به نام خدا

این پروژه از چهار قسمت اصلی تشکیل میشود:

1. Target Function
   1-B. Valid Function

2. Generating random trees(expressions)

3. Genetic Programming
   3-A. Select
   3-B. Crossover
   3-C. Mutation

4. Pick the Best

Target function & Valid function

برای این قسمت باید ابتدا تابع هدفمان را بنویسیم که میخواهیم با جنتیک پروگرمینگ به آن برسیم. که در این پروژه $x^2 + (x + 3)/2$ می باشد.

```
def Target_Function(x):
    return x**2 + (x+2)/3
```

تابع بعدی که در این قسمت باید پیاده سازی کنیم، تابعی است که معیار برای درخت ها را تعریف میکند و یک ابزار برای سنجش آن ها میباشد که mean squared error هر درخت را با تابع هدف روی داده های ورودی که به صورت رندوم انتخاب می شوند را برمی گرداند.

```
def Valid_Function(tree):
    mystack = []
    errors = []
    result = 0
    if(len(tree)>2):
        for i in range(len(inputs)):
            for x in tree:
                if(x not in operands):
                    mystack.append(x)
                else:
                    tmp1 = mystack.pop()
                    tmp2 = mystack.pop()

                    if(tmp1=='x'):
                        tmp1 = inputs[i]
                    if(tmp2=='x'):
                        tmp2 = inputs[i]

                    tmp1 = tmp1
                    tmp2 = tmp2

                    if(x == '+'):
                        mystack.append(tmp1+tmp2)

                    elif(x == '-'):
                        mystack.append(tmp1-tmp2)

                    elif(x == '^'):

                        if(tmp1!=0):
                            if(tmp1>100):
                                tmp1 = 100
                            elif(tmp1<-100):
                                tmp1 = -100
```

```
            elif(x == '^'):

                if(tmp1!=0):
                    if(tmp1>100):
                        tmp1 = 100
                    elif(tmp1<-100):
                        tmp1 = -100
                    if(tmp2>100):
                        tmp2 = 100
                    elif(tmp2<-100):
                        tmp2 = -100
                    resultofpower = tmp1**tmp2
                    if(abs(resultofpower)>1000000):
                        if(str(resultofpower)[0]=="-"):
                            resultofpower = -100000
                        else:
                            resultofpower = 100000
                else:
                    resultofpower = 0
                if(str(resultofpower)[0] == "-"): # for "complex" error
                    mystack.append(resultofpower)
                else:
                    mystack.append(resultofpower)



            elif(x == '*'):
                mystack.append(round(tmp1*tmp2,3))
```

```
                elif(x == '/'):
                    if(tmp2!=0):
                        mystack.append(tmp1/tmp2)
                    else:
                        mystack.append(tmp1*100)
                    continue


        result = mystack.pop()

        try: #for overflow catch
            error = (abs(result-Target_Function(inputs[i])))**2
            error = round(error,3)
        except:
            error = sys.maxsize//1000
        errors.append(error)

else:
    result = tree[-1]
    if(result!="x"):
        for i in range(len(inputs)):
            try: #for overflow catch
                error = round((abs(result-Target_Function(inputs[i])))**2,3)
            except:
                error = sys.maxsize//1000

            errors.append(error)

    else:
        for i in range(len(inputs)):

            result = inputs[i]

            try: #for overflow catch
                error = round((abs(result-Target_Function(inputs[i])))**2,3)
            except:
                error = sys.maxsize//1000

            errors.append(error)

return sum(errors)/len(errors)
```

و برای تسریع در زمان و جلوگیری از اورفلو در آخر برای محاسبه ی خطا از try catch استفاده شده و با توجه به آزمایش هایی که صورت گرفت ، در عملیات توان این تابع احتمال رخداد اورفلوو بیشتر از بقیه ی عملیات ها می باشد در نتیجه چهار شرط برای جلوگیری از این ارور در آن گذاشته شد که داده ها مقدار کمتری بگیرند.

```
        elif(x == '^'):

            if(tmp1!=0):
                if(tmp1>100):
                    tmp1 = 100
                elif(tmp1<-100):
                    tmp1 = -100
                if(tmp2>100):
                    tmp2 = 100
                elif(tmp2<-100):
                    tmp2 = -100
                resultofpower = tmp1**tmp2
                if(abs(resultofpower)>1000000):
                    if(str(resultofpower)[0]=="-"):
                        resultofpower = -100000
                    else:
                        resultofpower = 100000
            else:
                resultofpower = 0
            if(str(resultofpower)[0] == "-"): # for "complex" error
                mystack.append(resultofpower)
            else:
                mystack.append(resultofpower)
```

Generating random trees(expressions)

در این قسمت باید تابع مربوط به تولید جمعیت اولیه مان را بنویسیم که پارامتر ورودی n را میگیرد که تعداد درخت های جمعیت اولیه مان می باشد. که با آزمایش های مختلف بازه ی 50 تا 60 بهینه ترین بازه به دست آمد و تعداد جمعیت های بیشتر و کمتر از آن دارای خطاهای زیاد می شدند یا به یک عدد ثابت به عنوان درخت نهایی ختم می شدند.

```python
def Generating_Random(n):
    trees = []
    for i in range(n):
        operators_num = np.random.randint(0,7)
        if(operators_num==0):
            tree = []
            tree.append(np.random.randint(1,10))
            trees.append([tree,Valid_Function(tree)])
        else:
            counter = 1
            length = operators_num*2 + 1
            tree = []
            tree.append(np.random.randint(1,10))
            tree.append(np.random.randint(1,10))
            while(len(tree)!= length and ((length-len(tree))>counter)):
                to_select = np.random.randint(6)
                if(counter <=0 or to_select%3 !=2): #add number or variable / to_select %3 == 0, 1
                    if(to_select==0 or to_select==3): #add number
                        tree.append(np.random.randint(1,10))
                        counter += 1
                    elif(to_select==1 or to_select==4): #add varaible
                        tree.append('x')
                        counter += 1
                    else:
                        to_select = np.random.randint(2) #not allowed to add operator/counter<=0
                        if(to_select==2):
                            tree.append(np.random.randint(1,10)) #add number
                            counter += 1
                        elif(to_select==5):
                            tree.append('x') #add variable
                            counter += 1

                else:
                    tree.append(np.random.choice(operands)) #add operator / to_select %3 == 2
                    counter -= 1
            if not(length-len(tree)>counter):
                for i in range(counter):
                    tree.append(np.random.choice(operands))
            trees.append([tree,Valid_Function(tree)])
    return trees
```

که عملگرها، اعداد و متغیرها به صورت رندوم (در صورت مجاز بودن) انتخاب می شوند و در درخت قرار می گیرند که طول درخت بر حسب تعداد متغیرها که در ابتدا به صورت رندوم انتخاب می شود، تعیین می شود. که بیشترین تعداد عملگر مجاز برای یک درخت 6 می باشد که در پی آن بیشترین طول مجاز برای یک درخت، 13 می باشد.

که نودهای درخت ها در ایندکس های متناظر خود در یک لیست قرار میگیرند و در نتیجه ی ساخت n درخت رندوم، یک لیست n تایی از لیست ها به عنوان جمعیت اولیه برگردانده می شود.

که این جمعیت اولیه اولین نسل از نسل ها را تشکیل میدهد و به عنوان اولین عضو به نسل ها اضافه می شود.

```
Generations.append(Generating_Random(first_population))
```

Genetic Programming

این قسمت سه زیر شاخه دارد.

Select

این قسمت مربوط به انتخاب نسل برتر می باشد. در این قسمت درخت های برتر نسل های قبلی را به عنوان نسل جدید انتخاب میکنیم که معیار ارزیابی ما همان تابع Valid function می باشد.
یک پارامتر ورودی به عنوان جمعیت نسل بعدی دریافت می کنیم که 65 درصد آن را از درخت های برتر نسل قبلی و 0.35 آن را از نسل قبلی اش انتخاب میکنیم.

```
def Generic_Select(n):
    if(len(Generations)>1):
        Last_Gen = Generations[-1]
        Last_Gen.sort(key=lambda x : x[1])
        Next_Gen = []
        for i in range(int(0.65*n)):
            if(Last_Gen[i] not in Next_Gen):
                Next_Gen.append(Last_Gen[i])

        Previous_Gen = Generations[-2]
        Previous_Gen.sort(key=lambda x : x[1])
        for i in range(int(0.35*n)):
            if(Previous_Gen[i] not in Next_Gen):
                Next_Gen.append(Previous_Gen[i])

        Generations.append(Next_Gen)
    else:
        Last_Gen = Generations[-1]
        Last_Gen.sort(key=lambda x : x[1])
        Next_Gen = []
        for i in range(int(0.8*n)):
            if(Last_Gen[i] not in Next_Gen):
                Next_Gen.append(Last_Gen[i])

        Generations.append(Next_Gen)
```

Crossover

در این قسمت دو انتخاب داریم که به صورت رندوم انتخاب می شوند.
یک اینکه زیر درختی از یک درخت را به درخت دیگر اضافه کنیم و آن را به نسل فعلی اضافه کنیم که هر دو درخت و زیر درخت انتخابی به صورت رندوم انتخاب میشوند.
بعدی اینکه یک عضو یک درخت را با یک عضو از یک درخت دیگر که هر دو عضو و هر دو درخت به صورت رندوم انتخاب می شوند، جابجا کنیم.
یک پارامتر میگیرد که بیانگر این است که چند درصد درخت های موجود در نسل crossover شوند.

عملیات اول crossover :

```
def Generic_Crossover(percentage):
    indexes2crossover = [int(x) for x in np.random.uniform(0,len(Generations[-1]),int(percentage*len(Generations[-1])))]
    if not(len(indexes2crossover)%2==0):#must be even because each time select two trees
        del indexes2crossover[np.random.randint(len(indexes2crossover))]
    while(len(indexes2crossover)%2==0 and len(indexes2crossover)!=0): #select two trees and remove them from choices
        index1 = np.random.choice(indexes2crossover)
        indexes2crossover.remove(index1)
        index2 = np.random.choice(indexes2crossover)
        indexes2crossover.remove(index2)

        to_select = np.random.randint(2)

        if(to_select==0): #add random subtree from first tree to a random leaf from second tree and add the produced tree to the last generation
            if(len(Generations[-1][index1][0])!=1): #if the first tree is a constant value, it doesn't have a operand
                index2pick1 = np.random.randint(len(Generations[-1][index1][0]))
                while(Generations[-1][index1][0][index2pick1] not in operands):
                    index2pick1 = np.random.randint(len(Generations[-1][index1][0]))
                offspring1 = Generic_Crossover_ValidTree(Generations[-1][index1][0],index2pick1)
                index2pick2 = np.random.randint(len(Generations[-1][index2][0]))
                while(Generations[-1][index2][0][index2pick2] in operands):
                    index2pick2 = np.random.randint(len(Generations[-1][index2][0]))
                offspring2 = Generations[-1][index2][0][0:index2pick2] + offspring1 + Generations[-1][index2][0][index2pick2+1::]
                if(len(offspring2)>15): #cut the tree if has a lenght more than 15
                    subtreeindex = Cuttree(offspring2,15) #return the first index of the tree wich satisfies the condition
                    if(subtreeindex!=0): #its not the same tree
                        Generations[-1].append([offspring2[subtreeindex::],Valid_Function(offspring2[subtreeindex::])])
                else:
                    Generations[-1].append([offspring2,Valid_Function(offspring2)])
            else: #the first tree is a single constant value so it doesn't have operand so cant any subtree from it to add to the second tree, so the crossover operati
                continue
```

عملیات دوم crossover :

```
            continue
        else: #crossover two random nodes
            index2pick1 = np.random.randint(len(Generations[-1][index1][0])) #select random node from first tree
            if(Generations[-1][index1][0][index2pick1] in operands): #it's operand, swap with random operand from second tree
                if(len(Generations[-1][index2][0])!=1): #if the second tree is constant value, doesn't contain operand
                    index2pick2 = np.random.randint(len(Generations[-1][index2][0])) #until find node from second tree
                    while(Generations[-1][index2][0][index2pick2] not in operands):
                        index2pick2 = np.random.randint(len(Generations[-1][index2][0]))
                    tmp1 = Generations[-1][index1][0][index2pick1]
                    Generations[-1][index1][0][index2pick1] = Generations[-1][index2][0][index2pick2]
                    Generations[-1][index2][0][index2pick2] = tmp1
                else:#the second tree is a constant that doesn't contain operand so crossover won't be operated
                    continue
            else:                                               #it's not operand, swap with random number or variable from second tree
                index2pick2 = np.random.randint(len(Generations[-1][index2][0])) #select random node from  second tree
                while(Generations[-1][index2][0][index2pick2] in operands): #until find number or variable from second tree
                    index2pick2 = np.random.randint(len(Generations[-1][index2][0]))
                tmp1 = Generations[-1][index1][0][index2pick1]
                Generations[-1][index1][0][index2pick1] = Generations[-1][index2][0][index2pick2]
                Generations[-1][index2][0][index2pick2] = tmp1
```

Mutation

در این قسمت جهش ژنتیک صورت می گیرد که تابع آن پیاده سازی می شود.
دو پارامتر می گیرد که percentage مربوط به این هست که چند درصد درخت های نسل mutate شوند و جهش پیدا کنند.
Mutate_percentage مربوط به این است که چند درصد نودهای یک درخت جهش پیدا کنند.
انتخاب درختها برای جهش و نود ها برای جهش آن ها، به صورت رندوم اتفاق میافتد.
در انتخاب یک نود در صورتی که عدد یا متغیر باشد، یک عدد یا متغیر به صورت رندوم جایگزین آن می کند و در صورتی که عملگر باشد، یک عملگر به صورت رندوم جایگزین آن میکند.

```
def Generic_mutation(percentage,mutate_percentage):
    indexes2mutate = [int(x) for x in np.random.uniform(0,len(Generations[-1]),int(percentage*len(Generations[-1])))] #which trees to mutate
    for index in indexes2mutate:
        tomutate = Generations[-1][index][0]
        del Generations[-1][index]
        indexes2mutate = [int(x) for x in np.random.uniform(0,len(tomutate),int(mutate_percentage*len(tomutate)))] #which nodes to mutate
        for mutateindex in indexes2mutate:
            if(tomutate[mutateindex] not in operands): #mutate ints and variables
                to_select = np.random.randint(2)
                if(to_select==0):
                    tomutate[mutateindex] = np.random.randint(1,10) #change to number
                else:
                    tomutate[mutateindex] = 'x'                   #change to variable
            else:                                   #mutate operators
                tomutate[mutateindex] = np.random.choice(operands)

        # if([tomutate,Valid_Function(tomutate)] not in Generations[-1]):
        Generations[-1].append([tomutate,Valid_Function(tomutate)])
```

Pick the best

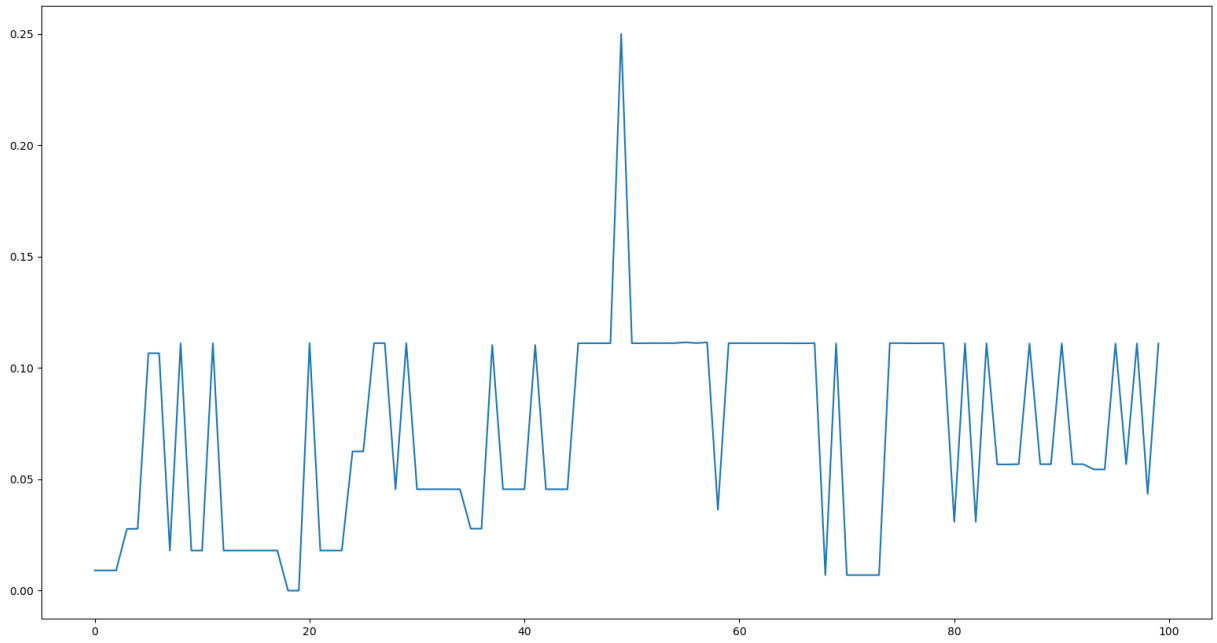
در آخرین قسمت، بهترین درخت را در آخرین نسل به عنوان جواب انتخاب میکنیم، که میتوانیم این انتخاب را محدود به نسل آخر نکنیم و از چند نسل آخر بهترین را بر مبنای کمترین خطا به عنوان بهترین جواب انتخاب کنیم پس پارامتر ورودی in_homany_generations را میگیرد و بر مبنای آن بین آخرین نسل ها بهترین را انتخاب می کند.

```
def Pick_the_Best(n):
    Generations[-1].sort(key=lambda x: x[1])
    the_best = Generations[-1][0]
    for i in range(n):
        Generations[i].sort(key=lambda x: x[1])
        if(Generations[i][0][1]<= the_best[1]):
            the_best = Generations[i][0]

    return the_best
```

ابتدا پارامترهای ورودی برای توابع مختلف مقدار دهی میشوند.
سپس جمعیت اولیه تولید می شود و به عنوان اولین نسل به نسل ها اضافه می شود.
حال نسل ها بوجود می آیند که در هر مرحله ی بوجود آمدن نسل جدید ابتدا بهترین های نسل قبل انتخاب می شوند سپس جهش می شوند و کراس آور روی آنها انجام میگیرد. و در آخر بهترین انتخاب می شود و خطای آن روی داده های نمایش داده می شود.

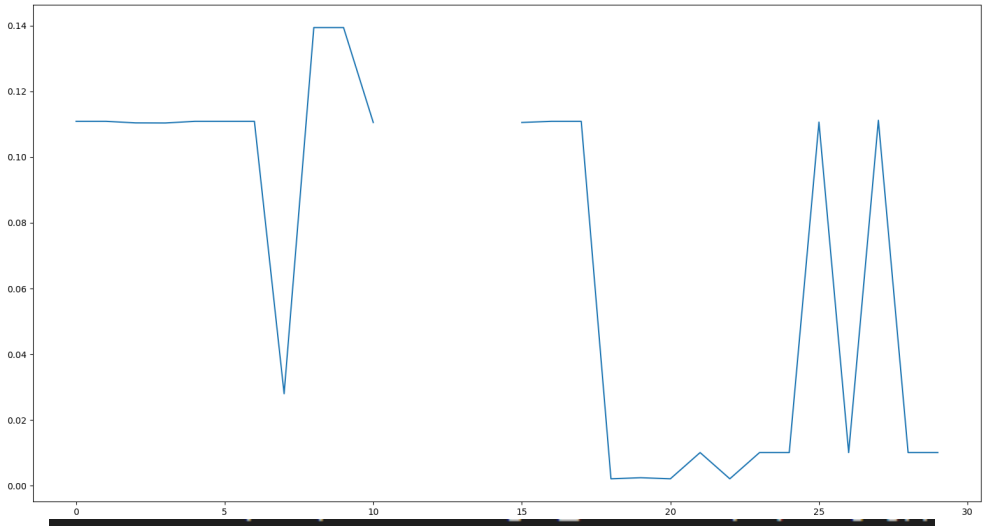
```
input_size = 70
Generations = []
inputs = np.random.uniform(-0.01,0.01,input_size)
inputs = [round(x,3) for x in inputs]
operands = ['+' , '^','-', '*', '/']
Generations_Num = 100
first_population = 50


Generations.append(Generating_Random(first_population))
for i in range(Generations_Num-1): #generations 50 to u
    Generic_Select(int(0.8*(first_population-(i//10))))
    Generic_mutation(0.7, 0.6)
    Generic_Crossover(0.8)


theBestTree = Pick_the_Best()

print(f"the mean squared error is : {theBestTree[1]}")
```

در زیر نمودار خطای بهترین درخت هر نسل بر حسب نسل آن آمده است.



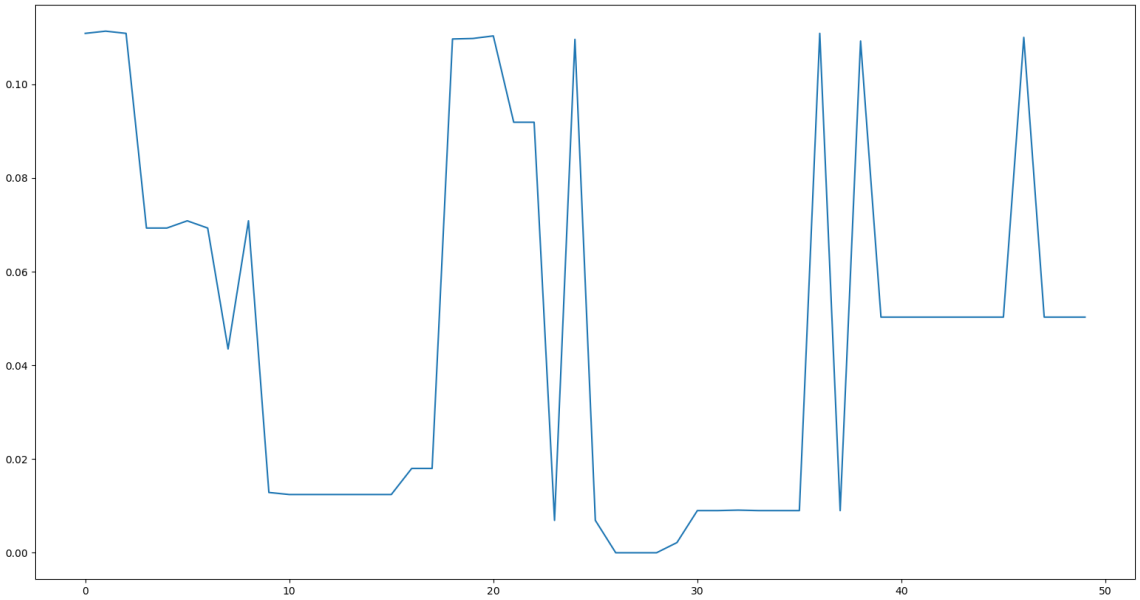
این برنامه برای حالت ها با تعداد نسل های مختلف اجرا و آزمایش شد که نتایج به صورت زیر می باشند.

برای وقتی که تعداد نسل ها برابر 30 باشد :



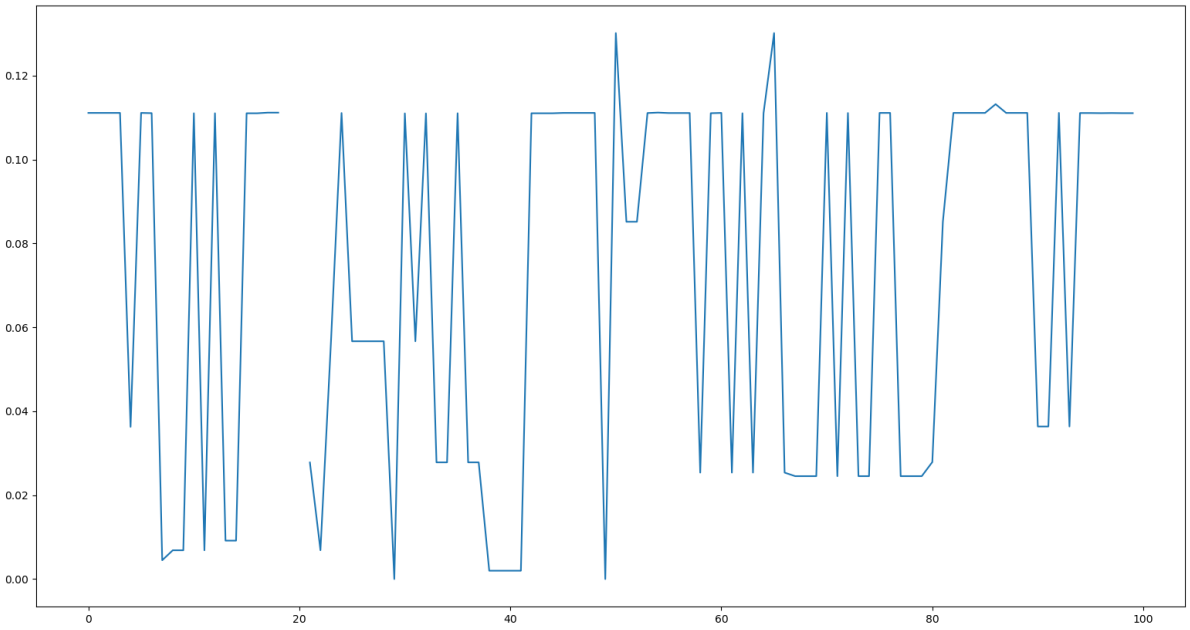

```
the mean squared error is : 0.010085714285714292
```

وقتی برابر 50 باشد :




```
the mean squared error is : 0.05030000000000003
```

و برای وقتی که 100 نسل تولید کنیم :




```
the mean squared error is : 0.11105714285714429
```

علی شیخ عطار
دکتر عبدی
هوش مصنوعی
دانشکده ی کامپیوتر علم و صنعت
1402