به نام خدا

گزارش تمرین کامپیوتری شماره ۴

علی درّانی

۸۱۰۱۹۵۵۲۲

پیاده سازی کلاس Input :

این کلاس به دلیل اینکه فقط مقدار یک ویژگی را نگه می دارد و وزنی به آن متصل نیست به صورت زیر پیاده سازی می شود.

```
class Input(ValuedElement,DifferentiableElement):

    def __init__(self,name,val):
        ValuedElement.__init__(self,name,val)
        DifferentiableElement.__init__(self)



    def output(self):
        return self.get_value()
    def dOutdX(self, elem):
        return 0
```

پیاده سازی شبکه عصبی

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 بالا، پیاده سازی توابع مشتق و خروجی دو کلاس Neuron و PerformanceElem در صفحه بعد ذکر شده است.

پیاده سازی کلاس Neuron :

```
    def compute_output(self):

        z = 0
        for elem in range(len(self.get_inputs())):
            inp = self.get_inputs()[elem]
            wei = self.get_weights()[elem]
            z+= wei.get_value()*inp.output()
        return 1.0/(1.0 + np.exp(-z))

    def dOutdX(self, elem):
        if self.use_cache:
            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)

    def compute_doutdx(self, elem):

        sigDev = (self.output())*(1-self.output())
        weights = self.get_weights

        if (self.has_weight(elem)):
            for i in range(0,len(self.get_inputs())):
                if (self.get_weights()[i] == elem):
                    return (sigDev * (self.get_inputs()[i].output()))
        else :
            inNeurons = self.get_inputs()
            inWeights = self.get_weights()
            dev = 0
            for i in range(len(self.get_weights())):
                if (self.isa_descendant_weight_of(elem, inWeights[i])):
                    input_deriv = self.get_inputs()[i].dOutdX(elem)
                    dev = (sigDev * ((self.get_weights()[i]).get_value()) * (inN
eurons[i]).dOutdX(elem))
        return dev
```

پیاده سازی کلاس PerfomanceElem :

```
class PerformanceElem(DifferentiableElement):
    """
    Representation of a performance computing output node.
    This element contains methods for setting the
    desired output (d) and also computing the final
    performance P of the network.
    This implementation assumes a single output.
    """
    def __init__(self,input,desired_value):
        assert isinstance(input,(Input,Neuron))
        DifferentiableElement.__init__(self)
        self.my_input = input
        self.my_desired_val = desired_value
    def output(self):
        return -0.5*((self.my_desired_val)-(self.my_input.output()))**2


    def dOutdX(self, elem):
        myInput = self.get_input()
        return ((self.my_desired_val - self.my_input.output())*myInput.dOutdX(el
em))

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

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

تست کردن شبکه Simple با پیاده سازی های صورت گرفته

در تست اول که مربوط به بخش Simple می شود، خروجی به شکل زیر است :

```
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
```

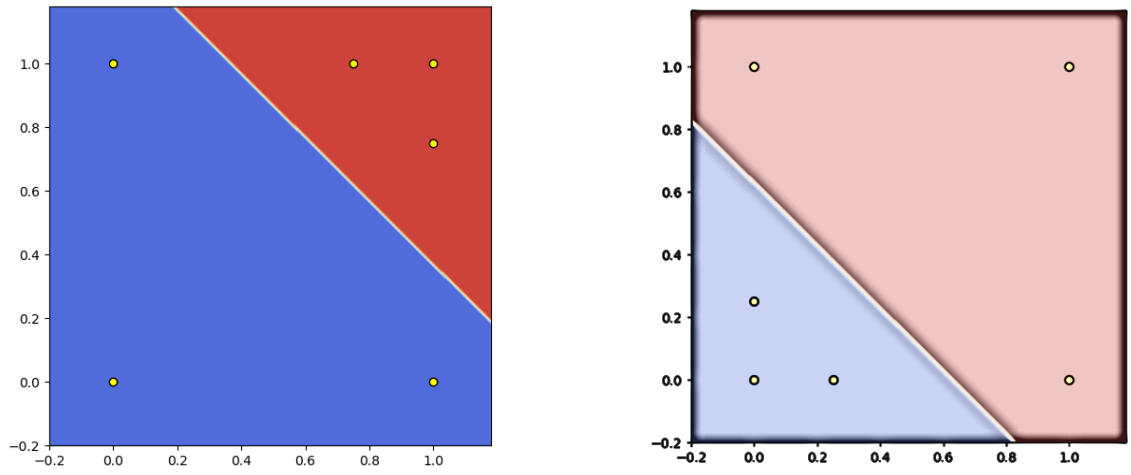
کشیدن ناحیه تصمیم گیری

۷. کشیدن ناحیه‌ی تصمیم‌گیری:

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)
```

ناحیه تصمیم گیری برای تست Simple به شکل زیر است:



پیاده سازی Finite Diffrence و مقایسه آن با dOutdX :

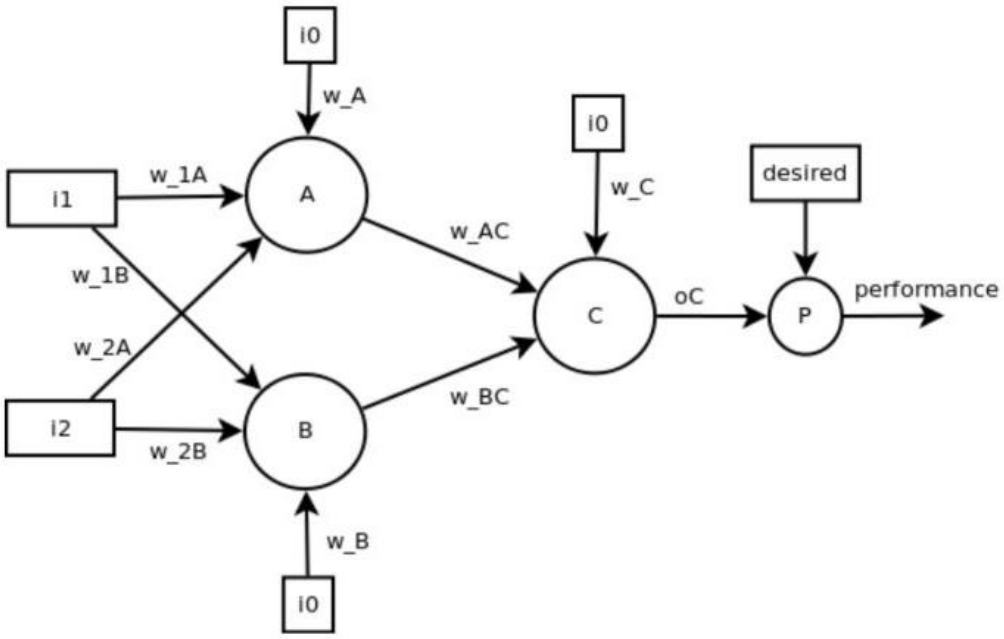
برای محاسبه در آخر Training باید مراحل زیر را انجام دهیم

۱. محاسبه Performance شبکه بدون تغییر وزن
۲. محاسبه dOutdX برای همان وزن
۳. پاک سازی کش شبکه و محاسبه Performance با تغییر وزن به اندازه اپسیلون
۴. محاسبه اختلاف فرمول گفته شده در صورت سوال و dOutdX و مقایسه در بازه مشخص
۵. برگرداندن وزن شبکه به حالت پیش فرض

پیاده سازی :

```
def finite_difference(network):
    weights = list()
    PerfElement = list()
    weights = network.weights
    PerfElement = network.performance

    for weight in weights:
        network.clear_cache()

        preWeight = weight.get_value()
        NewWeight = (weight.get_value() + 1e-8)
        oldPerf = PerfElement.output()
        dev = PerfElement.dOutdx(weight)
        weight.set_value(weight.get_value() + (1e-8))
        network.clear_cache()
        newPer = (network.performance).output()
        weight.set_value(preWeight)
        finite_diff = (newPer - oldPerf) / (1e-8)
        if abs(network.performance.dOutdX(weight) - finite_diff) < 1e-4:
            print("True")
        else:
            print("False")

    network.clear_cache()
```

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...
```

برای تست این شبکه دستور زیر را استفاده کنید:

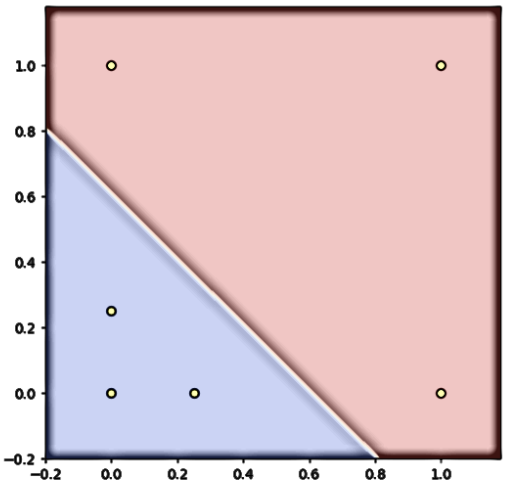
```
python neural_net_tester.py two_layer
```

برای پیاده سازی این بخش مطابق آنچه گفته شد وزن ها، ورودی ها و نورون هارا می سازیم. در همه نورون ها مقدار بایس (i0) موجود 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)
    i2 = Input('i2', 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
```

خروجی این تست به شکل زیر است :

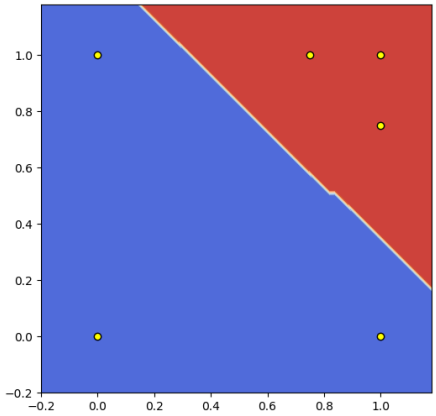
```
mean-abs-performance threshold 0.0001 reached(0.000100)

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)]

Train Acc:  1.0
```



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)]

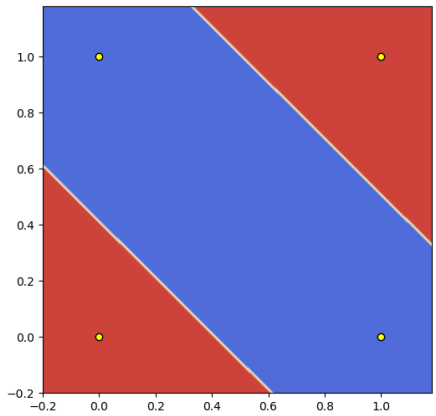
Train Acc: 1.0



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)]

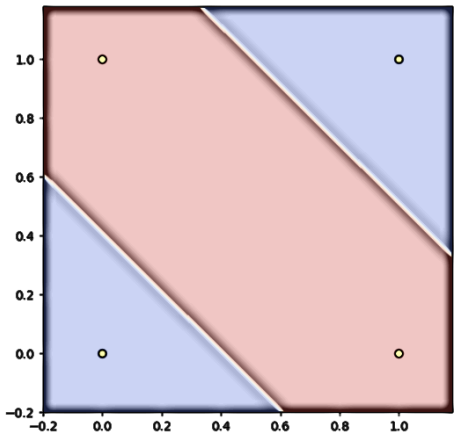
Train Acc: 1.0



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)]

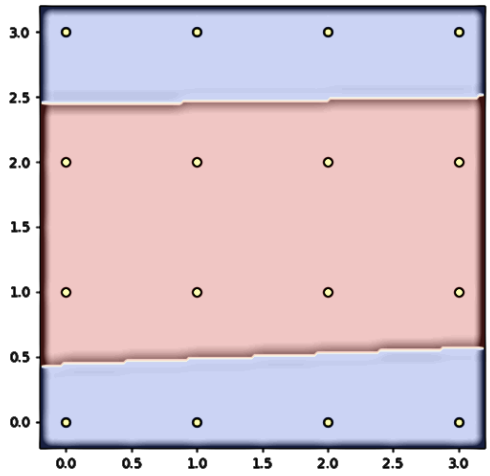
Train Acc: 1.0



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)]

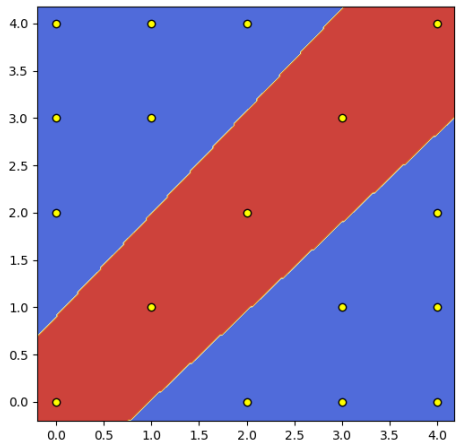
Train Acc: 1.0



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)]

Train Acc: 1.0



Testing on inverse-diagonal-band test-data

test((-1, -1, 0)) returned: 0.024887880686060723 => 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]

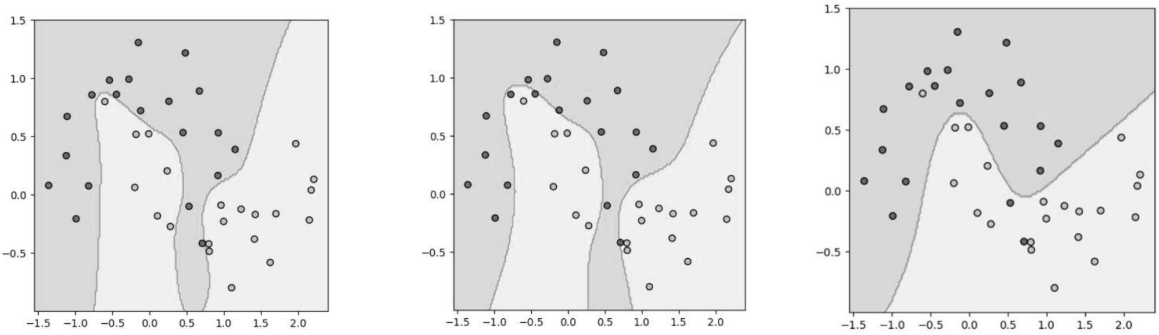
test((0, 4, 1)) returned: 0.9878824018126686 => 1 [correct]

Accuracy: 1.000000

Overfitting & regularization

زمانی که پیچیدگی شبکه عصبی بیشتر از داده ورودی باشد، شبکه سعی در منطبق قرار دادن داده ترین می کند و نتیجه حاصل از اختلاف ارور بین داده تست و ترین نشان می دهد بیش برازش اتفاق افتاده است.

برای داده های Two_moons به ترتیب ۱۰۰ ، ۵۰۰ و ۱۰۰۰ بار ترین می کنیم و سپس دقت اندازه گیری را به دست می آوریم. ملاحظه می شود با اضافه شدن پیمایش دیتای تست با خطای بیشتری نسبت به دیتا ترین دیده می شود. جدول دقت به شکل زیر است.



ایتریشن ۱۰۰	ایتریشن ۵۰۰	ایتریشن ۱۰۰۰
۹۸٪	۹۴٪	۹۵٪

به همین منظور برای حذف regularization از L2norm استفاده می کنیم. L2norm یک بردار به صورت زیر بدست می آید.

$$\text{l2norm(w)} = \sum w_i^2$$

به این ترتیب این عبارت به تابع performance استفاده می شود. تا تاثیر ویژگی ها در اورفیتینگ کمتر شود.

پیاده سازی کلاس :

```
class RegularizedPerformanceElem(PerformanceElem):
    def __init__(self, input, desired_value):
        if(type(Input) == Neuron):
            return 0
        DifferentiableElement.__init__(self)
        self.my_input = input
        self.my_desired_val = desired_value
        self.lambda__ = 0.0001
        self.weights = None

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

    def output(self):
        old_out = -0.5 * ((self.my_desired_val - self.my_input.output()) ** 2)
        np_w = np.array([item.get_value() for item in self.weights])
        OutPut = old_out - self.lambda__ * (np.linalg.norm(np_w))
        return OutPut
```

برای حساسیت Loss مقدار ۰.۰۰۰۱ را بر روی وزن ها اعمال می کنیم و اگر این مقدار کم باشد مانند آن است اصلا رگیولاریزیشن انجام نشده است.

ایتریشن ۱۰۰	ایتریشن ۵۰۰	ایتریشن ۱۰۰۰
۹۶٪	۹۶٪	۹۶٪