاهده می‌کنید و عدد زیر آن نیز برابر با میزان fitness است که برابر با یک است .

```
Elapsed time: 4.15 seconds
```

و همانطور که می‌بیند ۴ ثانیه بیشتر طول نکشیده است تا به این تابع برسیم .



شکل بالا نشان می‌دهد که هر دو نمودار روی همدیگر افتاده‌اند که به معنای دقت صد درصدی است .



حاصل بعد از گذر 700 نسل بدست آمده است .

در شکل زیر نیز نمودار پیش‌رفت fitness را در نسل‌های مختلف می‌بینید همانطور که می‌بینید فقط ۷۰۰ نسل گذشته تا به دقت ۱۰۰ درصد برسیم .

در ادامه تابع را یک مقدار سخت‌تر می‌کنم .

```
"""
ALGORITHM PARAMETERS
"""

SIZE = 1 # number of variables
MAX_DEPTH = 6 #maximum depth of a tree
FUNCTIONS = {1: ['sin','cos'], 2:['+', '-', '*', '/','^']} #unary and binary functions
TERMINAL_SET = ['x'+str(i) for i in range(SIZE)] #create parameters name
DEPTH = 2 #initial depth of trees
POP_SIZE = 300 #number of chromosomes in population
NUM_FOR_SELECTION = 300//3 #number of chromosomes to be chosen for selection
NUMBER_ITERS = 50000 #number of iterations
```

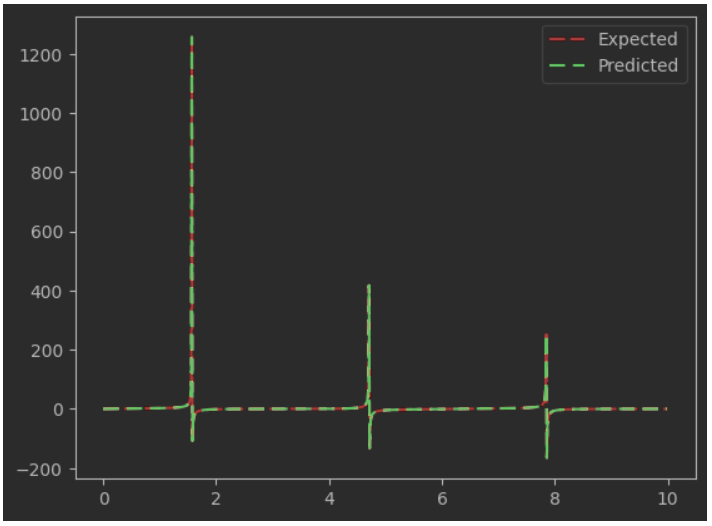
```
"""
DEFINE A FUNCTION TO PREDICT
"""
import math
def function_tan(x):
    """
    This function takes in three arguments, x, y, and returns the result of x cubed plus the sine of y.
    """
    return math.tan(x)+math.sin(x)
X = [[x] for x in np.arange(0, 10, 0.01)] #function inputs
X = sorted(X, key=lambda x: x[0])

y = [[function_tan(x[0])] for x in X] #function outputs

pop = Population(POP_SIZE, NUM_FOR_SELECTION, FUNCTIONS, TERMINAL_SET, 6, MAX_DEPTH) #create the populati
alg = Algorithm(pop, NUMBER_ITERS, X, y, epoch_feedback=500)  #create the algorithm
start_time = time.time()
best = alg.train() #train the algorithm
end_time = time.time()

elapsed_time = end_time - start_time
```

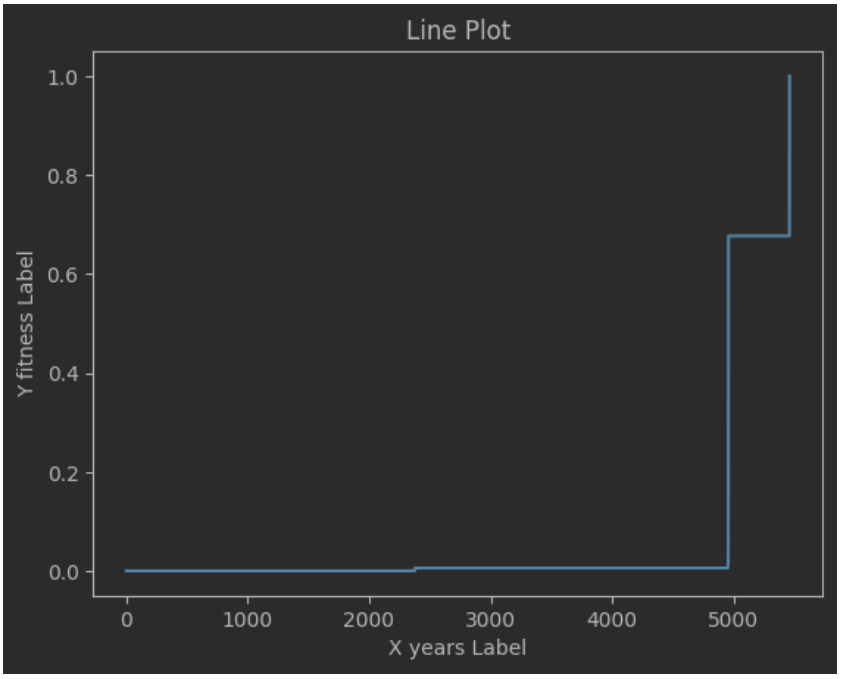
همانطور که مشاهده می‌کنید ما در فرمول tan داریم که در بین آیتم‌های مجاز برای ساخت ژن هر کروموزم مجاز به استفاده از آن نیستیم پس باید شکلی مانند sin/cos را بسازد تا بتواند به دقت بالایی از پیش بینی دست پیدا کند .

نتایج :

```
(sin(x0) + (sin(x0) / cos(x0)))
1.0
```

```
Elapsed time: 73.58 seconds
```





حاصل بعد از گذر 5000 نسل بدست آمده است .

حال برای ساخت تابعی مشابه شکل زیر :

یک تابع را به صورت رندوم به شکل زیر می‌سازیم .

```
SIZE = 1 # number of variables
MAX_DEPTH = 10 #maximum depth of a tree
FUNCTIONS = {1: ['sin','cos','e','ln','tg','tanh','abs'], 2:['+', '-', '*', '/','^']} #unary and binary functions
TERMINAL_SET = ['x'+str(i) for i in range(SIZE)] #create parameters name
DEPTH = 4 #initial depth of trees
POP_SIZE = 1000 #number of chromosomes in population
NUM_FOR_SELECTION = 1000//3 #number of chromosomes to be chosen for selection
NUMBER_ITERS = 50000 #number of iterations
```

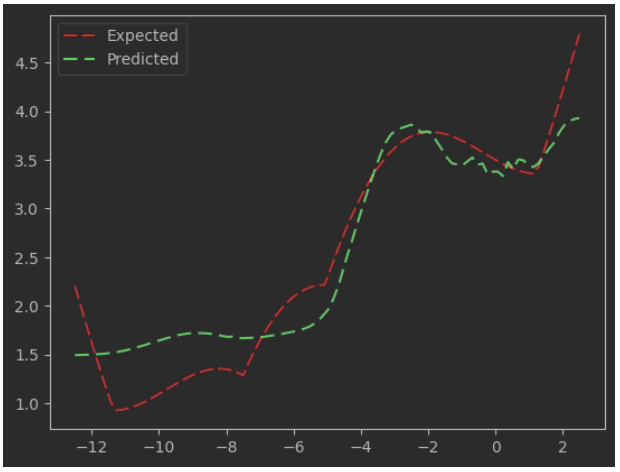
```
import numpy as np

X = np.linspace(-10, -2.5, 100)
y = np.abs(np.sin(X)) + np.random.binomial(0, 0.1, len(X))  + np.cos(X)
X2 = np.linspace(-2.5, 5, 100)
y += np.random.binomial(0, 0.05, len(X2))+ np.abs(X2)

X = X + X2

X = [[x] for x in X]
y = [[item] for item in y]

pop = Population(POP_SIZE, NUM_FOR_SELECTION, FUNCTIONS, TERMINAL_SET, 6, MAX_DEPTH) #create the population
alg = Algorithm(pop, NUMBER_ITERS, X, y, epoch_feedback=500)  #create the algorithm
start_time = time.time()
best = alg.train() #train the algorithm
end_time = time.time()

elapsed_time = end_time - start_time

print(f"Elapsed time: {elapsed_time:.2f} seconds")
```

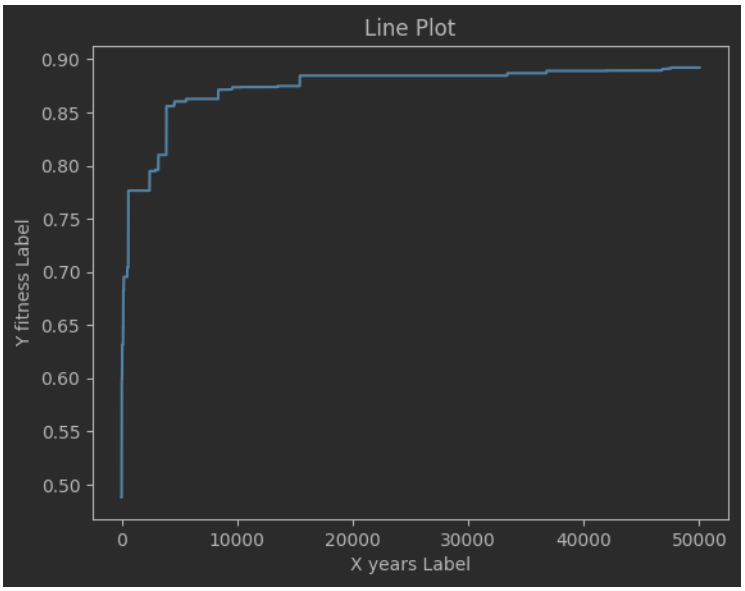
نتایج :

همانطور که در اینجا مشاهده می‌کنید در این سوال دقت ما به ۱۰۰ نرسیده است و به دقت 90 درصد نزدیک شده است .

```
0.8921784608570941
```

```
Elapsed time: 1427.36 seconds
```





حاصل بعد از گذر از 50000 نسل بدست آمده است .

در اینجا که مسئله دشوار تر شده است مشاهده می‌کنید که نمودار پیشرفت بهتر از موارد گذشته مشخص شده است و به صورت مناسبی تاثیر گذر سال‌ها و ساخت بچه‌های جدیدی که توانایی مدل را بهبود می‌دهند نشان داده شده است .

در نهایت مطابق با خواست سوال یک تابع چند ضابطه‌ای همانند خواست پروژه ساختم .

```
SIZE = 1 # number of variables
MAX_DEPTH = 20 #maximum depth of a tree
FUNCTIONS = {1: ['sin','cos','e','ln','tg','tanh','abs'], 2:['+', '-', '*', '/','^']} #unary and binary functions
TERMINAL_SET = ['x'+str(i) for i in range(SIZE)] #create parameters name
DEPTH = 4 #initial depth of trees
POP_SIZE = 1000 #number of chromosomes in population
NUM_FOR_SELECTION = 1000//3 #number of chromosomes to be chosen for selection
NUMBER_ITERS = 80000 #number of iterations
```

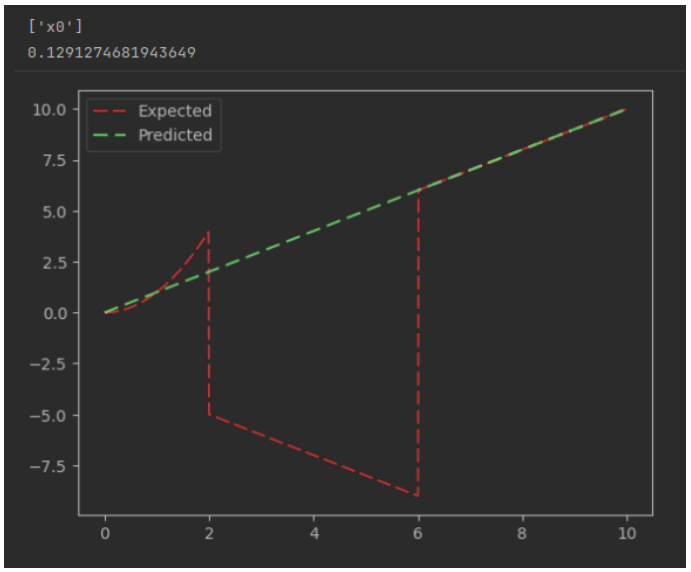
```
import numpy as np

"""
DEFINE A FUNCTION TO PREDICT
"""
def f(x):
    if x<2:
        return x**2
    elif x>=2 and x<=6:
        return -x-3
    else:
        return x


X = [[x] for x in np.arange(0, 10, 0.01)] #function inputs
X = sorted(X, key=lambda x: x[0])

y = [[f(x[0])] for x in X] #function outputs



pop = Population(POP_SIZE, NUM_FOR_SELECTION, FUNCTIONS, TERMINAL_SET, 6, MAX_DEPTH) #create the population
alg = Algorithm(pop, NUMBER_ITERS, X, y, epoch_feedback=500)  #create the algorithm
start_time = time.time()
best = alg.train() #train the algorithm
end_time = time.time()

elapsed_time = end_time - start_time
```

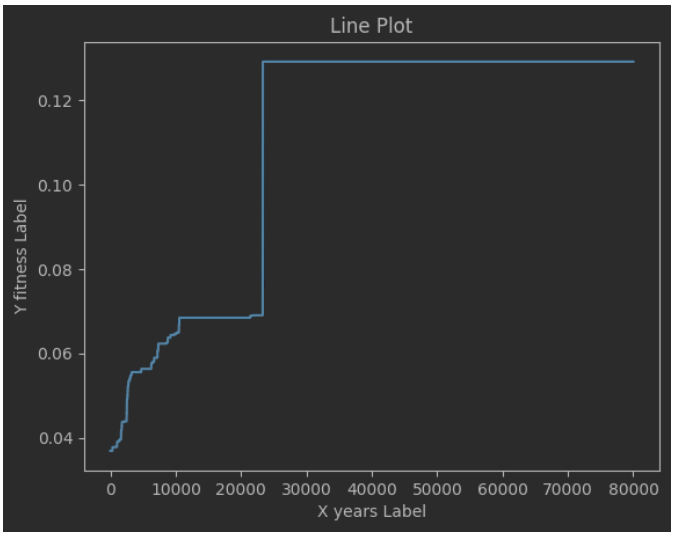
نتایج بدست آمده به شکل زیر است :

همانطور که در زیر مشاهده می‌کنید fitness برابر با 0.1 است که بسیار کم است و فرمولی که نهایتا بدست آورده برابر با y=x است . در مدت زمان 2604 ثانیه .

```
Elapsed time: 2604.90 seconds
```

```
['x0']
0.1291274681943649
```





همانطور که مشاهده می‌کنید پیشرفت و گذر نسل‌ها از یک جایی به بعد متوقف شده است است و دیگر تغییر نکرده هرچند که در گذشته این پیشرفت خوب بوده است .

چندین بار این آزمایش را انجام دادم و در نهایت در تمامی موارد به همچین شکلی می‌رسیدم که نشان می‌دهد که به کمک الگوریتم ژنتیک به سبکی که من پیاده سازی کردم توانایی تولید یک تابع که بتواند یک تابع دو ضابطه را پیش بینی کند را ندارد برای اینکه بتواند به این مهم دست پیدا کند باید یک سری فانکشن جدید اضافه کنیم که تابع اصلی را تبدیل به چندین ضابطه تقسیم کند و برای هر قسمت از الگوریتم ژنتیک انتظار داشته باشیم که یک ضابطه مناسب تولید کند تا بتواند چنین توابعی را نیز پیش بینی کند .