به نام خدا

دانشکده مهندسی برق

پروژه ۱

درس هوش مصنوعی

محمدامین حاجی خداوردیان

۹۷۱۰۱۵۱۸

استاد:دکتر عبدی

نیمسال اول ۱۴۰۰-۱۴۰۱

بخش اول: نگاشت مسئله به درخت

۱-۱ نحوه تولید درخت

در این بخش به توضیح نحوه ساخت درخت توابع(Expression Tree) می‌پردازیم. در ابتدای کار یک کلاس تحت عنوان گره برای درخت‌های خود تعریف کرده‌ایم که شامل سه بخش است:

- مقدار(Value) گره
- فرزند سمت چپ
- فرزند سمت راست
- احتمال

در شکل زیر این کلاس دیده می‌شود:

```
7   class node:
8       def __init__(self,value):
9           self.value = value
10          self.right = None
11          self.left = None
12          self.Probability = None
13
14      def __lt__(self, other):
15          return self.Probability > other.Probability
16
```

حال به شیوه تعریف توابع بر روی این ساختار می‌پردازیم. در ابتدا یک گره با عنوان ریشه درخت درنظر می‌گیریم. این ریشه در واقع نمایانگر درخت توابع(Expression Tree) ما خواهد بود. پس از آن هر کدام از بچه‌های چپ و راست این ریشه می‌توانند یک گره دیگر از کلاس بالا باشند که خودشان می‌توانند شامل مقادیر و بچه های دیگری باشند. مقادیری که می‌تواند گره‌های ما بگیرد شامل عملگرها و عملوندهایی است که در بخش بعد تعریف کرده‌ایم. بخش احتمال گره‌ها برای نسبت دادن شایستگی به درخت‌ها است و از آنجایی که گره ریشه نمایانگر درخت ما است بنابراین این عدد تنها برای ریشه درخت‌ها با اهمیت است. در بخش بعد و تولید جمعیت اولیه به طور کامل نحوه استفاده از این ساختار را نشان خواهیم داد.

۱-۲ محدودیت‌ها

محدودیت هایی که برای مقادیر این درخت در نظر گرفته‌ایم به این صورت است که گره‌های میانی و ریشه این درخت باید از عملگرهای گفته شده در صورت سوال باشد. از طرفی برای گسترده نشدن و تنوع درخت‌ها فرض را بر این گرفته‌ایم که خطای کمتر از ۰.۵ برای توابعی که با عملگرهای موجود ما ساخته می‌شوند قابل قبول است پس مقادیر ثابت را از بازه [-9,9]+0.5 انتخاب کرده‌ایم. در این بازه تنها مقدار 0 رو از بازه کنار گذاشته‌ایم به این علت که ممکن است باعث محدودیت در درخت ایجاد کند(به عنوان مثال تقسیم بر ۰ رخ دهد.)

یکی دیگر از محدودیت‌های در نظر گرفته شده برای این درخت‌ها این است که در ساخت آن‌ها مقدار x که قرار است ورودی تابع باشد و تابع ما با توجه به آن محاسبات را انجام دهد در سمت چپ گره‌های بالادستی خود قرار بگیرد. این مسئله فقط برای ساخت درخت‌های اولیه است.

اگر عملگر ما sin یا cos باشد فرزند سمت راست آن را تعیین نشده قرار می‌دهیم به این صورت که تنها یک فرزند دارد.

بخش دوم: الگوریتم ژنتیک

۱-۲ تولید جمعیت اولیه

برای جمعیت اولیه ما از تابع‌های ساده استفاده می‌کنیم. در واقع ساده‌ترین توابعی که می‌شود با عملگرهای گفته شده در صورت مسئله تولید کرد را تولید می‌کنیم. به عنوان مثال (cos(x) , x/4 , x*6 , x+2)

شکل زیر نحوه تولید درخت‌ها است:

```
24  def PopulationCreator(size):
25      Operators = ['+' , '-' , '*' , '/' , '^' , 'sin' , 'cos']
26      Operands = [-9 , -8 , -7 , -6 , -5 , -4 , -3 , -2 , -1 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 0.5 ]
27      FirstPopulation = list()
28      for i in range(size):
29          operator = random.choice(Operators)
30          operand = random.choice(Operands)
31          ParentNode = node(operator)
32          ParentNode.left = node('x')
33          if operator == 'sin' or operator == 'cos':
34              ParentNode.right = node(None)
35          else:
36              ParentNode.right = node(operand)
37          FirstPopulation.append(ParentNode)
38      return FirstPopulation
```

همانطور که در بخش اول گفته شد محدودیت های گفته شده در کد بالا دیده می‌شود. ابتدا به صورت تصادفی یک عملگر برای تابع ساده خود انتخاب می‌کنیم و آن را به عنوان ریشه درخت قرار می‌دهیم. سپس باتوجه به عملگر و محدودیت گفته شده برای sin و cos مقدار فرزند راست را از بازه عملوندهای تعریف شده در بخش اول و محدودیت‌ها انتخاب می‌کنیم. فرزند چپ نیز مطابق گفته‌های محدودیت بخش اول باید x قرار گیرد. جمعیت اولیه انتخاب شده تعداد آن ۲۰۰ است به این دلیل که از همه توابع ساده با مقادیر مختلف آن داشته باشیم.

۲-۲ تابع شایستگی

برای تعیین شایستگی یک تابع با عنوان FindProb در کد تهیه شده است. کد تابع به شکل زیر است:

```
135 def FindProb(Tree , x, y):
136     Sum = 0
137     Fx = list()
138     yx = copy.deepcopy(y)
139     for i in range(len(yx)):
140         if yx[i] < 0 :
141             yx[i] = abs(yx[i])
142     yx.sort()
143     Max = yx[len(yx) - 1]
144     for i in range(len(x)):
145         Fx.append(EvaluateExpressionTree(root=Tree , s=x[i]))
146     for i in range(len(y)):
147         DeltaY = abs(y[i] - Fx[i])
148         Sum = Sum + DeltaY
149     MeanDeltaY = Sum/(len(y))
150     Tree.Probability = (Max - MeanDeltaY)/Max
```

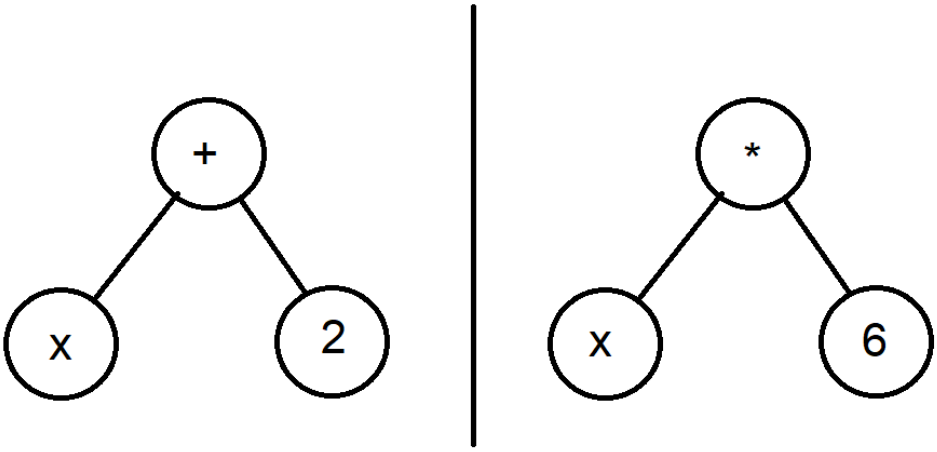
در این بخش برای نسبت دادن یک احتمال از این روش استفاده کرده‌ایم که بیشترین مقداری که می‌تواند خروجی‌های ما داشته باشند را با توجه به ورودی برنامه پیدا می‌کنیم. سپس به ریشه درخت و بخش احتمال آن عددی را نسبت می‌دهیم که میانگین تفاوت مقدار واقعی و تابع تخمینی ما را نسبت به بیشترین مقدار تابع واقعی را می‌سنجد و عددی را نسبت می‌دهد. اگر احتمال برابر ۱ شود یعنی تابع دقیق پیدا شده است.

برای محاسبه مقادیر تابع ما از یک تابع با عنوان EvaluateEpressionTree استفاده شده است که در یک بخش جداگانه به آن می‌پردازیم. پس از محاسبه مقادیر تابع تخمینی اختلاف آن را نسبت به تابع واقعی میسنجیم و از آن میانگین می‌گیریم.

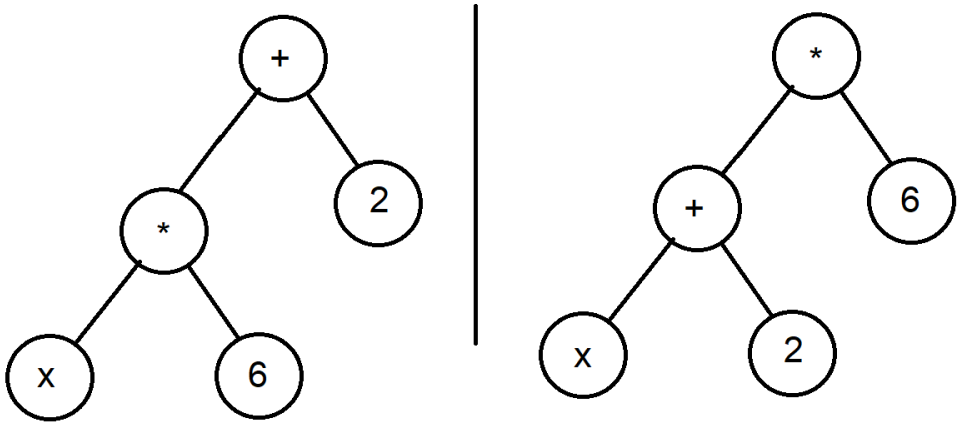
۲-۳ نحوه تولید نسل بعد

۲-۳-۱ ترکیب دو درخت و تولید فرزند از والدین

برای این بخش نحوه کار کرد ما به این صورت است که دو درخت از مجموعه درخت‌های انتخاب شده که نحوه انتخاب آن‌ها در بخش بعد توضیح داده خواهد شد را به صورت تصادفی انتخاب می‌کنیم. پس از انتخاب این دو درخت دوبار آن‌ها را ترکیب می‌کنیم. ترکیب ما به این صورت است که به جای مقدار x تابع دیگر قرار می‌گیرد. به عنوان مثال دو تابع x+2 و x*6 را در اختیار داریم، درخت‌های این دوتابع به شکل زیر است:



همانطور که دیده می‌شود تمام محدودیت‌های گفته شده در آن رعایت شده است حال برای ترکیب به صورت زیر دو تابع مختلف را بدست می‌آوریم:



همانطور که گفته شد دو تابعی که به صورت رندوم انتخاب شده‌اند از طریق مقدار X با یکدیگر ترکیب می‌شوند. به همین دلیل دو تابع پیچیده تر در جمعیت ما تولید شده است که می‌توان از آن در ادامه استفاده کرد.

کد این بخش به صورت زیر است و با توجه به آن ۶۰۰ فرزند جدید میسازیم:

```
195     for i in range(300):
196         p1 = random.choice(SelectedPopulation)
197         p2 = random.choice(SelectedPopulation)
198         NewPopulation.append(CombineTrees(p1 , p2))
199         NewPopulation.append(CombineTrees(p2 , p1))
```

در بخش بالا از جمعیت برتر که با توجه به شایستگی انتخاب شده‌اند به صورت رندوم دو تابع انتخاب می‌شود و در بخش بعد در نسل جدید دو بار این دو تابع مطابق مثالی که در قبل گفته شد ترکیب می‌شوند. نحوه ترکیب هم به صورت کد زیر است:

```
126 def CombineTrees(Tree1 , Tree2):
127     NewTree = copy.deepcopy(Tree1)
128     while(True):
129         if NewTree.left.value == 'x':
130             NewTree.left = Tree2
131             return NewTree
132         else:
133             NewTree = NewTree.left
```

۲-۳-۲ جهش

برای این بخش تابعی در کد با عنوان Mutation قرار دارد. در این بخش ما به این صورت عمل می‌کنیم که شروع به حرکت در درخت تابع(ExpressionTree) می‌کنیم و با یک احتمالی(۰.۱) گره‌ای که در آن قرار داریم را مقدارش را تغییر می‌دهیم. در این بخش عملگرهایی که از آن‌ها انتخاب می‌شود تغییری نکرده ولی در عملوندها علاوه بر مقادیر ثابت گفته شده در محدودیت‌های بخش اول مقدار x را نیز قرار داده‌ایم علت این کار را در بخش **چالش‌ها** توضیح خواهیم داد.

اگر گره انتخابی عملگر باشد و باید جهش رخ دهد چک می‌کنیم که اگر sin و cos باشد به فرزند سمت راست آن نیز عددی تصادفی از عملوندهای خود را نسبت دهیم.(در صورتی که جهشی که اتفاق میفتد مجدد sin و cos نباشد)

اگر گره انتخابی گره مقدار باشد در این صورت محدودیتی نداریم و به راحتی از عملوندها جایگزین را برای آن انتخاب می‌کنیم. کد بخش گفته شده به شکل زیر است:

```python
def Mutation(Tree):
    nodes = list()
    nodes.append(Tree)
    Operators = ['+' , '-' , '*' , '/' , '^' , 'sin' , 'cos']
    Operands = [-9 , -8 , -7 , -6 , -5 , -4 , -3 , -2 , -1 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9  , 0.5 , 'x']
    i = 0
    while len(nodes) > 0:
        UnderTestNode = nodes[i]
        i = i + 1
        probability = random.random()*100
        if probability < 10:
            if UnderTestNode.value in Operators:
                    if UnderTestNode.value == 'cos' or UnderTestNode.value == 'sin':
                        UnderTestNode.value = random.choice(Operators)
                        UnderTestNode.right.value = random.choice(Operands)
                        return
                    else:
                        UnderTestNode.value = random.choice(Operators)
                        return
            elif UnderTestNode.value == 'x':
                UnderTestNode.value = 'x'
                return
            elif UnderTestNode.value in Operands:
                UnderTestNode.value = random.choice(Operands)
                return
        else:
            if UnderTestNode.left is not None and UnderTestNode.right is not None:
                nodes.append(UnderTestNode.left)
                nodes.append(UnderTestNode.right)
            else:
                return
        if i==100:
            return
```

۲-۴ نحوه انتخاب والدین

در این بخش با توجه به شایستگی نسبت داده شده به ریشه درخت‌ها مقادیری که در نسل جدید وجود دارد را مقدار شایستگی آن‌ها را با تابع FindProb محاسبه می‌کنیم.(مقادیر موجود در نسل جدید شامل درخت‌های ترکیب شده و جهش یافته و درخت‌های والدین(دلیل این بخش در **چالش‌ها** توضیح داده خواهد شد) است.)

سپس جمعیت نسل جدید که مقادیر شایستگی آن حساب شده است را با توجه به مقدار شایستگی مرتب می‌کنیم و پس از آن به تعداد ۲۰۰ درخت که شایستگی بیشتری دارند و احتمال بیشتری به آن نسبت داده شده است را به عنوان نسل انتخاب شده برای تولید نسل بعد در نظر می‌گیریم.

۵-۲ تابع EvaluateEpressionTree

همانطور که در بخش محاسبه شایستگی به آن اشاره شد به توضیح این تابع می‌پردازیم. این تابع به این صورت کار می‌کند که به صورت بازگشتی به سراغ فرزندان چپ و راست می‌رود و مقادیر سمت چپ یک گره و مقادیر سمت راست آن گره را محاسبه می‌کند و با توجه به مقدار گره اصلی که یک عملگر است این دو مقدار را با توجه به عملگر گره با هم ترکیب می‌کند. کد آن به شکل زیر است:

```python
def EvaluateExpressionTree(root , s):
    if root is None:
        return 0
    if root.left is None and root.right is None:
        if root.value == 'x':
            return s
        else:
            return root.value
    LeftSum = EvaluateExpressionTree(root.left , s=s)
    RightSum = EvaluateExpressionTree(root.right , s=s)

    if type(LeftSum) is complex or type(RightSum) is complex:
        return 0
    if root.value == '+':
        return LeftSum + RightSum

    elif root.value == '-':
        return LeftSum - RightSum

    elif root.value == '*':
        return LeftSum * RightSum

    elif root.value == '^':
        if LeftSum == 0:
            return 0
        else:
            if LeftSum < -10000 or RightSum < -1000 or LeftSum > 10000 or RightSum > 1000:
                return 0
            else:
                if RightSum < 0:
                    if (1/LeftSum) < -10000 or (1/LeftSum) > 10000 or RightSum < -1000 :
                        return 0
                    else:
                        return (1/LeftSum) ** (-1*RightSum)
                else:
                    return LeftSum ** RightSum

    elif root.value == '/':
        if RightSum == 0:
            return 1000000000000
        else:
            return LeftSum / RightSum

    elif root.value == 'sin':
        return math.sin(LeftSum)

    elif root.value == 'cos':
        return math.cos(LeftSum)
```

شرط‌های گذاشته شده برای عملیاتی است که عملگرها توانایی محاسبه آن را ندارند به عنوان مثال اگر تقسیم بر ۰ رخ داد مقدار زیادی را برگرداند. و یا یک شرط دیگر وجود دارد که اگر توان منفی شد به این علت که عملیات توان برای مقادیر منفی جواب نمی‌دهد عکس پایه به مقدار مثبت توان برسد.

۲-۶ شرط خاتمه

برای خاتمه یافتن برنامه از آنجایی که تعداد نسل‌ها و فرزندان زیاد است تعداد نسل‌ها را کم انتخاب می‌کنیم که سرعت برنامه حفظ شود و اگر تعداد نسل‌ها را بیشتر کنیم دقت تابع های بدست آمده در ازای زمان بیشتر خواهد شد. به همین دلیل شرط خاتمه را برابر آن گرفتیم که یا ۲۰۰ نسل برنامه ادامه یابد یا احتمال(شایستگی) بیشتر از ۰.۹۸ شود.

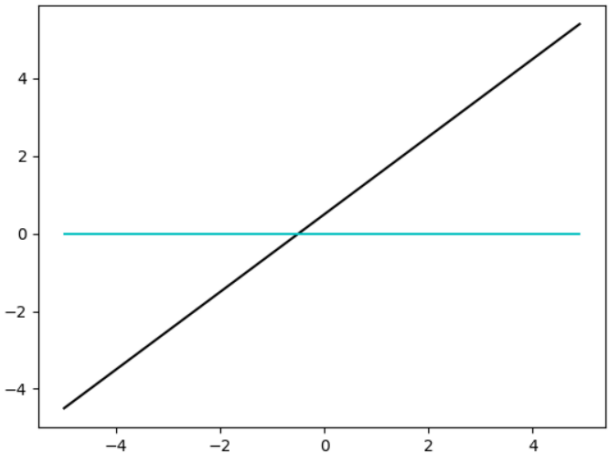
۲-۷ چالش‌های مواجه شده

چالش‌های مواجه شده به صورت زیر است:

- علت آنکه در نسل جدید والدین را نیز در نظر می‌گیریم این است که همانطور که می‌دانیم در الگوریتم ژنتیک گام‌های اولیه ما بزرگ است و این احتمال وجود دارد که ما از مسیر مدنظر خود دور شویم. برای آنکه الگوریتم بتواند اشتباه خود را اصلاح کند و برگردد والدین را نیز در نسل جدید قرار می‌دهیم تا در صورت اینکه شایستگی آن را داشتند در نسل منتخب و برتر قرار گیرند.
- در جهش برای مقادیر علاوه بر اعداد ثابت مقدار x را نیز در نظر گرفته‌ایم این به آن علت است که در ابتدا فرض برای حل مسئله به این صورت بود که تابع‌های ما تنها شامل یک x خواهد بود ولی برای دستیابی به تابع‌های پیچیده تر مانند x*sin(x) نیاز به آن داریم به همین دلیل برای رفع این مشکل احتمال جهش را بیشتر کردیم و این احتمال که x نیز انتخاب شود را قرار داده‌ایم
- یکی از چالش‌های مواجه شده با آن این است که برای محاسبه مقادیر محدودیت‌هایی از جمله تقسیم بر ۰ و ... مواجه شدیم که برای برطرف کردن آن‌ها شرط های زیادی را برای تابع EvaluateExpressionTree قرار دادیم.

بخش سوم: آزمایش و تست پروژه

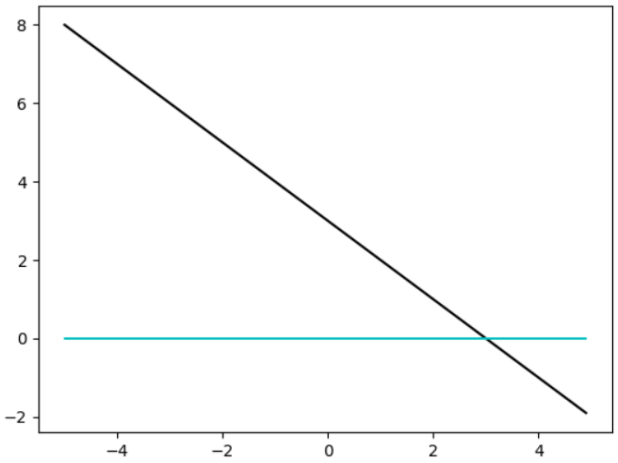
در این بخش به تست پروژه می‌پردازیم تست اول تابع ساده x+0.5 را تست می‌کنیم شکل خروجی زیر بدست می‌آید.(دو تابع و مقدار خطا در یک شکل رسم می‌شوند اگر تابع تخمینی دقیقا تابع واقعی باشد رنگ نمودار به رنگ مشکی خواهد بود):



```
Iteration: 13
Competency 1.0
RunTime is: 5.151986837387085
Time we calculate Compentency: 10400
('(', ('(', 'x', '+', 1, ')'), '-', 0.5, ')')
```

همانطور که دیده می‌شود تابع را دقیقا بدست آورد و مقدار شایستگی آن ۱ است و تعداد نسل های گذشته برابر با ۱۳ است و زمان الگوریتم ۵ ثانیه است. تابع بدست آمده از الگوریتم هم به صورت (X+1)-0.5 است که با تابع واقعی ما یکسان است.

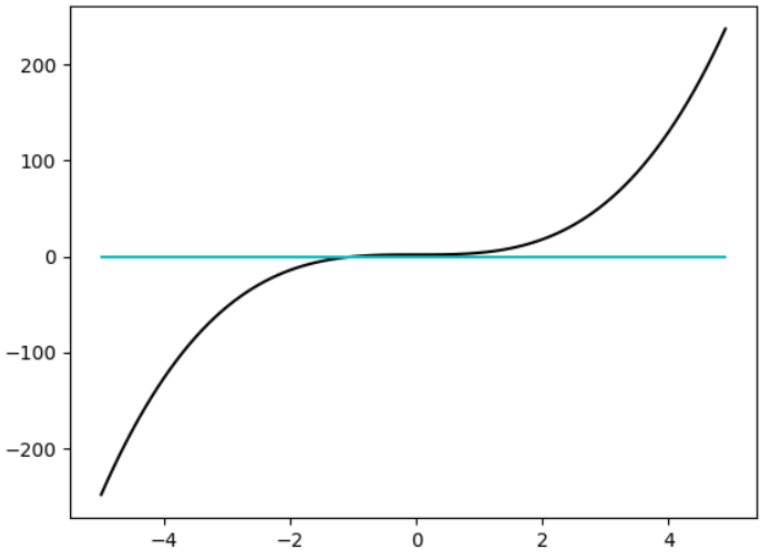
حال به سراغ تست تابعی ساده با شیب منفی می‌شویم. تابع تست x+3- است. برای آن داریم:



```
Iteration: 9
Competency 1.0
RunTime is: 3.7670016288757324
Time we calculate Compentency: 7200
('(', ('(', 'x', '-', 3, ')'), '/', -1, ')')
```

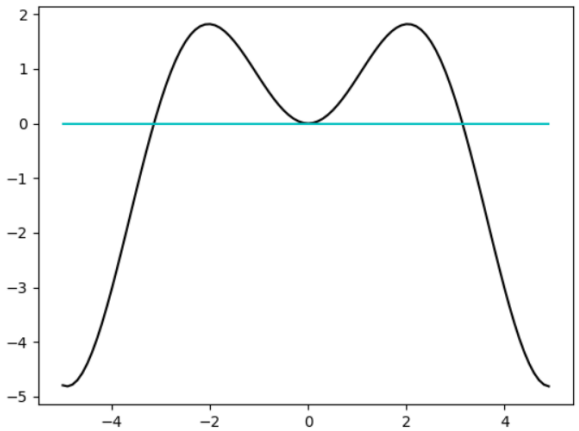
حال به سراغ توابع کمی پیچیده تر می‌رویم. تابع تست 2x^3 + 2 است و ورودی‌های ما مقادیر -۵ تا ۵ است.
با اجرای پروژه داریم:

```
Iteration: 43
Competency 1.0
RunTime is: 20.97300124168396
Time we calculate Compentency: 34400
('(', ('(', ('(', ('(', ('(', ('(', ('(', 'x', '^', 3, ')'), '/', 1, ')'), '+', 1, ')'), '*', 0.5, ')'), '*', 1, ')'), '*', -1, ')'), '*', -4, ')')
```



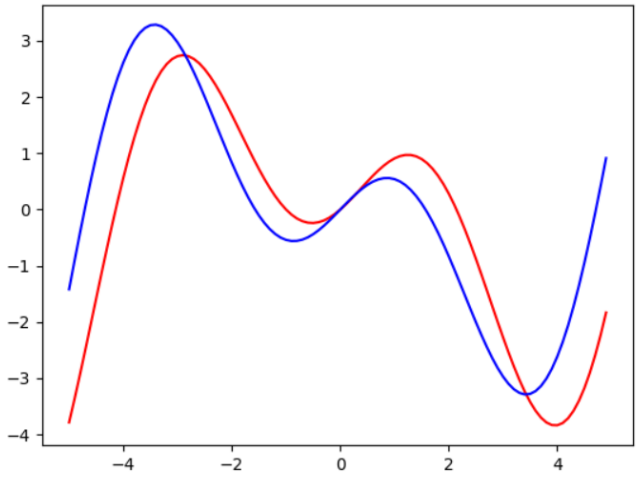
همانطور که دیده می‌شود این تابع نیز پس از ۴۳ نسل پیدا شد و نسبت به تابع ساده تر زمان بیشتری برای یافتن آن صرف شده و تقریبا ۲۱ ثانیه برای یافتن تابع دقیق زمان برده شده است. تابع آن نیز در صورت ساده کردن دقیقا با تابع تست یکسان خواهد بود.

تابع بعدی را کمی پیچیده‌تر در نظر گرفته‌ایم. تابع x*sin(x) به عنوان تابع تست در نظر گرفته شده است. با اجرای الگوریتم داریم:



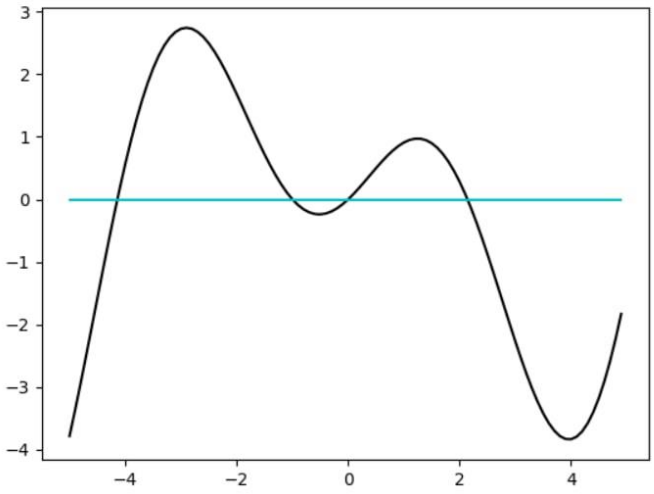

```
Iteration: 3
Competency 1.0
RunTime is: 1.1269755363464355
Time we calculate Compentency: 2400
('(', ('sin(', 'x', ')'), '*', 'x', ')')
```

همانطور که دیده می‌شود این تابع پیچیده نیز به راحتی تابع آن یافت شد برای چالشی شدن آن در تابع تست بعدی از تابع x*sin(x+1) استفاده می‌کنیم. برای این تابع تست داریم:



```
Iteration: 200
Competency 0.7489977128306796
RunTime is: 100.71899962425232
Time we calculate Compentency: 160000
(('(', ('cos(', 'x', ')'), '*', 'x', ')')
```

در یکی از تست ها شکل به صورت بالا درآمده رنگ آبی نشان دهنده تابع تخمینی و رنگ قرمز تابع اصلی و تست است که از دقت خوبی برخوردار نیست. بار دیگر الگوریتم را اجرا می‌کنیم. نتایج به شکل زیر است:



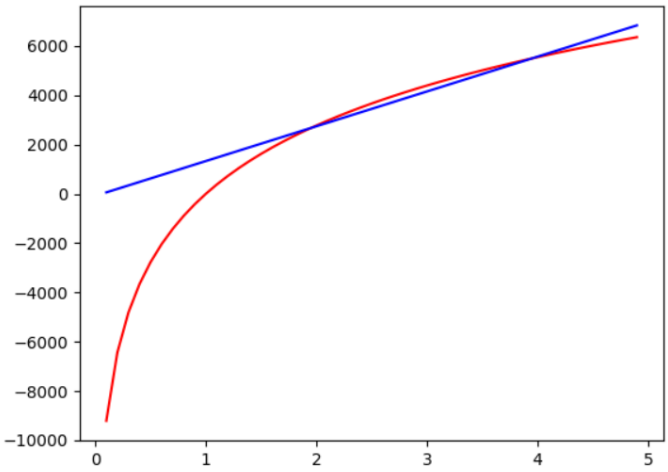
```
Iteration: 92
Competency 1.0
RunTime is: 55.74975919723511
Time we calculate Compentency: 73600
(('(', ('sin(', (('(', 'x', '-', -1, ')'), ')'), ')'), '*', 'x', ')')
```

توجه: الگوریتم به صورت تصادفی است و در اجرا های مختلف ممکن است نتایج یکسان نباشد اما در توابعی که عملگرهای آن را داریم با احتمال خوبی تابع دقیق پیدا می‌شود.

همانطور که در تست دوم دیده می‌شود تابع دقیق یافت شده و نمایش داده شده است.در این مجموعه تست ها اکثرا از سرعت و دقت خوبی برخوردار است و شایستگی اکثر تابع ها زیاد است و از طرفی زمان آن نیز بهتر است تابع های بالا در بخش کد زیر است(برای هر بخش کافی است که از حالت کامنت در آورده شود و استفاده شود)

```
#x=[x for x in np.arange(-5 , 5 , 0.1)]
#y = list()
#for i in range(len(x)):
    #y.append(x[i]*np.sin(x[i]+1))
    #y.append(x[i]+0.5)
    #y.append(7*x[i]**4 + 2)
    #y.append(x[i]*np.sin(x[i]))
    #y.append(2*x[i]**3 + 2)
    #y.append(6*np.sin(x[i]) + 3)
    #y.append(-1*x[i]+3)
    #y.append(2*x[i]**3 + 8*x[i]**2 + 2*x[i])
```

حال به تست تابعی می‌پردازیم که در عملگرهای ما وجود ندارد. تابع تست ما 4000*log(x) است. حال پروژه را برای این تابع اجرا می‌کنیم:



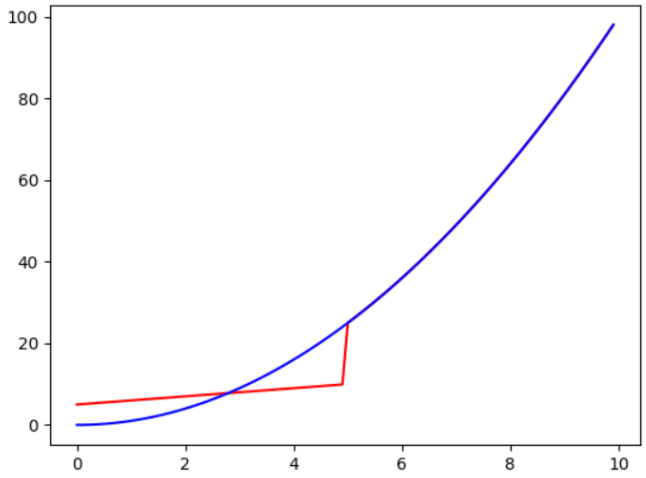

```
Iteration: 200
Competency 0.8923526994312817
RunTime is: 127.0793092250824
Time we calculate Compentency: 160000
(('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(
', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', 'x', '*', 4, ')'), '*', 8, ')'), '*', 8, ')'), '+', -7, ')'), '+', -6, ')'), '*', 6, ')'), '-', -9, ')'), '-
', 5, ')'), '-', 3, ')'), '+', 0.5, ')'), '-', 5, ')'), '/', 1, ')'), '+', -9, ')'), '+', 0.5, ')'), '-', 7, ')'), '-', -4, ')'), '-', 4, ')'), '-', 8, ')'), '-', 0.5, ')'), '-', -3, ')'), '/', 3, ')')
, '-', -1, ')'), '*', 1, ')'), '-', 3, ')'), '*', 4, ')'), '+', -6, ')'), '+', -6, ')'), '/', 'x', ')'), '*', 'x', ')'), '-', -7, ')'), '*', 3, ')'), '/', 7, ')'), '+', 9, ')'), '-', 5, ')'), '/', 1, '
)'), '-', -2, ')'), '-', 6, ')'), '/', -7, ')'), '*', -9, ')'), '-', -2, ')'), '-', 2, ')'), '+', -2, ')'), '+', 4, ')'), '+', 'x', ')'), '-', -7, ')'), '-', -2, ')'), '+', 6, ')'), '/', 4, ')'), '*',
5, ')')
```

تابع به صورت بالا در آمده است همانطور که دیده می‌شود به خوبی تلاش شده که تابع بر روی مقادیر تابع واقعی منطبق شود. شایستگی تابع برابر با ۰.۸۹ است و در ۲ دقیقه محاسبات انجام شده است. در این حالت عملکرد الگوریتم همچنان قابل قبول است و تابعی منطبق شده از دقت خوبی برخوردار است

تابع لگاریتم صفحه قبل در کد زیر آورده شده است:

```
236    #y = 4000*log(x) + 2
237    #x = [x for x in np.arange(0.1 , 5 , 0.1)]
238    #y = list()
239    #for i in range(len(x)):
240    #    y.append(4000*math.log(x[i]))
```

حال به سراغ یک تابع ناپیوسته می‌رویم. در مقادیر کمتر از ۵ باشد تابع x+5 و مقادیر بین ۵ تا ۱۰ تابع x^2 است. حال پروژه را مجدد اجرا می‌کنیم:

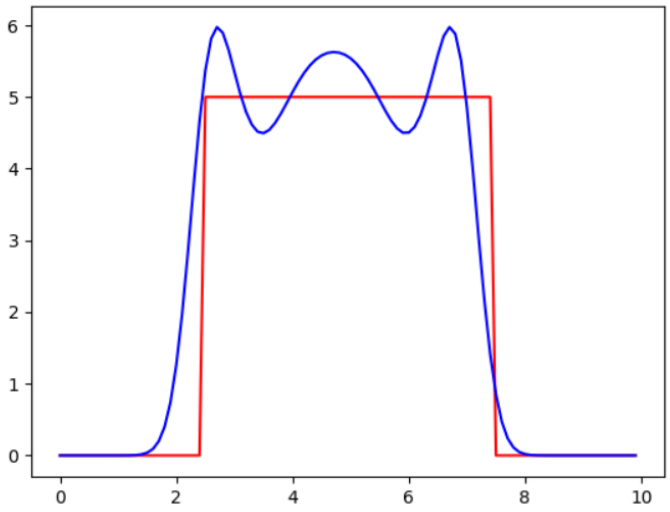



```
Iteration: 200
Competency 0.9746199367411489
RunTime is: 184.86866569519043
Time we calculate Compentency: 160000
('(', ('(', 'x', '/', -1, ')'), '^', 2, ')')
```

همانطور که دیده می‌شود الگوریتم ژنتیک ما تلاش کرده است که بخشی از تابع را که مقادیر بیشتری دارد را تطبیق دهد به همین دلیل تابع را در بخش دوم آن منطبق کرده است شایستگی آن به ۰.۹۷ رسیده و تقریبا زمان رسیدن به آن ۳ دقیقه است.از لحاظ زمانی نسبت به تابع های قبل رفتار نامناسب تری داشته است ولی دقت خوبی را دارد و تطبیق خوبی تابع اصلی و تخمینی دارد. تابع بالا نیز در کد زیر آورده شده است:

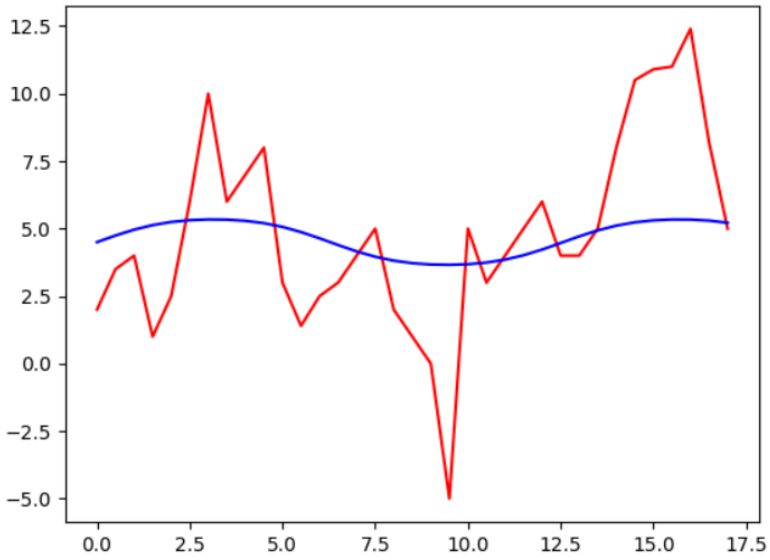
```
242    # y = x + 5 for 0<=x<=5 and y = x^2 for 5<x<=10
243    x = [x for x in np.arange(0 , 10 , 0.1)]
244    y = list()
245    for i in range(len(x)):
246        if i < (len(x)/2):
247            y.append(x[i]+5)
248        else:
249            y.append(x[i]**2)
```

یک نمونه دیگر از توابع غیرپیوسته یک تابع پله را در نظر میگیریم. نتایج آن به شکل زیر است:




```
Iteration: 200
Competency 0.905653587638759
RunTime is: 231.36961269378662
Time we calculate Compentency: 160000
('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('sin(', ('(', ('sin(', ('(', ('(', ('(', ('(', ('(', ('sin(', ('sin(', ('sin(', ('(', ('(', ('sin(', ('sin(', ('(', ('(', ('sin(', ('(', 'x', '/', 3, ')'), ')'), '*', 0.5, ')'), '/', 0.5, ')'), ')'), ')'), '^', 6, ')'), '*', 0.5, ')'), ')'), ')'), ')'), '*', 6, ')'), '*', 2, ')'), '*', 1, ')'), '/', 0.5, ')'), '*', 1, ')'), ')'), '*', 2, ')'), ')'), '*', 0.5, ')'), '*', 7, ')'), '/', 0.5, ')'), '*', 0.5, ')'), '/', 3, ')'), '/', 0.5, ')'), '/', 3, ')'), '*', 7, ')'), '*', 3, ')'), '/', 3, ')'), '*', 3, ')'), '/', 3, ')'), '/', 3, ')'), '*', 1, ')'), '*', 3, ')'), '/', 0.5, ')'), '/', 3, ')'), '^', 3, ')'), '/', 8, ')')
```

حال به سراغ یک تابع کاملا تصادفی می‌رویم. به تست آن می‌پردازیم:




```
Iteration: 200
Competency 0.8040409833882917
RunTime is: 45.406994581222534
Time we calculate Compentency: 160000
('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('(', ('sin(', ('sin(', ('(', ('(', 'x', '/', 2, ')'), '^', 1, ')'), ')'), ')'), '+', 5, ')'), '*', 1, ')'), '/', 1, ')'), '-', 1, ')'), '*', 1, ')'), '^', 1, ')'), '^', 1, ')'), '-', 0.5, ')'), '-', 0.5, ')'), '-', -2, ')'), '-', 0.5, ')'), '^', 1, ')')
```

شایستگی نسبت داده شده به آن برابر است با ۰.۸۰ نسبتا تابع مناسبی انتخاب شده است و تابعی میانگین انتخاب شده است. تابع نسبت داده شده دقت کمی برخوردار است ولی از طرفی زمان محاسبه آن بسیار بالاتر بوده است با توجه به آن نکته که در ابتدا گفتیم بین زمان اجرا و دقت یک TradeOff وجود دارد که هرچقدر محاسبات به سمت دقیق تر شدن برود زمان بیشتری نیاز دارد.

بخش آخر: جمع بندی

با توجه به نتایج بدست آمده الگوریتم ژنتیک برای تخمین روشی به شدت مناسب است و جزو الگوریتم‌های قدرتمند و سریع برای جستجوهای محلی است. اگر بخواهیم تعداد نسل‌ها را افزایش دهیم دقت افزایش میابد ولی از طرفی سرعت آن به شدت کاهش میابد. اگر تعداد نسل‌ها را کم کنیم در این صورت سرعت بالاتری خواهیم داشت اما دقت ما کاهش خواهد یافت.

از طرفی افزایش تعداد فرزندان و جمعیت اولیه احتمال اینکه توابع در جهت درستی حرکت کنند به شدت افزایش میابد. یکی از نکاتی که در این پروژه به آن رسیدیم آن بود که در گام‌های ابتدایی اگر در جهت درستی حرکت نکند به علت اینکه گام‌های ما بزرگ است ممکن است در نهایت نتایج مطلوبی را دریافت نکنیم این اتفاق را با افزایش تعداد فرزندان تولید شده یا قرار دادن والدین در نسل بعدی می‌توان به نحوی آن را جبران کرد.

در این مثال که ما اطلاعات محدودی فقط ورودی و خروجی‌های تابع را دراختیار داشتیم ولی با این حال این الگوریتم به خوبی توانست نتایج خوبی را به ما بدهد در حالی که تقریبا آگاهانه نیست و به شدت به الگوریتم-های جستجوی محلی نزدیک است.