به نام خدا

پروژه اول درس هوش مصنوعی و سیستم های خبره

علیرضا اسلامی خواه 99521064

استاد عبدی

پاییز 1401

در ابتدا در نظر دارم منابعی که در این پروژه و همچنین یادگیری استفاده کرده ام را قید کنم.

https://en.wikipedia.org/wiki/Genetic_programming

https://www.youtube.com/watch?v=K2Hl7m2Ty_4

https://en.wikipedia.org/wiki/Gene_expression_programming

/https://www.geeksforgeeks.org/expression-tree

https://github.com/primaryobjects/genetic-programming

https://github.com/GeorgianBadita

https://www.geeksforgeeks.org/diagonal-traversal-of-/binary-tree

در کنار این داک 3 فایل ضمیمه شده که در آنها کنار هر تکه کد توضیحات مربوطه را داده ام.

فایل های به ترتیب:

Main.py که وظیفه ران کردن و راه اندازی پروژه را دارد و همچنین در آن تابع اولیه و جمعیت اولیه مشخص میشود.

Random_expression_tree.py که در این فایل نحوه ساختن درخت و محاسبه fitnessو همچین محاسبه اینپوت دلخواه آورده شده.

Genetic_programing.py که در آن توابع اصلی ما مانند mutate و cross over وجود دارند.

ابتدایی ترین چالش این پروژه نگاشت درخت عبارت ریاضی رندوم بود که در ابتدا چندین نظریه برای درخت مطرح شد که معروف ترین آنها اینگونه بود که یک روت در نظر گرفته و بیاییم چپ و راست برای آن رندوم تولید کنیم ولی در این روش برای کراس اور به دردسر میخوریم زیرا تولید نود تصادفی کمی دشوار میشد بعد از آن گفتیم از لیست استفاده کنیم ولی مشکلی که این روش داشت این بود که ما چون از اول باید نودها را پر میکردیم و تعداد زیادی نود چپ و راست نداشتند پس درون لیست ما ناچارا تعدادی زیادی نال قرار میگرفت برای حل این مشکل گراف را اینگونه تعریف کردیم از چپ شروع میکنیم و تا هرجا که لازم بود همین چپ را در عمق ادامه میدهیم تا به یک متغیر برسیم سپس آن را برمیگردانیم البته برای این روش بنده از اینترنت کمک گرفتم که این روش پویا را معرفی کرده بود. لذا در آخر تابع :

:def random_expression_tree(self,height = 0)
تولید درخت تصادفی

در نوشتم. البته در این راه از منابعی هم کمک گرفتم.

تابع بعدی که برای فرآیند درخت ما لازم بود تابع calculate_tre

بود که وظیفه گرفتن ورودی و دادن خروجی خا درخت ایجاد شده را داشت. بعد از ساختن درختها اینبار نوبت به محاسبه درخت میباشد به طوری که اگر یک یا چند ورودی به آن بدهیم بتواند خروجی را به ما برگرداند به طوری که تمام ورودیها در آن دخیل بوده انددر این تابع ما به صورت بازگشتی حرکت میکنیم به طوری که اگر به عبارتی رسیدیم که دو ورودی میگرفت عبارت چپ و راست برایش مشخص میکنیم و در آخر آنها را با هم جمع یا تفریق و غیره میکنیم.

در اصل برای این پروژه اگر بخواهیم به طور خلاصه توضیح دهیم ما میخواهیم از تولید درخت های زیاد و زاد و ولد آنها به تابعی برسیم که از ابتدا میخواستیم یا حداقل نزدیک شویم. اگر به موضوع دقیق تر شویم زاد و ولد هنگامی اتفاق میفتد که قوی ترین درخت ها انتخاب شوند. حال قوی ترین ها جه زمانی

انتخاب میشوند؟ زمانی که به ازای ورودی های یکسان کمترین اختلاف را با تابع اصلی داشته باشند.

سپس با توجه به خواست مسئله میتوانیم از یک یا چند درخت قوی یک یا چند فرزند درست کنیم. در انتها هم قوی ترین درخت را برداشته و به عنوان جواب مسئله طرح میکنیم. پس با توجه به توضیحات فوق اولین قدم ساخت درخت و توابع مربوط به آن بود که توضیح داده شد.

برای این پروژه بنده مجموعا 6 عملگر و 2 متغیر را در نظر گفتم که بنظرم کافی میباشد برای حل مسئله که آنها را در دو ارایه نوشته ام.

```
exp = {1:['+', '-', '*', '/'],2:['sin','cos']}
val = ['x','y']
```

در اینجا در قسمت اندیس اول ارایه exp چهار عملگر اصلی جمع ضرب تقسیم و منها قرار دارد و در اندیس دوم دو عملگرد sin و cos قرار دارد. علت جدا کردن آنها هم این بود که دو عملگر دیگر یک ورودی میگیرند. در ارایه val هم که x و y که متغیر های اصلی ما هستند وجود دارند.

در فایل اصلی ما ، برای اینکه تابع سمپل اولیه را مشخص کنیم آن را اینگونه تعریف کردیم:

```python
def main_function(x,y):
    """
    این تابع در واقع تابع اصلی ما و همان چیزی است که ما باید به آن با
    استفاده از الگوریتم ژنتیک برسیم سپس میتوانیم با استفاده از این تابع
    مقادیر مختلف را بدست آوریم
    """
    result = (x + y) * (x - y)
    return result
```

و همینطور تابع شایستگی ما هم طبق چیزهایی که در جزوه درس نوشته شده بود اینگونه تعریف شد:

```python
    def fitnessss (self,input1,input2,output):
        """
        این تابع برای محاسبه فیتنس درخت است که به طوری که اگر خروجی درخت با
        خروجی داده شده مطابقت داشت
        فیتنس بیشتری داشته باشد
        """
        result = 0
        for i in range(len(input1)):
            result += (self.calculate_tree(input1[i], input2[i], 0)[0] - output[i]) ** 2
        # print(result)
        r = str(result)
        n = len(r)
        if n > 20:
            r = r[:25]
            result = float(r)
        self.fitness = result / (len(input1))
        return self.fitness
```

که دلیل شرط آخر این بود که ما گاها برای شایستگی ها اعداد بسیار بزرگی داشتیم و باید آنها را کوتاه میکردیم.

و همینطور درختی که استفاده شده از درخت های رندوم معروف که در یوتیوب توضیح داده شده بود الهام گرفتم که بنظرم روشی منطقی تر از روش های دیگر بود.

در این پروژه ابتدا جمعیت اولیه ای به اندازه 4000 تا و درخت هایی به عمق 4 ایجاد شدند که بنظرم برای پروژه کفایت میکردند.

نحوه انتخاب والدین بدین صورت بود که ما خیلی خطی طور آمدیم و از درخت هایی با شایستگی بالا بهترینها را انتخاب و برای زاد و ولد انتخاب کردیم.

سپس برای عمل cross over هم از والد اول و هم از والد دوم یک عدد رندوم برداشتیم، حالا یک تابع دیگر تعریف کردیم که می آمد و تا جایی که درخت به یک عملگر میرسید را جدا و سپس این بازه را به درخت بچه که ایجاد شده بود متصل میکرد.

این تابع که در واقع اصلی ترین تابع ما هست برای کراس اور کردن و به نوعی ساختن یک بچه از بین دو والد و اضافه کردن آن به جمعیت است و بدین صورت کار میکند که ابتدا از والد اول یک ایندکس رندوم

و سپس از والد دوم یک ایندکس رندوم دیگر انتخاب کرده و اینها را به هم متصل میکند. نحوه اتصال تکه های والد به فرزند همانطور که در لینکی که به همین تابع الصاق کردم توضیح داده بدین صورت است که

ابتدا مانند تکه کد زیر عملیات cross over را انجام دادم ولی سپس به روش موثرتری رسیدم.

```
end2 = self.traverse(index2,parent2)
# result.graph = parent1.graph[:index1] + parent2.graph[index2:]
result.graph = parent1.graph[:index1] + parent2.graph[index2 : end
```

در اینجا خیلی عادی تا عدد رندوم اول را از مادر و سپس از پدر به ارث میبرد. ولی سپس از روش بهتری که استفاده کردم باعث شد جواب های نزدیک تر داشته باشم.

```
# result.graph = parent1.graph[:index1] + parent2.graph[index2:]
result.graph = parent1.graph[:index1] + parent2.graph[index2 : end2] + parent1.graph[end2 :]
return result
```

در اینجا end هم تعریف شده بدین صورت که زمانی که ما به یک متغیر میرسیم آن نقطه نقطه پایان ماست. در این تکه کد به ترتیب اول و آخر از مادر و وسط درخت فرزند از پدر به ارث میبرد.

در آخر هم mutation انجام شد که به جای عبارت ها عبارت دیگر و به جای متغیرها هم متغیری دیگر میگذاشت.

تست ورودی های مختلف:

تابع سمپل :

```
result = (x + y) * (x - y)
```

خروجی:

```
[ x ]
The final expression is : * * + + x x / x y - / y x / x y * + - y x / x x x
```

که ظابطه آن به این صورت است:

$$(((2x)+(x/y))*((y/x)-(x/y)))*((y-x)+1)*x*x)$$

```
result = x+y+100
```

```
['y']
The final expression is : * + * sin cos x y y sin cos x x
```

```
return x**2
```

```
['*', 'x0', 'x0']
```