پروژه‌ی پنجم در هوش مصنوعی
نام و نام خانوادگی: فاطمه حقیقی
شماره دانشجویی: 810195385

بخش سوم: پیاده‌سازی شبکه عصبی

در این بخش لازم است که ما ساختار توابع output و dOutdx را برای کلاس های input, performanceElem, Neuron کامل کنیم.

پس از کامل شدن، پیاده سازی کامل شده این دو تابع در کلاس Input به صورت زیر می باشد:

```
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)
        """
        # raise NotImplementedError("Implement me!")
        return self.get_value()


    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)
        """
        # raise NotImplementedError("Implement me!")
        return 0
```

و نیز پیاده سازی این دو تابع در کلاس Neuron به صورت زیر می باشد:

به عبارت دیگر دو تابع output و dOutdx در این کلاس پیاده سازی شده است و توابع زیر مجموعه ی آن ها یعنی compute_output و compute_doutdx نیاز به پیاده سازی توسط ما دارند. که کامل شده آن ها به صورت زیر است:

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

    returns: number (float or int)
    """
    # raise NotImplementedError("Implement me!")
    x = 0.0
    for i in range(len(self.my_inputs)):
        x += self.my_inputs[i].output() * self.my_weights[i].get_value()
    return 1 / (1 + np.exp(-x))
```

```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)
    """
    # raise NotImplementedError("Implement me!")
    dOut = self.output() * (1 - self.output())
    # direct
    if(self.has_weight(elem)):
        # moshtaq sigmod g (1 - g)
        for i in range(len(self.my_weights)):
            if self.my_weights[i] == elem:
                factor = self.my_inputs[i].output()
        return factor * dOut

    else:
        sumation = 0
        for i, w in enumerate(self.my_weights):
            if(self.isa_descendant_weight_of(elem, w)):
                sumation += w.get_value() * self.my_inputs[i].dOutdX(elem)
        return sumation * dOut
```

و نیز پیاده سازی این دو تابع در کلاس performanceElem به صورت زیر می باشد:

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

    returns: number (float/int)
    """
    # raise NotImplementedError("Implement me!")
    return 1 - (1/2) * ((self.my_desired_val -  self.my_input.output()) ** 2)

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)
    """
    # raise NotImplementedError("Implement me!")
    return (self.my_desired_val - self.my_input.output()) * self.my_input.dOutdX(elem)
```

بخش چهارم : تست کردن شبکه

پس از اتمام پیاده سازی، برنامه را با دستور python3 neural_net_tester.py simple تست کردیم. در این حالت دقت همواره هم برای ساختار AND و هم برای ساختار OR برابر ۱ یا ۱۰۰٪ می باشد.

```
--------------------------------------
Training on OR data
weights: [w1A(11.17), w2A(11.17), wA(6.93)]
Trained weights:
Weight 'w1A': 11.172709
Weight 'w2A': 11.173043
Weight 'wA': 6.933418
Testing on OR test-data
test((0.1, 0.1, 0)) returned: 0.009023640720038306 => 0 [correct]
test((0.1, 0.9, 1)) returned: 0.9857913164734841 => 1 [correct]
test((0.9, 0.1, 1)) returned: 0.9857875715894618 => 1 [correct]
test((0.9, 0.9, 1)) returned: 0.999998107780629 => 1 [correct]
Accuracy: 1.000000
--------------------------------------
Training on AND data
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: 0.020284490369173127 => 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
→  Codes
```

بخش پنجم: finite Difference

در این بخش پیاده سازی تابع finite_difference به عنوان یک متد از کلاس Network به صورت زیر می باشد:

```python
def finite_difference(self):
    fault_counter = 0
    e = 10 ** (-8)
    for n in self.neurons:
        for j, w in enumerate(n.my_weights):
            self.clear_cache()
            fx = n.output()
            formulated_derivative = n.dOutdX(w)
            self.clear_cache()
            original_weight = n.my_weights[j].get_value()
            n.my_weights[j].set_value(original_weight + e)
            fxe = n.output()
            calculated_derivative = (fxe - fx) / e
            n.my_weights[j].set_value(original_weight)
            if (np.abs(calculated_derivative - formulated_derivative) > 10**(-5)):
                fault_counter += 1
                print("fault accur with weight ", w, f"calculated Derivative: {calculated_derivative}",\
                      f"and formulated Derivative: {formulated_derivative}")
                return
            else:
                print("ok")
    print(f"number of fault is {fault_counter}")
    return
```

و در قسمت Train می توانیم آن را تست کنیم، که نتیجه به صورت زیر است:

```
→  Codes python3 neural_net_tester.py simple
--------------------------------------
Training on OR data
weights: [w1A(11.17), w2A(11.17), wA(6.93)]
True
Trained weights:
Weight 'w1A': 11.172709
Weight 'w2A': 11.173043
Weight 'wA': 6.933418
Testing on OR test-data
test((0.1, 0.1, 0)) returned: 0.009023640720038306 => 0 [correct]
test((0.1, 0.9, 1)) returned: 0.9857913164734841 => 1 [correct]
test((0.9, 0.1, 1)) returned: 0.9857875715894618 => 1 [correct]
test((0.9, 0.9, 1)) returned: 0.999998107780629 => 1 [correct]
Accuracy: 1.000000
--------------------------------------
Training on AND data
weights: [w1A(10.50), w2A(10.50), wA(14.37)]
True
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
```

همانطور که مشاهده می شود، اختلاف دو مقدار مشتق به ازای هر وزن، همواره از یک مقدار معقولی (10 ^-5) کمتر است .

بخش ششم: پیاده سازی شبکه ی عصبی دولایه ای

پیاده سازی شبکه عصبی دولایه داخل تابع make_neural_net_two_layer() به صورت زیر است:

```python
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)
    i1 = Input('i1', 0)
    i2 = Input('i2', 0)

    w_A = Weight('w_A', random_weight())
    w_1A = Weight('w_1A', random_weight())
    w_1B = Weight('w_1B', random_weight())

    w_B = Weight('w_B', random_weight())
    w_2A = Weight('w_2A', random_weight())
    w_2B = Weight('w_2B', random_weight())

    A = Neuron('A', [i1,i2,i0], [w_1A,w_2A,w_A])
    B = Neuron('B', [i1,i2,i0], [w_1B,w_2B,w_B])

    w_C = Weight('w_C', random_weight())
    w_AC = Weight('w_AC', random_weight())
    w_BC = Weight('w_BC', random_weight())

    C = Neuron('C', [A, B, i0], [w_AC, w_BC, w_C])

    P = PerformanceElem(C, 0.0)

    return Network(P, [A, B, C])
    # raise NotImplementedError("Implement me!")
```

پس از پیاده سازی، به وسیله ی دستور python3 neural_net_tester.py two_layer آن را تست می کنیم. و همه ی دقت های آن برابر با یک یا همان ۱۰۰٪ می باشد:

```
---------------------------------------
Training on OR data
weights: [w_1A(4.47), w_2A(4.42), w_A(2.62), w_1B(-4.24), w_2B(-4.30), w_B(-2.51), w_AC(7.79), w_BC(-7.34), w_C(0.56)]
Trained weights:
Weight 'w_1A': 4.471370
Weight 'w_2A': 4.419079
Weight 'w_A': 2.622611
Weight 'w_1B': -4.243544
Weight 'w_2B': -4.297716
Weight 'w_B': -2.508835
Weight 'w_AC': 7.785762
Weight 'w_BC': -7.335022
Weight 'w_C': 0.555203
Testing on OR test-data
test((0.1, 0.1, 0)) returned: 0.0038946205602074617 => 0 [correct]
test((0.1, 0.9, 1)) returned: 0.9937717616156505 => 1 [correct]
test((0.9, 0.1, 1)) returned: 0.9937670953788678 => 1 [correct]
test((0.9, 0.9, 1)) returned: 0.9992186339759043 => 1 [correct]
Accuracy: 1.000000
---------------------------------------
Training on AND data
weights: [w_1A(2.95), w_2A(3.34), w_A(3.98), w_1B(-4.49), w_2B(-4.13), w_B(-5.66), w_AC(7.17), w_BC(-9.53), w_C(-0.35)]
Trained weights:
Weight 'w_1A': 2.951971
Weight 'w_2A': 3.340751
Weight 'w_A': 3.978100
Weight 'w_1B': -4.491011
Weight 'w_2B': -4.129295
Weight 'w_B': -5.661480
Weight 'w_AC': 7.166471
Weight 'w_BC': -9.527051
Weight 'w_C': -0.354753
Testing on AND test-data
test((0.1, 0.1, 0)) returned: 0.00014317401518460169 => 0 [correct]
test((0.1, 0.9, 0)) returned: 0.0066082713475809065 => 0 [correct]
test((0.9, 0.1, 0)) returned: 0.006477776235925617 => 0 [correct]
test((0.9, 0.9, 1)) returned: 0.9953123701612101 => 1 [correct]
Accuracy: 1.000000
```

```
---------------------------------------
Training on EQUAL data
weights: [w_1A(2.83), w_2A(-9.61), w_A(1.88), w_1B(-5.16), w_2B(-9.99), w_B(-1.02), w_AC(-6.19), w_BC(7.29), w_C(-0.06)]
Trained weights:
Weight 'w_1A': 2.831431
Weight 'w_2A': -9.614049
Weight 'w_A': 1.876121
Weight 'w_1B': -5.158514
Weight 'w_2B': -9.993999
Weight 'w_B': -1.017947
Weight 'w_AC': -6.186990
Weight 'w_BC': 7.294413
Weight 'w_C': -0.064180
Testing on EQUAL test-data
test((0.1, 0.1, 1)) returned: 0.9150142878681388 => 1 [correct]
test((0.1, 0.9, 0)) returned: 0.5163580785761048 => 0 [wrong]
test((0.9, 0.1, 0)) returned: 0.0748721966507282 => 0 [correct]
test((0.9, 0.9, 1)) returned: 0.5155172078533236 => 1 [correct]
Accuracy: 0.750000
---------------------------------------
Training on NOT_EQUAL data
weights: [w_1A(-6.09), w_2A(-6.09), w_A(-2.40), w_1B(5.69), w_2B(5.69), w_B(8.81), w_AC(-10.41), w_BC(-10.23), w_C(-5.06)]
Trained weights:
Weight 'w_1A': -6.091185
Weight 'w_2A': -6.089324
Weight 'w_A': -2.401487
Weight 'w_1B': 5.688892
Weight 'w_2B': 5.686573
Weight 'w_B': 8.805299
Weight 'w_AC': -10.407586
Weight 'w_BC': -10.226546
Weight 'w_C': -5.055354
Testing on NOT_EQUAL test-data
test((0.1, 0.1, 0)) returned: 0.05131416543507837 => 0 [correct]
test((0.1, 0.9, 1)) returned: 0.9874845389004957 => 1 [correct]
test((0.9, 0.1, 1)) returned: 0.9874795772995518 => 1 [correct]
test((0.9, 0.9, 0)) returned: 0.039067217275299534 => 0 [correct]
Accuracy: 1.000000
```

```
----------------------------------------
Training on horizontal-bands data
weights: [w_1A(0.07), w_2A(-4.67), w_A(-11.38), w_1B(0.22), w_2B(-6.72), w_B(-3.21), w_AC(12.13), w_BC(-11.85), w_C(5.91)]
Trained weights:
Weight 'w_1A': 0.074498
Weight 'w_2A': -4.666823
Weight 'w_A': -11.379889
Weight 'w_1B': 0.216935
Weight 'w_2B': -6.718208
Weight 'w_B': -3.205975
Weight 'w_AC': 12.129981
Weight 'w_BC': -11.848466
Weight 'w_C': 5.905549
Testing on horizontal-bands test-data
test((1, 1.5, 1)) returned: 0.9976935471086034 => 1 [correct]
test((2, 1.5, 1)) returned: 0.9977078089374308 => 1 [correct]
test((3, 1.5, 1)) returned: 0.9977183740293424 => 1 [correct]
test((0, 1.5, 1)) returned: 0.9976758017071272 => 1 [correct]
test((4, 0, 0)) returned: 0.004381737892437479 => 0 [correct]
test((4, 4, 0)) returned: 0.0027473142318270366 => 0 [correct]
test((-1, 0, 0)) returned: 0.0063283772591907535 => 0 [correct]
test((-1, 4, 0)) returned: 0.00273782084049732 => 0 [correct]
Accuracy: 1.000000
```

```
----------------------------------------
Training on vertical-bands data
weights: [w_1A(-4.51), w_2A(0.07), w_A(-11.02), w_1B(-9.71), w_2B(0.97), w_B(-3.26), w_AC(11.90), w_BC(-11.41), w_C(5.85)]
Trained weights:
Weight 'w_1A': -4.506870
Weight 'w_2A': 0.072547
Weight 'w_A': -11.024224
Weight 'w_1B': -9.710062
Weight 'w_2B': 0.965477
Weight 'w_B': -3.256550
Weight 'w_AC': 11.904178
Weight 'w_BC': -11.408017
Weight 'w_C': 5.847530
Testing on vertical-bands test-data
test((0, 1, 0)) returned: 0.005559757103607995 => 0 [correct]
test((0, 2, 0)) returned: 0.005025630319752322 => 0 [correct]
test((0, 1.5, 0)) returned: 0.005224264366854344 => 0 [correct]
test((1.5, 2, 1)) returned: 0.9973021111294399 => 1 [correct]
test((1.5, 5, 1)) returned: 0.9973321614914142 => 1 [correct]
test((1.5, 1, 1)) returned: 0.9972749874561931 => 1 [correct]
test((3, 1, 0)) returned: 0.007548308718209909 => 0 [correct]
test((3, 1.5, 0)) returned: 0.007797818527481219 => 0 [correct]
test((3, 2, 0)) returned: 0.008063533668732166 => 0 [correct]
test((1, 1.5, 1)) returned: 0.9974393807305806 => 1 [correct]
test((1, -1.5, 1)) returned: 0.9976070889291458 => 1 [correct]
test((2, 1.5, 1)) returned: 0.9916747485296661 => 1 [correct]
test((2, -1.5, 1)) returned: 0.9891217589456359 => 1 [correct]
test((4, 0, 0)) returned: 0.0029099080668377702 => 0 [correct]
test((4, 4, 0)) returned: 0.0029204741360524888 => 0 [correct]
test((-1, 0, 0)) returned: 0.004719394871382647 => 0 [correct]
test((-1, 4, 0)) returned: 0.0047192748080959485 => 0 [correct]
Accuracy: 1.000000
```

```
----------------------------------------
Training on diagonal-band data
weights: [w_1A(4.75), w_2A(-4.27), w_A(-3.64), w_1B(3.82), w_2B(-4.04), w_B(3.82), w_AC(10.69), w_BC(-11.00), w_C(5.35)]
Trained weights:
Weight 'w_1A': 4.747488
Weight 'w_2A': -4.274671
Weight 'w_A': -3.640627
Weight 'w_1B': 3.822508
Weight 'w_2B': -4.040037
Weight 'w_B': 3.817979
Weight 'w_AC': 10.691860
Weight 'w_BC': -10.995463
Weight 'w_C': 5.345907
Testing on diagonal-band test-data
test((-1, -1, 1)) returned: 0.9902600876561047 => 1 [correct]
test((5, 5, 1)) returned: 0.9947214917998429 => 1 [correct]
test((-2, -2, 1)) returned: 0.9867244428269898 => 1 [correct]
test((6, 6, 1)) returned: 0.9948542453770578 => 1 [correct]
test((3.5, 3.5, 1)) returned: 0.9944086256622482 => 1 [correct]
test((1.5, 1.5, 1)) returned: 0.9935556242615637 => 1 [correct]
test((4, 0, 0)) returned: 0.0035072998228320804 => 0 [correct]
test((0, 4, 0)) returned: 0.0047450763162799935 => 0 [correct]
Accuracy: 1.000000
```

```
--------------------------------------
Training on inverse-diagonal-band data
weights: [w_1A(4.74), w_2A(-4.27), w_A(-3.63), w_1B(3.85), w_2B(-4.06), w_B(3.86), w_AC(-10.68), w_BC(10.98), w_C(-5.34)]
Trained weights:
Weight 'w_1A': 4.738647
Weight 'w_2A': -4.266682
Weight 'w_A': -3.634781
Weight 'w_1B': 3.852600
Weight 'w_2B': -4.064160
Weight 'w_B': 3.864729
Weight 'w_AC': -10.681968
Weight 'w_BC': 10.980752
Weight 'w_C': -5.344373
Testing on inverse-diagonal-band test-data
test((-1, -1, 0)) returned: 0.00969206855555521 => 0 [correct]
test((5, 5, 0)) returned: 0.005316177973027873 => 0 [correct]
test((-2, -2, 0)) returned: 0.013144728073778993 => 0 [correct]
test((6, 6, 0)) returned: 0.005185495328262319 => 0 [correct]
test((3.5, 3.5, 0)) returned: 0.005623775324050439 => 0 [correct]
test((1.5, 1.5, 0)) returned: 0.006461701121862004 => 0 [correct]
test((4, 0, 1)) returned: 0.9964704564520406 => 1 [correct]
test((0, 4, 1)) returned: 0.9952476735193524 => 1 [correct]
Accuracy: 1.000000
```

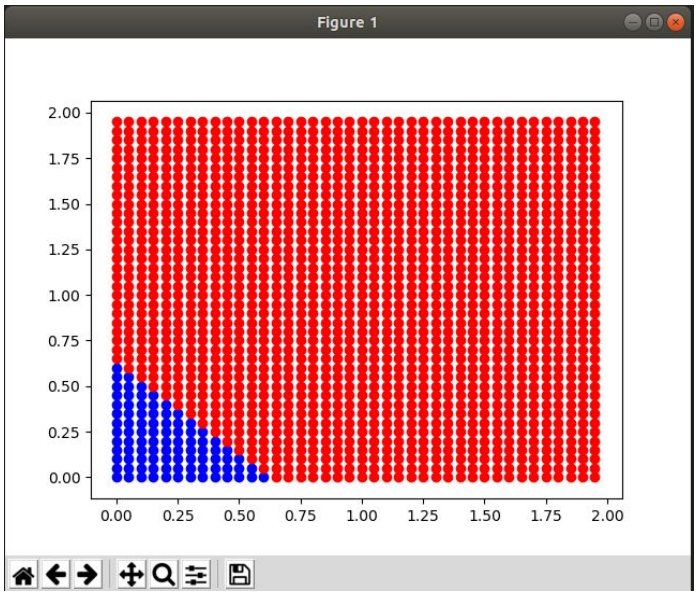
بخش هفتم: کشیدن ناحیه ی تصمیم گیری

پیاده سازی تابع plot_decision_boundary به صورت زیر می باشد:

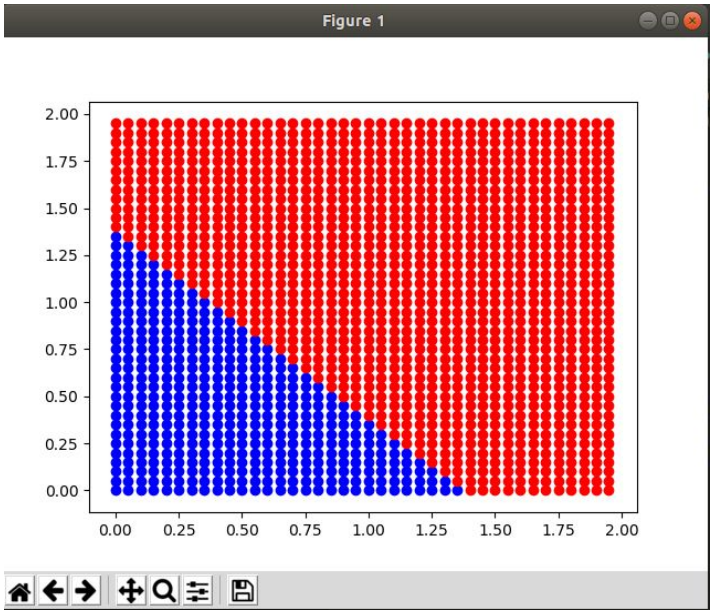
که در آن برای کشیدن grid از تابع scatter در کتابخانه ی mathplotlib استفاده شده است

```python
def plot_decision_boundary(network, xmin, xmax, ymin, ymax, step = 0.05):
    x = np.arange(xmin, xmax, step)
    y = np.arange(ymin, ymax, step)
    for i in x:
        for j in y:
            network.clear_cache()
            network.inputs[0].set_value(i)
            network.inputs[1].set_value(j)
            out = network.output.output()
            if out < 0.5:
                plt.scatter(i, j, c='blue')
            else:
                plt.scatter(i, j, c='red')
    plt.show()
```

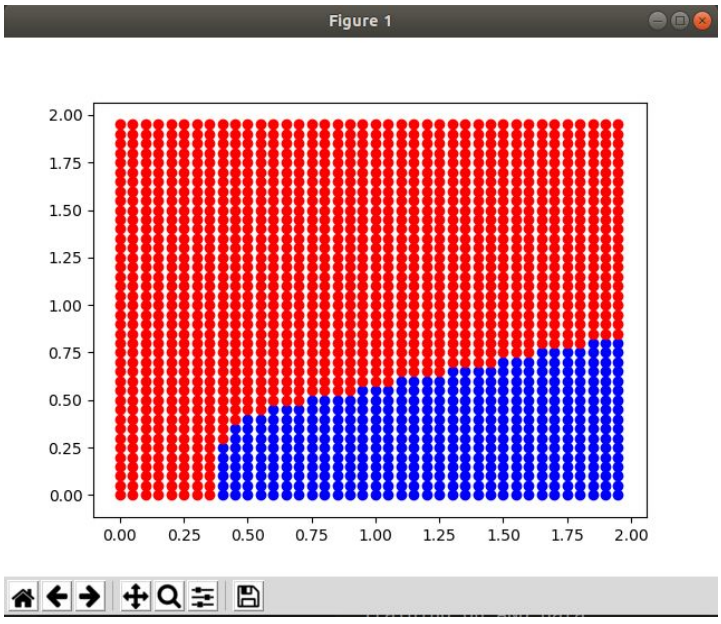
در صورتی که برای دیدن خروجی این تابع از دستور python3 neural_net_tester.py simple استفاده کنیم، نتیجه این تابع برای OR به صورت زیر است:



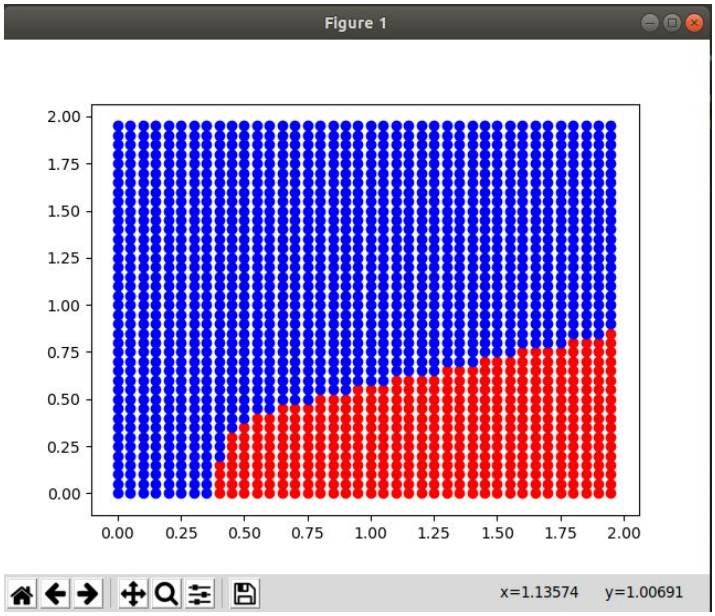
و برای تابع AND نیز به صورت زیر است:



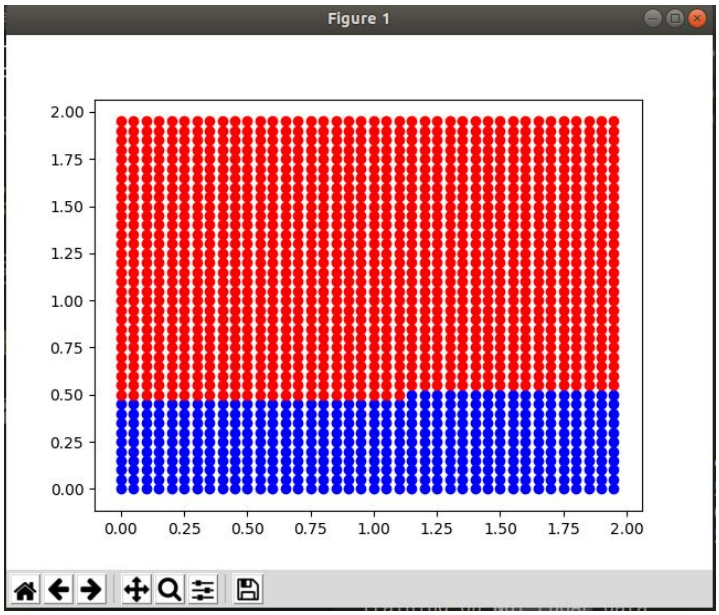
در شرایطی که برای تست این تابع از دستور python3 neural_net_tester.py two_layer استفاده کنیم،به ازای EQUAL خروجی به صورت زیر می باشد:



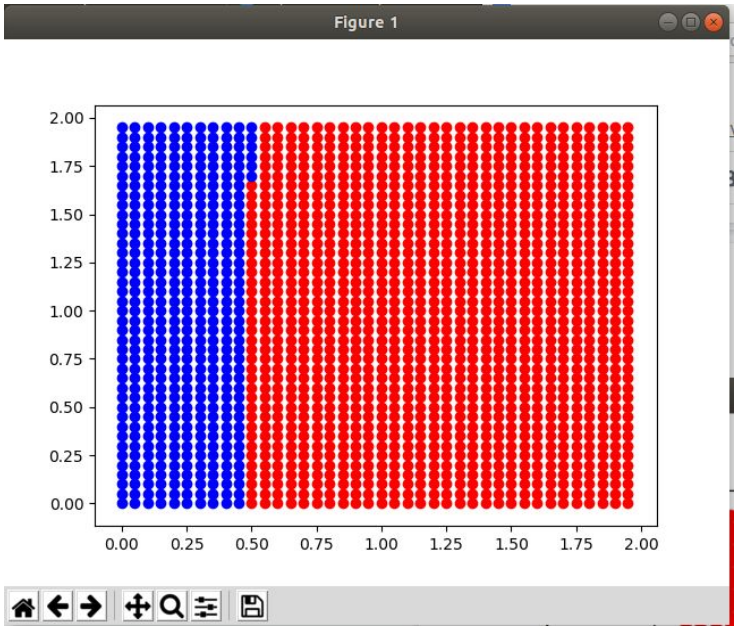
در شرایط NOT_EQUAL خروجی به صورت زیر است:



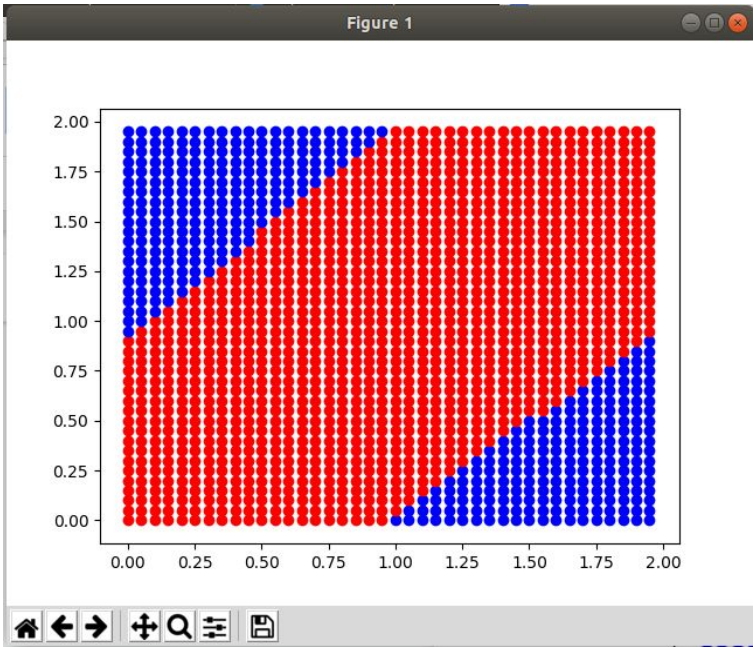
و نیز در حالت horizontal-bands نتیجه به صورت زیر می باشد:



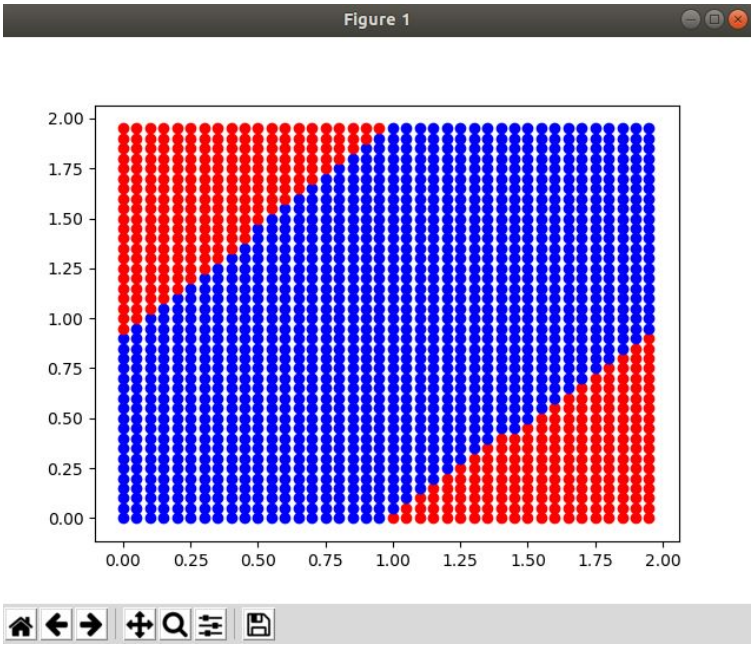
در حالت vertical-bands نتیجه به صورت زیر می باشد



در حالت diagonal-band نتیجه به صورت زیر می باشد:



در حالت inverse-diagonal-band نتیجه به صورت زیر می باشد:



بخش هشتم: بیش برازش و regularization

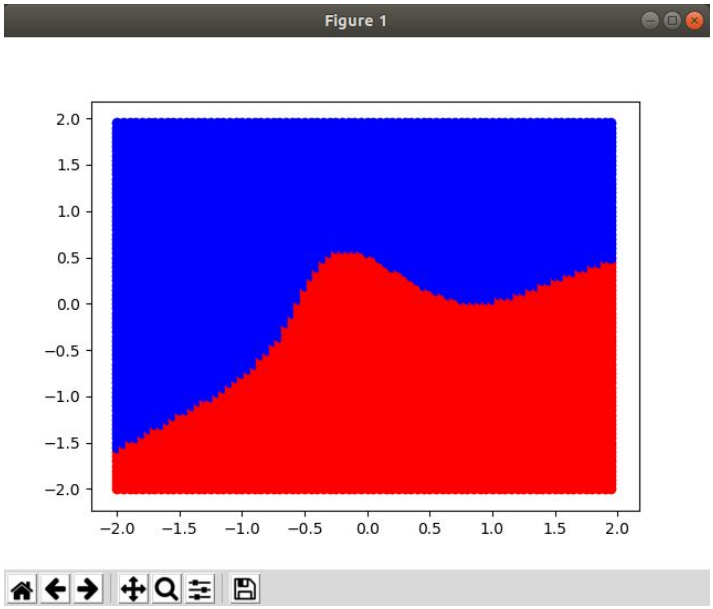
در این بخش ابتدا یک شبکه عصبی با دو ورودی در لایه ی اول و ۴۰ نورون در لایه ی دوم و ۱ نورون خروجی ایجادمی کنیم، پیاده سازی آن به صورت زیر می باشد:

```
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)
    i1 = Input('i1', 0)
    i2 = Input('i2', 0)

    Neurons = list()
    c_wights = list()

    names = list()
    elements = string.ascii_uppercase[0:20]
    for i in range(2):
        for j in range(20):
            names.append(elements[i]+elements[j])

    for i in range(len(names)):
        w_A = Weight('w_' + names[i], random_weight())
        w_1A = Weight('w_1' + names[i], random_weight())
        w_2A = Weight('w_2' + names[i], random_weight())

        w_c = Weight('w_' + names[i] + 'C', random_weight())
        c_wights.append(w_c)

        A = Neuron(names[i], [i1, i2, i0], [w_1A, w_2A, w_A])
        Neurons.append(A)

    C = Neuron('C', Neurons + [i0], c_wights + [Weight('w_C', random_weight())])
    P = PerformanceElem(C, 0.0)
    Neurons.append(C)
    network = Network(P, Neurons)
    return network
```

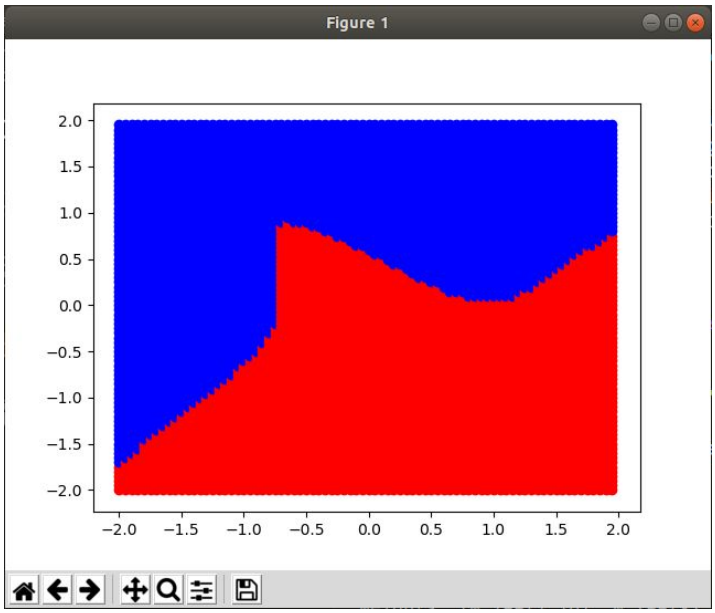
جدول نتایج دقت تست طی چند iteration به صورت زیر است:

Iteration number	accuracy
100	0.99
500	0.94
1000	0.87

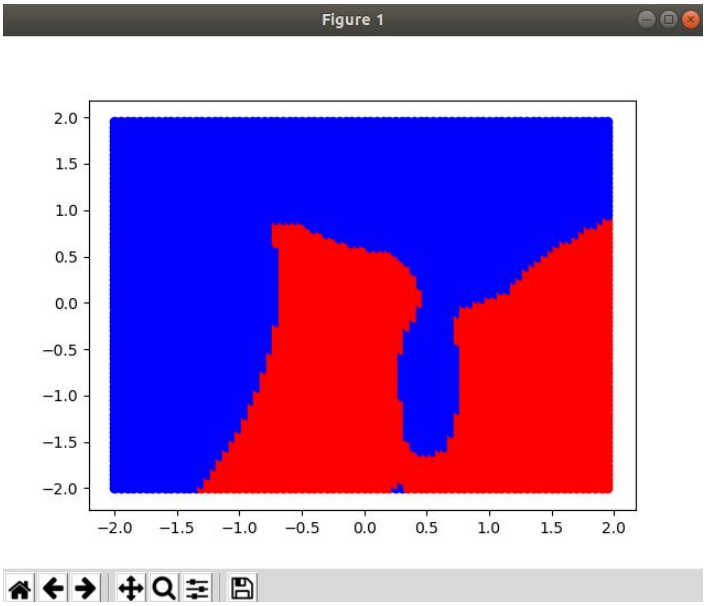
ناحیه ی تصمیم گیری در iteration = 100



ناحیه ی تصمیم گیری در iteration = 500



ناحیه ی تصمیم گیری در iteration = 1000



در این قسمت، با افزایش عدد iteration ، مدل ما بر روی داده ی مربوط به یادگیری overfit می شود. زیرا شبکه عصبی بسیار انعطاف پذیر می باشد. همچنین در در شرایطی که dicision boundary ما پیچیده باشد، می توان گفت با overfitting روبرو هستیم ، بنابراین تصاویر ضمیمه شده از dicision boundary خود بیانگر overfit کردن شبکه عصبی بر روی train data می باشد.

پیاده سازی کلاس RegularizedPerformanceElem که از کلاس PerformanceElem ارث می برد به صورت زیر می باشد:

```
class RegularizedPerformanceElem(PerformanceElem):
    def set_weights(self, weights):
        self.my_weights = weights

    def output(self):
        output = 1 - (1/2) * ((self.my_desired_val -  self.my_input.output()) ** 2)
        offset = 0
        a = 1 * (10 ** 0)
        for w in self.my_weights:
            offset += w.get_value() ** 2
        offset = offset* (a/2)
        return output - offset

    def dOutdX(self, elem):
        dout = (self.my_desired_val - self.my_input.output()) * self.my_input.dOutdX(elem)
        offset = 1* (10 ** 0) * elem
        return dout - offset
```

برای آنکه بتوانیم به هر instance از این کلاس مقادیر weight را بدهیم، متد زیر را در کلاس Network پیاده سازی کردیم، که این متد در انتهای init این کلاس صدا زده می شود. به عبارتی پس از set شدن مقدار آرایه ی weight در Network، این مقادیر به هر نمونه از کلاس RegularizedPerformanceElem که به عنوان ورودی به Network داده شده است، نیز داده می شود. این متد در کلاس Network به صورت زیر می باشد:

```
def set_performance_weights(self):
    if(isinstance(self.performance, RegularizedPerformanceElem)):
        self.performance.set_weights(self.weights)
```

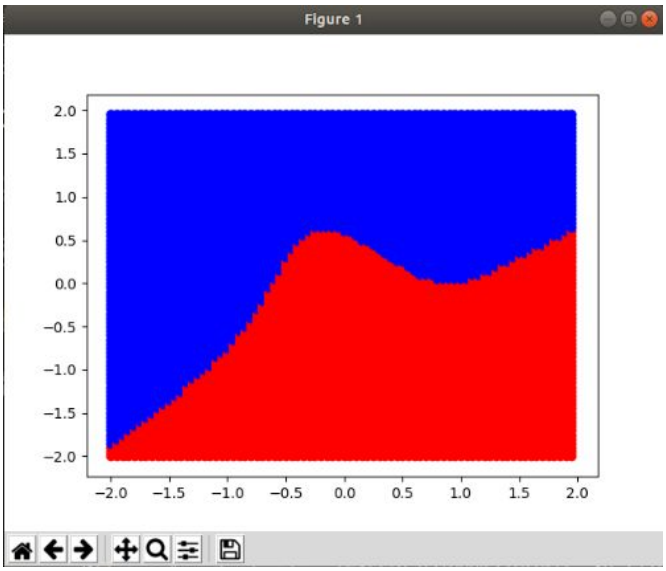
و نیز init کلاس Network به صورت زیر تغییر می کند:

```
def __init__(self,performance_node,neurons):
    self.inputs =  []
    self.weights = []
    self.performance = performance_node
    self.output = performance_node.get_input()
    self.neurons = neurons[:]
    self.neurons.sort(key=functools.cmp_to_key(alphabetize))
    for neuron in self.neurons:
        self.weights.extend(neuron.get_weights())
        for i in neuron.get_inputs():
            if isinstance(i,Input) and not ('i0' in i.get_name()) and not i in self.inputs:
                self.inputs.append(i)
    self.weights.reverse()
    self.weights = []
    for n in self.neurons:
        self.weights += n.get_weight_nodes()
    # change(add)
    self.set_performance_weights()
```

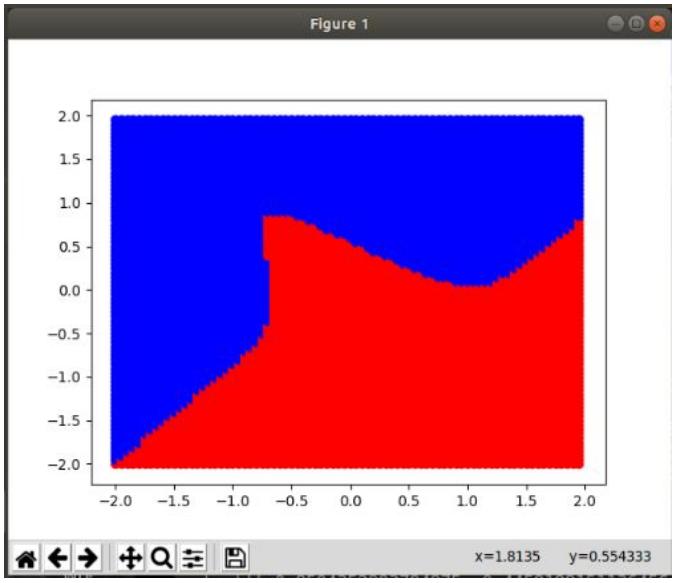
نتیجه بدست آمده پس از اجرای برنامه با این تغییرات ذکر شده طی چند iteration به صورت زیر است:

Iteration number	accuracy
100	1.00
500	0.95
1000	0.94

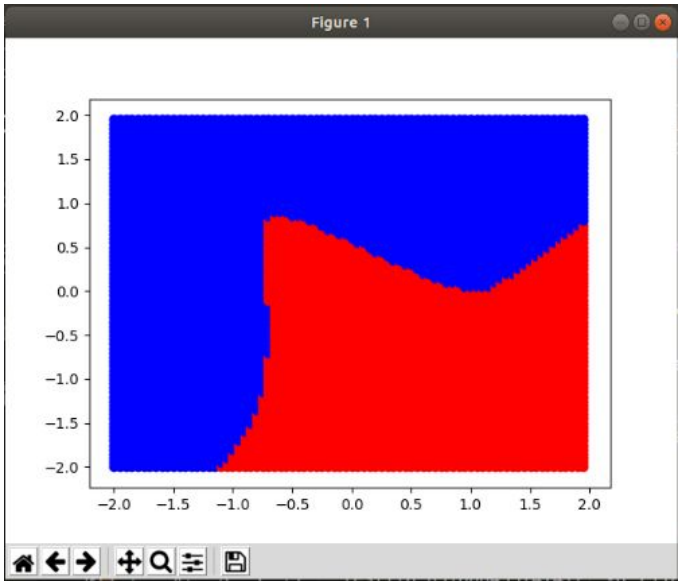
ناحیه ی تصمیم گیری در iteration = 100 پس از regularization:



ناحیه ی تصمیم گیری در iteration = 500 پس از regularization:



ناحیه ی تصمیم گیری در iteration = 1000 پس از regularization:



با این کار مشکل بیش برازش یا overfitting حل شده.

دلیل حل شدن مشکل overfitting آن است که هنگامی که عبارت مربوط به regularization را اضافه می کنیم، باعث می شود پارامترها دائماً به 0 برسند به عبارت دیگر باعث می شویم با قرار دادن محدودیت های اضافی و در عین حال مطلوب مانع overfiting شویم.