Assignment one

MLQP and min-max modular networks.

* Name:Xin Xu Student ID:519021910726 Email: xuxin20010203@sjtu.edu.cn

1. *back-propagation algorithm.* Please derive a back-propagation algorithm for multilayer quadratic perceptron (MLQP) in on-line or sequential mode. The output of each neuron in a multilayer perceptron is:
$x_{kj} = f(\sum_{i=1}^{N_{k-1}}(u_{kji}x_{k-1,i}^2 + v_{kji}x_{k-1,i}) + b_{kj})$.

   Solution. Suppose that the loss of output is E, and the gradient propagated back to x_{kj} is $\frac{\partial E}{\partial x_{kj}} = G_{kj}$. And we define $net_{kj} = \sum_{i=1}^{N_{k-1}}(u_{kji}x_{k-1,i}^2 + v_{kji}x_{k-1,i}) + b_{kj}$, so $x_{kj} = f(net_{kj})$.
   So, $\frac{\partial E}{\partial u_{kji}} = \frac{\partial E}{\partial x_{kj}}\frac{\partial x_{kj}}{\partial net_{kj}}\frac{\partial net_{kj}}{\partial u_{kji}} = G_{kj}f'(net_{kj})x_{k-1,i}^2$.
   $\frac{\partial E}{\partial v_{kji}} = \frac{\partial E}{\partial x_{kj}}\frac{\partial x_{kj}}{\partial net_{kj}}\frac{\partial net_{kj}}{\partial v_{kji}} = G_{kj}f'(net_{kj})x_{k-1,i}$.
   $\frac{\partial E}{\partial b_{kj}} = \frac{\partial E}{\partial x_{kj}}\frac{\partial x_{kj}}{\partial net_{kj}}\frac{\partial net_{kj}}{\partial b_{kj}} = G_{kj}f'(net_{kj})$.
   $\frac{\partial E}{\partial x_{k-1,j}} = \frac{\partial E}{\partial x_{kj}}\frac{\partial x_{kj}}{\partial net_{kj}}\frac{\partial net_{kj}}{\partial x_{k-1,j}} = G_{kj}f'(net_{kj}) * \sum_{i=1}^{N_{k-1}}(2u_{kji}x_{k-1,i} + v_{kji})$, and it's the same with layer $k-1$.
   If the learning rate is η, $\partial u'_{kji} = \partial u_{kji} - \eta * \frac{\partial E}{\partial u_{kji}}$; $\partial v'_{kji} = \partial v_{kji} - \eta * \frac{\partial E}{\partial v_{kji}}$; $\partial b'_{kj} = \partial b_{kj} - \eta * \frac{\partial E}{\partial b_{kji}}$.
   If x_{kj} is the output unit and this model is trained in sequential mode, $E = \frac{(o_{kj}-x_{kj})^2}{2}$, o_{kj} is the ideal output of unit x_{kj}. So, $\frac{\partial E}{\partial x_{kj}} = G_{kj} = x_{kj} - o_{kj}$. So, the BP algorithm can continue. □

2. *two spiral problem.* Using MLQP with one hidden layer to classify two spirals.
   First, we should prepare the data, we use DataLoader to prepare data iterator for batch mode or sequential mode. The code is as follows:

```
1   data_train = torch.from_numpy(np.loadtxt('two_spiral_train_data.txt')). float()
2   data_test = torch.from_numpy(np.loadtxt('two_spiral_test_data.txt')). float()
3
4   def load_data(data_ori, batch_size, is_train = True):
5       data_in = data_ori[:, 0:2]
6       data_out = data_ori[:, 2]
7       return data.DataLoader(data.TensorDataset(data_in, data_out), batch_size, shuffle =
            is_train)
8
9   batch_size = 30
10  data_train_iter = load_data(data_train, batch_size, True)
11  data_test_iter = load_data(data_test, batch_size, False)
```

   Then, we define the basic part of MLQP, such as weights, bias, net, loss function, optimation function and so on.
   In my opinion, two spiral problem is a classification problem. And there are two classes: 0 and 1. The number of input features is 2, x axis and y axis. The number of output features is 2, probabilities of two classes.
   The activation function is Sigmoid. To make output satisfy conditions of probability(which is larger or equal than 0 and less or equal than 1), I use softmax algorithm to map output to [0, 1]. To avoid overflow, elements in output all subtract their maximum.
   I use cross entropy function to express loss. In practice, I meet gradient explosion, which may result in log in cross entropy function.
   The code of net structure is:

```
        u1 = torch.normal(0, 0.01, size = (input_num, hidden_num), requires_grad = True)
        v1 = torch.normal(0, 0.01, size = (input_num, hidden_num), requires_grad = True)
        b1 = torch.zeros(hidden_num, requires_grad=True)

        u2 = torch.normal(0, 0.01, size = (hidden_num, output_num), requires_grad = True)
        v2 = torch.normal(0, 0.01, size = (hidden_num, output_num), requires_grad = True)
        b2 = torch.zeros(output_num, requires_grad=True)

        u1_max = torch.normal(0, 0.01, size = (input_num, hidden_num), requires_grad = True)
        v1_max = torch.normal(0, 0.01, size = (input_num, hidden_num), requires_grad = True)
        b1_max = torch.zeros(hidden_num, requires_grad=True)

        u2_max = torch.normal(0, 0.01, size = (hidden_num, output_num), requires_grad = True)
        v2_max = torch.normal(0, 0.01, size = (hidden_num, output_num), requires_grad = True)
        b2_max = torch.zeros(output_num, requires_grad=True)

        #print(u1, v1, b1, u2, v2, b2)

        def quadratic_layer(u, v, b, x):
            return torch.matmul(x**2, u) + torch.matmul(x, v) + b

        def sigmoid(x):
            x_exp = torch.exp(-x)
            return 1 / (1 + x_exp)

        def softmax(x):
            x_max = torch. max(x)
            x_exp = torch.exp(x - x_max)
            partition = x_exp. sum(1, keepdim=True)
            return x_exp / partition

        def mlqp(x, u1, v1, b1, u2, v2, b2):
            out1 = sigmoid(quadratic_layer(u1, v1, b1, x))
            out2 = quadratic_layer(u2, v2, b2, out1)
            return softmax(out2)

        def cross_entropy(y_hat, y):
            return -1 * torch.log(y_hat[ range( len(y_hat)), y])

        def sgd(params, lr, batch_size):
            with torch.no_grad():
                for param in params:
                    #print(param.grad)
                    param -= lr * param.grad / batch_size
                    param.grad.zero_()
```

The code of training and testing is:

```
        def train_spiral(net, train_iter, loss, updater, epoch_num, lr, batch_size):
            accuracy_max = 0
            for i in  range(epoch_num):
                accuracy_rate = [0., 0., 0.] # loss, predict right, num
                for x, y in train_iter:
                    y_hat = net(x, u1, v1, b1, u2, v2, b2)
                    #print(y.long())
                    l = loss(y_hat, y. long())
                    #print(l)
                    #updater.zero_grad()
                    l. sum().backward()
                    #updater.step()
```

```
                    updater([u1, v1, b1, u2, v2, b2], lr, batch_size)
                    accuracy_rate[0] = accuracy_rate[0] + l. sum()
                    accuracy_rate[1] = accuracy_rate[1] + accuracy(y_hat, y)
                    accuracy_rate[2] = accuracy_rate[2] + y.numel()
                accuracy_now = accuracy_rate[1] / accuracy_rate[2]
                if accuracy_now > accuracy_max:
                    accuracy_max = accuracy_now
                    print('max accuracy:', accuracy_max)
                    u1_max, v1_max, b1_max, u2_max, v2_max, b2_max = u1 * 1, v1 * 1, b1 * 1, u2
                        * 1, v2 * 1, b2 * 1
                print('loss: {0}, accuracy: {1}'. format(accuracy_rate[0] / accuracy_rate[2],
                    accuracy_rate[1] / accuracy_rate[2]))

        def evaluate_spiral(net, test_iter, u1_g, v1_g, b1_g, u2_g, v2_g, b2_g):
            accuracy_rate = [0., 0.]
            with torch.no_grad():
                for x, y in test_iter:
                    y_hat = net(x, u1_g, v1_g, b1_g, u2_g, v2_g, b2_g)
                    plot_decision_boundary(x, y_hat)
                    accuracy_rate[0] = accuracy_rate[0] + accuracy(y_hat, y)
                    accuracy_rate[1] = accuracy_rate[1] + y.numel()
            print('accuracy: {0}'. format(accuracy_rate[0] / accuracy_rate[1]))
```

Other codes are in the file q2.ipynb.

The learning rates I choose are 0.05, 0.075 and 0.1, and the training time and the decision boundaries are as follows(with epoch number is 1000):
The learning rate is 0.05, its training time is 11.74s, accuracy is 0.59.

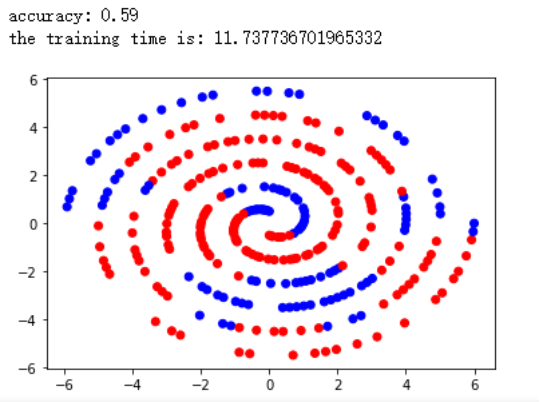

Figure 1: training of learning rate 0.05

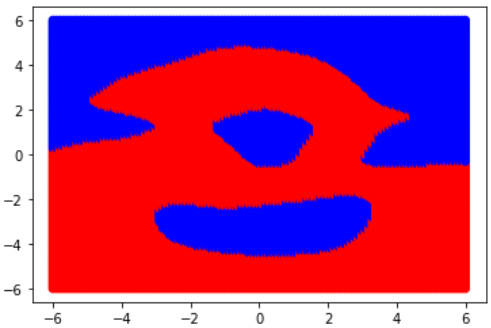

Figure 2: decision boundary of learning rate 0.05

The learning rate is 0.075, its training time is 11.93s, accuracy is 0.48.

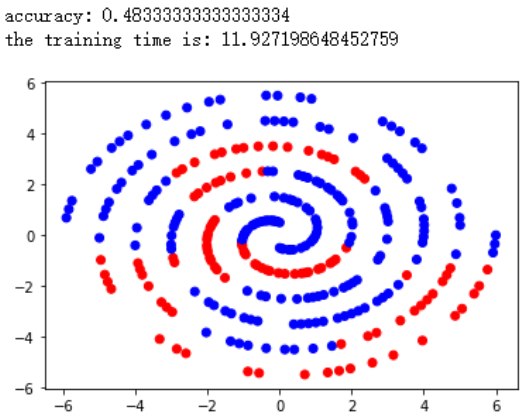

Figure 3: training of learning rate 0.075

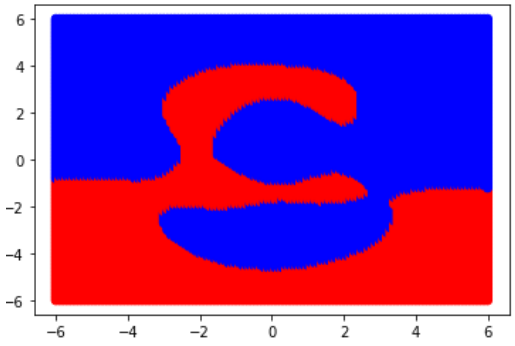

Figure 4: decision boundary of learning rate 0.075

The learning rate is 0.1, its training time is 11.66s, accuracy is 0.62.

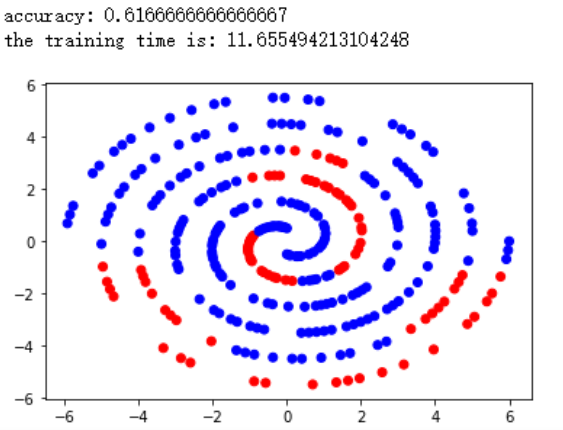

Figure 5: training of learning rate 0.1

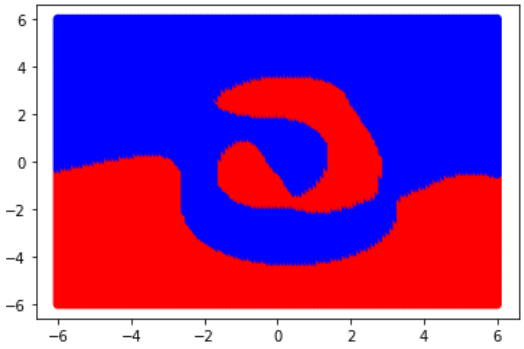

Figure 6: decision boundary of learning rate 0.1

As we can see, in a limited range, higher learning rate has a better performance(decision boundary). But training time almost unchanged.

3. *max min module*. Use max min module to increase the accuracy.
The code is similar to question 2, and I modify code in question 2 into a class to make it clearer. Please see code in file q3.ipynb.
I divide two spiral problem into four subproblem: to classify nodes with label 0 and distance to origin less than $r/2$; to classify nodes with label 1 and distance to origin less than $r/2$; to classify nodes with label 0 and distance to origin larger than $r/2$; to classify nodes with label 1 and distance to origin larger than $r/2$.
To classify specific subproblem, we construct data like this: remove train data with the same label but different class of distance. For example, to classify nodes with label 0 and distance to origin less than $r/2$, the data set we use is all the training date except data with label 0 and distance to origin larger than $r/2$.
The process of min-max module is as follows:
Firstly, we respectively train net for these four subproblems(which is the same with question 2). Then we figure subproblems with the same distance class by "min" method, thus leaving two subproblems. Finally, we figure this two subproblems by using "max" method.
The code is as follows:

```
        # to prepare four data set for these four subproblems
        data_train_in_0 = []
        data_train_in_1 = []
        data_train_out_0 = []
        data_train_out_1 = []

        for i in  range( len(data_train)):
            if data_train[i][0] ** 2 + data_train[i][1] ** 2 <= 9.5:
                if data_train[i][2]:
                    data_train_in_1.append( list(data_train[i]))
                else:
                    data_train_in_0.append( list(data_train[i]))
            if data_train[i][0] ** 2 + data_train[i][1] ** 2 >= 8.5:
                if data_train[i][2]:
                    data_train_out_1.append( list(data_train[i]))
                else:
                    data_train_out_0.append( list(data_train[i]))

        data_train_in_0 = torch.from_numpy(np.array(data_train_in_0))
        data_train_in_1 = torch.from_numpy(np.array(data_train_in_1))
```

```
21          data_train_out_0 = torch.from_numpy(np.array(data_train_out_0))
22          data_train_out_1 = torch.from_numpy(np.array(data_train_out_1))
23
24          q3_in_0_data = torch.cat((data_train_in_0, data_train_in_1, data_train_out_1), axis =
                0)
25          q3_in_1_data = torch.cat((data_train_in_0, data_train_in_1, data_train_out_0), axis =
                0)
26          q3_out_0_data = torch.cat((data_train_out_0, data_train_in_1, data_train_out_1), axis =
                 0)
27          q3_out_1_data = torch.cat((data_train_in_0, data_train_out_1, data_train_out_0), axis =
                 0)
28
29          print(data_train_in_0.shape, data_train_in_1.shape, data_train_out_0.shape,
                data_train_out_1.shape)
30
31          # to get the four networks
32          q3_in_0 = SpiralNet(hidden = 512, batch = 30, r = 0.1, epoch = 8000)
33          q3_in_1 = SpiralNet(hidden = 512, batch = 30, r = 0.1, epoch = 8000)
34          q3_out_0 = SpiralNet(hidden = 512, batch = 30, r = 0.1, epoch = 8000)
35          q3_out_1 = SpiralNet(hidden = 512, batch = 30, r = 0.1, epoch = 8000)
36
37          q3_in_0_iter = q3_in_0.load_data(q3_in_0_data, True)
38          q3_in_1_iter = q3_in_1.load_data(q3_in_1_data, True)
39          q3_out_0_iter = q3_out_0.load_data(q3_out_0_data, True)
40          q3_out_1_iter = q3_out_1.load_data(q3_out_1_data, True)
41
42          # min-max module
43          def plot_decision_boundary(x, y_class, color = ['b', 'r']):
44              col = []
45              for i in  range(0,  len(y_class)):
46                  col.append(color[y_class[i]])
47              plt.scatter(x[:, 0], x[:, 1], c = col)
48
49          def max_min(in_0, in_1, out_0, out_1, test_iter): #      max
50              accuracy_rate = [0., 0.]
51              with torch.no_grad():
52                  for x, y in test_iter:
53                      y_hat_in_0 = in_0.mlqp(x, in_0.u1_max, in_0.v1_max, in_0.b1_max, in_0.
                            u2_max, in_0.v2_max, in_0.b2_max)
54                      y_hat_in_0 = y_hat_in_0.argmax(axis=1)
55
56                      y_hat_in_1 = in_1.mlqp(x, in_1.u1_max, in_1.v1_max, in_1.b1_max, in_1.
                            u2_max, in_1.v2_max, in_1.b2_max)
57                      y_hat_in_1 = y_hat_in_1.argmax(axis=1)
58
59                      y_hat_out_0 = out_0.mlqp(x, out_0.u1_max, out_0.v1_max, out_0.b1_max, out_0
                            .u2_max, out_0.v2_max, out_0.b2_max)
60                      y_hat_out_0 = y_hat_out_0.argmax(axis=1)
61
62                      y_hat_out_1 = out_1.mlqp(x, out_1.u1_max, out_1.v1_max, out_1.b1_max, out_1
                            .u2_max, out_1.v2_max, out_1.b2_max)
63                      y_hat_out_1 = y_hat_out_1.argmax(axis=1)
64
65                      y_hat = torch. max(torch. min(y_hat_in_0, y_hat_in_1), torch.
                            min(y_hat_out_0, y_hat_out_1))
66
67                      plot_decision_boundary(x, y_hat)
68                      accuracy_rate[0] = accuracy_rate[0] +  float((y_hat. type(y.dtype) == y).
                            type(y.dtype). sum())
69                      accuracy_rate[1] = accuracy_rate[1] + y.numel()
70                  print('accuracy: {0}'. format(accuracy_rate[0] / accuracy_rate[1]))
```

```
71
72          # draw decision boundary
73          def max_min_boundary(in_0, in_1, out_0, out_1):
74              x = torch.from_numpy(in_0.create_points(-6, 6, 121)). float()
75              with torch.no_grad():
76                  y_hat_in_0 = in_0.mlqp(x, in_0.u1_max, in_0.v1_max, in_0.b1_max, in_0.u2_max,
                        in_0.v2_max, in_0.b2_max)
77                  y_hat_in_0 = y_hat_in_0.argmax(axis=1)
78
79                  y_hat_in_1 = in_1.mlqp(x, in_1.u1_max, in_1.v1_max, in_1.b1_max, in_1.u2_max,
                        in_1.v2_max, in_1.b2_max)
80                  y_hat_in_1 = y_hat_in_1.argmax(axis=1)
81
82                  y_hat_out_0 = out_0.mlqp(x, out_0.u1_max, out_0.v1_max, out_0.b1_max, out_0.
                        u2_max, out_0.v2_max, out_0.b2_max)
83                  y_hat_out_0 = y_hat_out_0.argmax(axis=1)
84
85                  y_hat_out_1 = out_1.mlqp(x, out_1.u1_max, out_1.v1_max, out_1.b1_max, out_1.
                        u2_max, out_1.v2_max, out_1.b2_max)
86                  y_hat_out_1 = y_hat_out_1.argmax(axis=1)
87
88                  y_hat = torch. max(torch. min(y_hat_in_0, y_hat_in_1), torch.
                        min(y_hat_out_0, y_hat_out_1))
89                  plot_decision_boundary(x, y_hat)
```

The process to train subproblem with label 0 and distance to origin less than $r/2$ is:
We can see the blue part, it matches well.

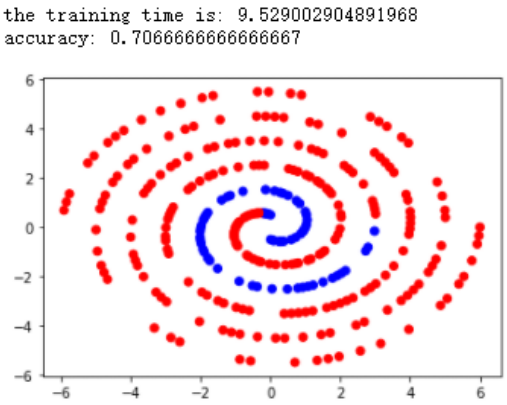

Figure 7: label 0 and distance less

And the training time is 9.53s(with learning rate 0.1 and epoch 1000).
The decision boundary is:

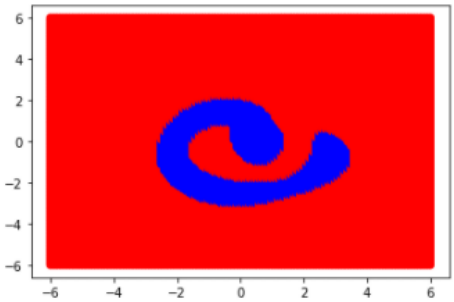

Figure 8: label 0 and distance less

The process to train subproblem with label 1 and distance to origin less than $r/2$ is:
We can see the red part, it matches well.

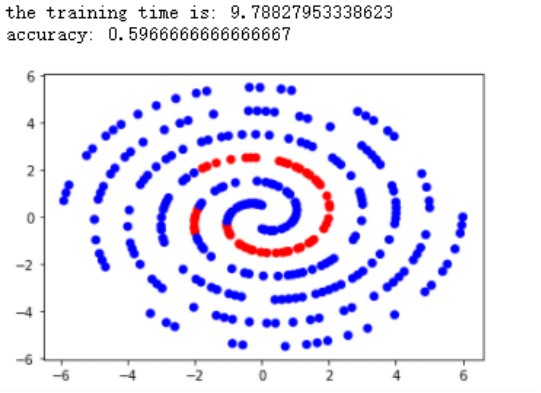

Figure 9: label 1 and distance less

And the training time is 9.79s(with learning rate 0.1 and epoch 1000).
The decision boundary is:

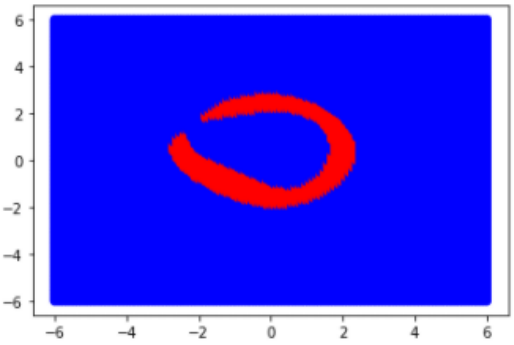

Figure 10: label 1 and distance less

The process to train subproblem with label 0 and distance to origin larger than $r/2$ is:
We can see the blue part, it matches well.

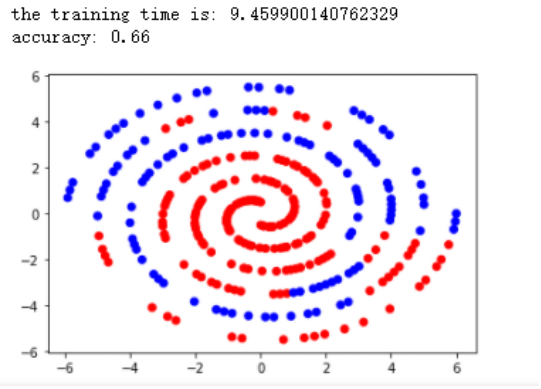

Figure 11: label 0 and distance larger

And the training time is 9.46s(with learning rate 0.1 and epoch 1000).
The decision boundary is:

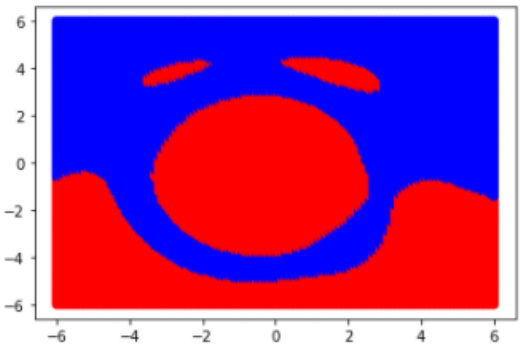

Figure 12: label 0 and distance larger

The process to train subproblem with label 1 and distance to origin larger than $r/2$ is:
We can see the red part, it matches well.

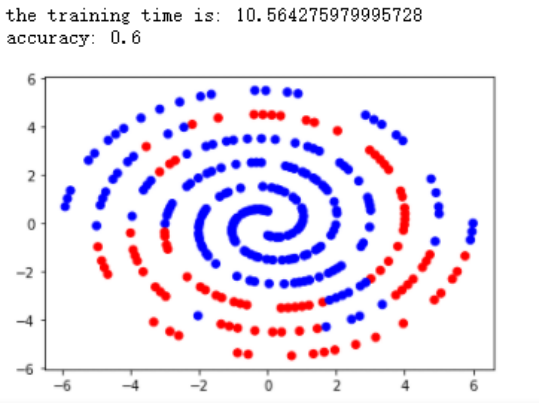

Figure 13: label 1 and distance larger

And the training time is 10.56s(with learning rate 0.1 and epoch 1000).
The decision boundary is:

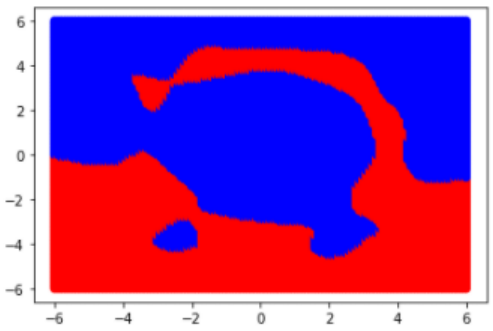

Figure 14: label 1 and distance larger

And the training output of max min module is Figure 15:
We can see that the accuracy is 0.74, higher than q2.
The decision boundary of max min module is Figure 16:

4. *question.* In question 3, I find the larger the number of epoch is, the higher accuracy we will achieve. But in practice, there is a condition where loss becomes nan. I think there may be

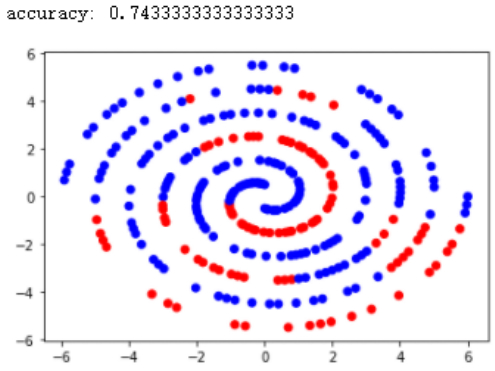

Figure 15: max min module

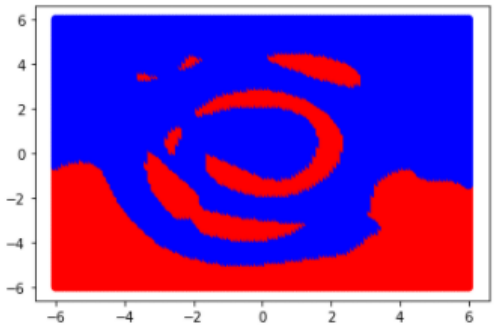

Figure 16: max min decision boundary

gradient explosion. But I can't figure out why it happens.
When I set epoch number to 8000, there is gradient explosion in some subproblems:

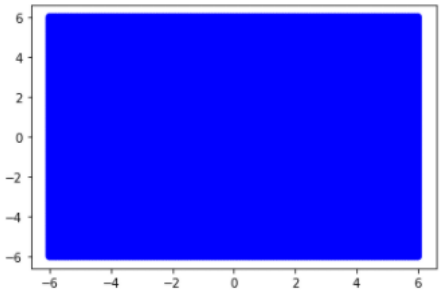

Figure 17: gradient explosion

But in some subproblems, high epoch number results in high accuracy, which can even achieve 1.0 like this!

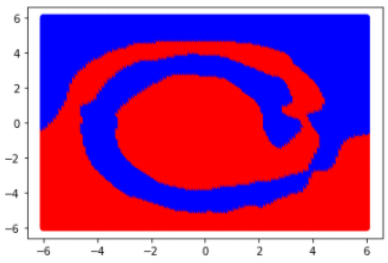

Figure 18: highest accuracy of 1.0

To tackle this problem, I reserve parameters with best accuracy in all epoches and use these best parameters to construct max min module. The output is:
We can see that the accuracy in test data is 0.92!

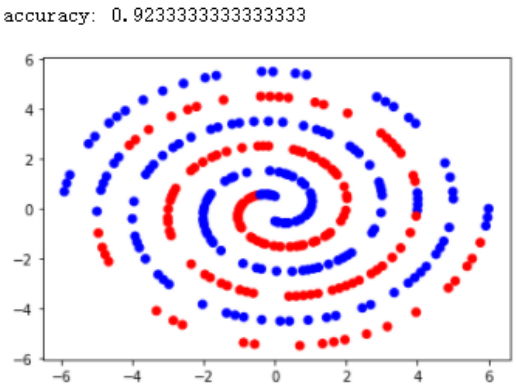

Figure 19: max min module with best parameters

And the decision boundary is: That is quite good!

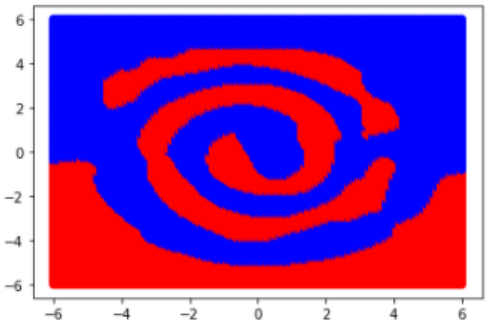

Figure 20: max min module's decision boundary with best parameters