به نام خدا

گزارش تمرین کامپیوتری پنجم
شبکه‌های عصبی

امیرمحمد رنجبرپازکی ۸۱۰۱۹۵۴۰۲

درس هوش مصنوعی

دانشکده مهندسی برق و کامپیوتر
زمستان ۹۸

● پیاده‌سازی شبکه عصبی:

در این بخش، بخش‌های ناقص اسکلت شبکه‌ی عصبی در سه کلاس Input و PerformanceElem و Neuron تکمیل شد. توضیحات و کد بخش‌ها در زیر به تفکیک آمده‌است.

● کلاس Input:

● تابع Output:

این تابع خروجی المان ورودی را مشخص می‌کند. این المان تنها مقدار ورودی را نگه می‌دارد پس خروجی آن مقدار ورودی است.

● تابع dOutDx:

این المان هیچ وزنی ندارد پس مشتق آن به ازای هر المانی مقدار صفر است.

● کلاس Neuron:

● تابع Output:

محاسبه خروجی هر یک از neuronها به این صورت است که ورودی‌هایش را در وزن متناظر آن‌ها ضرب می‌کند و بر روی آن activation function اعمال می‌کند که ببیند متعلق به کدام کلاس است.

$$WeightSum = \Sigma x_i w_i$$

$$Sigmoid(x) = \frac{1}{1 + e^x}$$

$$output = sigmoid(WeightSum)$$

● تابع dOutDx:

برای محاسبه‌ی مشتق، از output نسبت به وزن مورد نظر مشتق می‌گیریم.

در صورتی‌که این وزن، وزنی باشد که مستقیما به آن وصل است مشتق آن ورودی به ازای آن وزن است.

$$\frac{\sigma Output}{\sigma w_i} = sigmoid' \times x_i$$

در صورتی که این وزن مربوط به لایه‌های قبل شود، مجبور به استفاده از قاعده مشتق زنجیره‌اس هستیم و باید از مشتق نورونی که از آن ورودی گرفته‌ایم استفاده کنیم. x_j تمام ورودی‌هایی هستند که در محاسبه آن‌ها w_i تاثیرگذار بوده‌است.

$$\frac{\sigma Output}{\sigma w_i} = sigmoid' \times \Sigma w_j \frac{\sigma x_j}{\sigma w_i}$$

- **کلاس PerformanceElem**:

- **تابع Output**:

برای محاسبه خروجی این المان از فرمول زیر استفاده می‌کنیم. این فرمول بیانگر نزدیک‌بودن جواب شبکه ما به جواب مورد نظر است.

$$Performance = -0.5 \times (d - o)^2$$

d جوابی است که مورد نظر است و o خروجی شبکه ماست. علامت منفی به این دلیل است که هر چقدر اختلاف کمتر باشد، مقدار performance بیشتر است و این تابع صعودی است.

- **تابع dOutDx**:

برای محاسبه‌ی مشتق، از output نسبت به وزن مورد نظر مشتق می‌گیریم.

$$\frac{\sigma Performance}{\sigma w_i} = \frac{\sigma Performance}{\sigma NetOutput} \times \frac{\sigma NetOutput}{\sigma w_i} = (d - o) \times \frac{\sigma NetOutput}{\sigma w_i}$$

حال که پیاده سازی شبکه به اتمام رسید، به سراغ تست‌آن می‌رویم.

- **تست کردن شبکه ساده با تست ساده:**

در این بخش، شبکه عصبی خود را با تست ساده And و OR تست کردیم و نتایج ۱۰۰ درصد گرفته شد که در عکس‌های زیر قابل مشاهده است.

```
Testing on OR test-data
test((0.1, 0.1, 0)) returned: 0.010654939503118913 => 0 [correct]
test((0.1, 0.9, 1)) returned: 0.9835615677616205 => 1 [correct]
test((0.9, 0.1, 1)) returned: 0.9835557959042682 => 1 [correct]
test((0.9, 0.9, 1)) returned: 0.9999969906368888 => 1 [correct]
Accuracy: 1.000000
```

```
Testing on AND test-data
test((0.1, 0.1, 0)) returned: 4.704254617957318e-06 => 0 [correct]
test((0.1, 0.9, 0)) returned: 0.020484490369173127 => 0 [correct]
test((0.9, 0.1, 0)) returned: 0.02048903863720659 => 0 [correct]
test((0.9, 0.9, 1)) returned: 0.9893604979736043 => 1 [correct]
Accuracy: 1.000000
```

- **پیاده سازی Finite Difference برای تست درستی مشتق گیری:**

در این بخش، با استفاده از تکنیک Finite Difference که در پایین آمده است، مقدار تقریبی مشتق را می‌توان محاسبه کرد.

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

در مسئله ما، تابع f همان خروجی performanceElement است و x وزن‌های شبکه ما هستند. این نتیجه باید به طور تقریبی با مقدار dOutdX المان برابر باشد. در تابع checkDerivatives که در فایل utility.py قرار گرفته‌است، به ازای هر وزن، مقدار FiniteDifference محاسبه می‌شود، مقدار dOutdX نیز محاسبه می‌شود و اگر اختلاف آن‌ها از حد کوچکی کمتر بود، برابر درنظر گرفته می‌شوند. اگر تمام وزن‌ها این شرط را داشتند یعنی مشتق‌ها دارند درست محاسبه می‌شود و تابع مقدار True را برمی‌گرداند. در غیر این صورت، تابع False برمی‌گرداند.

نکته‌ای که در این تابع وجود دارد، برای محاسبه f نمی‌توان خروجی مستقیم داد. به همین دلیل، وزن مورد نظر را به اندازه ϵ تغییر می‌دهیم و روی شبکه عوض می‌کنیم. حال با مقدار جدید output را به دست می‌آوریم و با حالت قبل تفریق می‌کنیم.

```
def check_derivatives(network):
    weights = network.weights
    performance_element = network.performance
    epsilon = (10 ** (-8))
    for weight in weights:
        initial_weight = weight.get_value()
        initial_performance = performance_element.output()
        weight_derivative = performance_element.dOutdX(weight)
        new_weight = initial_weight + epsilon
        weight.set_value(new_weight)
        network.clear_cache()
        new_performance = performance_element.output()
        finite_difference = (new_performance - initial_performance) / epsilon
        weight.set_value(initial_weight)
        network.clear_cache()
        if abs(finite_difference-weight_derivative) > 0.001:
            return False
    return True
```

فقط باید cache شبکه را پاک کرد که از مقادیر قبلی استفاده نکند.

- **پیاده سازی شبکه عصبی دولایه‌ای:**

با استفاده از APIهای کامل شده، ابتدا ورودی‌ها به عنوان کلاس Input تعریف شدند. سپس، وزن‌ها به عنوان نمونه کلاس Weight تعریف شدند و با استفاده از این دو موجودیت، نمونه‌های نورون تعریف شدند و به آن‌ها ورودی‌ها و وزن‌های متناظر داده‌شد. در انتها یک PerformanceElem نیز قرارداده‌شد و شبکه با استفاده از نورون‌ها و واحد ارزیابی ساخته شد. کد آن در زیر قابل مشاهده‌است.

```
def make_neural_net_two_layer():
    """
    Create a 2-input, 1-output Network with three neurons.
    There should be two neurons at the first level, each receiving both inputs
    Both of the first level neurons should feed into the second layer neuron.

    See 'make_neural_net_basic' for required naming convention for inputs,
    weights, and neurons.
    """
    i0 = Input('i0', -1.0)  # bias
    i1 = Input('i1', 0.0)
    i2 = Input('i2', 0.0)

    seed_random()
    w1A = Weight('w1A', random_weight())
    w2A = Weight('w2A', random_weight())
    wA = Weight('wA', random_weight())

    w1B = Weight('w1B', random_weight())
    w2B = Weight('w2B', random_weight())
    wB = Weight('wB', random_weight())

    wAC = Weight('wAC', random_weight())
    wBC = Weight('wBC', random_weight())
    wC = Weight('wC', random_weight())

    A = Neuron('A', [i1, i2, i0], [w1A, w2A, wA])
    B = Neuron('B', [i1, i2, i0], [w1B, w2B, wB])
    C = Neuron('C', [A, B, i0], [wAC, wBC, wC])

    P = PerformanceElem(C, 0.0)

    net = Network(P, [A,B,C])
    return net
```

این شبکه بر روی simple و harder_data_sets آزموده شد و دقت ۱۰۰ درصد خروجی داد. نتیجه دو نمونه تست در ادامه آمده است.

```
Testing on EQUAL test-data
test((0.1, 0.1, 1)) returned: 0.950586289111031 => 1 [correct]
test((0.1, 0.9, 0)) returned: 0.013485594079712695 => 0 [correct]
test((0.9, 0.1, 0)) returned: 0.01338481954291819 => 0 [correct]
test((0.9, 0.9, 1)) returned: 0.9539681376290063 => 1 [correct]
Accuracy: 1.000000
```

```
Testing on NOT_EQUAL test-data
test((0.1, 0.1, 0)) returned: 0.049413710888969085 => 0 [correct]
test((0.1, 0.9, 1)) returned: 0.9865144059202873 => 1 [correct]
test((0.9, 0.1, 1)) returned: 0.986615180457082 => 1 [correct]
test((0.9, 0.9, 0)) returned: 0.046031862370993895 => 0 [correct]
Accuracy: 1.000000
```

- **کشیدن ناحیه تصمیم‌گیری:**

در این بخش، تابع boundary_decision_plot در utility.py پیاده‌سازی شد. این تابع با گرفتن یک ناحیه از صفحه مختصات، تصمیماتی که توسط شبکه عصبی برای نقاط صفحه به عنوان ورودی گرفته می‌شود را در آن صفحه رسم می‌کند.

این تابع ناحیه مورد نظر را یک grid 500×500 تقسیم می‌کند و در صورتی‌که در هر نقطه خروجی شبکه از ۰.۵ کمتر باشد(کلاس صفر باشد)، آن را آبی می‌کند. کد پیاده سازی در تصویر زیر قابل مشاهده است.

```python
def plot_decision_boundary(network, xmin, xmax, ymin, ymax):
    x_diff = xmax - xmin
    y_diff = ymax - ymin
    x_coords = [xmin+(x_diff/500)*index for index in range(501)]
    y_coords = [ymin+(y_diff/500)*index for index in range(501)]
    decision_points_x = []
    decision_points_y = []
    for x, y in product(x_coords, y_coords):
        network.inputs[0].set_value(x)
        network.inputs[1].set_value(y)
        network.clear_cache()
        result = network.output.output()
        if result < 0.5:
            decision_points_x.append(x)
            decision_points_y.append(y)
    plot.scatter(decision_points_x, decision_points_y, color='skyblue')
    plot.show()
```

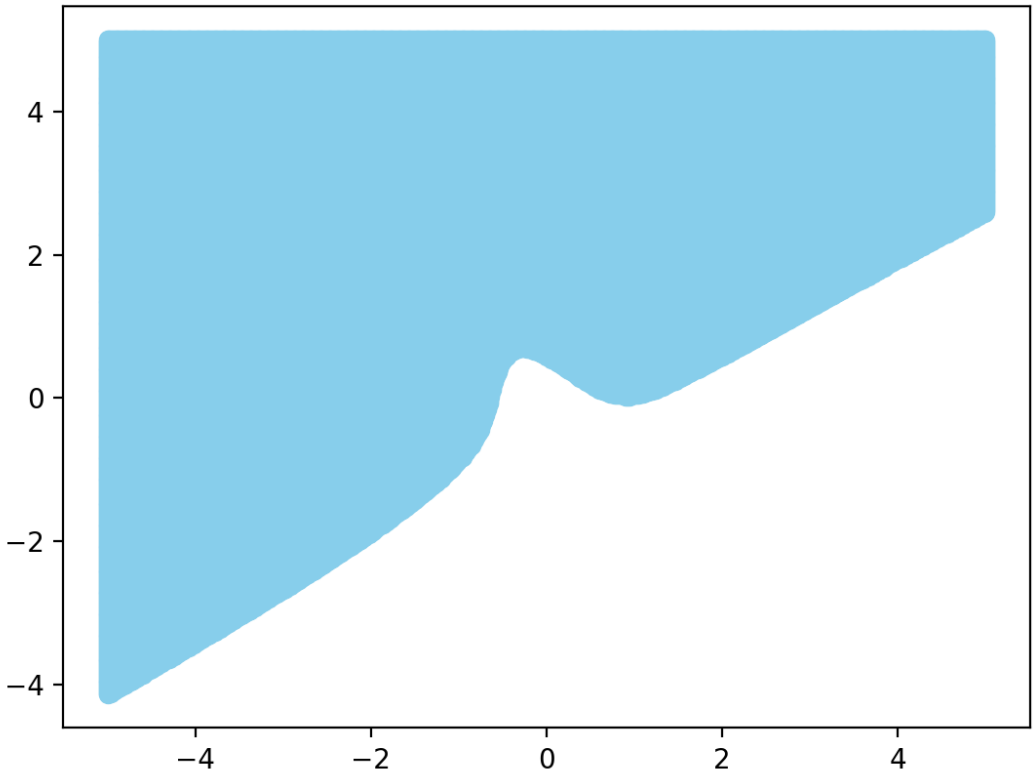
- **Overfitting و Regularization:**

در شبکه two moon در لایه اول، ۴۰ نورون گذاشته‌شده است که پیچیدگی‌اش بسیار بیشتر از داده‌های مسئله ماست. این باعث می‌شود که نویزها نیز با دقت کلاس بندی شوند و دقت ما روی داده مورد نظر بالاتر برود و حتی جزییات بی‌ارزش نیز به عنوان ویژگی به خاطر سپرده شود.

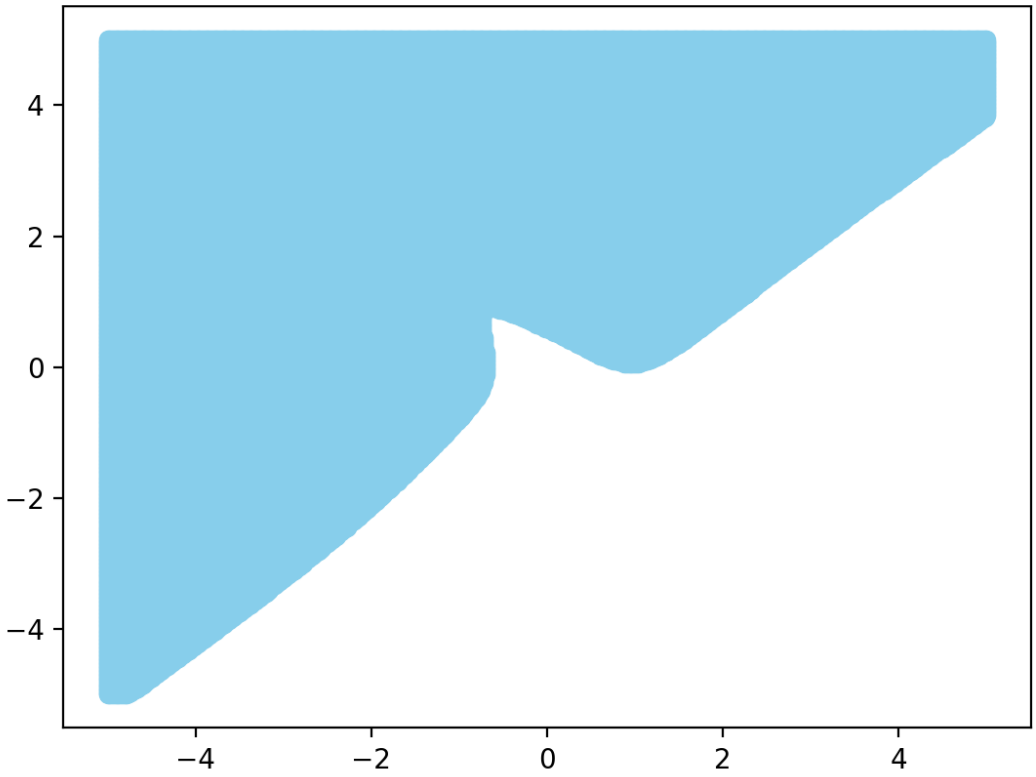
پیاده‌سازی شبکه two moon در کد زیر قابل مشاهده است.

```
def make_neural_net_two_moons():
    """
    Create an overparametrized network with 40 neurons in the first layer
    and a single neuron in the last. This network is more than enough to solve
    the two-moons dataset, and as a result will over-fit the data if trained
    excessively.

    See 'make_neural_net_basic' for required naming convention for inputs,
    weights, and neurons.
    """
    i0 = Input('i0', -1.0)  # bias
    i1 = Input('i1', 0.0)
    i2 = Input('i2', 0.0)

    seed_random()

    neurons = []
    for index in range(1, 41):
        first_input_weight = Weight('w1A' + str(index), random_weight())
        second_input_weight = Weight('w2A' + str(index), random_weight())
        bias_weight = Weight('wA' + str(index), random_weight())
        new_neuron = Neuron('A' + str(index), [i1, i2, i0], [first_input_weight, second_input_weight, bias_weight])
        neurons.append(new_neuron)

    o_weights = []
    for index in range(1, 41):
        o_weight = Weight('wA' + str(index) + 'O', random_weight())
        o_weights.append(o_weight)
    wO = Weight('wO', random_weight())
    O = Neuron('O', neurons + [i0], o_weights + [wO])
    neurons.append(O)

    # P = PerformanceElem(O, 0.0)
    P = RegularizedPerformanceElement(O, 0.0, 0.001)
    net = Network(P, neurons)
    return net
```

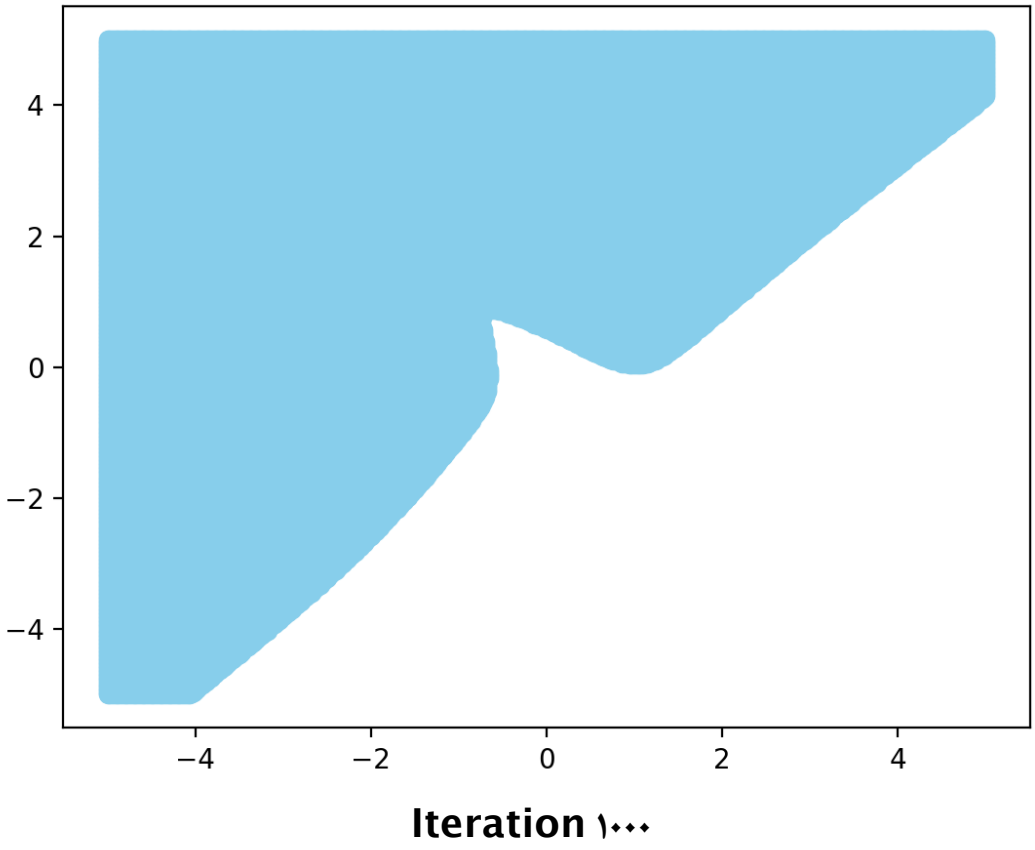
این پیاده‌سازی با تعداد ۱۰۰، ۵۰۰ و ۱۰۰۰ iteration آموزش داده‌شد و نمودارهای تصمیم‌آن روی داده‌تمرینی به صورت زیر است.



Iteration ۱۰۰



Iteration ۵۰۰



این سه تصویر نشان‌می‌دهد که هر چقدر آموزش را ادامه می‌دهیم، جزئیات بی‌ارزش بیشتری در شبکه عصبی می‌آید و گرچه دقت داده‌ی تست بالا می‌رود اما آن داده‌های اضافی درست در واقع نویز هستند و این وسواس!، دقت را در داده تستی پایین می‌آورد. جدول زیر گواه این ماجراست. از ۱۰۰ به ۵۰۰ و ۱۰۰۰ افت دقت داریم. ۱۰۰ و ۵۰۰ به ناحیه تصمیم نزدیکی دارند و حال یک داده به صورت تصادفی در آن درست شده‌است و دقت را بالا برده‌است.

Iteration #	Accuracy
100	98
500	94
1000	95

در حالت بالا، پدیده overfitting مشاهده شد.

حال برای جلوگیری از این پدیده باید به سراغ Regularization برویم. در این قسمت از l۲norm استفاده می‌کنیم. L۲norm برای یک آرایه مجموع مربعات اعضای آن آرایه می‌باشد.

برای تاثیر دادن آن، l۲norm وزن‌های شبکه را از خروجی PerformanceElem قبلی کم می‌کنیم و این خروجی کلاس RegularizedPerformanceElem را تشکیل می‌دهد. تفریق

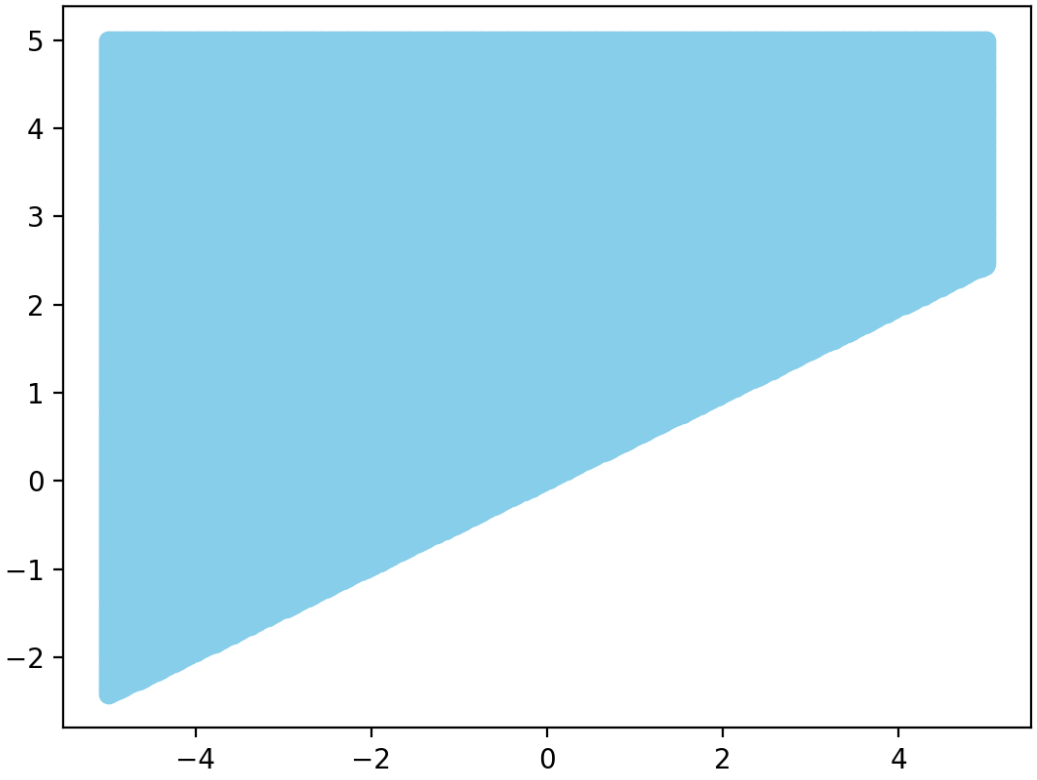
برای صعودی ماندن تابع PerformanceElem است چراکه مقدار آن منفی است. پیاده‌سازی این کلاس در زیر آمده‌است.

```
class RegularizedPerformanceElement(PerformanceElem):
    def __init__(self, input, desired_value, lambdaa):
        PerformanceElem.__init__(self, input, desired_value)
        self.lambdaa = lambdaa

    def output(self):
        weights_norm = 0
        network_weights = []
        network_weights = get_network_weights(self.get_input(), network_weights)
        for weight in network_weights:
            weights_norm += (weight.get_value() ** 2)
        weights_norm_effect = self.lambdaa * weights_norm
        return -0.5 * ((self.my_desired_val - self.my_input.output()) ** 2) - weights_norm_effect

    def dOutdX(self, elem):
        return (self.my_desired_val - self.my_input.output()) * self.my_input.dOutdX(elem) - \
               2 * self.lambdaa * elem.get_value()
```

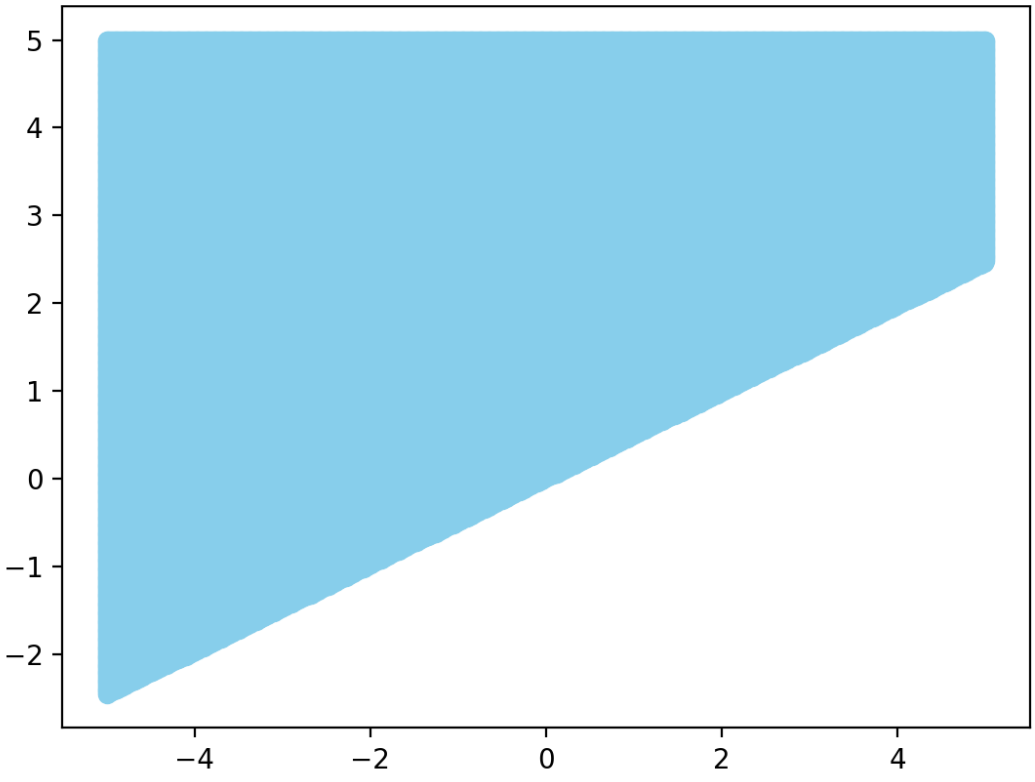
تابع get_network_weights به صورت بازگشتی وزن‌های شبکه را گردآوری می‌کند چراکه به کلاس شبکه و وزن‌های آن از داخل PerformanceElem دسترسی نداریم.

حال با جایگزنی این کلاس، مسئله two_moon را مجددا حل می‌کنیم. نمودارهای ناحیه تصمیم به صورت زیر است.

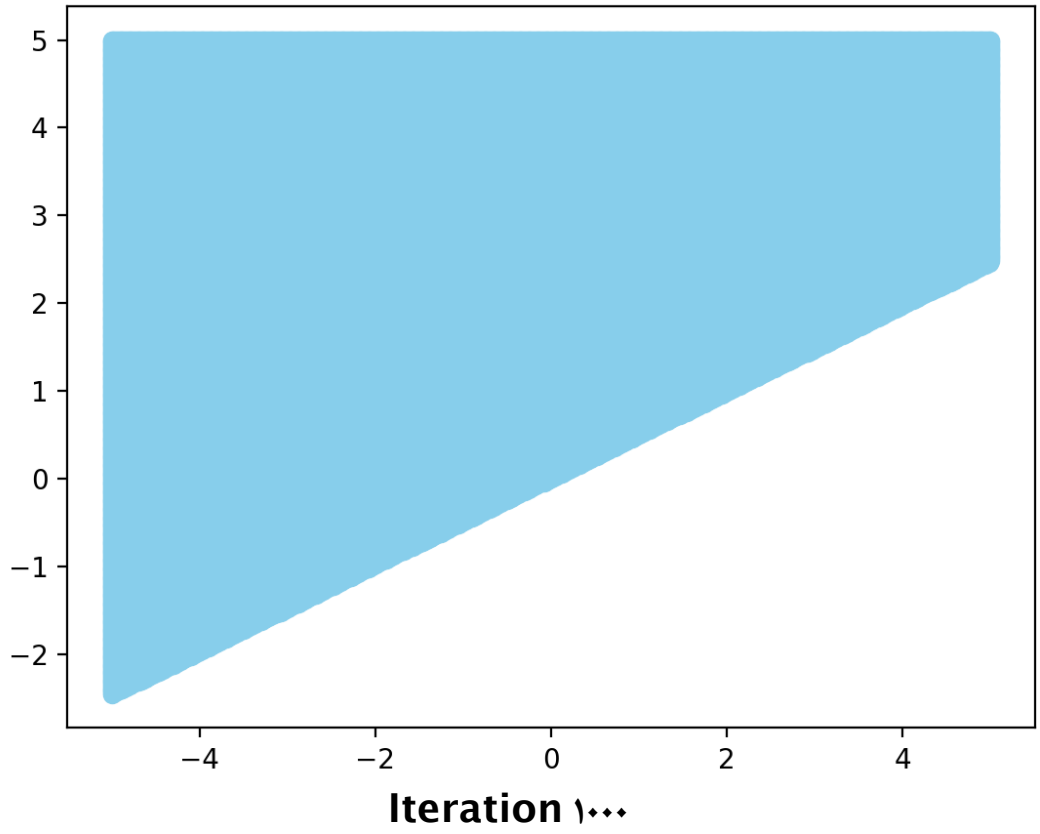
(مقدار λ با استفاده از آزمون و خطا ۰.۰۰۱ به‌دست‌آمد. این مقدار مرز تشخیص ندادن داده‌ای جزئی و بی‌ارزش بود.)



Iteration ۱۰۰



Iteration ۵۰۰



همان‌طور که در این ناحیه‌ها مشخص است داده‌های بی‌ارزش دیگر تشخیص داده نشده‌اند و جلوی overfitting گرفته شده‌است. این بهبود در جدول دقت‌ها نیز قابل مشاهده است.

Iteration #	Accuracy
100	86
500	86
1000	86

دقت ثابت شده‌است و افتی در آن مشاهده نمی‌شود چراکه مدل منطبق بر داده آموزشی نیست و روی داده تست عملکرد مساعدی دارد.

Regularization با دخیل کردن وزن‌های شبکه این مفهوم را اضافه کرد که اگر وزن‌ها مقدار زیادی تغییر بکنند، در خروجی المان Performance دیده‌می‌شود و کلاس را تغییر می‌دهد. به عبارت دیگر، نقطه‌های یک ناحیه تمایل بیشتری برای همبستگی نشان می‌دهند.