به نام خدا

دانشگاه تهران
پردیس دانشکده‌های فنی
دانشکده برق و کامپیوتر

هوش مصنوعی پاییز 98

پروژه پنجم
شبکه عصبی

نام و نام خانوادگی
علیرضا زارع نژاد اشکذری
شماره دانشجویی
810196474

۳. پیاده‌سازی شبکه‌ی عصبی:

در فایل neural_net.py بدنه‌ی اصلی یک شبکه‌ی عصبی آورده شده است. سه کلاس Input، PerformanceElem، و Neuron هر سه پیاده‌سازی‌های ناقصی از توابع زیر را دارند که باید توسط شما کامل شوند:

```
def output(self)
def dOutdX(self, elem)
```

تابع **output(self)**:

این تابع، خروجیِ هر کدام از المان‌های شبکه‌ی عصبی را تولید می‌کند. در این تابع باید از activation function سیگموید (لاجیستیک) استفاده کنید.

تابع **dOutdX(self, elem)**:

این تابع مشتق جزئی خروجی را نسبت به المان وزنی که به عنوان ورودی داده شده محاسبه می‌کند. از این مقدار برای بروزرسانی وزن‌های شبکه استفاده خواهد شد. البته در این پیاده‌سازی، به جای مفهوم loss از مفهوم performance استفاده شده که همان عکس loss است. یعنی هرچه performance بالاتر باشد بهتر است. در نتیجه فرمول بروزرسانی وزن‌های شبکه به شکل زیر خواهد بود:

```
wi' = wi + rate * dP / dwi
```

که در آن P همان مقدار Performance است.

توجه کنید که المان ورودی در این تابع، همواره یک وزن خواهد بود. شما باید فکر کنید که چطور می‌توان این تابع را با استفاده از فراخوانی‌های بازگشتی توابع dOutdX و output روی ورودی های شبکه یا سایر وزن‌ها به دست آورد. برای این پیاده‌سازی شما باید از قانون زنجیره‌ای[1] در مشتق گیری استفاده کنید.

برای مثال برای یک Performance Element با نام P، پیاده‌سازی تابع dOutdX می‌تواند به صورت زیر باشد: (در اینجا o خروجی نورونی است که که مستقیما به P متصل شده)

```
dP / d(w) = dP / do * do / dw = (d - o) * o.dOutdX(w)
```

با توجه به توضیحات فوق ابتدا توابع فوق را در کلاس‌های داده شده پیاده‌سازی می کنیم.

```python
class Input(ValuedElement,DifferentiableElement):
    """
    Representation of an Input into the network.
    These may represent variable inputs as well as fixed inputs
    (Thresholds) that are always set to -1.
    """
    def __init__(self,name,val):
        ValuedElement.__init__(self,name,val)
        DifferentiableElement.__init__(self)
    def output(self):
        """
        Returns the output of this Input node.
        returns: number (float or int)
        """
        return self.get_value()
        # raise NotImplementedError("Implement me!")

    def dOutdX(self, elem):
        """
        Returns the derivative of this Input node with respect to
        elem.
        elem: an instance of Weight
        returns: number (float or int)
        """
        return 0
        # raise NotImplementedError("Implement me!")
```

توابع پیاده‌سازی شده در performance elem

```python
def output(self):
    """
    Returns the output of this PerformanceElem node.

    returns: number (float/int)
    """
    return -0.5*(self.my_desired_val-self.my_input.output())**2
    # raise NotImplementedError("Implement me!")
def dOutdX(self, elem):
    """
    Returns the derivative of this PerformanceElem node with respect
    to some weight, given by elem.
    elem: an instance of Weight
    returns: number (int/float)
    """
    return (self.my_desired_val - self.my_input.output())*self.my_input.dOutdX(elem)
    # raise NotImplementedError("Implement me!")
```

توابع پیاده‌سازی شده در neuron:

```
def output(self):
    # Implement compute_output instead!!
    if self.use_cache:
        # caching optimization, saves previously computed output.
        if self.my_output is None:
            self.my_output = self.compute_output()
        return self.my_output
    return self.compute_output()

def compute_output(self):
    """
    Returns the output of this Neuron node, using a sigmoid as
    the threshold function.

    returns: number (float or int)
    """
    z = 0
    inputs = self.get_inputs()
    weights = self.get_weights()
    for i in range(len(inputs)):
        inp = inputs[i]
        wei = weights[i]
        z+= wei.get_value()*inp.output()
    return 1.0/(1.0 + math.exp(-z))
    # raise NotImplementedError("Implement me!")
```

```
def dOutdX(self, elem):
    # Implement compute_doutdx instead!!
    if self.use_cache:
        # caching optimization, saves previously computed dOutdx.
        if elem not in self.my_doutdx:
            self.my_doutdx[elem] = self.compute_doutdx(elem)
        return self.my_doutdx[elem]
    return self.compute_doutdx(elem)
```

```python
def compute_doutdx(self, elem):
    """
    Returns the derivative of this Neuron node, with respect to weight
    elem, calling output() and/or dOutdX() recursively over the inputs.
    elem: an instance of Weight
    returns: number (float/int)
    """
    out = self.output()
    octerm = out*(1-out)

    if self.has_weight(elem):
        index = self.my_weights.index(elem)
        oa = self.get_inputs()[index].output()
        d = octerm*oa
    else:
        d = 0
        for i in range(len(self.get_weights())):
            current_weight = self.my_weights[i]
            if self.isa_descendant_weight_of(elem, current_weight):
                input_deriv = self.get_inputs()[i].dOutdX(elem)
                d += current_weight.get_value()*input_deriv
        d *= octerm
    return d
```

تست کرد شبکه

۴. تست کردن شبکه:

بعد از اتمام پیاده‌سازی توابع، می‌توانید با استفاده از اسکریپت پایتون neural_net_tester.py کار خود را تست کنید. با اجرای دستور زیر مطمئن خواهید بود که برنامه‌ی شما برای کاربردهای ساده‌ای مثل AND و OR کار خواهد کرد:

```
Python neural_net_tester.py simple
```

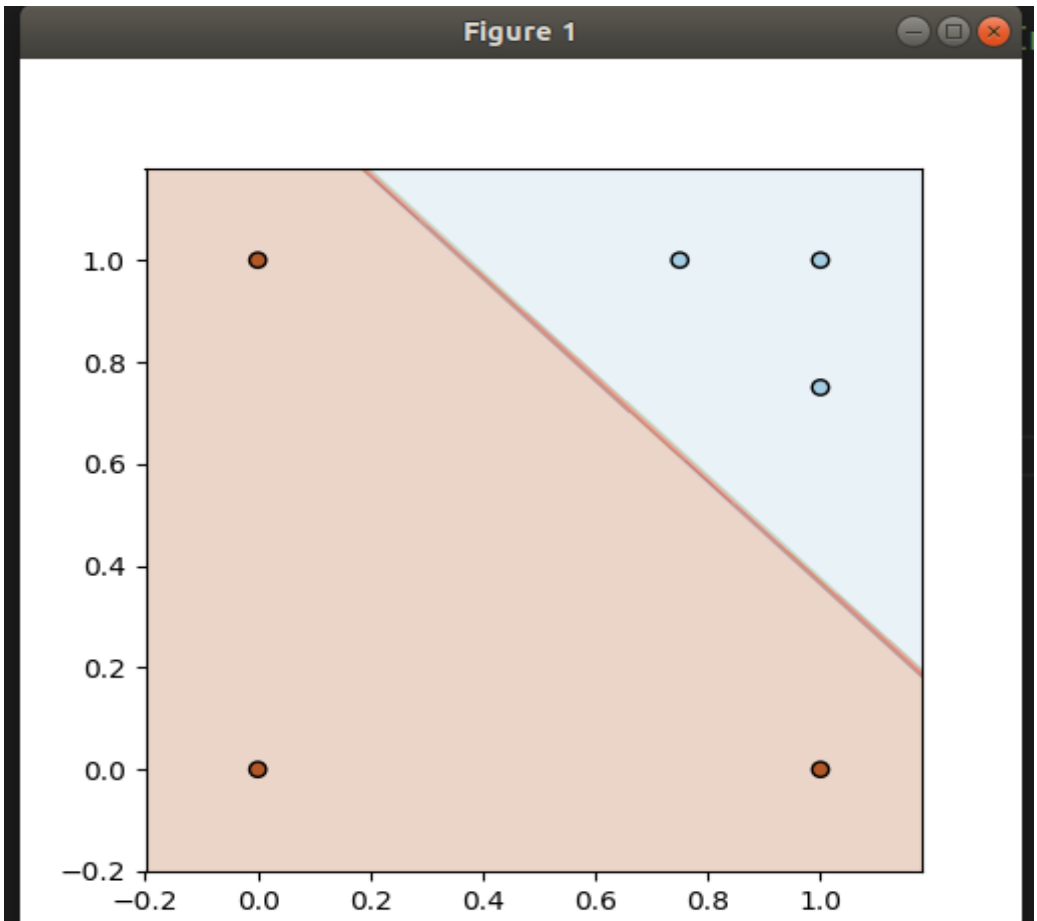
در زیر نتیجه ی تست آورده شده است.

کشیدن ناحیه تصمیم گیری

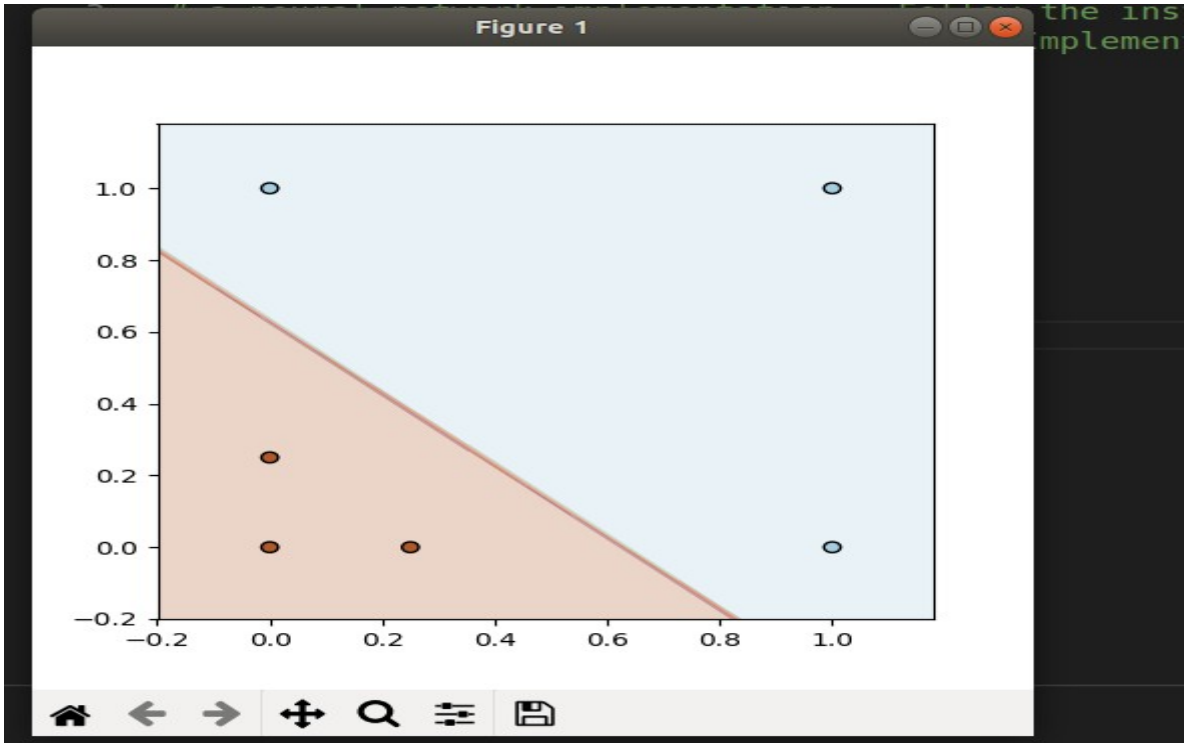
در این بخش شما باید تابعی بنویسید که با دریافت یک شبکه‌ی عصبی، و یک محدوده از صفحه در قالب یک مربع، ناحیه‌ی تصمیم گیری شبکه را در آن قسمت از صفحه رسم کند. به منظور این کار کافی است که در آن محدوده از صفحه، نقاط زیادی را به شکل یک grid ریزدانه انتخاب کنید و به ازای هر نقطه معین کنید که آیا خروجی شبکه کمتر از ۰.۵ است یا خیر، و اگر جواب مثبت بود آن نقطه‌را به نحوی روی صفحه نمایش بدهید. امضای این تابع باید به شکل زیر باشد:

```
def plot_decision_boundary(network, xmin, xmax, ymin, ymax)
```

```python
def plot_decision_boundary(network, data, xmin=-10, xmax=10, ymin=-10, ymax=10):
    print("PLOTING")
    X = np.array([[item[0], item[1]] for item in data])
    y = np.array([item[2] for item in data])
    h = 0.02
    x_min, x_max = X[:, 0].min() - 10*h, X[:, 0].max() + 10*h
    y_min, y_max = X[:, 1].min() - 10*h, X[:, 1].max() + 10*h

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    _chiz = np.c_[xx.ravel(), yy.ravel()]
    _new_data = []
    for i in range(len(_chiz)):
        _new_data.append((_chiz[i, 0], _chiz[i, 1]))
    z = []
    for datum in _new_data:
        for i in range(len(network.inputs)):
            network.inputs[i].set_value(datum[i])
        network.clear_cache()
        result = network.output.output()
        prediction = round(result)
        network.clear_cache()
        z.append(prediction)
    z = np.array(z)
    z = z.reshape(xx.shape)
    plt.figure(figsize=(5, 5))
    plt.contourf(xx, yy, z, cmap='Paired_r', alpha=0.25)
    plt.contour(xx, yy, z, colors='k', linewidths=0.02)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='Paired_r', edgecolors='k')
    plt.show()
    input('press <ENTER> to see Weights and results')
```




```
iter 9910: mean-abs-performance = 0.000124
iter 9920: mean-abs-performance = 0.000124
iter 9930: mean-abs-performance = 0.000124
iter 9940: mean-abs-performance = 0.000124
iter 9950: mean-abs-performance = 0.000124
iter 9960: mean-abs-performance = 0.000124
iter 9970: mean-abs-performance = 0.000124
iter 9980: mean-abs-performance = 0.000124
iter 9990: mean-abs-performance = 0.000123
iter 10000: mean-abs-performance = 0.000123
weights: [w1A(10.50), w2A(10.50), wA(14.37)]
```




```
iter 9970: mean-abs-performance = 0.000124
iter 9980: mean-abs-performance = 0.000124
iter 9990: mean-abs-performance = 0.000123
iter 10000: mean-abs-performance = 0.000123
weights: [w1A(10.50), w2A(10.50), wA(14.37)]
Trained weights:
Weight 'w1A': 10.499844
Weight 'w2A': 10.499561
Weight 'wA': 14.366979
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
```

قواعد نامگذاری

قواعد نام‌گذاری

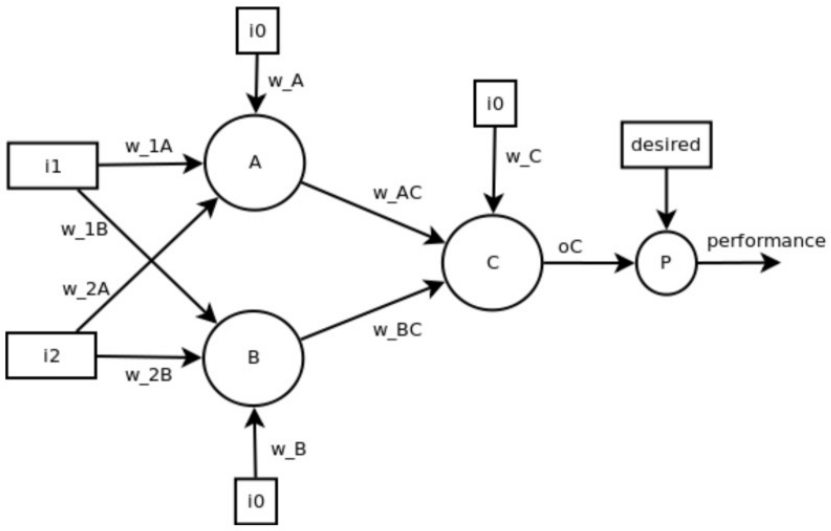
هنگام نام‌گذاری المان‌هایی که طراحی می‌کنید حتما از قواعد زیر پیروی کنید:

۱) ورودی‌ها به صورت 'i' + input_number باشند. Input_number باید از یک شروع شود. مثلا: i1, i2. از i0 برای ورودی‌های با مقدار ثابت (که ضریب آن‌ها نقش bias را در شبکه دارند) از مقدار 1- استفاده کنید.

۲) برای وزن‌ها به صورت 'w' + from_id + to_id نام‌گذاری کنید. مثلا برای وزنی که که از ورودی شماره یک به نورون A می‌رود w1A مناسب است. یا برای وزنی که از نورون A به B می‌رود کافی است wAB را به عنوان نام انتخاب کنید.

۳) برای اسم نورون‌ها باید یک حرف از حروف الفبا اختصاص دهید. به این صورت که نزدیک‌ترین نورون به ورودی A نام می‌گیرد و نورون‌های دورتر B و ... . اگر دو نورون فاصله‌ی یکسانی با ورودی دارند به هر صورت که مایل هستید ترتیبی را قائل شوید.

پیاده سازی شبکه عصبی دولایه



این کار را داخل تابع make_neural_net_two_layer() در داخل فایل neural_net.py انجام بدهید. شبکه عصبی شما باید قادر باشد دیتاست‌های کمی سخت تر مثل NOT EQUAL (XOR) یا EQUAL را طبقه‌بندی کند.

برای وزن‌های این شبکه مقادیر اولیه‌ای به صورت رندم در نظر بگیرید. دقت کنید که برای تکرارپذیر بودن تست‌ها مقدار seed را قبل از هر چیزی تعیین کنید و بعد با استفاده از تابع random_weight مقدار وزن‌های مورد نیاز را تولید کنید:

```
seed_random()
wt = random_weight()
...use wt...
wt2 = random_weight()
...use wt2...
```

نحوه پیاده سازی در زیر آورده شده است:

```
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)
    i1 = Input('i1', 0.0)
    i2 = Input('i2', 0.0)
    seed_random()
    w1A = Weight('w1A', random_weight())
    w1B = Weight('w1B', random_weight())
    w2A = Weight('w2A', random_weight())
    w2B = Weight('w2B', random_weight())
    wA = Weight('wA', random_weight())
    wB = Weight('wB', random_weight())
    wAC = Weight('wAC', random_weight())
    wBC = Weight('wBC', random_weight())
    wC = Weight('wAC', random_weight())
    A = Neuron('A', [i0,i1,i2], [wA,w1A,w2A])
    B = Neuron('B', [i0,i1,i2], [wB,w1B,w2B])
    C = Neuron('C', [i0,A,B], [wC,wAC,wBC])
    P = PerformanceElem(C, 0.0)
    net = Network(P,[A,B,C])
    return net
```

در نهایت برای تست خواهیم داشت:

```
Weight 'wAC': 9.031638
Weight 'wBC': -8.787502
Testing on inverse-diagonal-band test-data
test((-1, -1, 0)) returned: 0.02488788068606077 => 0 [correct]
test((5, 5, 0)) returned: 0.014085019322693562 => 0 [correct]
test((-2, -2, 0)) returned: 0.030910450355458887 => 0 [correct]
test((6, 6, 0)) returned: 0.013631659638460891 => 0 [correct]
test((3.5, 3.5, 0)) returned: 0.015100332454052038 => 0 [correct]
test((1.5, 1.5, 0)) returned: 0.0175615259703364 => 0 [correct]
test((4, 0, 1)) returned: 0.9904819750411343 => 1 [correct]
test((0, 4, 1)) returned: 0.9878824018126686 => 1 [correct]
Accuracy: 1.000000
```

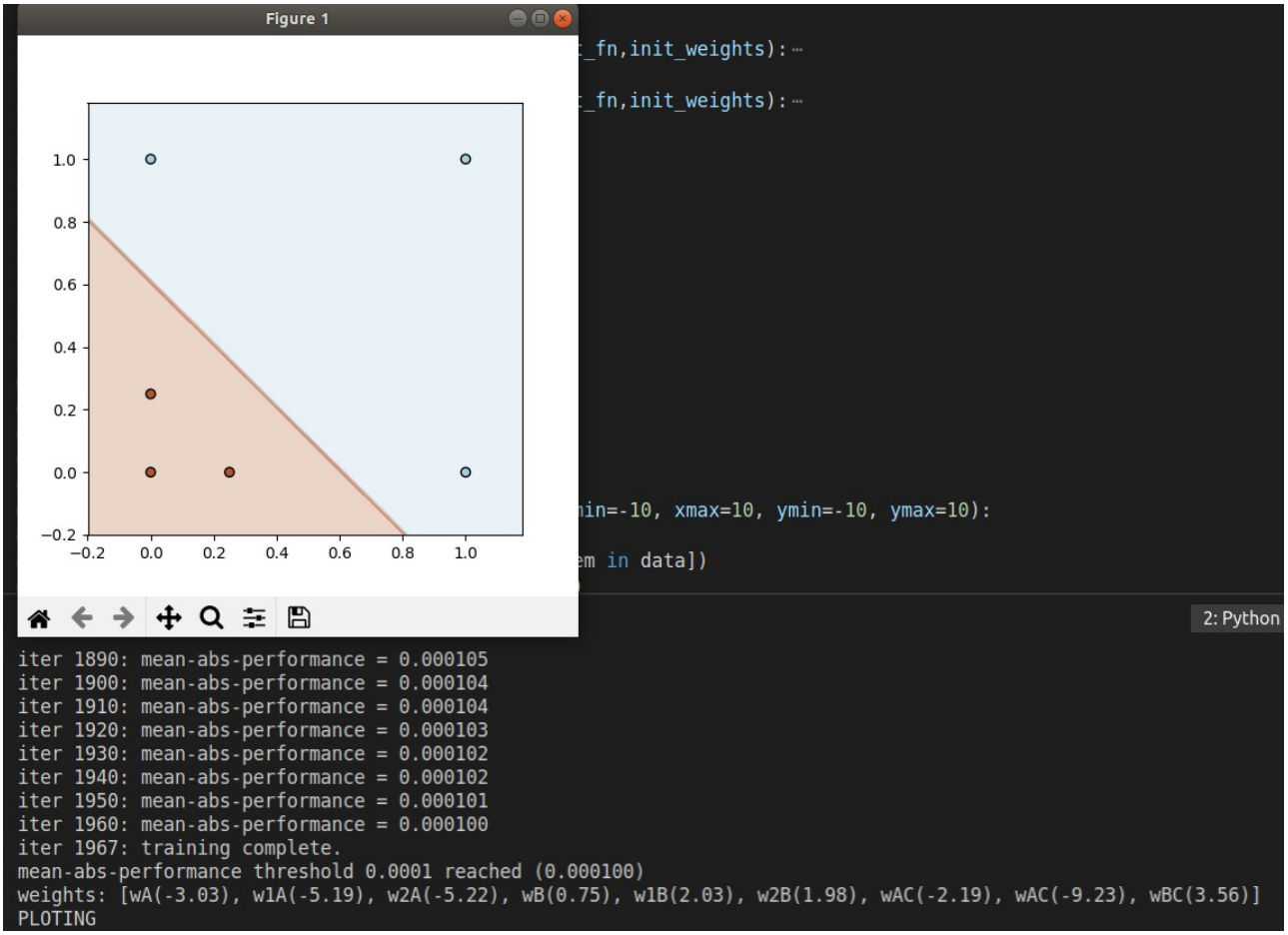


Figure 1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
_fn,init_weights):…
_fn,init_weights):…
nin=-10, xmax=10, ymin=-10, ymax=10):
em in data])
2: Python
iter 1890: mean-abs-performance = 0.000105
iter 1900: mean-abs-performance = 0.000104
iter 1910: mean-abs-performance = 0.000104
iter 1920: mean-abs-performance = 0.000103
iter 1930: mean-abs-performance = 0.000102
iter 1940: mean-abs-performance = 0.000102
iter 1950: mean-abs-performance = 0.000101
iter 1960: mean-abs-performance = 0.000100
iter 1967: training complete.
mean-abs-performance threshold 0.0001 reached (0.000100)
weights: [wA(-3.03), w1A(-5.19), w2A(-5.22), wB(0.75), w1B(2.03), w2B(1.98), wAC(-2.19), wAC(-9.23), wBC(3.56)]
PLOTING


475
476 > def make_neural_net_with_weights(): …
Figure 1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
_fn,init_weights): …
_fn,init_weights): …
in=-10, xmax=10, ymin=-10, ymax=10):
m in data])
2: Python
iter 2800: mean-abs-performance = 0.000103
iter 2810: mean-abs-performance = 0.000103
iter 2820: mean-abs-performance = 0.000102
iter 2830: mean-abs-performance = 0.000102
iter 2840: mean-abs-performance = 0.000101
iter 2850: mean-abs-performance = 0.000101
iter 2860: mean-abs-performance = 0.000101
iter 2870: mean-abs-performance = 0.000100
iter 2872: training complete.
mean-abs-performance threshold 0.0001 reached (0.000100)
weights: [wA(-6.46), w1A(-4.88), w2A(-5.07), wB(-1.12), w1B(-1.99), w2B(-1.37), wAC(-5.38), wAC(-10.49), wBC(-2.65)]
PLOTING

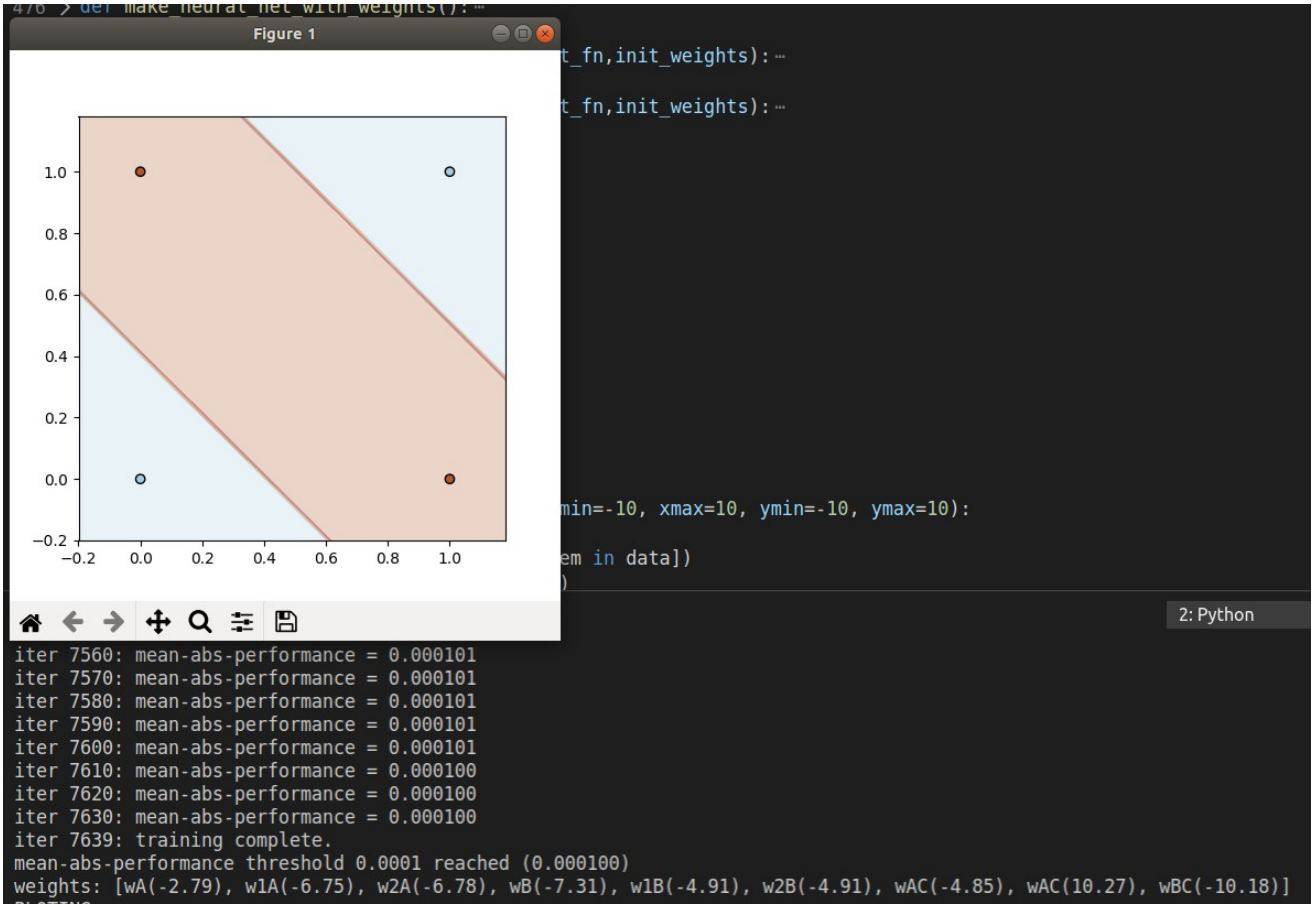


Figure 1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
t_fn,init_weights):
t_fn,init_weights):
min=-10, xmax=10, ymin=-10, ymax=10):
em in data])
2: Python
iter 7560: mean-abs-performance = 0.000101
iter 7570: mean-abs-performance = 0.000101
iter 7580: mean-abs-performance = 0.000101
iter 7590: mean-abs-performance = 0.000101
iter 7600: mean-abs-performance = 0.000101
iter 7610: mean-abs-performance = 0.000100
iter 7620: mean-abs-performance = 0.000100
iter 7630: mean-abs-performance = 0.000100
iter 7639: training complete.
mean-abs-performance threshold 0.0001 reached (0.000100)
weights: [wA(-2.79), w1A(-6.75), w2A(-6.78), wB(-7.31), w1B(-4.91), w2B(-4.91), wAC(-4.85), wAC(10.27), wBC(-10.18)]

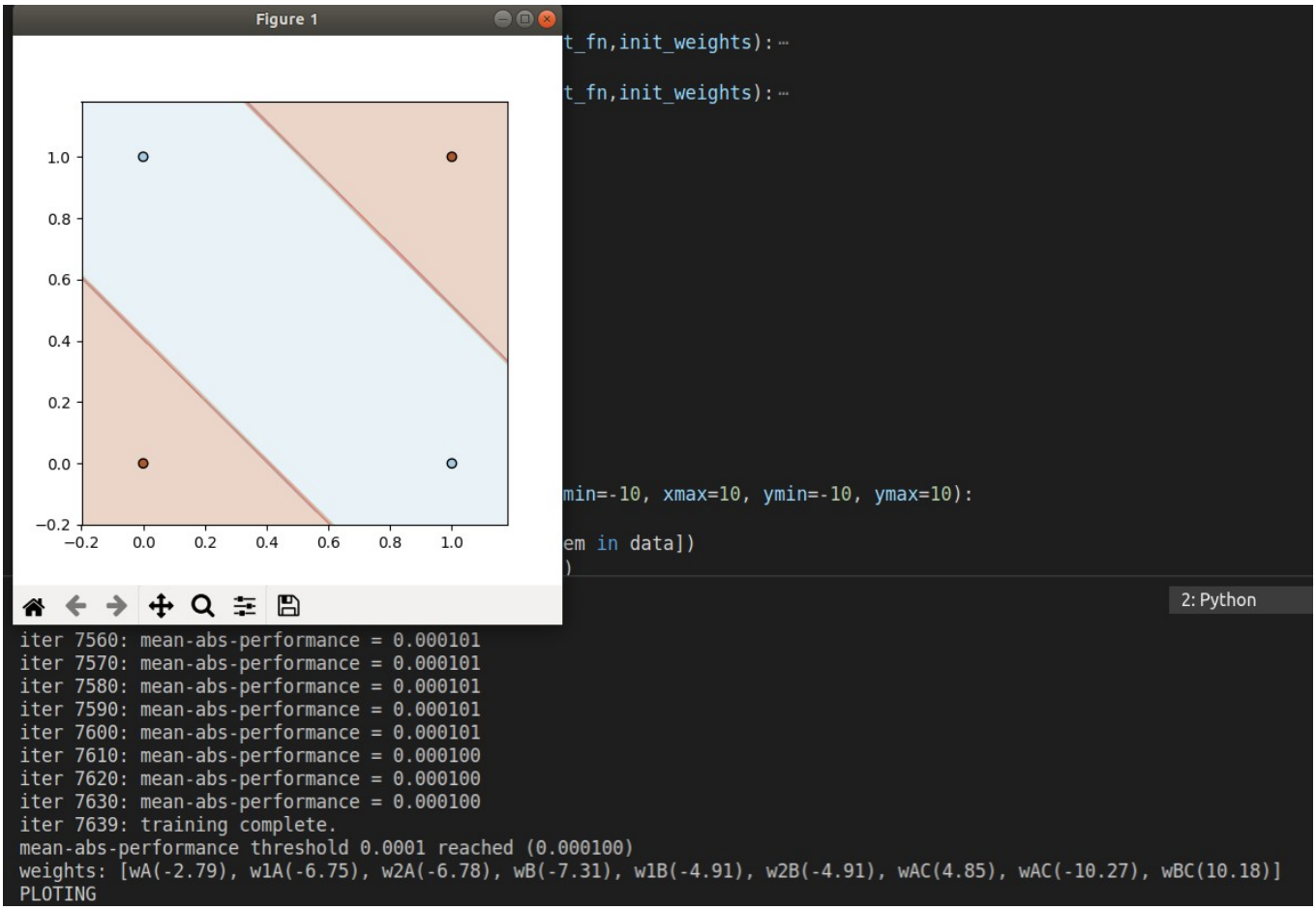


Figure 1
t_fn,init_weights):…
t_fn,init_weights):…
min=-10, xmax=10, ymin=-10, ymax=10):
em in data])
)
2: Python
iter 7560: mean-abs-performance = 0.000101
iter 7570: mean-abs-performance = 0.000101
iter 7580: mean-abs-performance = 0.000101
iter 7590: mean-abs-performance = 0.000101
iter 7600: mean-abs-performance = 0.000101
iter 7610: mean-abs-performance = 0.000100
iter 7620: mean-abs-performance = 0.000100
iter 7630: mean-abs-performance = 0.000100
iter 7639: training complete.
mean-abs-performance threshold 0.0001 reached (0.000100)
weights: [wA(-2.79), w1A(-6.75), w2A(-6.78), wB(-7.31), w1B(-4.91), w2B(-4.91), wAC(4.85), wAC(-10.27), wBC(10.18)]
PLOTING

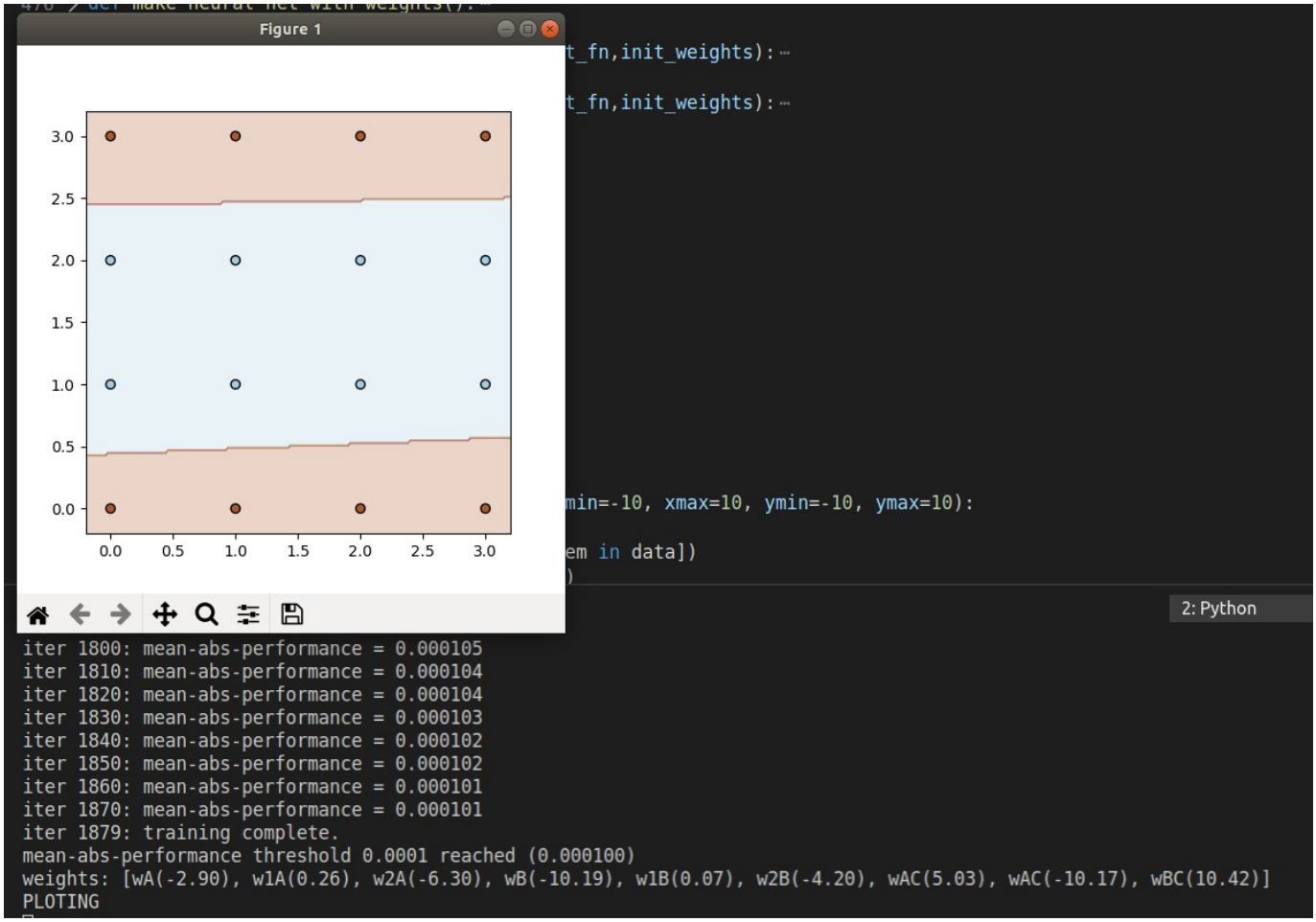


Figure 1
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
t_fn,init_weights):
t_fn,init_weights):
min=-10, xmax=10, ymin=-10, ymax=10):
em in data])
2: Python
iter 1800: mean-abs-performance = 0.000105
iter 1810: mean-abs-performance = 0.000104
iter 1820: mean-abs-performance = 0.000104
iter 1830: mean-abs-performance = 0.000103
iter 1840: mean-abs-performance = 0.000102
iter 1850: mean-abs-performance = 0.000102
iter 1860: mean-abs-performance = 0.000101
iter 1870: mean-abs-performance = 0.000101
iter 1879: training complete.
mean-abs-performance threshold 0.0001 reached (0.000100)
weights: [wA(-2.90), w1A(0.26), w2A(-6.30), wB(-10.19), w1B(0.07), w2B(-4.20), wAC(5.03), wAC(-10.17), wBC(10.42)]
PLOTING

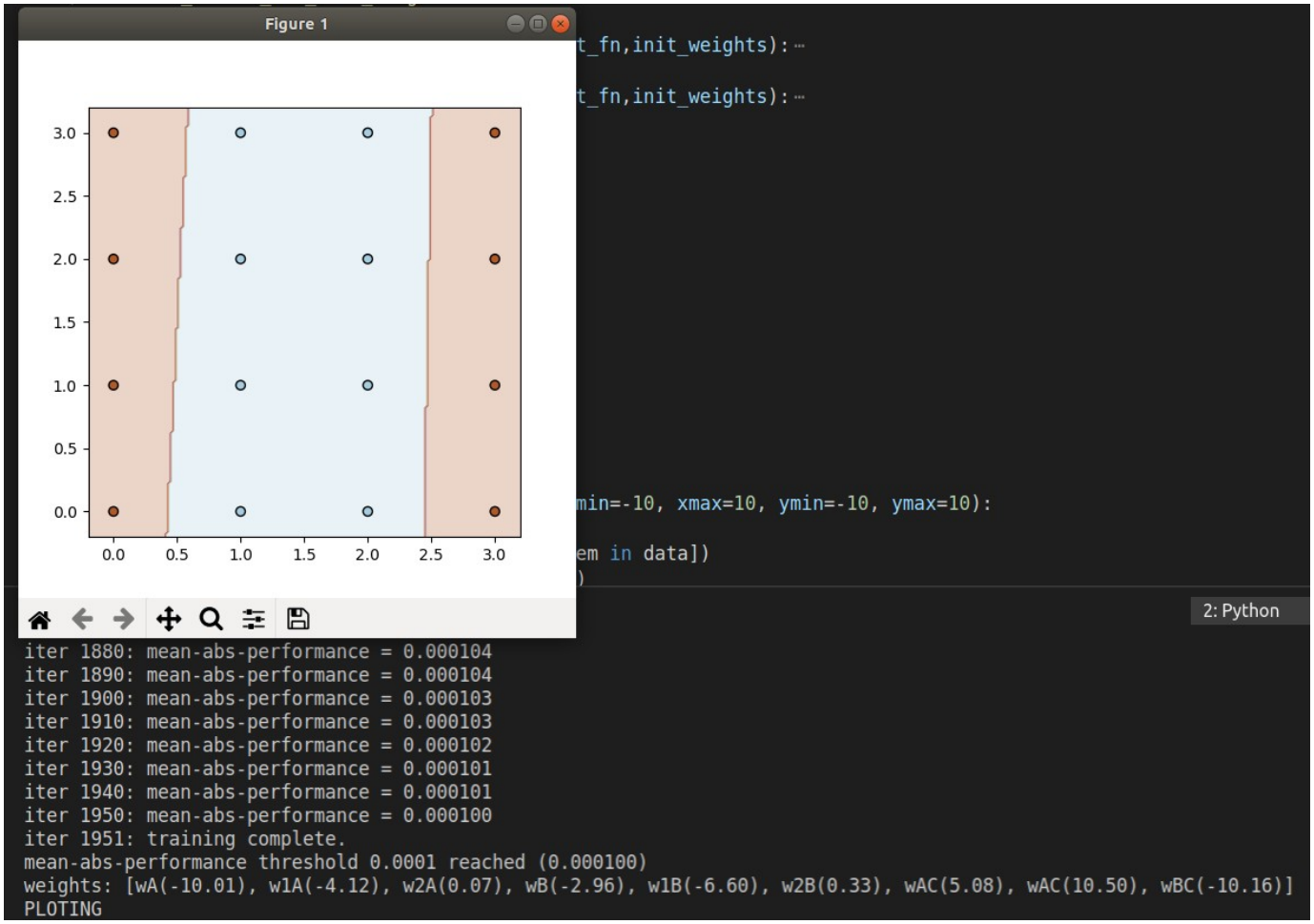
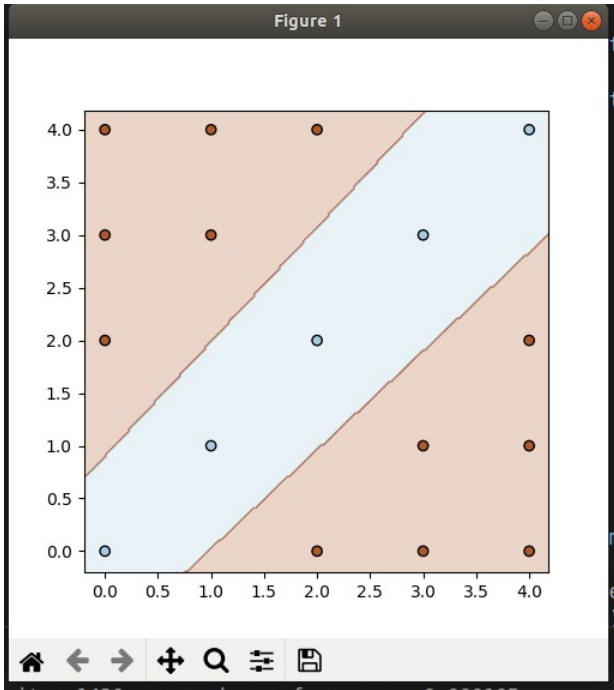


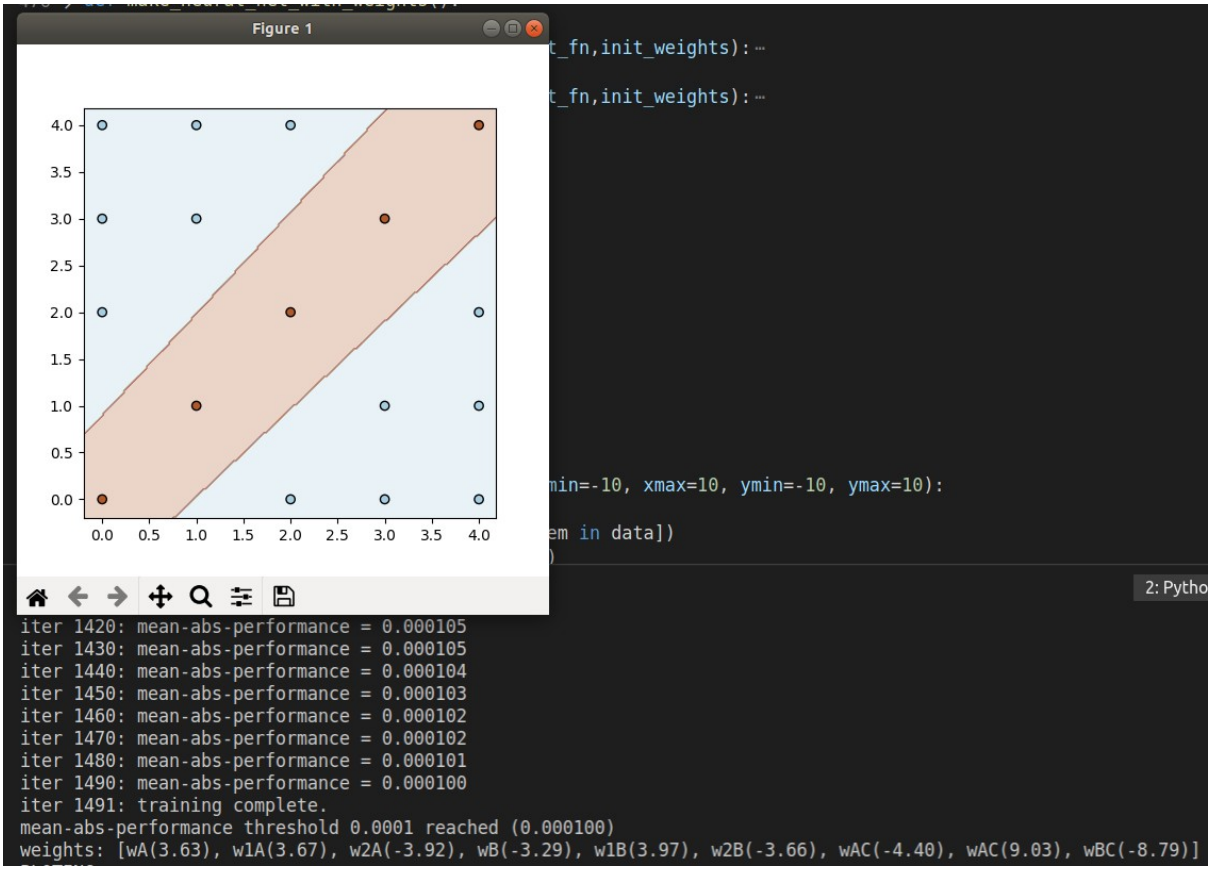
Figure 1
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
t_fn,init_weights):…
t_fn,init_weights):…
min=-10, xmax=10, ymin=-10, ymax=10):
em in data])
)
2: Python
iter 1880: mean-abs-performance = 0.000104
iter 1890: mean-abs-performance = 0.000104
iter 1900: mean-abs-performance = 0.000103
iter 1910: mean-abs-performance = 0.000103
iter 1920: mean-abs-performance = 0.000102
iter 1930: mean-abs-performance = 0.000101
iter 1940: mean-abs-performance = 0.000101
iter 1950: mean-abs-performance = 0.000100
iter 1951: training complete.
mean-abs-performance threshold 0.0001 reached (0.000100)
weights: [wA(-10.01), w1A(-4.12), w2A(0.07), wB(-2.96), w1B(-6.60), w2B(0.33), wAC(5.08), wAC(10.50), wBC(-10.16)]
PLOTING




```
t_fn,init_weights):…

t_fn,init_weights):…

min=-10, xmax=10, ymin=-10, ymax=10):

em in data])
)
```

2: Pyth

```
iter 1420: mean-abs-performance = 0.000105
iter 1430: mean-abs-performance = 0.000105
iter 1440: mean-abs-performance = 0.000104
iter 1450: mean-abs-performance = 0.000103
iter 1460: mean-abs-performance = 0.000102
iter 1470: mean-abs-performance = 0.000102
iter 1480: mean-abs-performance = 0.000101
iter 1490: mean-abs-performance = 0.000100
iter 1491: training complete.
mean-abs-performance threshold 0.0001 reached (0.000100)
weights: [wA(3.63), w1A(3.67), w2A(-3.92), wB(-3.29), w1B(3.97), w2B(-3.66), wAC(4.40), wAC(-9.03), wBC(8.79)]
```



تست شبکه با تست مشتق گیری

۵. Finite Difference

رسیدن به دقت ۱۰۰ درصد روی مسائل ساده‌ی OR و AND لزوما به معنای درست بودن کامل پیاده‌سازی نیست. برای اینکه از درستی پیاده‌سازی که در بالا داشته اید تا حد خوبی مطمئن بشوید و همچنین اگر که پیاده‌سازی درست نیست تکنیکی برای debug کردن آن داشته باشید، باید در این بخش از متد finite difference برای تخمین زدن مشتق وزن‌ها استفاده کرده، و بعد آن را با مقدار حاصل از توابعی که نوشته‌اید مقایسه کنید. فرمول مورد استفاده در این متد برای تقریب زدن مشتق به شکل زیر است:

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

که در آن مقدار ε باید برابر مقدار بسیار کوچکی باشد (مثلا ۱۰ به توان منفی ۸). برای کاربرد ما، x هر کدام از وزن‌های شبکه خواهد بود، و تابع f، تابعی است که خروجی PerformanceElem را نمایش می‌دهد. شما باید در این بخش تابعی بنویسید که با گرفتن یک شبکه، روی وزن‌های آن iterate بکند، و به ازای هر وزن، مقدار مشتق PerformanceElem را نسبت به آن وزن یک بار از متد تخمین بالا، و یکبار از طریق تابع dOutdX محاسبه کرده و برابری (تقریبی) آن‌ها را چک بکند و در نهایت اگر همه برابر بودند مقدار True برگرداند. در نوشتن این تابع حواستان به خالی کردن cache شبکه بین دو محاسبه باشد.

در زیر پیاده سازی تابع آورده شده است:

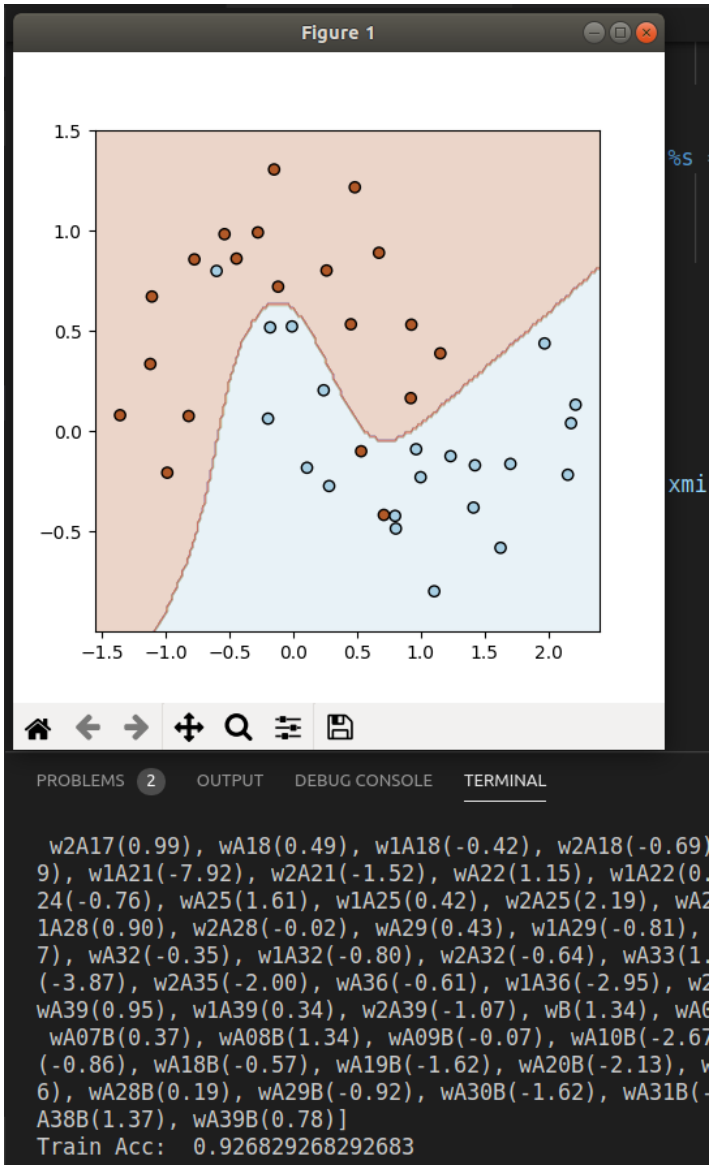
```
def finite_difference(network):
    for w in network.weights:
        network.clear_cache()
        cur_val = network.performance.output()
        w.set_value(w.get_value() + 1e-8)
        network.clear_cache()
        new_value = network.performance.output()
        w.set_value(w.get_value() - 1e-8)
        finite_diff = (new_value - cur_val) / 1e-8
        print(f'{w.get_name():5s} finite:{finite_diff: 2.4f} real:{network.performance.dOutdX(w): 2.4f}', end='')
        if abs(network.performance.dOutdX(w) - finite_diff) < 1e-4:
            print(" Correct")
        else:
            print(" Incorrect")
    network.clear_cache()
```

over fitting and regularization

وقتی پیچیدگی شبکه عصبی بیشتر از داده ای باشد که می خواهیم ان را یاد بگیریم شبکه عصبی سعی در منطبق قرار دادن روال کار خود مبتنی برا داده ی ترین می کند و نهایتا نتیجه حاصل از اختلاف ارور بین داده ی تست و ترین نشان می دهد بیش برازش اتفاق افتاده است.

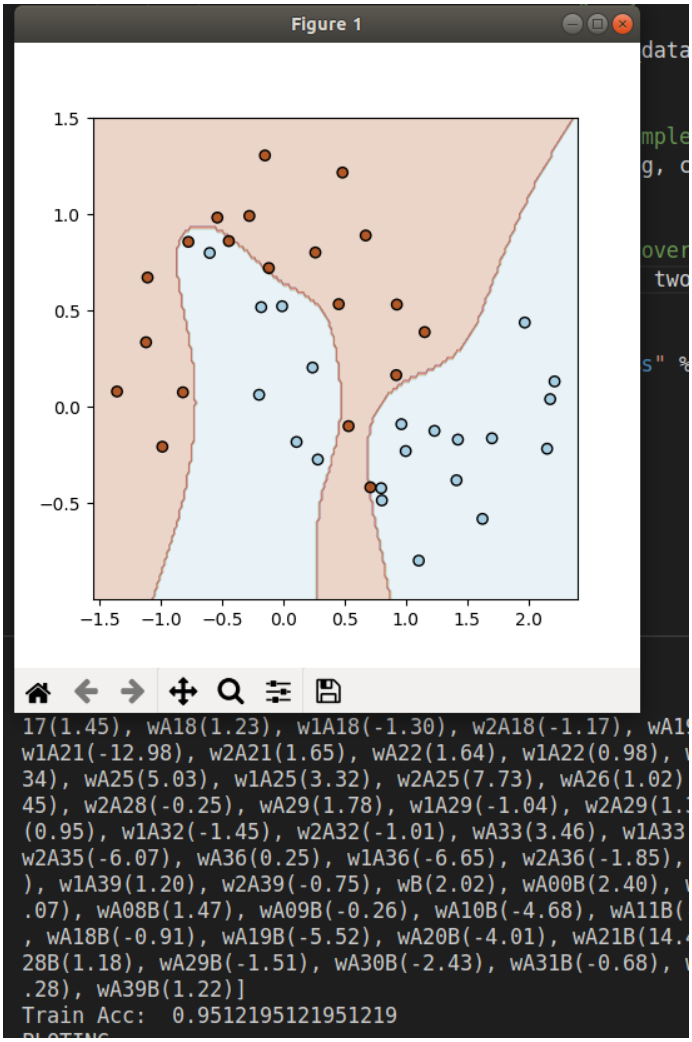
حال برای کار کردن بر روی داده های two_moons خواهیم داشت:
دو ورودی در لایه اول و ۴۰ نورون در لایه دوم و یک نورون برای لایه خروجی خواهیم داشت.
حال آن را به دفعات ۱۰۰ و ۵۰۰ و ۱۰۰۰ بار آموزش می دهیم.
و با استفاده از تابع تست دقت اندازه گیری را به دست می آوریم.
و ناحیه تصمیم گیری را نیز می کشیم.

: Iteration = 100



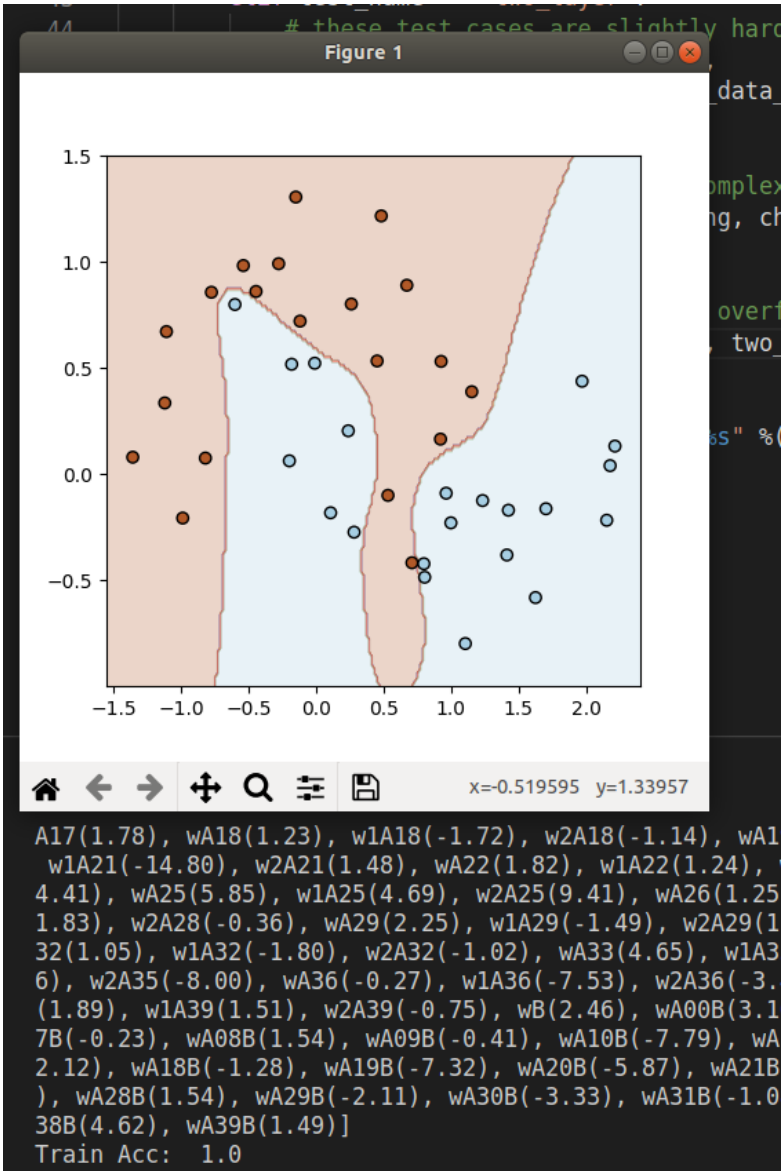

```
test([0.8063775806264301, 0.31389154012736115, 0.0]) returned: 0.06854291091988816 => 0.0 [correct]
test([-0.8350499250048773, 0.9757438805989312, 0.0]) returned: 0.008057076580269959 => 0.0 [correct]
test([-0.3056349962060481, 0.8480665351250637, 0.0]) returned: 0.1924355745821355 => 0.0 [correct]
test([1.342395809182781, -0.6628775105845491, 1.0]) returned: 0.9933291838793752 => 1.0 [correct]
test([-0.8504352993794975, 0.4459199151136456, 0.0]) returned: 0.021674890340769935 => 0.0 [correct]
test([1.1576995999372781, 0.3431944106118581, 0.0]) returned: 0.16062918677188637 => 0.0 [correct]
test([0.7693882306818469, -0.6514034954565945, 1.0]) returned: 0.9792113643673407 => 1.0 [correct]
test([0.9818376618443154, 0.4232921505544805, 0.0]) returned: 0.05109036005018022 => 0.0 [correct]
test([1.5108529245285525, -0.4695580852097725, 1.0]) returned: 0.9912880401094931 => 1.0 [correct]
test([0.6621843718069381, 0.8357685698483136, 0.0]) returned: 0.0011840134893738852 => 0.0 [correct]
test([0.651668455747412, -0.7352289907924181, 1.0]) returned: 0.9927270505682864 => 1.0 [correct]
test([0.2360202948238896, -0.4276736056557813, 1.0]) returned: 0.9985798178338635 => 1.0 [correct]
Accuracy: 1.000000
```

iteration =500




```
test([0.8063775806264301, 0.31389154012736115, 0.0]) returned: 0.0014986503903669632 => 0.0 [correct]
test([-0.8350499250048773, 0.9757438805989312, 0.0]) returned: 0.17481968250990468 => 0.0 [correct]
test([-0.3056349962060481, 0.8480665351250637, 0.0]) returned: 0.2238533109075733 => 0.0 [correct]
test([1.342395809182781, -0.6628775105845491, 1.0]) returned: 0.9999949851187063 => 1.0 [correct]
test([-0.8504352993794975, 0.4459199151136456, 0.0]) returned: 0.24243974807812663 => 0.0 [correct]
test([1.1576995999372781, 0.3431944106118581, 0.0]) returned: 0.016071620915005243 => 0.0 [correct]
test([0.7693882306818469, -0.6514034954565945, 1.0]) returned: 0.4848778421067173 => 1.0 [wrong]
test([0.9818376618443154, 0.4232921505544805, 0.0]) returned: 0.00036239390347678684 => 0.0 [correct]
test([1.5108529245285525, -0.4695580852097725, 1.0]) returned: 0.9999987056510997 => 1.0 [correct]
test([0.6621843718069381, 0.8357685698483136, 0.0]) returned: 1.0163752996277333e-07 => 0.0 [correct]
test([0.651668455747412, -0.7352289907924181, 1.0]) returned: 0.01977970654718134 => 1.0 [wrong]
test([0.2360202948238896, -0.4276736056557813, 1.0]) returned: 0.934367497938512 => 1.0 [correct]
Accuracy: 0.820000
```

:iteration = 1000




```
test([0.8063775806264301, 0.31389154012736115, 0.0]) returned: 1.2494727336081958e-05 => 0.0 [correct]
test([-0.8350499250048773, 0.9757438805989312, 0.0]) returned: 0.0022404528423108604 => 0.0 [correct]
test([-0.3056349962060481, 0.8480665351250637, 0.0]) returned: 0.003735048570901114 => 0.0 [correct]
test([1.342395809182781, -0.6628775105845491, 1.0]) returned: 0.9999999930965973 => 1.0 [correct]
test([-0.8504352993794975, 0.4459199151136456, 0.0]) returned: 0.0037958615788008195 => 0.0 [correct]
test([1.1576995999372781, 0.3431944106118581, 0.0]) returned: 0.014743864233811412 => 0.0 [correct]
test([0.7693882306818469, -0.6514034954565945, 1.0]) returned: 0.21428166973531654 => 1.0 [wrong]
test([0.9818376618443154, 0.4232921505544805, 0.0]) returned: 1.86822309641041e-05 => 0.0 [correct]
test([1.5108529245285525, -0.4695580852097725, 1.0]) returned: 0.999999998649209 => 1.0 [correct]
test([0.6621843718069381, 0.8357685698483136, 0.0]) returned: 4.947560756698069e-10 => 0.0 [correct]
test([0.651668455747412, -0.7352289907924181, 1.0]) returned: 0.007579101753114175 => 1.0 [wrong]
test([0.2360202948238896, -0.4276736056557813, 1.0]) returned: 0.995709297788865 => 1.0 [correct]
Accuracy: 0.870000
```

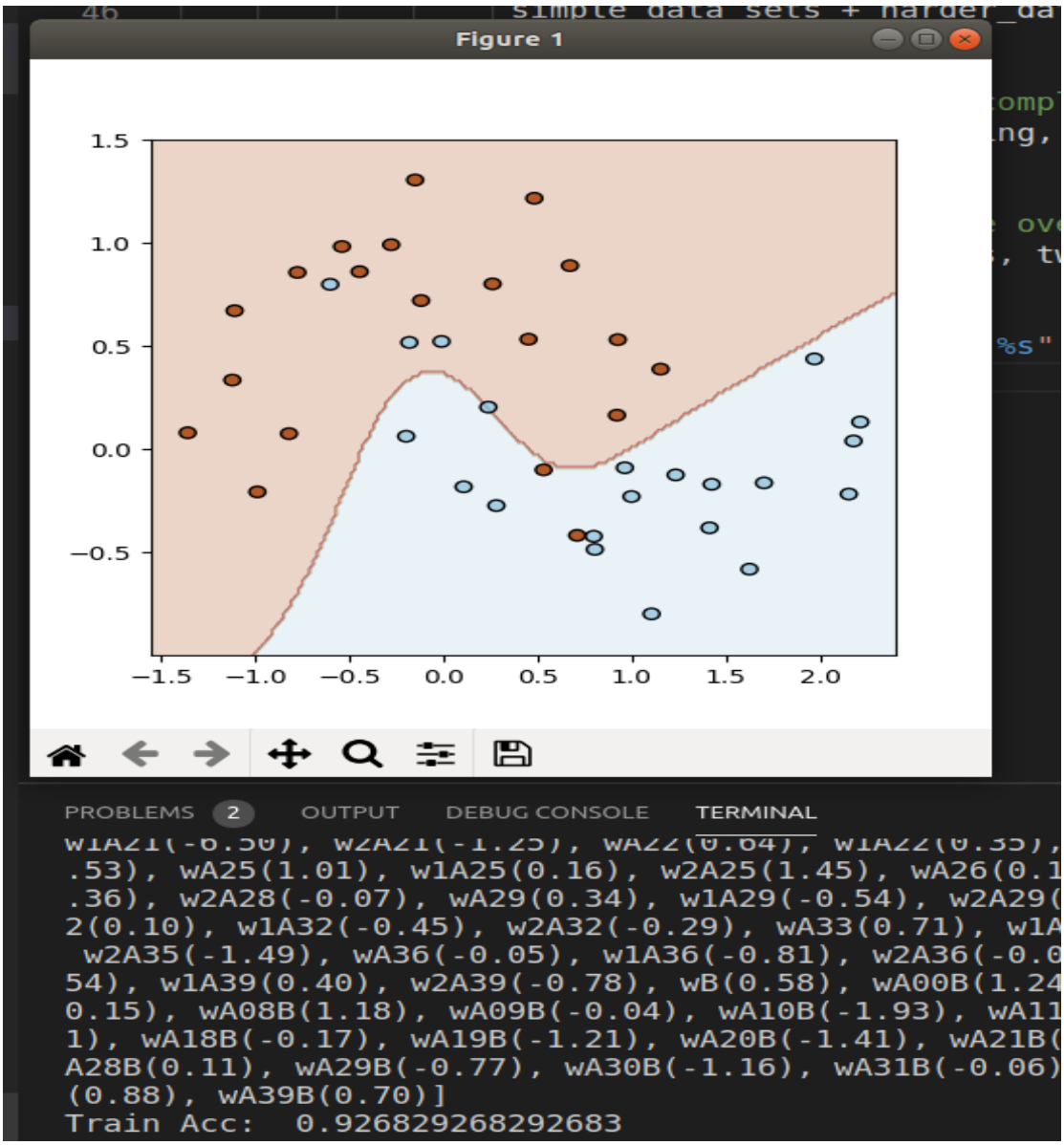
ملاحظه می‌کنیم با زیاد شدن تعداد ایتریشن ها شبکه عصبی سعی در نزدیک شدن به دیتای ترین می‌کند و در نتیجه دیتای تست با خطای بیشتری نسبت به دیتای ترین دیده خواهد شد.

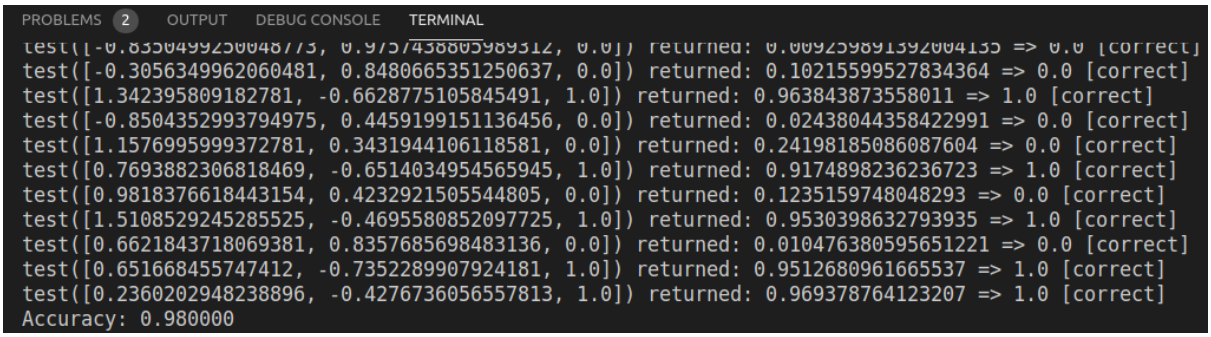
حال سعی در رفع این مورد با requlariزation هستیم :

```
class RegularizedPerformanceElem(PerformanceElem):
    def __init__(self, input, desired_value):
        assert isinstance(input, (Input, Neuron))
        DifferentiableElement.__init__(self)
        self.my_input = input
        self.my_desired_val = desired_value
        self._lambda = 0.00001
        self.weights = None

    def set_weights(self, _weights):
        self.weights = _weights

    def output(self):
        old_out = -.5 * ((self.my_desired_val - self.my_input.output()) ** 2)
        out = old_out - self._lambda * self.add_regularization_l2()
        return out

    def add_regularization_l2(self):
        np_w = np.array([item.get_value() for item in self.weights])
        return np.linalg.norm(np_w)

    def dOutdX(self, elem):
        old_dout = (self.my_desired_val - self.my_input.output()) * \
            self.my_input.dOutdX(elem)
        dout = old_dout - self._lambda * elem.get_value() * 2
        return dout

    def set_desired(self, new_desired):
        self.my_desired_val = new_desired

    def get_input(self):
        return self.my_input
```

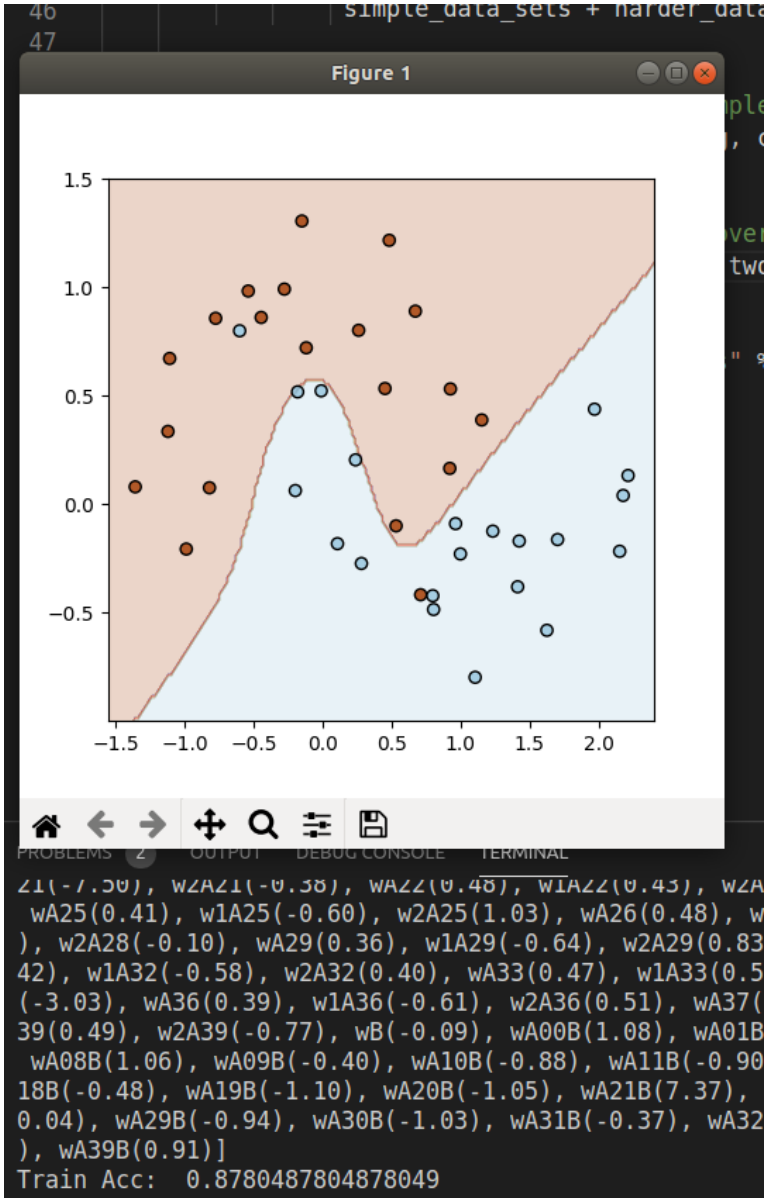
نهایتاً برای حساسیت loss بر روی وزن ها مقدار 0.0001 را انتخاب می‌کنیم و اگر این مقدار کم باشد مثل آن می‌ماند که اصلاً رگیولاریزیشن انجام نشده.

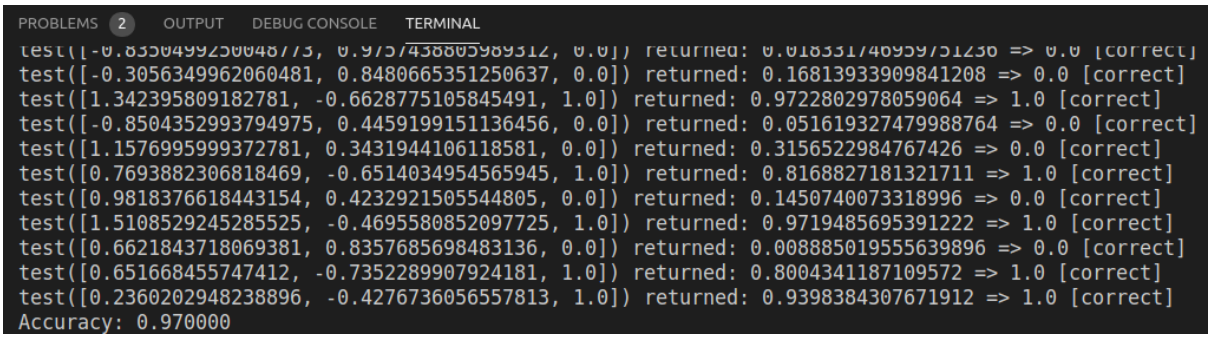
:iteration = 100



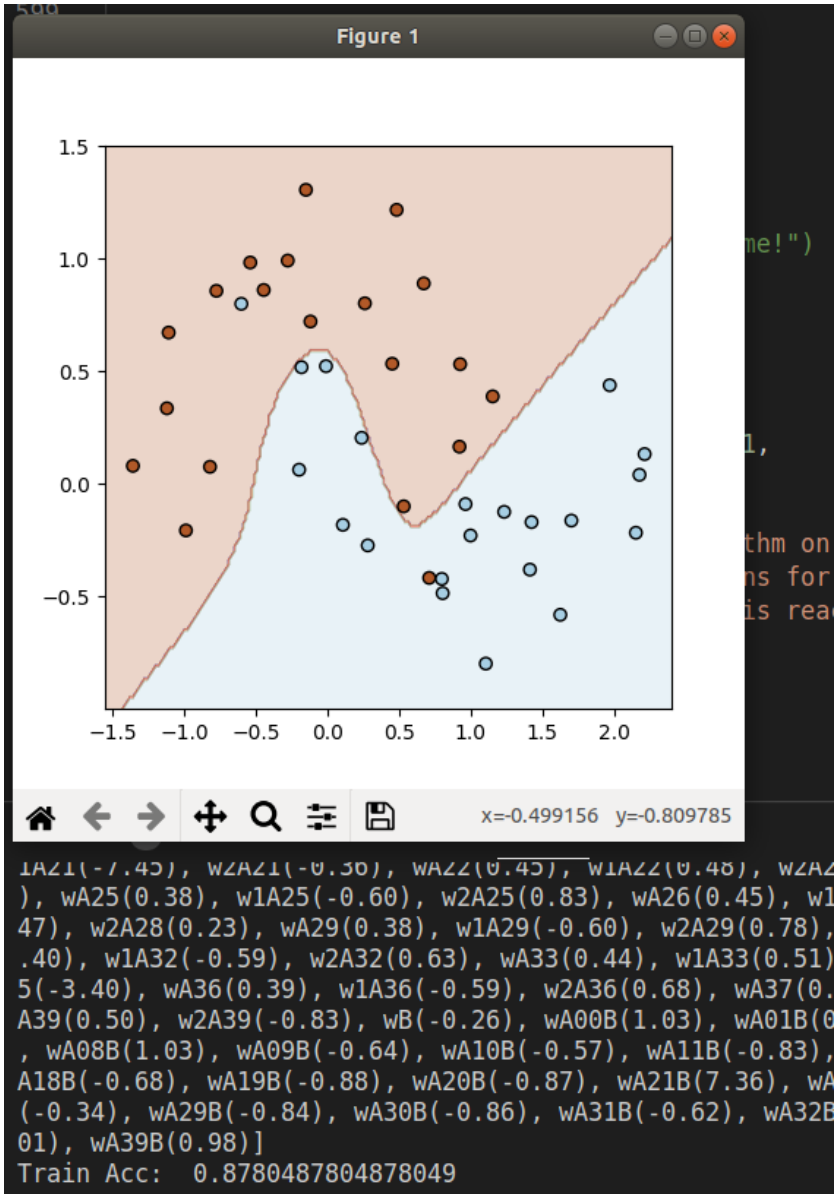


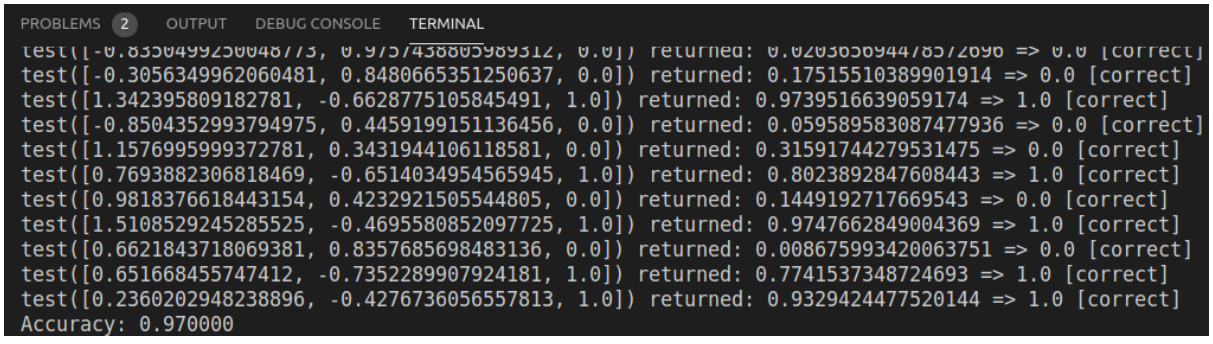
: iteration = 500





iteration = 1000





می‌بینیم که با این کار دقت دیتای ترین را پایین آورده‌ایم ولی دقت حاصله از تست بیشتر شده است و این یعنی مشکل بیش بردازش را حل شده است.

	دقت مدل ترین عادی	دقت مدل تست عادی	دقت مدل ترین بعد از رگیولاریزیشن	دقت مدل تست بعد رگیولاریزیشن
۱۰۰	0.92	1	0.92	0.98
۵۰۰	0.95	0.82	0.87	0.97
۱۰۰۰	1	0.87	0.87	0.

پایان