به نام خدا

دانشکده مهندسی کامپیوتر

درس هوش مصنوعی- ترم بهار 1401-1400

استاد درس: دکتر محمدحسین رهبان

گزارش سوال اول عملی تمرین دوم درس هوش مصنوعی

نام و نام خانوادگی: امیرحسین حاجی محمد رضایی

شماره دانشجویی: 99109252

بخش اول:

در تابع plot_func در ابتدا تابع و ابتدا و انتها دامنه تابع ورودی گرفته میشود و برای رسم نمودار، با گام های به اندازه 0.01 از ابتدا به انتها دامنه نقاط را برای رسم نمودار انتخاب کرده و در لیست x قرار داده و مقادیر منتاظر آنها در تابع را د ر لیست y قرار داده و در نهایت نمودار آنها را رسم میکنیم:

```
def plot_func(func, start, end):
    x = []
    y = []

    point = start
    while point <= end:
        x.append(point)
        y.append(func(point))
        point += .01

    plt.plot(x, y)
    plt.xlabel('x')
    plt.ylabel('y')
```

```
    plt.title('function')
    plt.show()
```

و تابع های f1 و f2 و f3 به ترتیب تابع های داده شده هستند که این توابع مقدار ورودی را دریافت کرده و خروجی میدهند:

```
def f1(x):
    ans = ((x**4)*(math.e ** x) - math.sin(x)) / 2
    return ans



def f2(x):


    ans = 5 * math.log(math.sin(5 * x) + math.sqrt(x), 10)
    return ans



def f3(x):
    ans = math.cos(5*math.log(x, 10)) - x ** 3/10
    return ans
```

بخش آ) با توجه به شکل نمودارها، تابع اول f1 محدب است، تابع f2 نه محدب است و نه مقعر، تابع f3 مقعر است.

بخش ب) میتوانیم این روش را بگوییم که به تعداد دفعات مشخص، یک نقطه رندوم از دامنه را انتخاب کرده و در بازه‌ای از همسایگی آن نقطه، هربار نقطه‌ای با کمترین/ بیشترین مقدار تابع را انتخاب کنیم تا اینکه به یک نقطه ماکسیمم محلی و یا مینیمم محلی برسیم و نتایج در هر دفعه را مقایسه کنیم و بیشترین/کمترین آنهار را انتخاب کنیم. (مانند روش صخره نوردی)

بخش دوم:

در اینجا در ابتدا به توضیح تابع روش کاهش گرادیان میپردازم:

```
def gradient_descent_1d(func, start, end, learning_rate, max_iter):
    x_point = random.uniform(start, end)

    while max_iter:
        derivative = first_order_derivative(func, x_point)

        x_point -= learning_rate * derivative

        max_iter -= 1
```

```
    print(x_point, func(x_point))
```

این تابع فقط برای توابع با یک ورودی قابل استفاده است. در این تابع در ابتدا خود تابع، ابتدا و انتها دامنه تابع، مقدار learning rate و تعداد بیشینه دفعات تکرار ورودی گرفت میشود. در ابتدا نقطه ای رندوم از دامنه انتخاب شده و در حلقه اصلی، هردفعه در نقطه‌ای که در زمان حال داشتیم، مشتق تابع را در آن نقطه حساب کرده و با ضرب کردن مقدار learning rate در آن، آنرا از x_point کم میکنیم اینکار را به تعداد دفعات max_iter انجام میدهیم. برای مشتق گرفتن از تعریف آن (تعریف حدی) و به صورت زیر استفاده میکنیم:

```
def first_order_derivative(func, x):
    ans = (func(x + .00001) - func(x)) / .00001
    return ans
```

مقدار تابع را در دو نقطه x+0.0001 و x حساب کرده و تقسیم بر 0.0001 میکنم تا مقدار مشتق تابع در نقطه x بدست بیاید.

بخش آ) به ازای همه مقادیر learning_rate ، تعداد دبیشینه دفعات برابر 100 قرار داده شده است.

Learning_rate = 0.1: x=0.49292181460844925, y= -0.1882777780598724

Learning_rate = 0.4: x=0.4929218146104918, y= -0.1882777780598724

Learning_rate = 0.6: x=0.5421050872886206, y= -0.18371323830403488

Learning_rate = 0.9: x=0.30191360569973297, y= -0.14305540386179277

بخش ب) با توجه به نتایج بدست آمده میتوان گفت که هر چه مقدار learning rate کوچکتر باشد، درنهایت به جواب درست میرسیم و همگرایی به جواب بیشتر است. دلیل آن این است که هرچه با قدم های کمتری حرکت کنیم، دقت حرکت و کمینه کردن تابع نیز بیشتر خواهد شد اما اگر قدم های بزرگ تری برداریم، ممکن است از مسیر درست برای کمینه کردن تابع منحرف شویم.

بخش ج) با توجه به اینکه فقط تابع f2 است که نه محدب است و نه مقعر و دامنه آن $[2,6]$ است، باتوجه به نمودار کمینه سراسری آن در نقطه 2.186 با مقدار 1.598- میباشد. با 1000 بار اجرا روش کاهش گرادیان با هرکدام از مقادیر learning_rate ، تقریبا در 0 درصد موقع این نقطه پیدا شد یا به عبارتی نتوانستند این کمینه را بیابند (میتوان به این موضوع اشاره کرد که تابع رفتاری مانند توابع تناوبی دارد که همین موضوع میتواند باغث پیدا نشدن این کمینه شود.)

بخش سوم:

در اینجا به روش نیوتون-رافسون میپردازیم که تابع آن در فایل ارسالی به صورت زیر است:

```
def newton_raphson(func, start, end, max_iter):
    x_point = random.uniform(start, end)

    while max_iter:
```

```
        f_derivative = first_order_derivative(func, x_point)
        s_derivative = second_derivative(func, x_point)

        x_point -= (f_derivative / s_derivative)
        max_iter -= 1

    print(x_point, func(x_point))
```

در این تابع، یک تابع و نقاط ابتداو انتها دامنه و بیشیته دفعات تکرار ورودی گرفته میشوند. در ابتدا یک نقطه رندوم از دامنه انتخاب شده و در حلقه اصلی طبق رابطه داده شده، مشتق اول و دوم محاسبه شده و نقطه جدید دست می‌آید و اینکار به اندازه تعداد دفعات ورودی گرفته شده انجام خواهد شد. برای محاسبه مشتق دوم، از تعریف آن و همان روش مشتق گرفتن به صورت زیر استفاده شده است:

```
def second_derivative(func, x):
    ans = (first_order_derivative(func, x + .00001) - first_order_derivative(func, x)) / .00001
    return ans
```

که در اینجا مشتق تابع در دو نقطه x+0.00001 و x محاسبه شده و اختلاف آنها تقسیم بر 0.00001 میشود.

با تکرار 1000 بار اجرا این تابع بر تابع f2 که نه محدب است و نه مقعر، تقریبا در 5 درصد مواقع میتوانیم کمینه سراسری را بیابیم که این مورد نسبت به روش قبلی بهتر است چراکه نسبت به روش قبلی توانسته در مواقعی کمینه را بیابد.

بخش چهارم:

در این بخش به روش کاهش گرادیان برای تابع دو متغیره میپردازیم.

تابع روش کاهش گرادیان برای تابع دو متغیره به صورت زیر تعریف شده است:

```
def gradient_descent_2d(func, start_x, end_x, start_y, end_y, learning_rate, max_iter):
    x_point = random.uniform(start_x, end_x)
    y_point = random.uniform(start_y, end_y)

    x_ans = [x_point]
    y_ans = [y_point]

    while max_iter:
        grad = gradient(func, x_point, y_point)
        x_point -= learning_rate * grad[0]
        y_point -= learning_rate * grad[1]

        x_ans.append(x_point)
        y_ans.append(y_point)
```

```
        max_iter -= 1

    return x_ans, y_ans
```

در اینجا به عنوان ورودی یک تابع، ابتدا و انتها بازه های دامنه، مقدار learning rate و بیشینه دفعات تکرار ورودی گرفته میشوند. در ابتدا یک نقطه رندوم از دامنه انتخاب میشود و در حلقه اصلی و به تعداد ورودی گرفته شده هر دفعه بردار گرادیان تابع در نقطه [x_point, y_point] محاسبه شده و با ضرب کردن در مقدار learning_rate و کم کردن از مولفه های نقطه، نقطه جدید بدست می‌آیند. مختصات های نقاط در لیست x_ans و y_ans ذخیره میشوند. برای محاسبه گرادیان از تعریف آن به صورت زیر استفاده میکنم:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}$$

تابع گرادیان به صورت زیر است:

```
def gradient(func, x, y):
    partial_x = (func(x + .00001, y) - func(x, y)) / .00001
    partial_y = (func(x, y + .00001) - func(x, y)) / .00001

    return [partial_x, partial_y]
```

در اینجا در ابتدا مشتق جزیی نسبت به x و مشتق جزیی نسبت به y گرفته میشود و نتیجه آن در یک لیست خروجی داده خواهد شد.

تابع دو متغیره داده شده نیز به صورت تابع f4 در فایل ارسالی وجود دارد که دو نقطه را به عنوان ورودی میگیرد و خروجی میدهد:

```
def f4(x, y):
    ans = (2 ** x / 10000) + (math.e ** y / 20000) + (x ** 2) + (4 * y ** 2) -
(2*x) - (3*y)
    return ans
```

در اینجا با توجه به مقدار learning rate های مختلفی که داریم، با توجه به تصاویر از نمودارها، هر چه قدر که مقدار learning rate کمتری داشته باشیم، در یک تابع محدب به صورت پیوسته به کمینه سراسری نزدیک و به آن میرسیم و هر چه قدر که مقدار learning rate بزرگتر خواهد شد، مسیر رفتن به سمت کمینه سراسری گسسته تر خواهد شد و واگراتر میشود(به خاطر اندازه بزرگ قدم ها در مسیر حرکت)

بخش پنجم:

در این بخش در مورد روش simulated annealing صحبت خواهیم کرد. در اینجا تابع این روش که در فایل ارسالی قرار دارد به صورت زیر است:

```
def simulated_annealing(func, start_x, end_x, initial_t, stopping_t, max_iter,
gamma, alpha):

    current = random.uniform(start_x, end_x)

    t = initial_t

    while t > stopping_t and max_iter:

        t *= gamma

        new = random.uniform(current - alpha, current + alpha)

        delta = -func(new) + func(current)

        if delta > 0:

            current = new

        else:

            p = math.e ** (delta / t)

            number = random.uniform(0, 1)

            if number <= p:

                current = new


        max_iter -= 1


    return current
```

در اینجا در ابتدا تابع هدف، ابتدا و انتها دامنه، مقدار دمای اولیه، مقدار پایین ترین دمای مجاز، بیشینه دفعات تکرار و مقدار gamma و alpha را ورودی میگیرم. درون تابغ، در ابتدا یک نقطه رندوم از دامنه را انتخاب کرده و درن حلقه اصلی تا زمانی که max_iter مخالف صفر و t بزرگتر از stopping_t باشد، هربار یک نقطه تصادفی از بازه مشخص شده حول نقطه current انتخاب شده و طبق تابع هزینه در اینجا، مقدار اختلاف هزینه محاسبه میشود که اگر مثبت بود، مقدار current را به نقطه جدید تغییر خواهیم داد و درغیر این صورت، یک نقطه رندوم بین صفر تا یک انتخاب میکنیم که اگر از مقدار $e^{\frac{delta}{T}}$ کمتر باشد، در این صورت و با این احتمال، این نقطه را مقدار current در نظر بگیرد. مقدار دما نیز از رابطه schedule و به صورت $T_n = gamma \times T_{n-1}$ تعیین خواهد شد.

باتوجه به اینکه تابع f2 ، نه محدب است و نه مقعر، از این تابع در ارزیابی این روش استفاده خواهیم کرد و نقطه کمینه سراسری در دامنه [2, 6] برابر x=2.186 خواهد بود. دراینجا با 1000 بار اجرا این تابع simulated_annealing بر تابع f2 ، در تکه برنامه زیر بررسی میکنیم که اگر اختلاف تابع جواب با x=2.186 کمتر از 0.001 باشد، در این صورت کمینه سراسری پیدا شده است.

```
count = 0

for _ in range(1000):

    try:
```

```
        x = simulated_annealing(f2, 2, 6, 10**-100, 10**-300, 10**6, 0.9, 0.1)
        if abs(x - 2.186) < 0.001:
            count += 1
    except ValueError:
        pass

print(count)
```

خطای ValueError به این خاطر در نظر گرفته شده است، که به خاطر انتخاب رندوم در بازه ها، در بعضی از اجراها مقدار current منفی خواهد شد و در این صورت در مقدار لگاریتم موجود در تابع f2 خطا ایجاد خواهد شد. در اجرای این تابع، مقدار gamma را نزدیک به یک در نظر میگیریم که دما به صورت پیوسته و آهسته کاهش یابد. در اینجا مقدار تعداد پیدا کردن کمینه سراسری به ازای هر مقدار alpha را مشاهده میکنید:

Alpha = 0.1: 238 دفعه

Alpha = 0.2: 232 دفعه

Alpha = 0.3: 226 دفعه

Alpha = 0.4: 215 دفعه

Alpha = 0.5: 200 دفعه

Alpha = 0.6: 220 دفعه

Alpha = 0.7: 209 دفعه

Alpha = 0.8: 211 دفعه

Alpha = 0.9: 234 دفعه

Alpha = 1: 204 دفعه

همانطور که در اینجا مشاهده میکنید، حداقل در 20 درصد موارد این روش میتواند کمینه محلی را بیابد. مقادیر بدست آمده توسط مقادیر مختلف alpha تقریبا به این صورت هستند که با افزایش مقدار alpha ، تعداد دفعات یافتن جواب کاهش میابد و دلیل آن میتواند این باشد که با افزایش بازه اطراف نقطه current ، احتمال انجام حرکت رندوم، با افزایش مقدار تابع در نقطه current ، افزایش میابد. البته که در بعضی از مقادیر کمتر alpha با افزایش تعداد یافتن جواب مواجه هستیم که این مورد نیز به انتخاب تصادفی در الگوریتم simulated annealing نیز وابسته است.

در نهایت این گزارش، برای تابع ای مانند f2 که در دامنه تعریف شده نه محدب است و نه مقعر، روش simulated annealing بهتر از روش نیوتون-رافسون میتواند جواب را پیدا کند (در تعداد مواقع بیشتری جواب را پیدا میکند) و همچنین

روش نیوتون-رافسون نیز نسبت به روش کاهش گرادیان برای این نوع توابع عملکرو بهتری دارد. در نتیجه میتوان برای یافتن کیمنه سراسری چنین توابعی روش simulated annealing را پیشنهاد داد.