Z5304897 Junyu Zhang

Part 1: Japanese Character Recognition

1. This is the final confusion matrix and final accuracy of the model *NetLin*.

```
<class 'numpy.ndarray'>
[[767.   5.   8.  13.  30.  63.   2.  62.  31.  19.]
 [  7. 668. 108.  19.  29.  24.  56.  15.  26.  48.]
 [  9.  61. 689.  27.  24.  20.  46.  38.  47.  39.]
 [  5.  34.  60. 758.  15.  56.  14.  18.  32.   8.]
 [ 61.  52.  78.  20. 625.  20.  33.  36.  21.  54.]
 [  8.  28. 124.  17.  19. 722.  30.   9.  33.  10.]
 [  4.  23. 146.  10.  25.  25. 722.  21.  11.  13.]
 [ 17.  29.  28.  11.  82.  17.  56. 620.  90.  50.]
 [ 10.  34.  96.  40.   7.  29.  44.   7. 712.  21.]
 [  8.  53.  86.   3.  57.  30.  20.  31.  40. 672.]]

Test set: Average loss: 1.0084, Accuracy: 6955/10000 (70%)
```

2. I tried different values for the number of hidden nodes and got corresponding accuracy as follows.[20(75%), 50(82%), 80(83%), 100(84%), 130(84%), 150(84%), 180(85%), 200(84%), 230(85%), 250(85%), 300(85%), 350(85%), 400(85%), 500(85%),800(85%)]. I could see that the accuracy is stable at around 85% when the number of hidden nodes is larger than 180. This is the final confusion matrix and final accuracy of the model when the number of hidden nodes is 180.

```
<class 'numpy.ndarray'>
[[856.   3.   2.   7.  28.  29.   3.  40.  28.   4.]
 [  5. 820.  31.   4.  18.   9.  54.   5.  23.  31.]
 [  6.   6. 849.  43.  14.  14.  22.  12.  21.  13.]
 [  3.   6.  32. 918.   3.  13.   4.   4.   7.  10.]
 [ 42.  35.  20.   7. 810.  10.  28.  17.  17.  14.]
 [  9.   9.  87.  11.  13. 833.  19.   2.  11.   6.]
 [  3.  13.  58.   8.  14.   8. 880.   7.   2.   7.]
 [ 15.  20.  20.   3.  26.  10.  29. 816.  27.  34.]
 [ 10.  25.  24.  50.   4.  11.  28.   3. 836.   9.]
 [  2.  16.  52.   4.  28.   5.  25.  12.  16. 840.]]

Test set: Average loss: 0.4978, Accuracy: 8458/10000 (85%)
```

3. This is the final confusion matrix and final accuracy of the model *NetConv*. I also tried many different values of parameters and discuss them in 4(c) in details.

```
<class 'numpy.ndarray'>
[[964.   4.   2.   2.  18.   2.   0.   4.   3.   1.]
 [  0. 927.   9.   1.  13.   2.  26.   4.   5.  13.]
 [ 10.   7. 876.  56.   4.   8.  20.   5.   7.   7.]
 [  2.   1.  13. 969.   2.   6.   5.   1.   1.   0.]
 [ 18.  14.   3.   6. 920.   3.  16.   7.   7.   6.]
 [  4.  10.  47.   6.   5. 892.  25.   4.   4.   3.]
 [  4.   5.  15.   2.   3.   1. 960.   4.   2.   4.]
 [  8.   9.   4.   1.   3.   0.   7. 950.   6.  12.]
 [  3.  19.   9.   5.   6.   3.   9.   3. 937.   6.]
 [  6.   8.  10.   1.   7.   0.   8.   6.   4. 950.]]

Test set: Average loss: 0.2554, Accuracy: 9345/10000 (93%)
```

4.

a. The accuracy of convolutional network is the highest. It could achieve 93% as above. The accuracy of a model which a linear function is low, which achieved only 70% accuracy. The accuracy of a fully connected 2-layer network is the middle number which is 85%. Therefore, the convolutional network should be more suitable than a simple linear function and a fully connected 2-layer network when processing the image task in this part.

b. From the three confusion matrixes for each model as above, I could conclude that which characters are more likely to be mistaken for which other characters as follows.

	0	1	2	3	4	5	6	7	8	9
NetLin	5	2	1	2	2	2	2	8	2	2
NetFull	7	6	3	2	0	2	2	9	3	2
NetConv	4	6	3	2	0	2	2	9	1	2

From the table as above, I could see that 1=”ki” is more likely to be mistaken for 6=”ma”, 2=”su” is more likely to be mistaken for 3=”tsu”, 3=”tsu” is more likely to be mistaken for 2=”su”, 4=”na” is more likely to be mistaken for 0=”o”, 5=”ha” is more likely to be mistaken for 2=”su”, 6=”ma” is more likely to be mistaken for 2=”su”, 7=”ya” is more likely to be mistaken for 9=”wo”, 9=”wo” is more likely to be mistaken for 2=”su”.

The reason is that these written characters are similar in some parts of pixels, so these networks could not distinguish them correctly.

c.

(1) This model doesn’t have any parameters to change except learning rate and momentum. Therefore, I tried different values for the learning rate in 0.01(default), 0.05, 0.1, 0.2. I found that when the learning rate equals to 0.01, the accuracy of this linear function could achieve 70%. However, other values achieved lower

accuracy which is around 67%. I also tried different momentum. But it did not have huge impact. In conclusion, I thinik that NetLin is limited by the architecture of itself instead of parameters so changing the parameters of training only has little effect.

(2) In this model, I also tried different value of the learning rate and the momentum except the number of hidden nodes which I said above. For the learning rate, I got that when the learning rate is 0.04 and the momentum is 0.05, the accuracy is 89%. When the momentum is 0.9 and learning rate is 0.01, the accuracy is 88%.

(3) At the beginning, I used the model as follows. However, the accuracy is only 92%.

```
NetConv(
  (conv2d1): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU(inplace=True)
  )
  (conv2d2): Sequential(
    (0): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU(inplace=True)
  )
  (linear1): Sequential(
    (0): Linear(in_features=9216, out_features=625, bias=True)
    (1): ReLU(inplace=True)
  )
  (output): Linear(in_features=625, out_features=10, bias=True)
)
```

Then, I tried to increase the kernel size of the convolution operation, but the accuracy is still 92%. Therefore, I tried to change the architecture of the network by adding max pool layer as follows. The accuracy became 93%.

```
NetConv(
  (conv2d1): Sequential(
    (0): Conv2d(1, 24, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU(inplace=True)
  )
  (maxpool1): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2d2): Sequential(
    (0): Conv2d(24, 48, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU(inplace=True)
  )
  (maxpool2): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (linear1): Sequential(
    (0): Linear(in_features=1200, out_features=625, bias=True)
    (1): ReLU(inplace=True)
  )
  (output): Linear(in_features=625, out_features=10, bias=True)
)
```

Next, I wanted to increase the accuracy by increasing the kernel size of the max pool layer from 2 to 3, but accuracy became 91%, which is less than ideal.

Then, I tried to increase the kernel size of the convolution layer from 3 to 5 and the model is as follows. The accuracy became 94%.

```
NetConv(
  (conv2d1): Sequential(
    (0): Conv2d(1, 24, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace=True)
  )
  (maxpool1): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2d2): Sequential(
    (0): Conv2d(24, 48, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU(inplace=True)
  )
  (maxpool2): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (linear1): Sequential(
    (0): Linear(in_features=768, out_features=625, bias=True)
    (1): ReLU(inplace=True)
  )
  (output): Linear(in_features=625, out_features=10, bias=True)
)
```

I also tried to use padding(2*2). The accuracy increased to 95%. The architecture is as follows.

```
NetConv(
  (conv2d1): Sequential(
    (0): Conv2d(1, 24, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU(inplace=True)
  )
  (maxpool1): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2d2): Sequential(
    (0): Conv2d(24, 48, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU(inplace=True)
  )
  (maxpool2): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (linear1): Sequential(
    (0): Linear(in_features=2352, out_features=625, bias=True)
    (1): ReLU(inplace=True)
  )
  (output): Linear(in_features=625, out_features=10, bias=True)
)
```

Except the parameters of the model, I also thought that the learning rate and momentum may could increase the accuracy. So, I tried different values of learning rate in [0.02, 0.05, 0.1, 0.2, 0.3]. I found that when the learning rate is 0.1, the accuracy is 96%. Then, I tried to change the momentum from 0.1 to 0.99. When the momentum is 0.7, the accuracy is 97%. It is better than the previous cases. The details are as follows.

```
<class 'numpy.ndarray'>
[[972.   5.   2.   0.  12.   1.   0.   4.   3.   1.]
 [  2. 955.   8.   0.   3.   1.  22.   1.   7.   1.]
 [ 11.   1. 939.  19.   6.   6.  11.   3.   3.   1.]
 [  1.   1.   6. 986.   0.   0.   2.   1.   3.   0.]
 [ 14.   5.   2.   8. 948.   3.   3.   5.  11.   1.]
 [  1.   1.  19.   5.   2. 955.   6.   2.   4.   5.]
 [  2.   1.   6.   3.   4.   2. 982.   0.   0.   0.]
 [ 10.   3.   3.   1.   0.   0.   4. 972.   2.   5.]
 [  2.   2.   4.   1.   2.   1.   1.   1. 986.   0.]
 [  6.   5.   3.   2.   4.   0.   3.   1.  12. 964.]]

Test set: Average loss: 0.2308, Accuracy: 9659/10000 (97%)
```

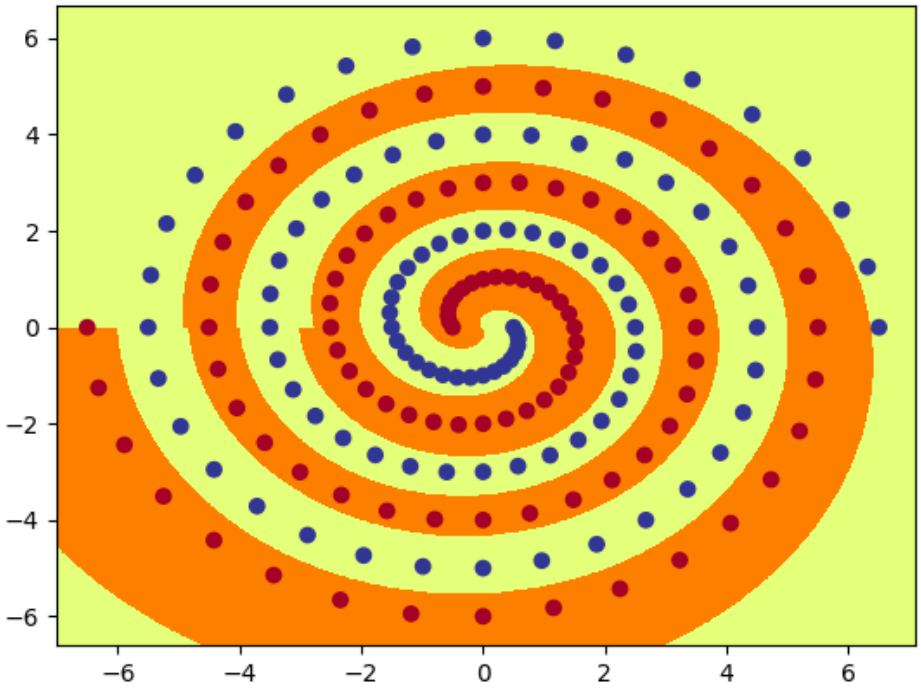
In conclusion, I want to say that changing architecture and the value of different parameters of the model is very import if we want to get better result. Some parameters are more important than the others, such as the kernel size of the convolution layer, the learning rate and the max pooling layer. The learning rate will have a huge impact on training, especially when the number of epochs is small. Therefore, I think that when we try to fit a model, we need choose the important parameters firstly and change others slightly afterwards. But we still need to determine how to choose the best values of these parameters basing the tasks.

Part 2: Twin Spirals Task

1. The codes are showed in the spiral.py
2. I tried to decrease the value of the number of hidden nodes from 10. The details are as follows.

The number of Hidden nodes	Details
10	ep: 3400 loss: 0.0072 acc: 100.00
9	ep: 2700 loss: 0.0245 acc: 100.00
8	ep: 2500 loss: 0.0226 acc: 100.00
7	ep:10800 loss: 0.0599 acc: 100.00
6	ep: 6700 loss: 0.0366 acc: 100.00(Sometimes)
5	ep:99900 loss: 0.0508 acc: 91.24

In the table, I could see that when the number of hidden nodes is 6, it could achieve 100% accuracy. However, when I tried it again, I found that it could not reach 100% accuracy sometimes. Therefore, the minimum number of hidden nodes if 7.
The polar_out.png is as follows.



3. The codes are showed in the spiral.py
4. I tried different combinations of the value of the number of hidden nodes in [10, 9, 8, 7, 6, 5] and the initial weight in [0.001, 0.01, 0.1, 0.2, 0.3] The details are as follows.

The number of Hidden nodes	Init weight	Details (The best result)
10	0.2	ep: 5500 loss: 0.0948 acc: 100.00
9	0.1	ep: 9300 loss: 0.0407 acc: 100.00
8		could not reach 100% with 20000 epochs
7		could not reach 100% with 20000 epochs
6		could not reach 100% with 20000 epochs
5		could not reach 100% with 20000 epochs

When the number of hidden nodes is 10, if the initial weight is 0.1, 0.2 and 0.3, it could achieve 100% accuracy. But if I set 0.1 or 0.3, it could not achieve 100% accuracy within 20000 epochs. When the number of hidden nodes is 9, if the initial weight is 0.1, it could achieve 100% accuracy before 200000 epochs. When the number of hidden nodes is 8, it could not achieve 100% accuracy within 2000 epochs when the initial weight is in [0.001, 0.01, 0.1, 0.2, 0.3]. When the number of hidden nodes is 7, it could not achieve 100% accuracy within 2000 epochs when the initial weight is in [0.001, 0.01, 0.1, 0.2, 0.3]. When the number of hidden nodes

is 6, it could not achieve 100% accuracy within 2000 epochs when the initial weight is in [0.001, 0.01, 0.1, 0.2, 0.3]. When the number of hidden nodes is 5, it could not achieve 100% accuracy within 2000 epochs when the initial weight is in [0.001, 0.01, 0.1, 0.2, 0.3].

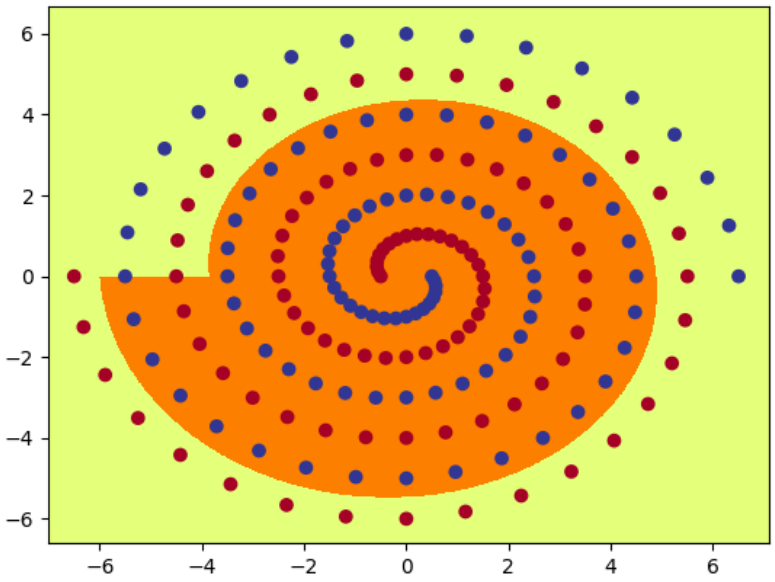
5.

(1) PolarNet

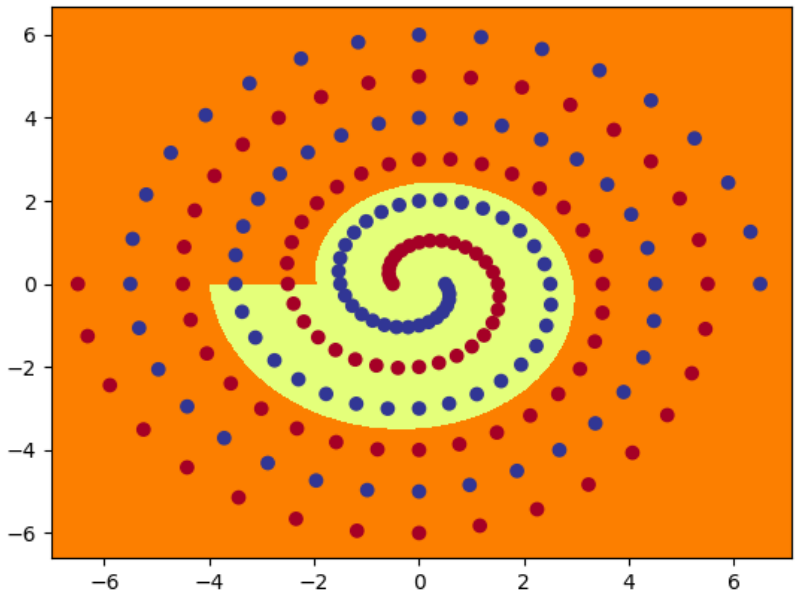
python3 spiral_main.py --net polar --hid 7

ep:10200 loss: 0.0523 acc: 100.00

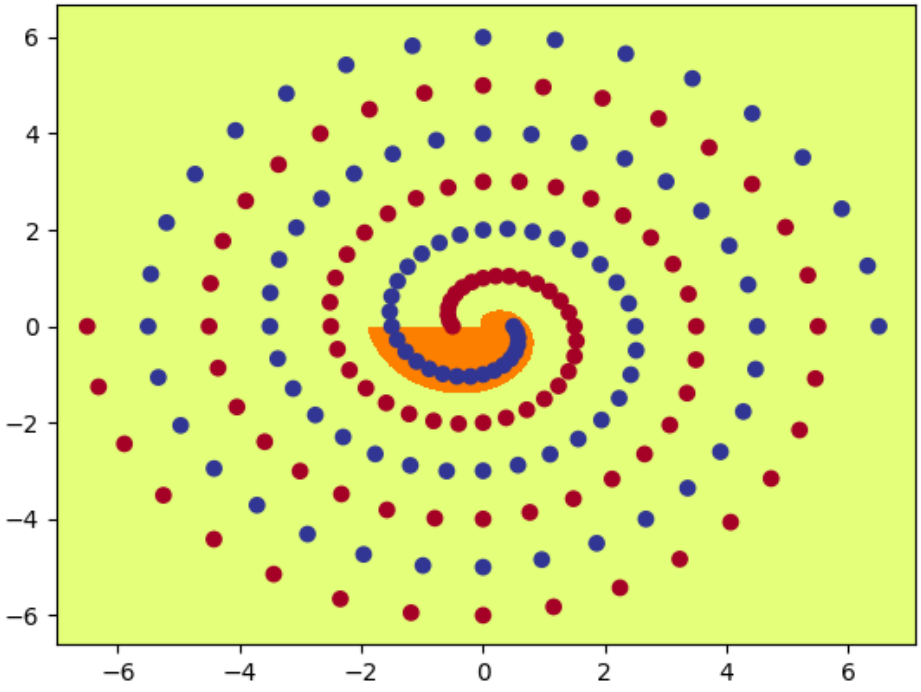
polar1_0.png



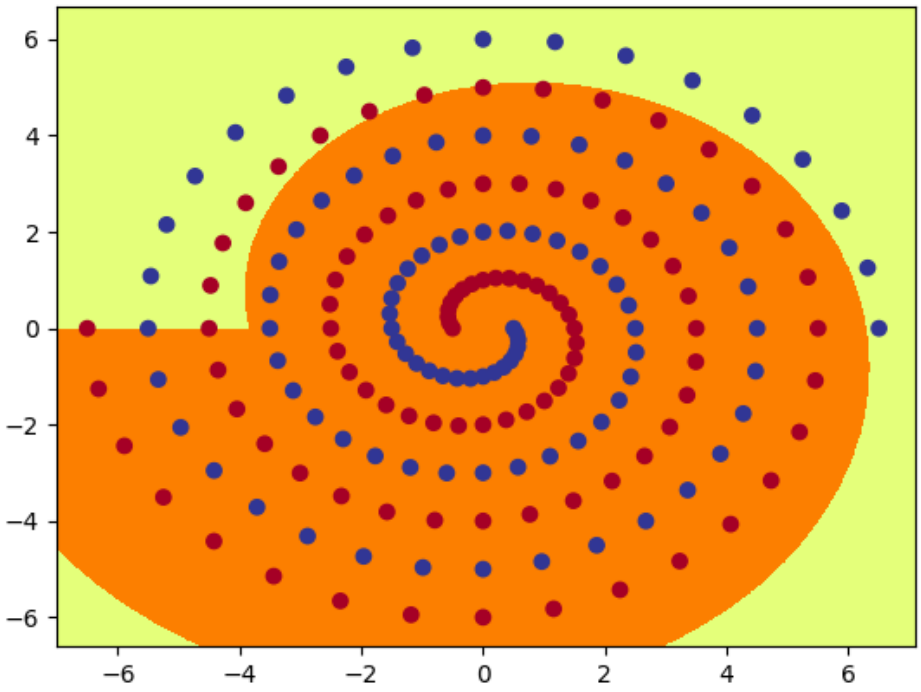
polar1_1.png



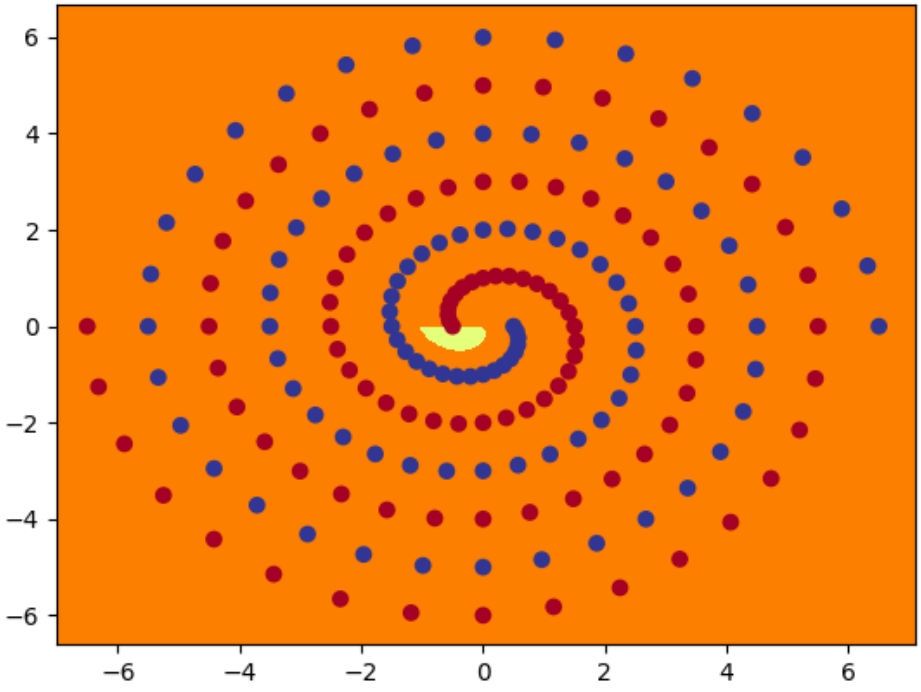
polar1_2.png



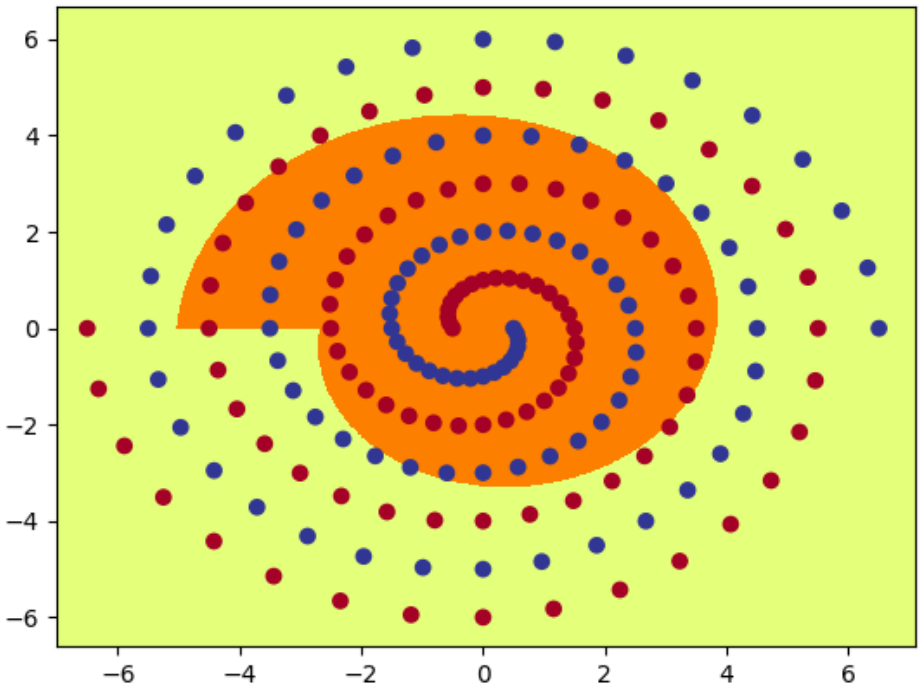
polar1_3.png



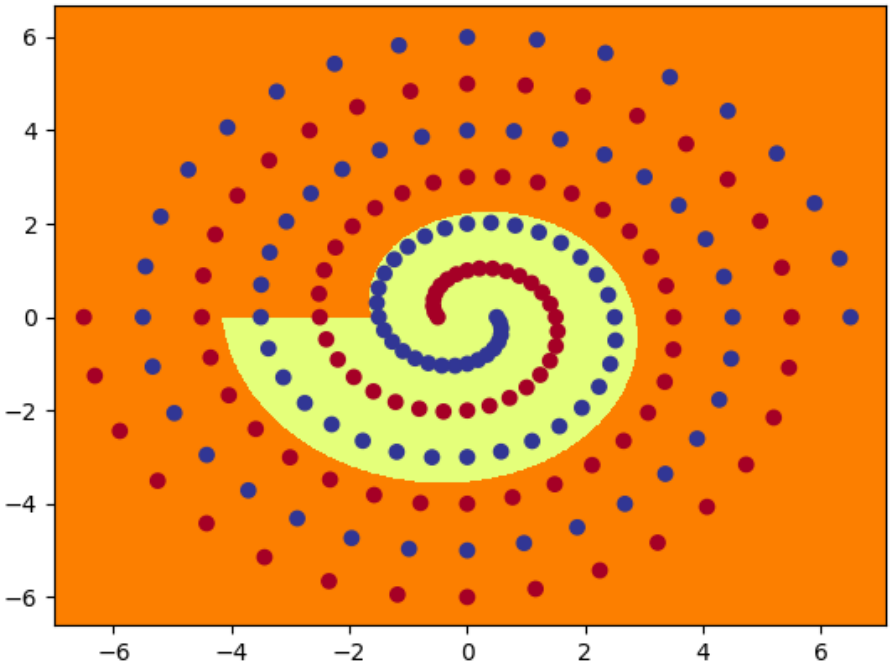
polar1_4.png



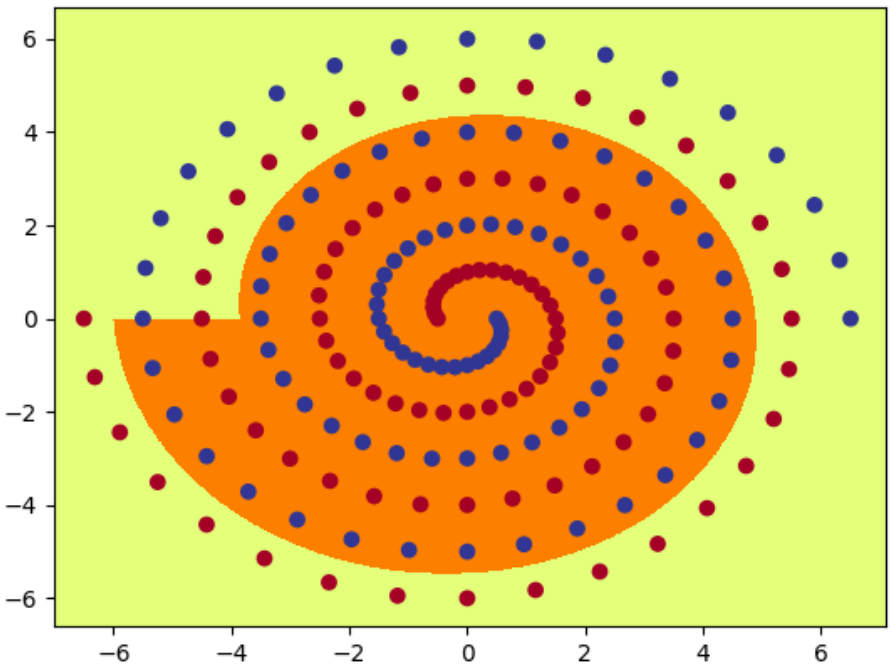
polar1_5.png



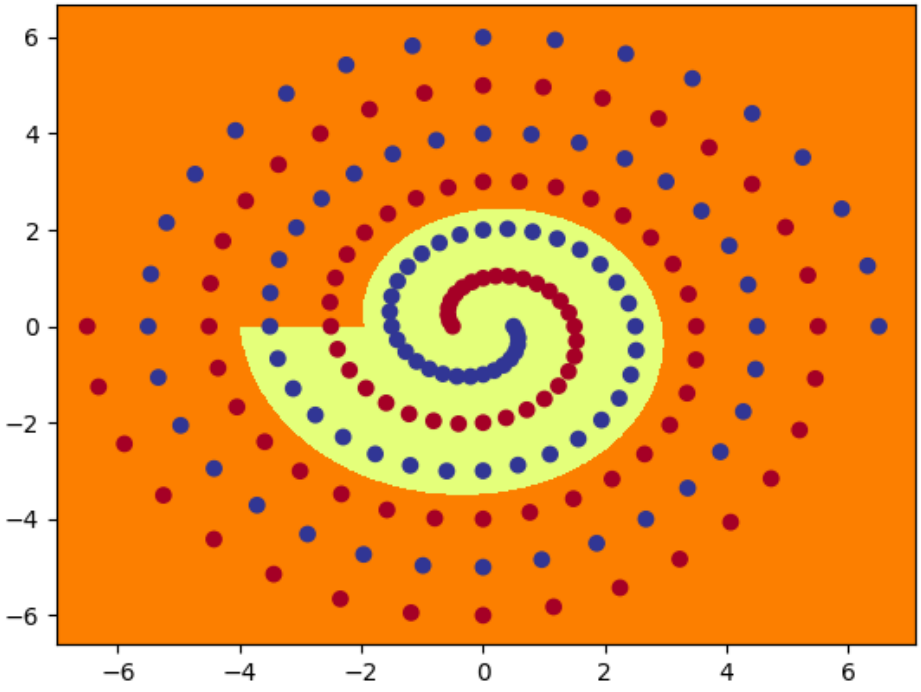
polar1_6.png



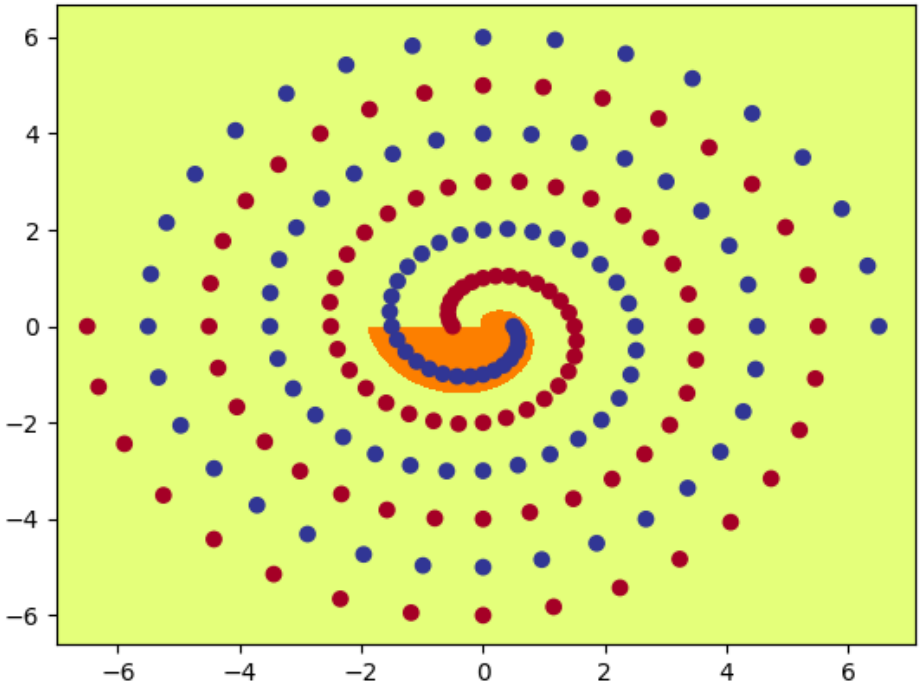
(2)
python3 spiral_main.py --net raw --hid 10 --init 0.2
ep:10200 loss: 0.0523 acc: 100.00
polar1_0.png



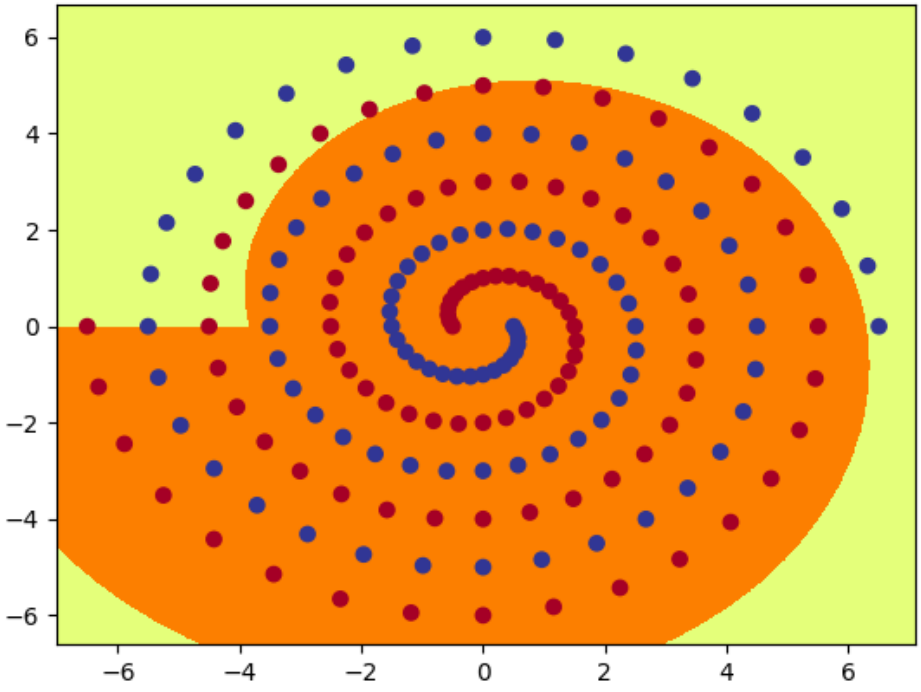
polar1_1.png



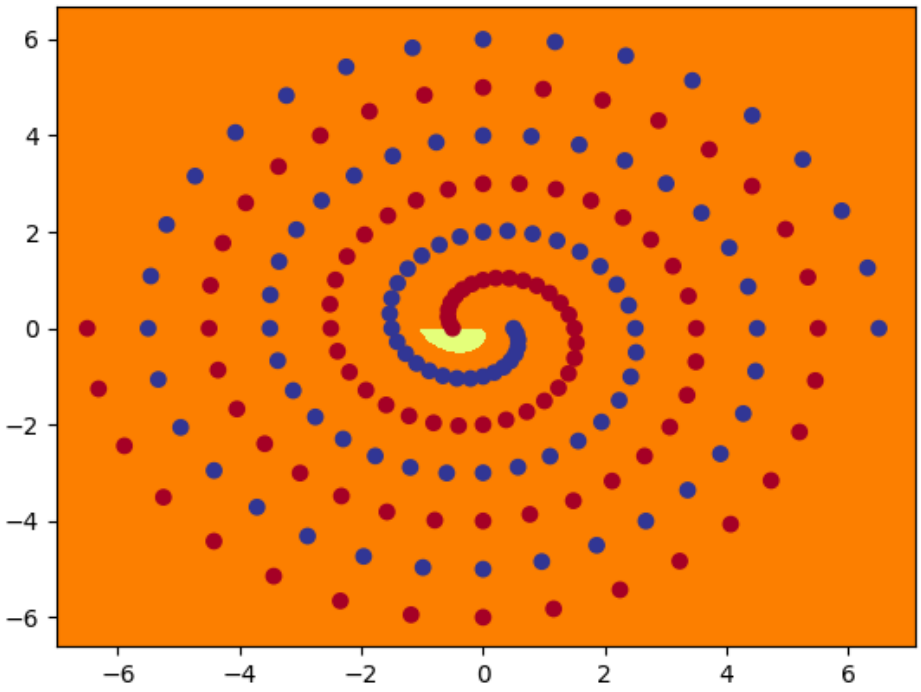
polar1_2.png



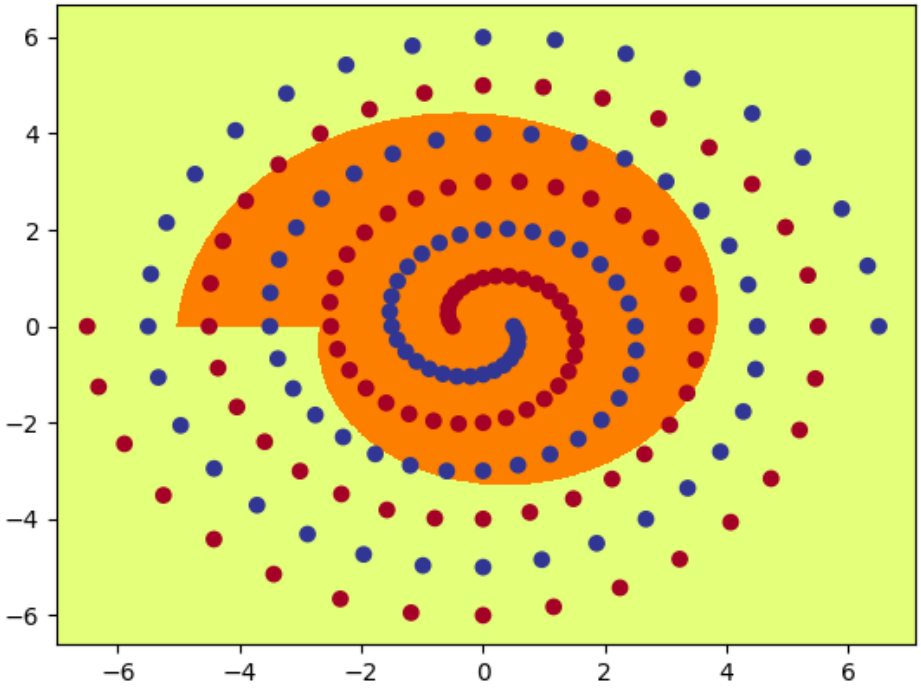
polar1_3.png



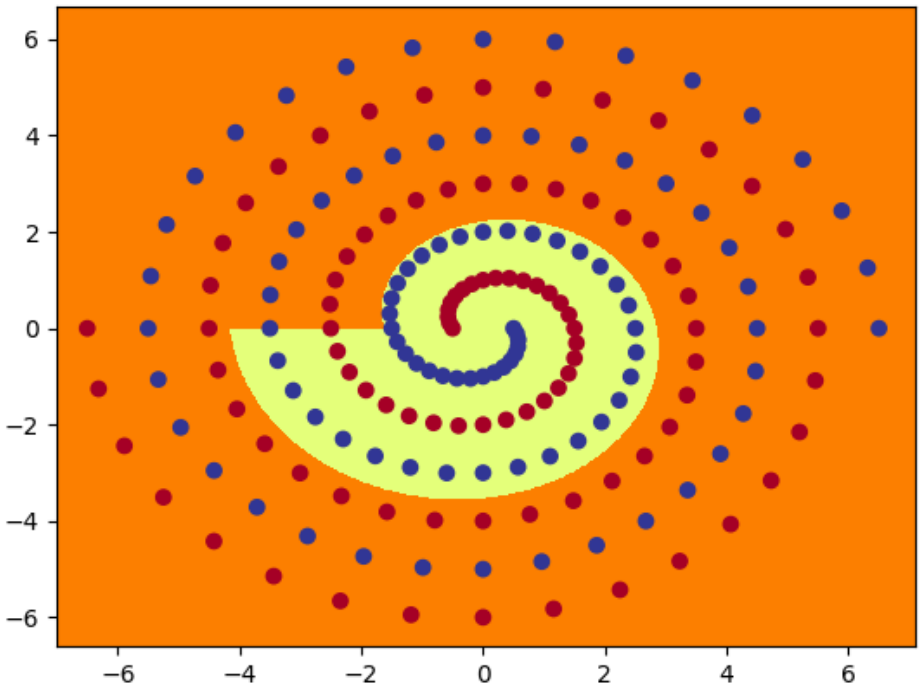
polar1_4.png



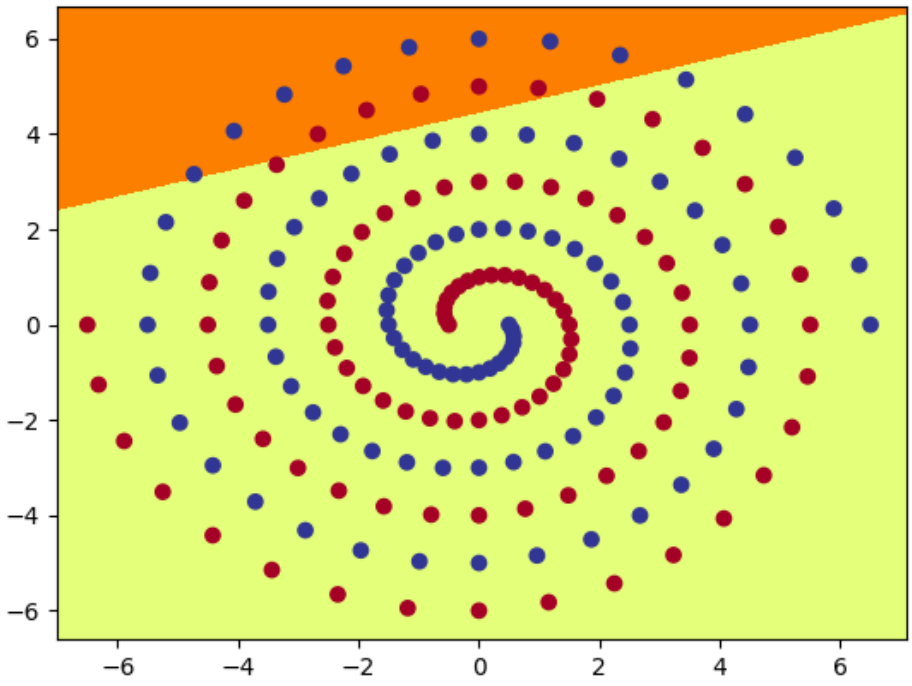
polar1_5.png



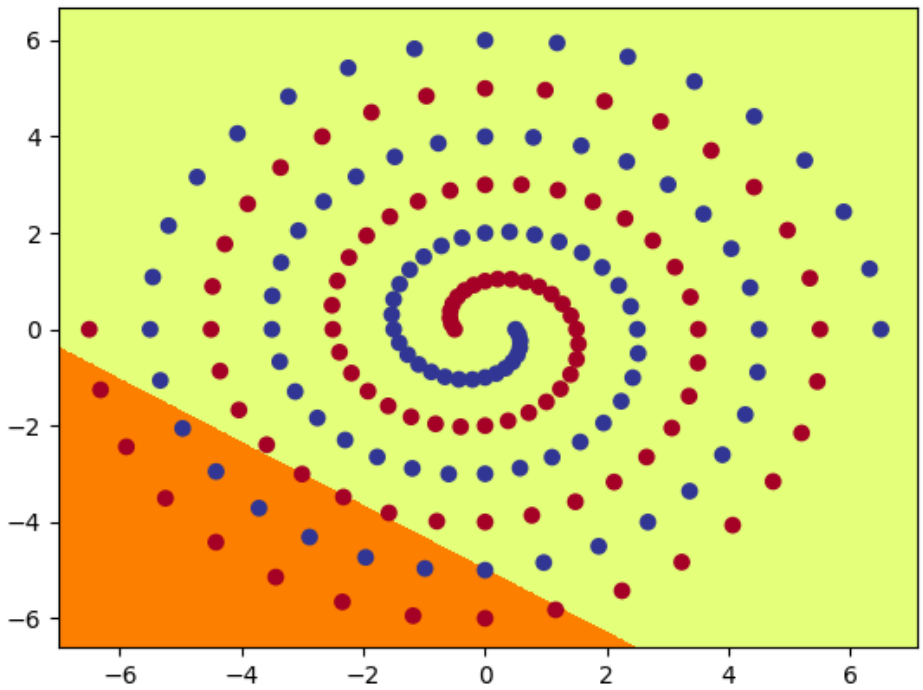
polar1_6.png



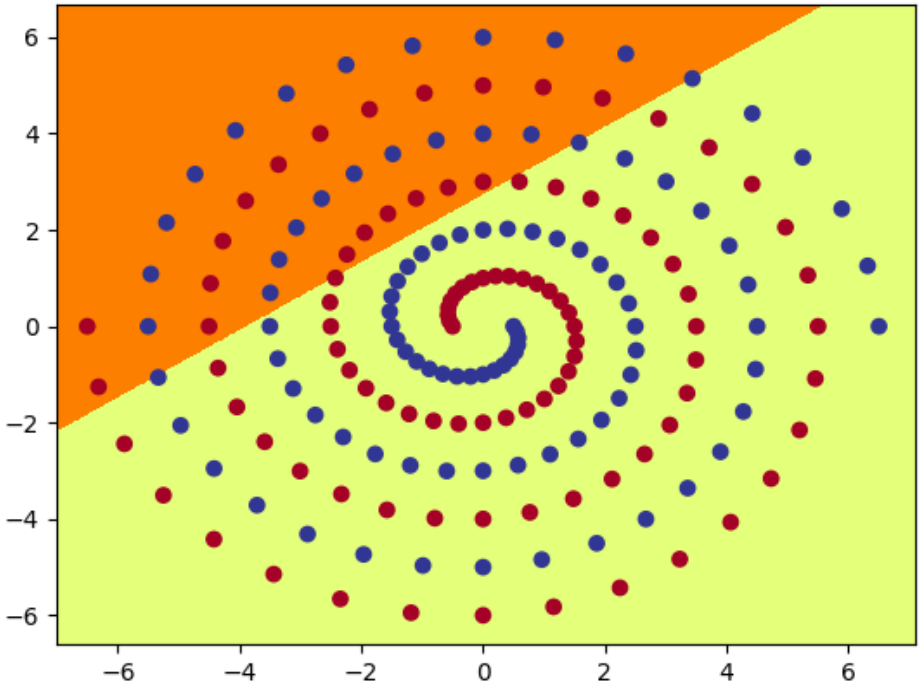
python spiral_main.py --net raw --hid 10 --init 0.2
ep:18800 loss: 0.0199 acc: 100.00
raw1_0.png



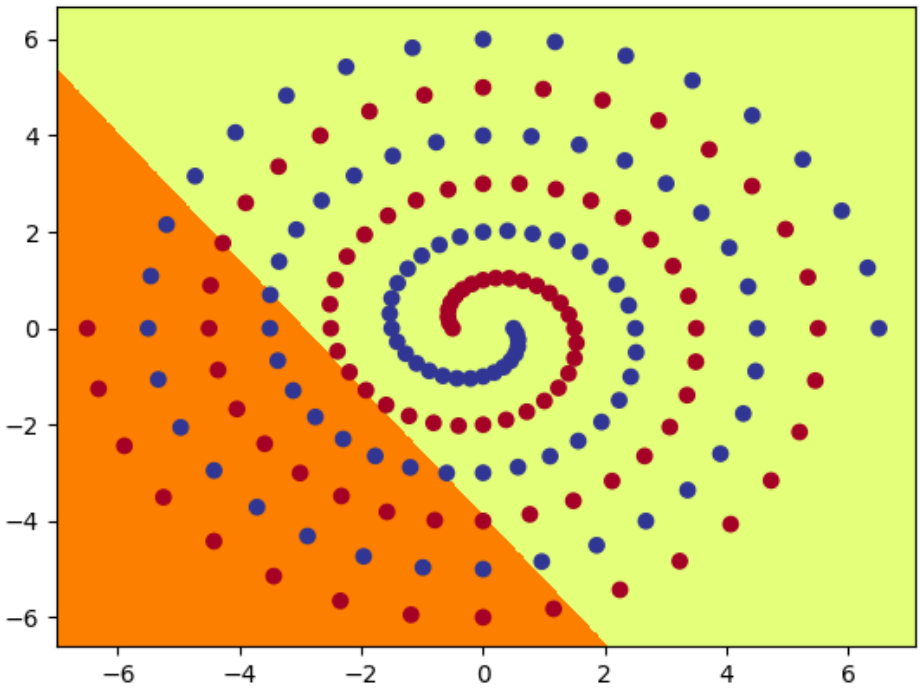
raw1_1.png



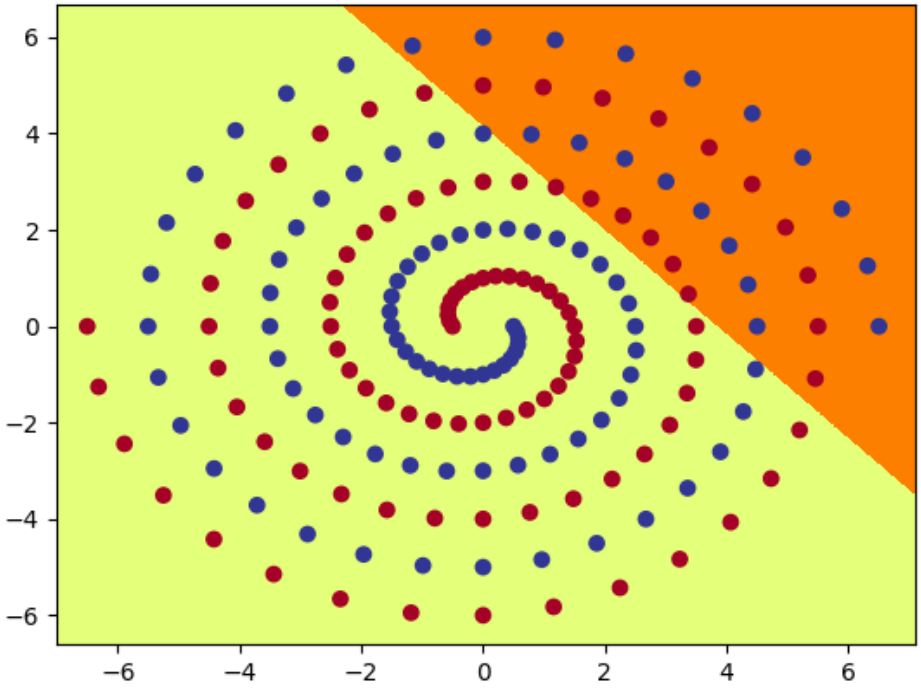
raw1_2.png



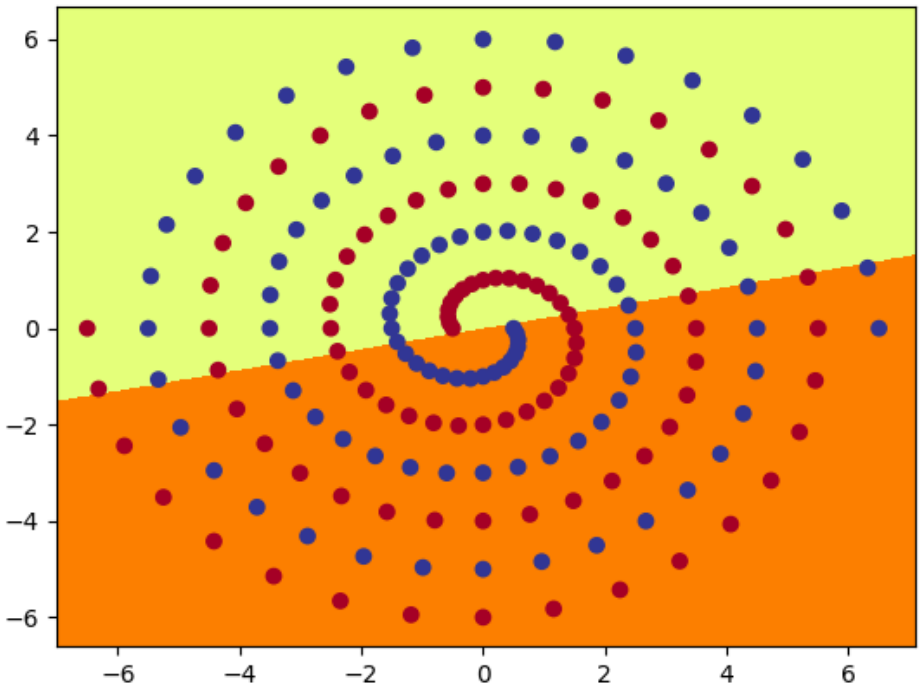
raw1_3.png



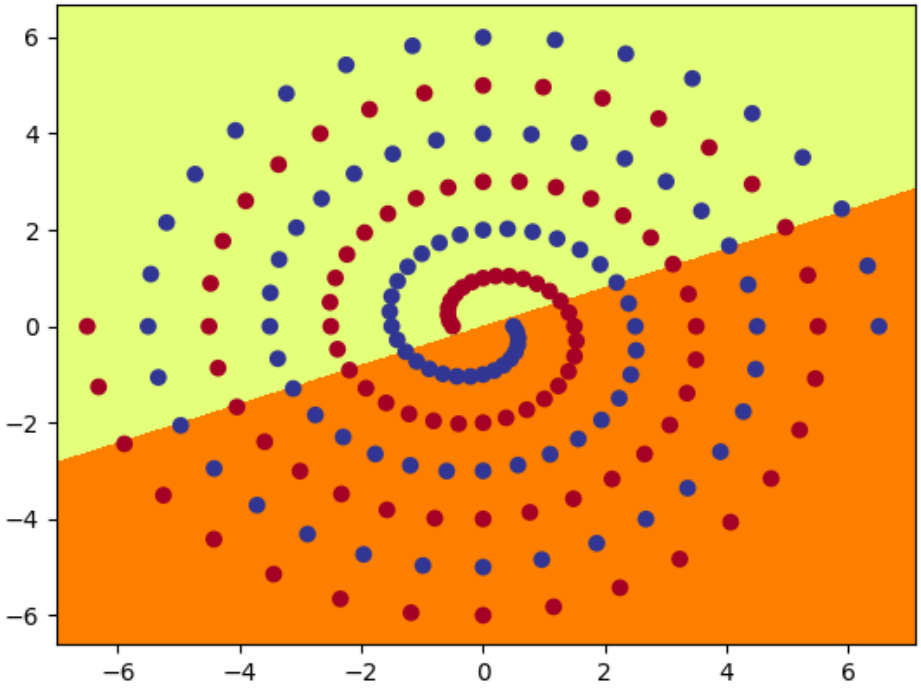
raw1_4.png



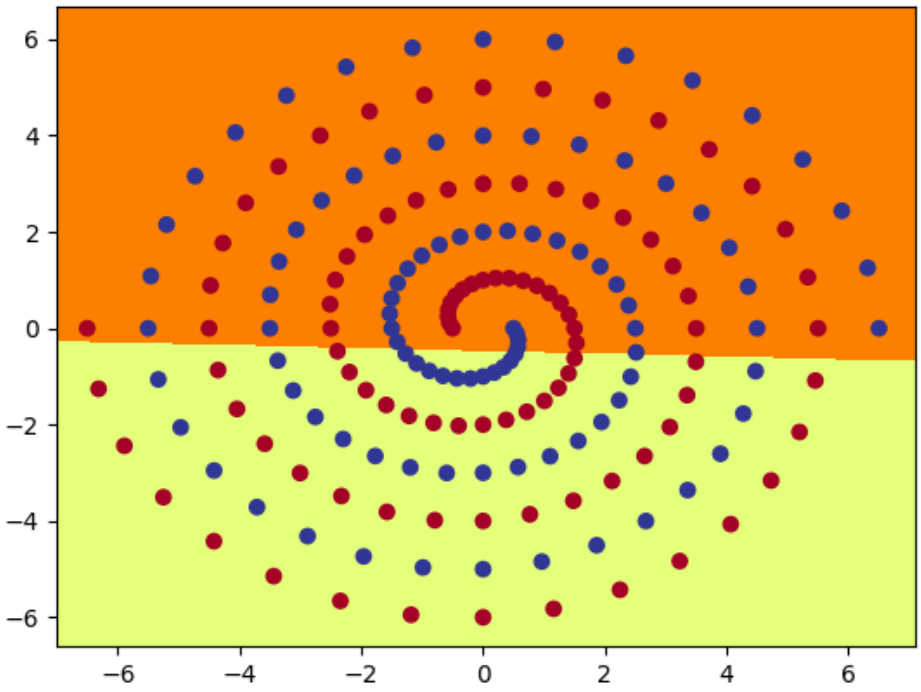
raw1_5.png



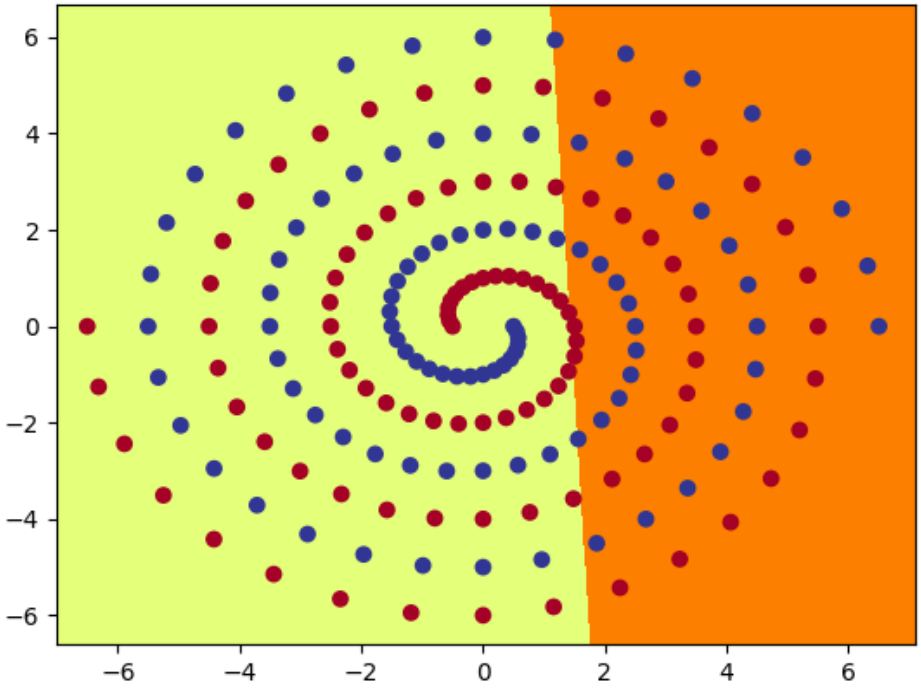
raw1_6.png



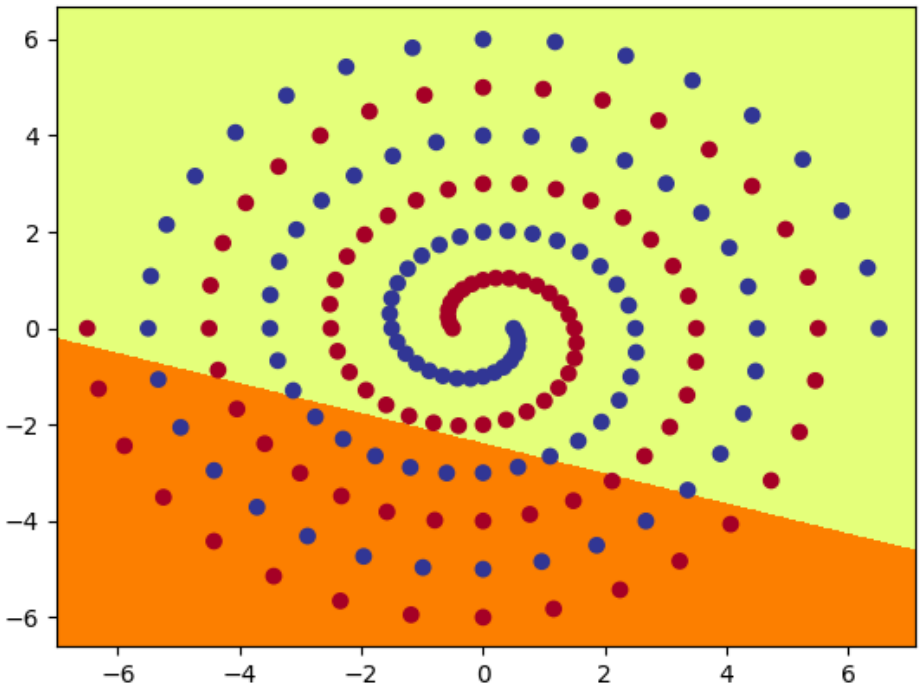
raw1_7.png



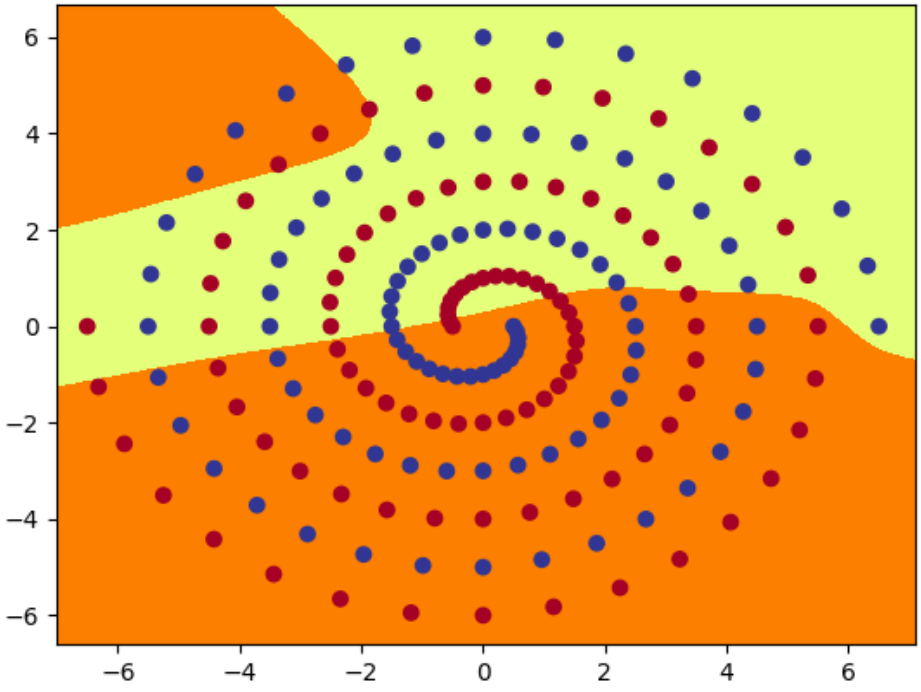
raw1_8.png



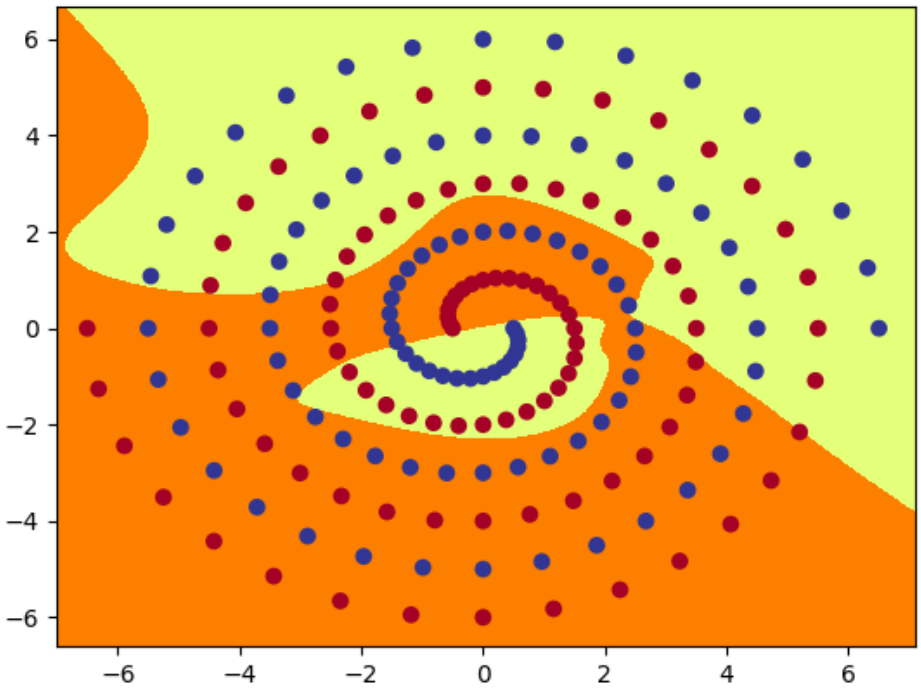
raw1_9.png



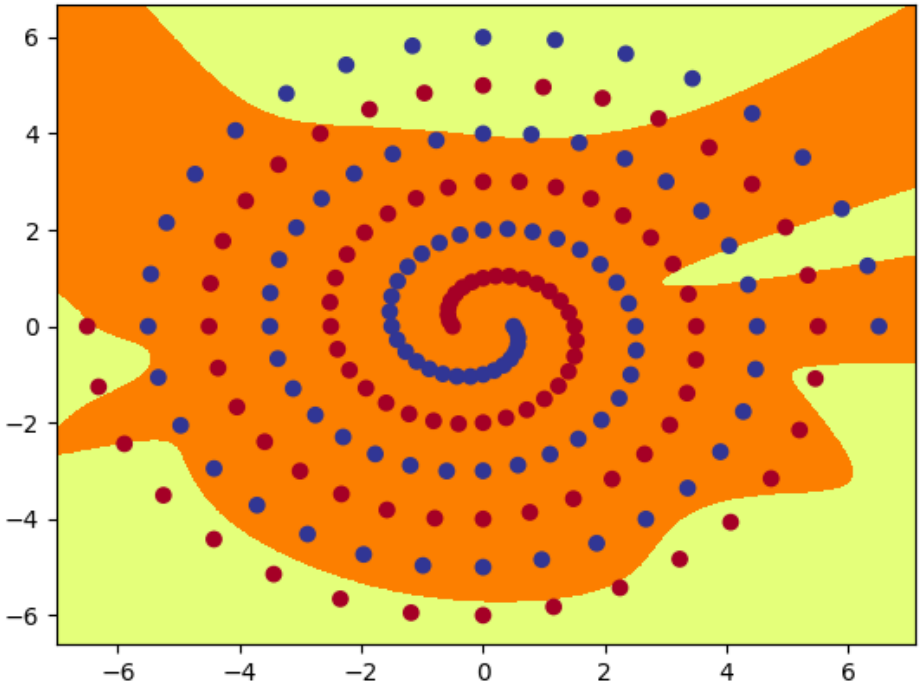
raw2_0.png



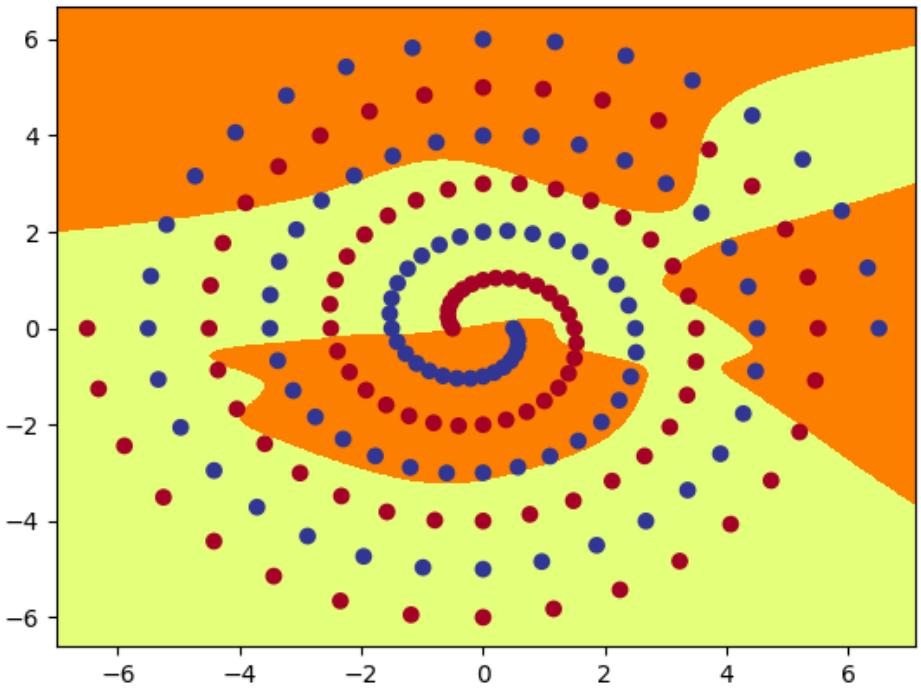
raw2_1.png



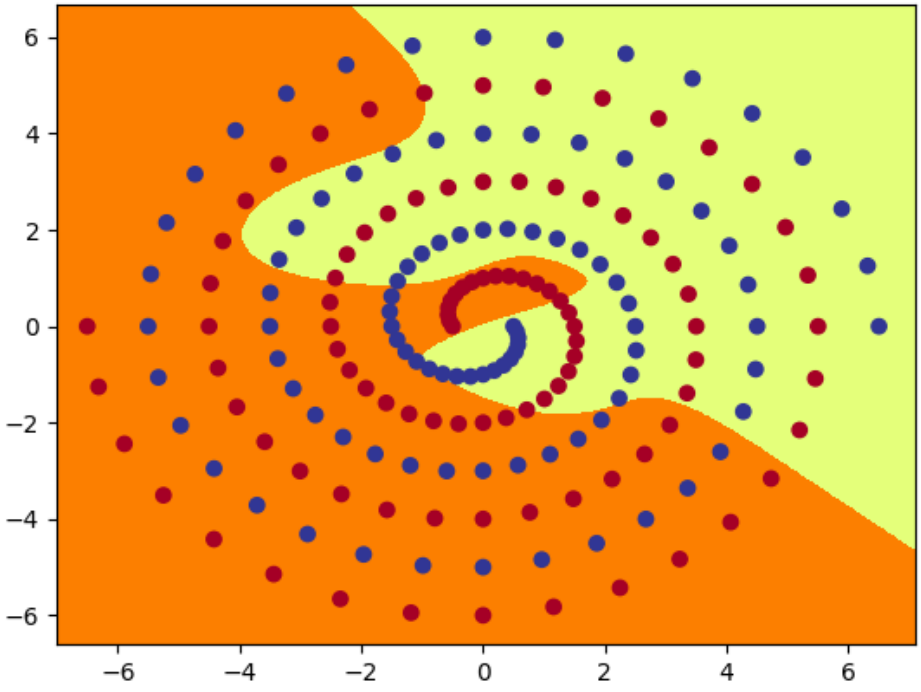
raw2_2.png



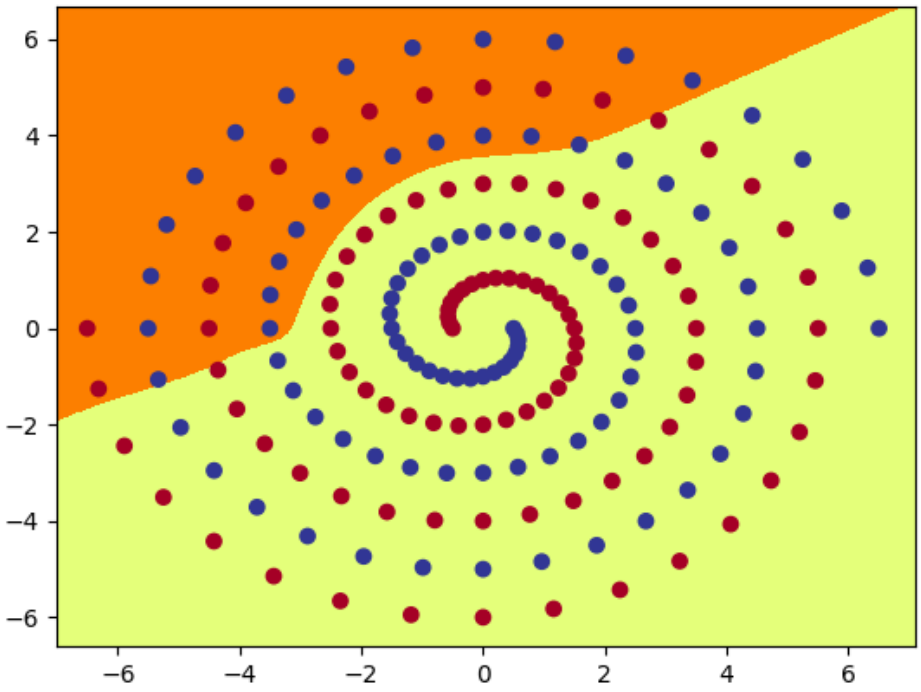
raw2_3.png



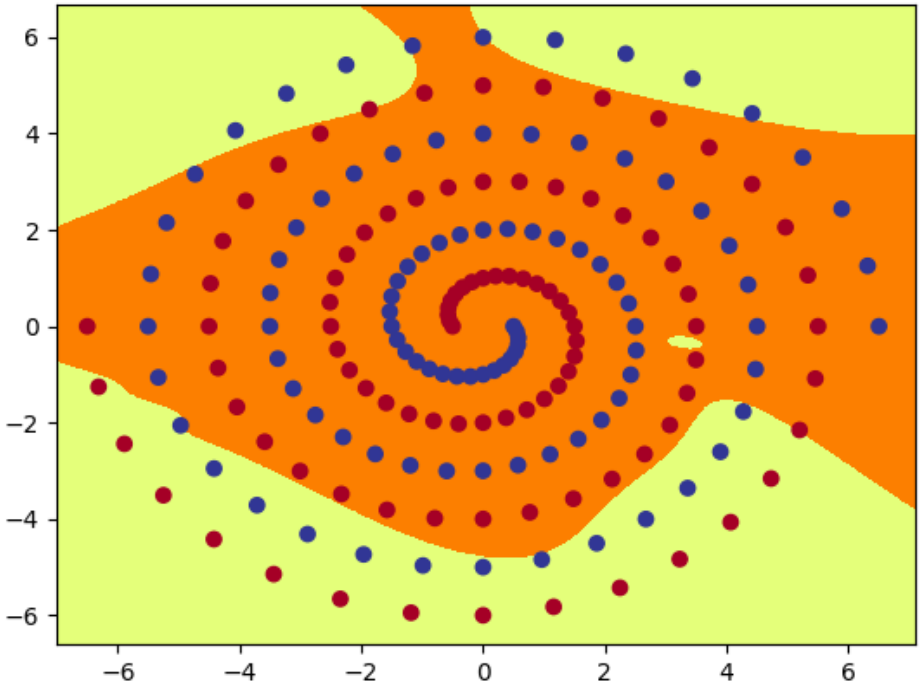
raw2_4.png



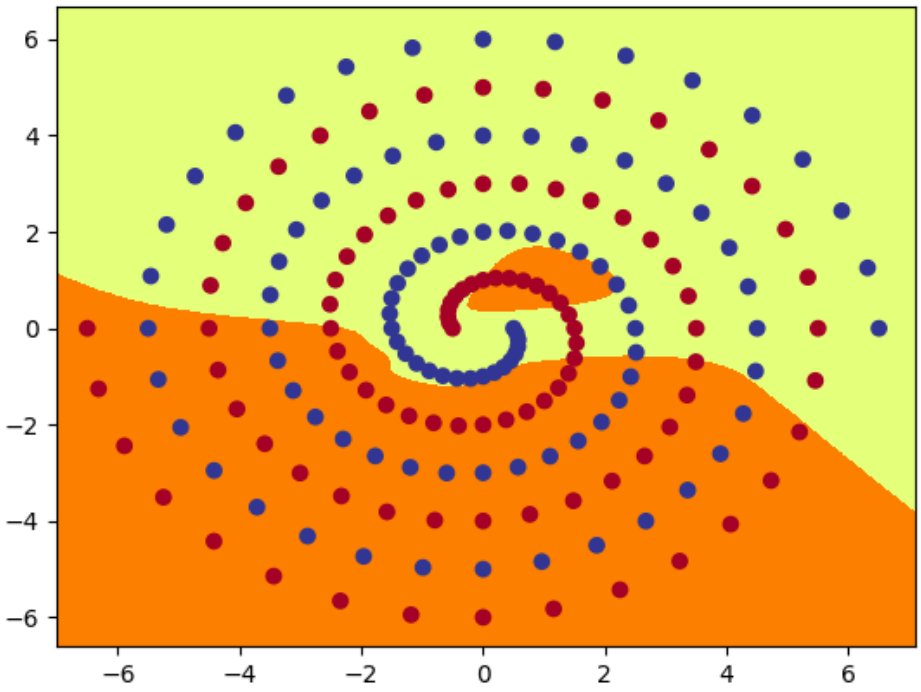
raw2_5.png



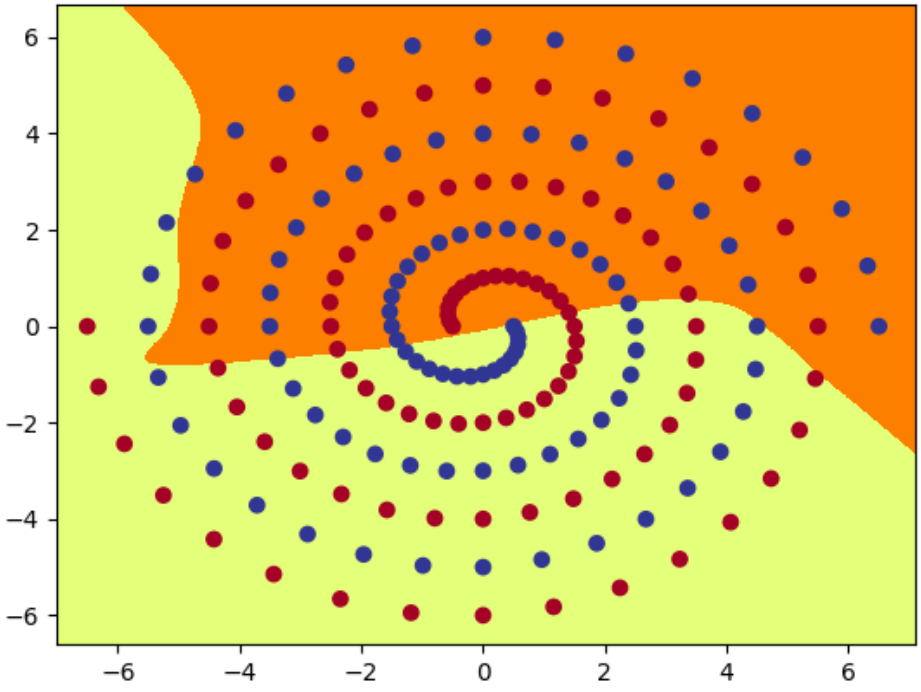
raw2_6.png



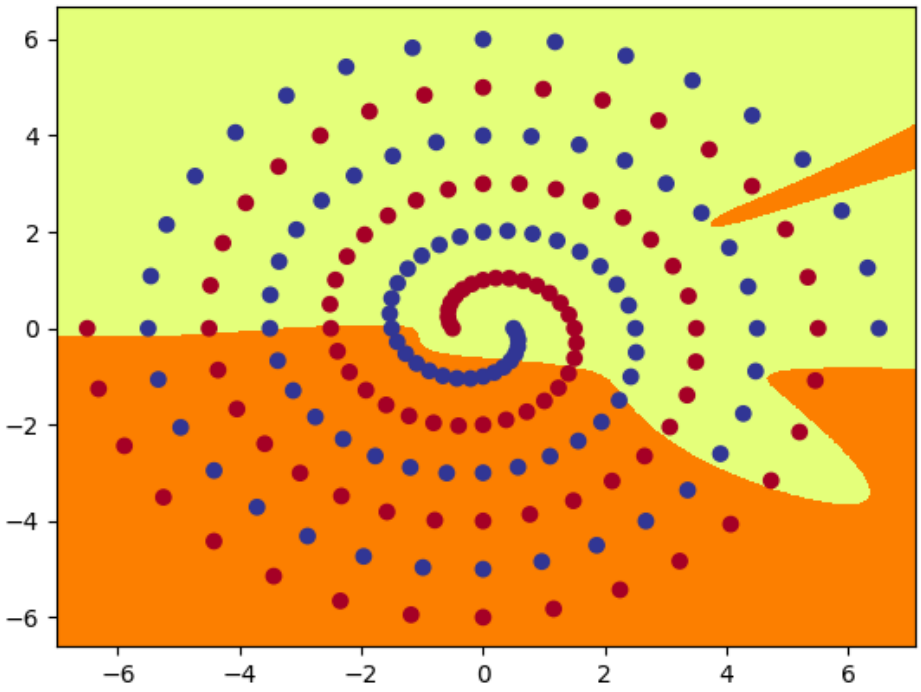
raw2_7.png



raw2_8.png



raw2_9.png



6.

(a) For the PolarNet, I could see that the function generated non-linear features in the hidden layer from the pictures of the hidden layer in question 5. For the RawNet, I could see that the function generated linear features in the first hidden layer and generated non-linear features in the second hidden layer from the pictures in question 5. From these pictures, we also could get that every hidden node could only get a part of information of the classification boundary. Maybe, because the values of the hidden nodes multiply the weights plus the bias and then using the activation function is the result of the function. Therefore, the net need to use all the hidden nodes and the weights to achieve the classification.

(b) In this part, I tried different values of initial weights in [0.001, 0.01, 0.1, 0.2, 0.3] in RawNet, which is recommended by teacher. I found that if the value of the initial weight is 0.1 or 0.2, it is more likely to succeed. However, when I used 0.001 and 0.1, it always ended in failure. When I used 0.3, it could succeed sometimes. Besides, talking about the speed, in the question 2, I tried to set the initial weight as 0.1 and 0.2 to compare the speed when the number of hidden nodes is 10, I got the result that if the weight is 0.2, it used 5900 epochs to achieve 100% accuracy. However, when the initial weight is 0.1, it used more than 20000 epochs to achieve 100% accuracy sometimes. Therefore, I want to conclude that if the number of the initial weight is too larger or too small, it is very hard to reach the best point. So, we need to consider and try to find the most suitable value of the initial weight in order to get good results basing on different conditions of the task.

(c)

(1) Changing batch size. I tried to change the batch size from 97 to 194.

In the PolarNet, it could achieve 100% accuracy in 3400 epochs if the number of hidden nodes is 10 and the batch size is 97. However, if I set the batch is 194, the number of epochs is only 900.
ep: 900 loss: 0.1756 acc: 100.00

In the RawNet, it could achieve 100% accuracy in 3000 epochs sometimes. However, it also could not achieve 100% sometimes.

Therefore, I conclude that if the batch size is larger, the speed of learning of some tasks is faster in some tasks.

(2) I tried the SGD optimizer instead of Adam optimizer.
Optimizer = torch.optim.SGD(net.parameters(), lr=args.lr, momentum=0.9, weight_decay=0.0001)

It does not perform well as Adam optimizer.
ep:12900 loss: 0.0130 acc: 100.00

I think the reason is that Adam is an extended SGD optimizer. Adam optimizer uses momentum and adaptive learning rate to speed up. Therefore, Adam optimizer is

better than SGD optimizer in general.

(3) I tried to change the activation from tanh to relu.
self.hidden_layer_1 = nn.functional.relu(self.linear1(input))

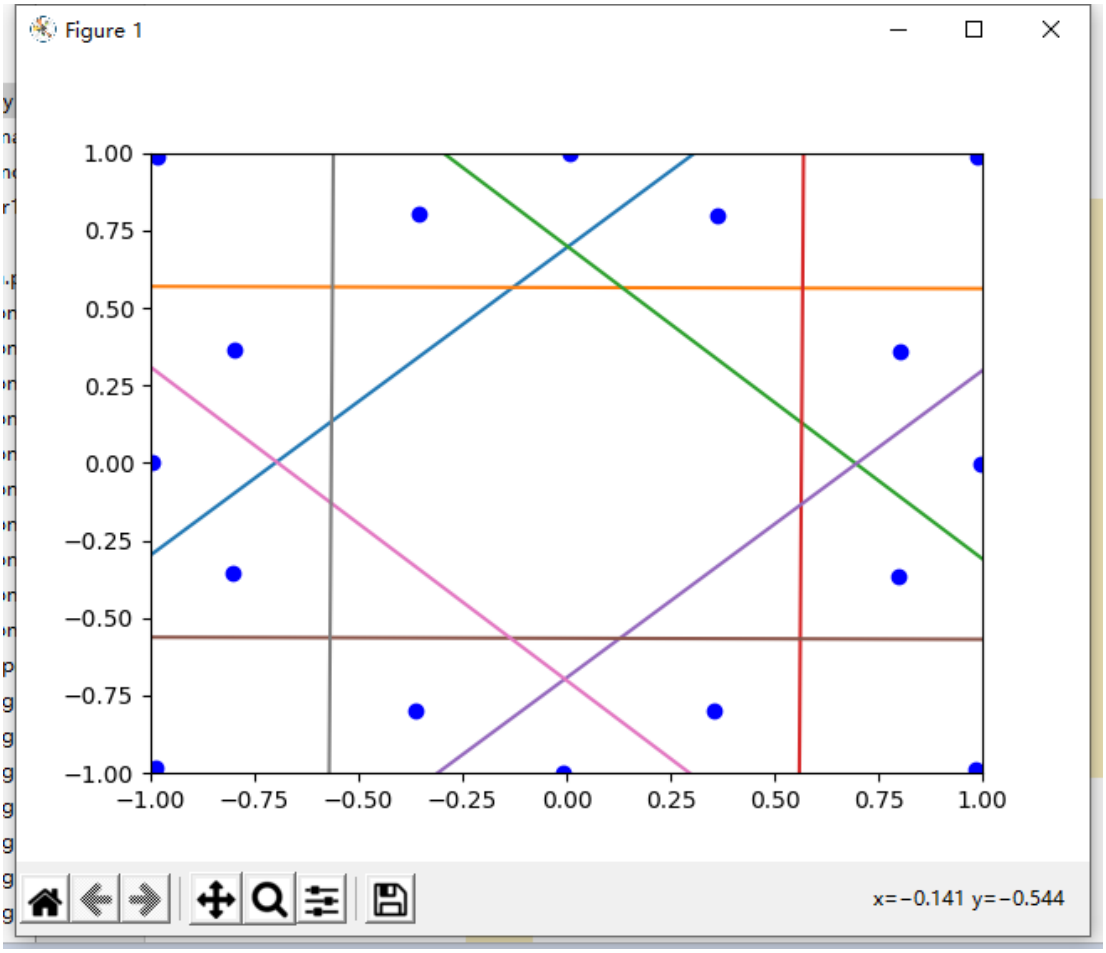
However, the model performed worse. I think the reason is that when the x is smaller than 0, the output is 0. So, in the learning process, zero output will make the gradient become zero. It is difficult for the model to learn.

(4) I tried to add the third hidden layer in the RawNet. It also has a good performance.
ep: 5700 loss: 0.0063 acc: 100.00

I think that the hidden layer is useful to fit data to some extent. If the model has more hidden layer, the performance of representing data is better and the data is more abstract. So, it could perform better in the training set. However, if the number of hidden layers is too lager, it will over-fitting and does not perform well in testing set.

Part 3: Hidden Unit Dynamics

1.



2.

(1)

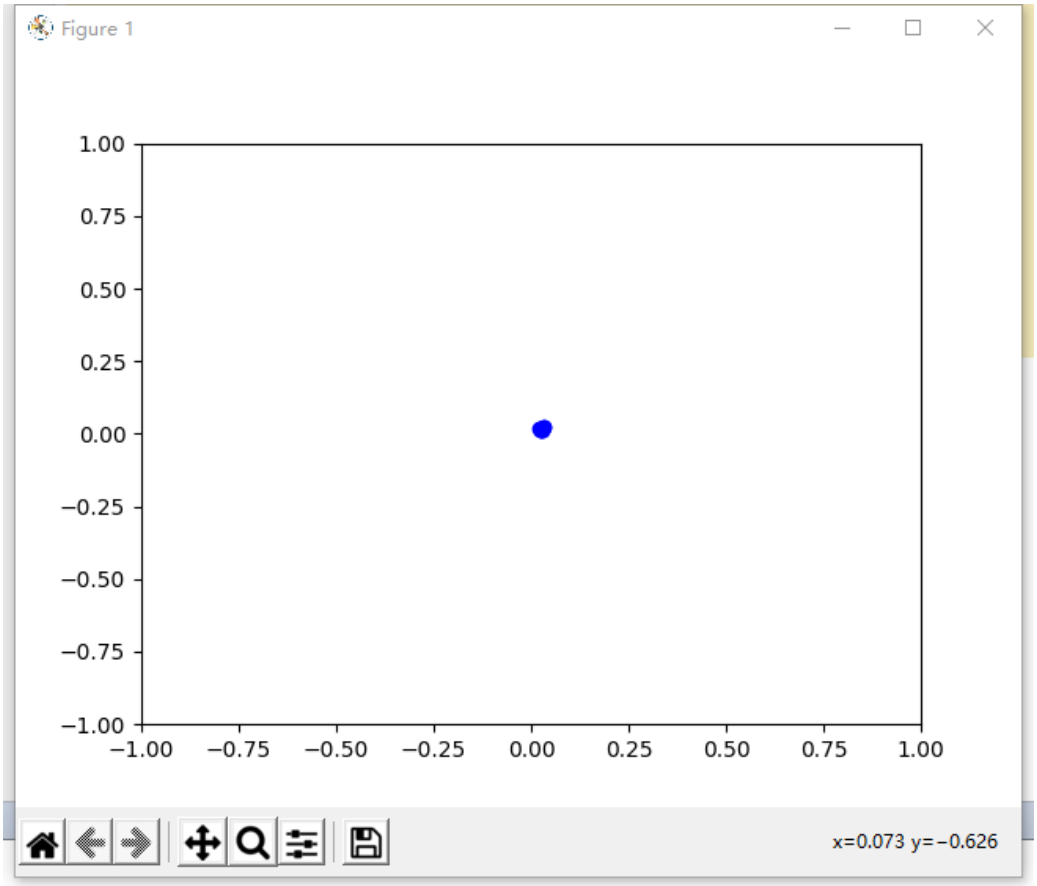


Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x=0.073 y=−0.626

(2)

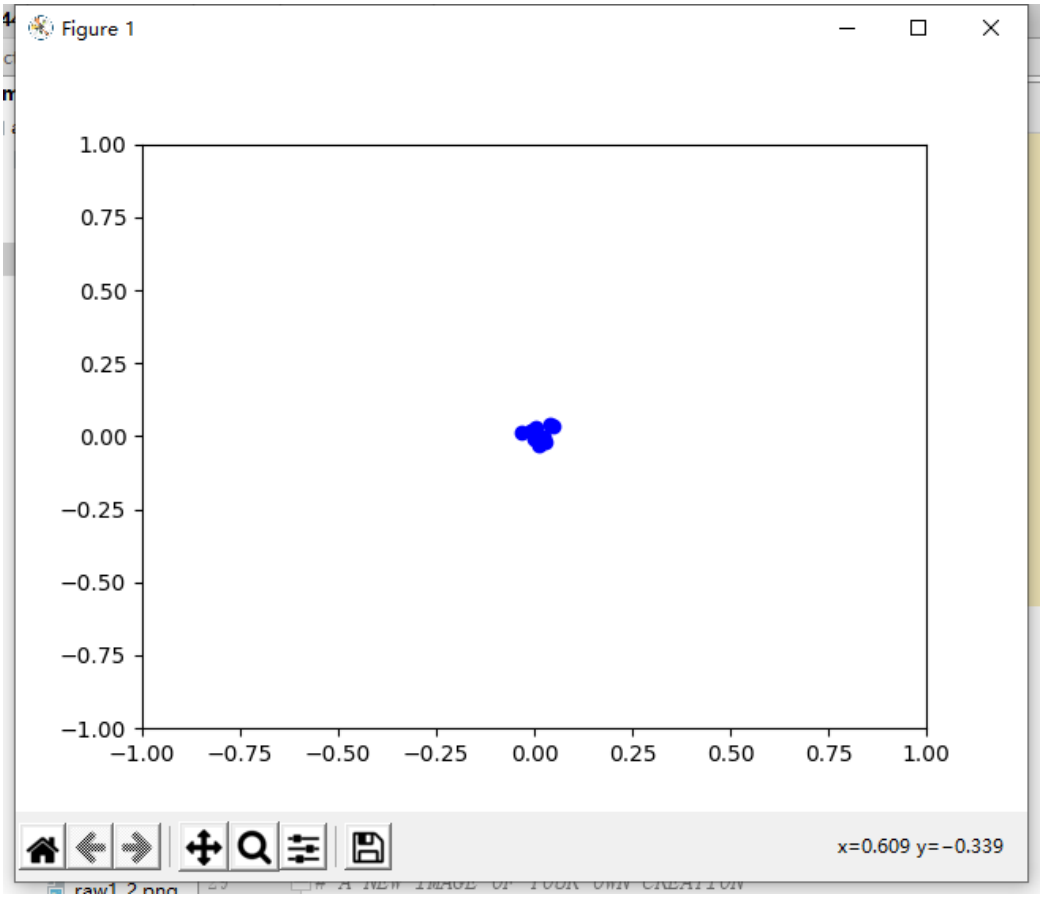


Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x=0.609 y=−0.339

(3)

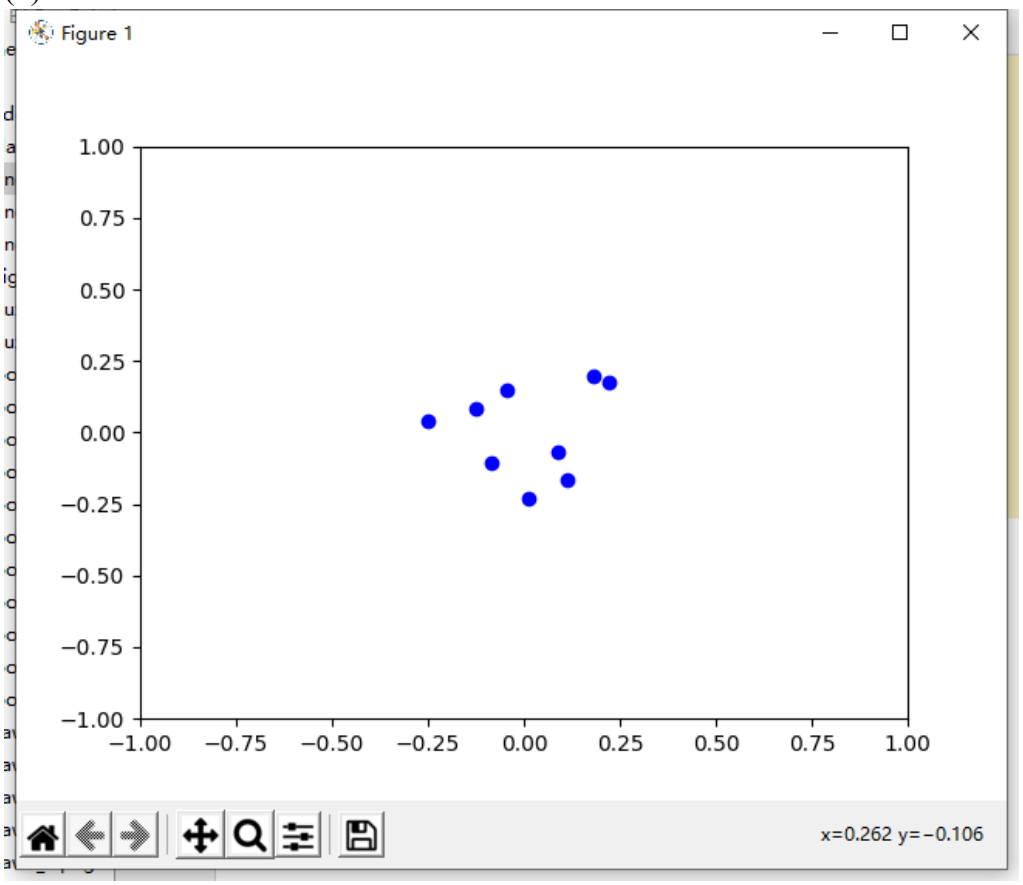


Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x=0.262 y=−0.106

(4)

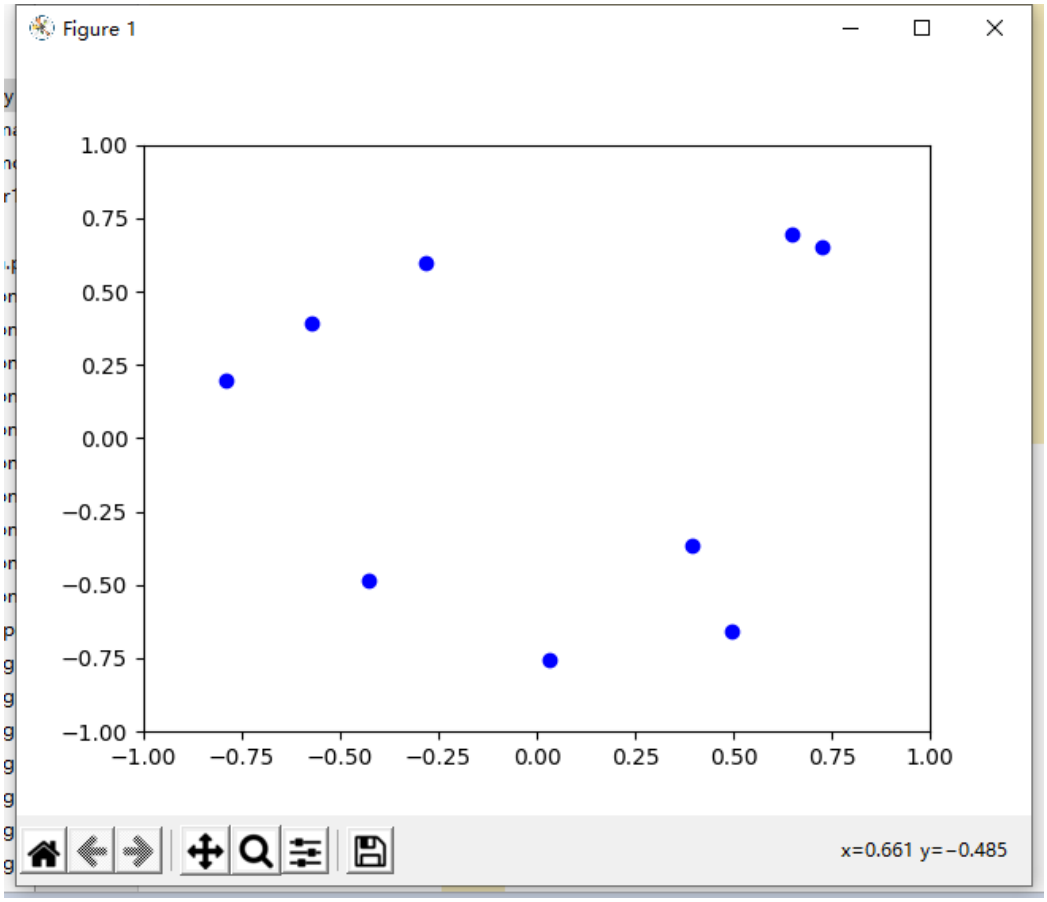


Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x=0.661 y=−0.485

(5)

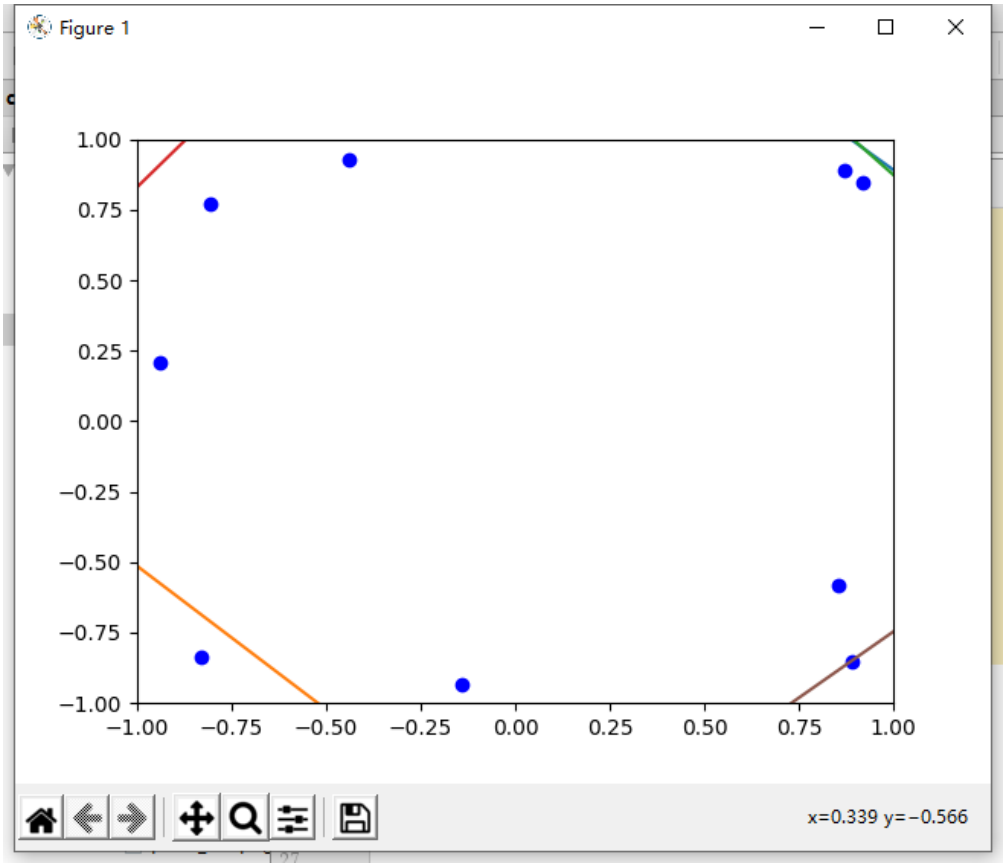


Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x=0.339 y=−0.566

(6)

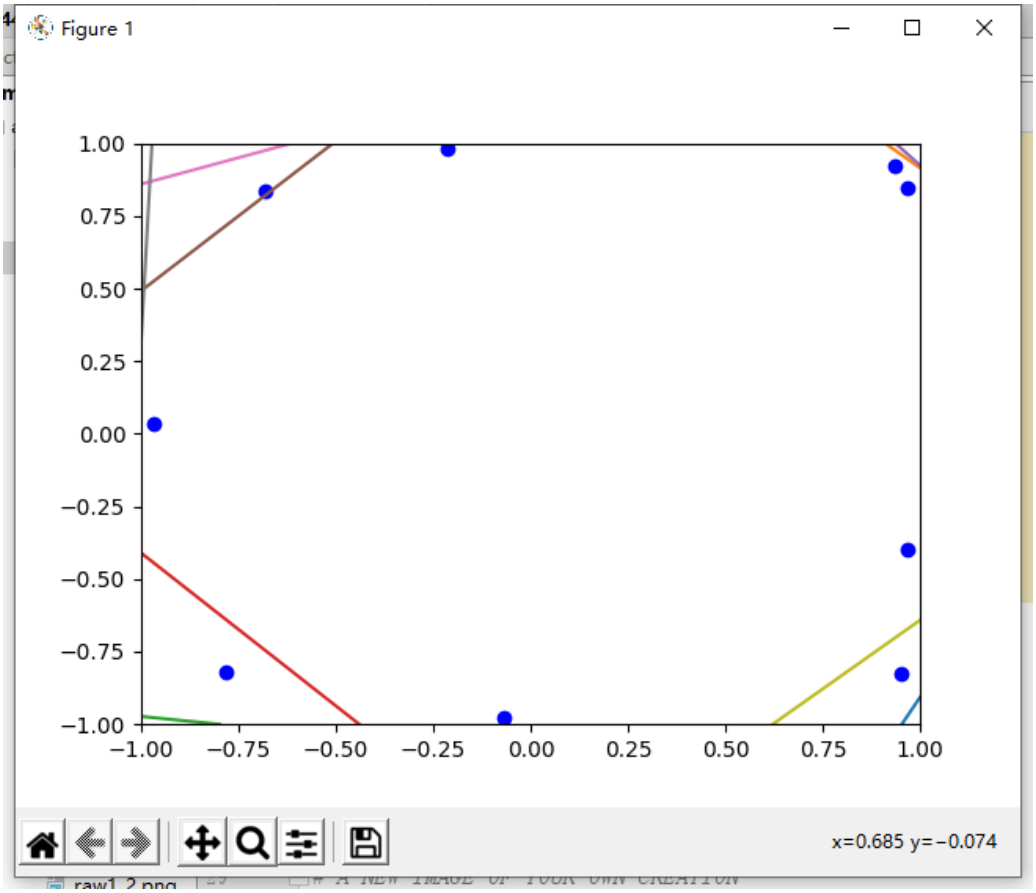
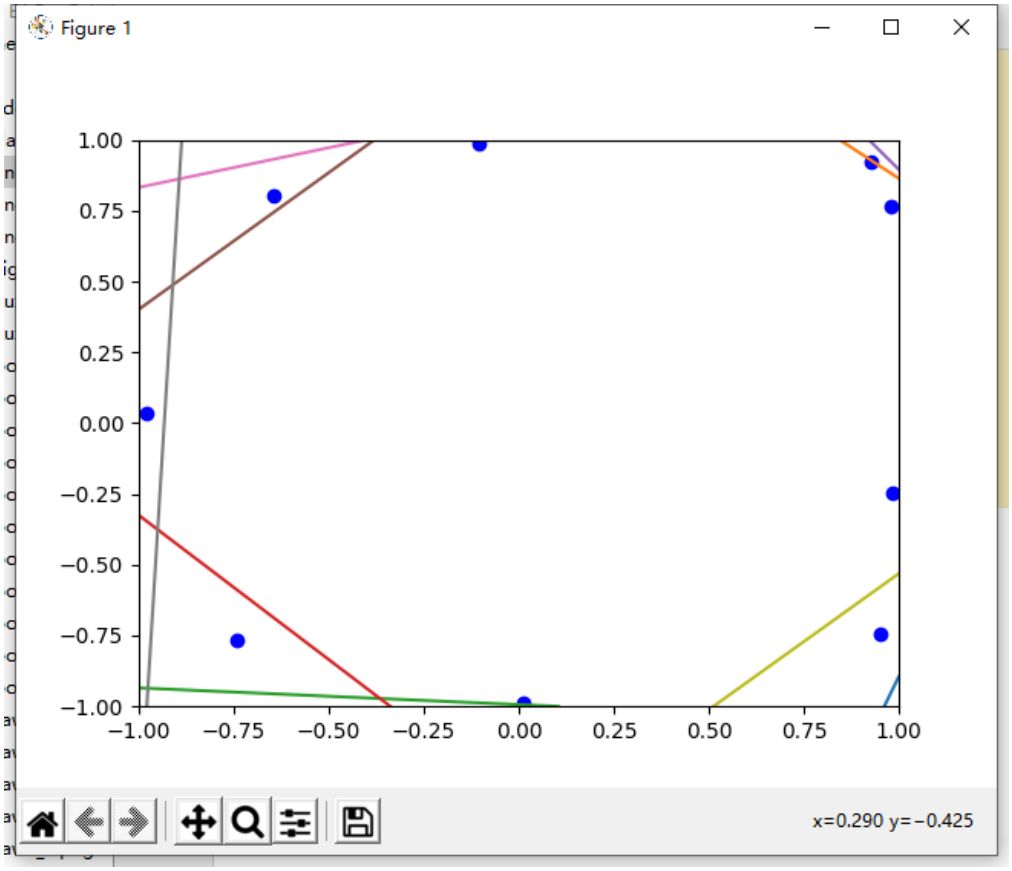
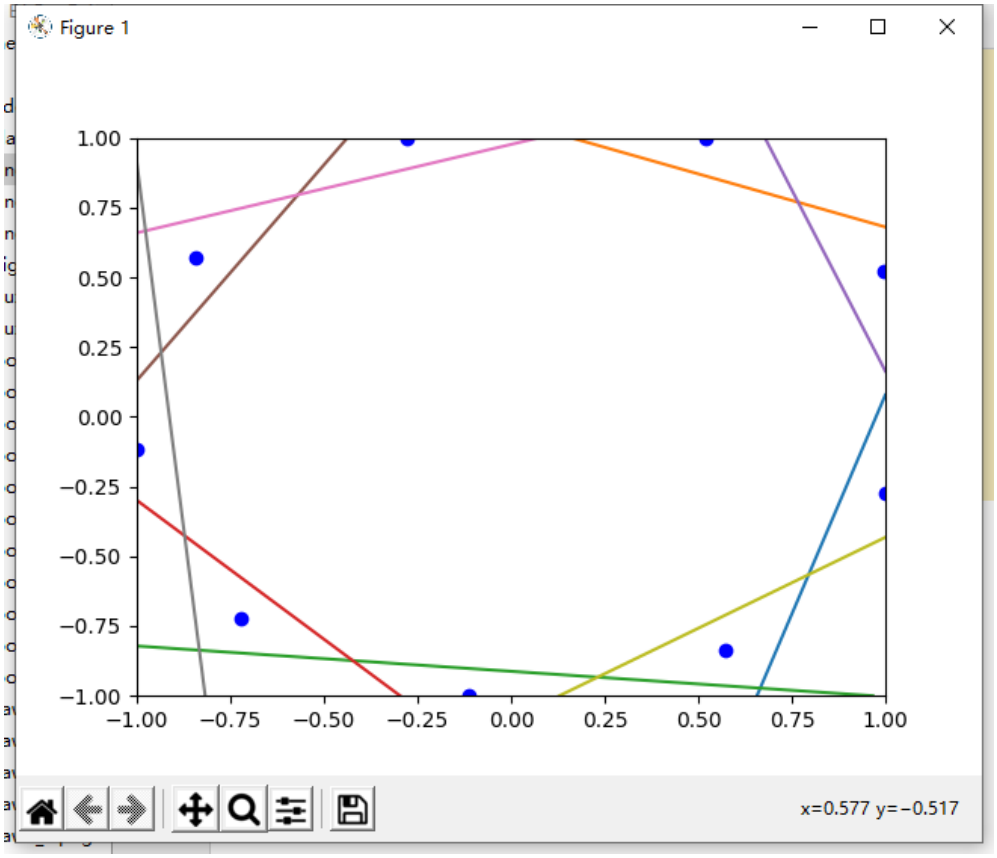


Figure 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
x=0.685 y=−0.074

(7)

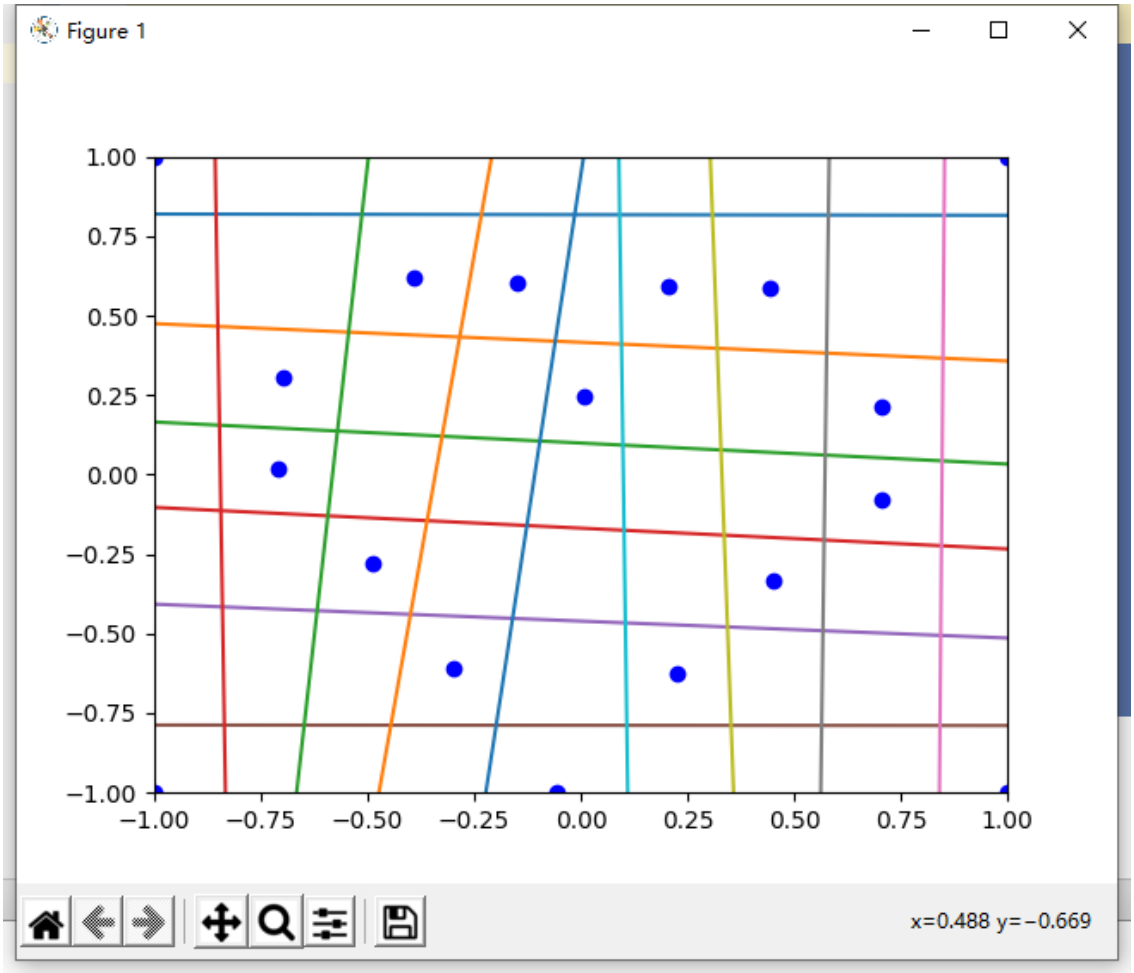


（8） This is the final image.

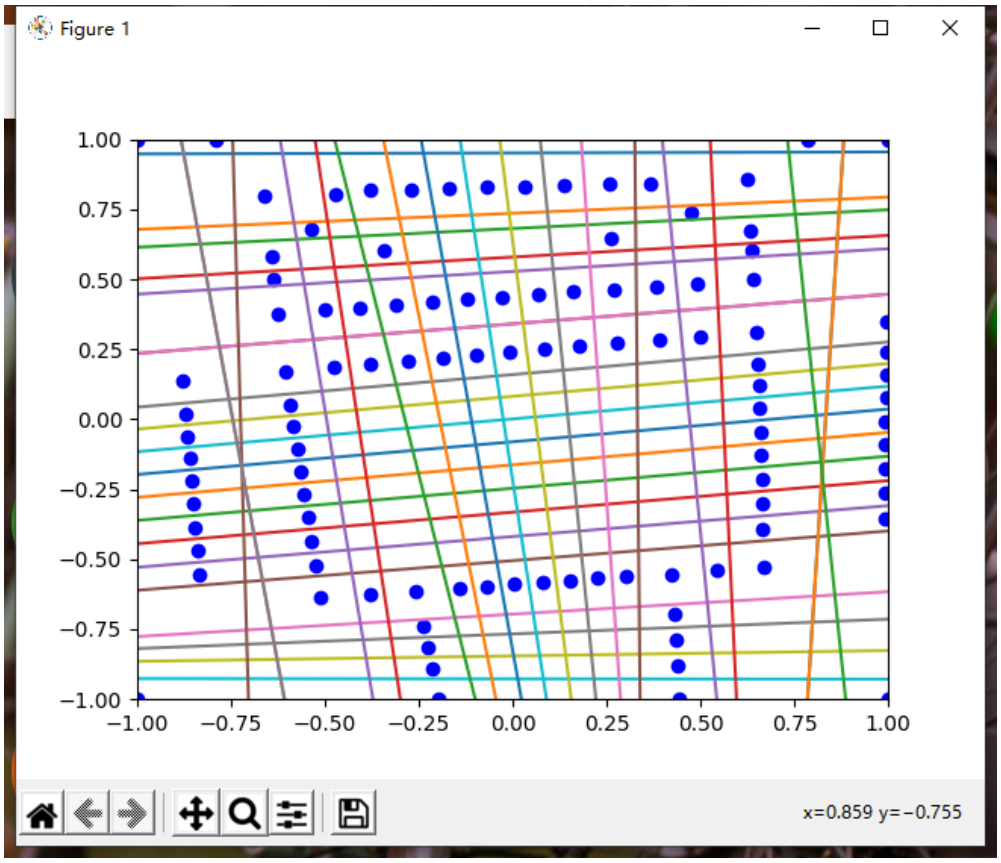


At the beginning of this process, nine points are gathering in the middle of the figure. No line could divide any point with others. Therefore, in order to divide these points and improve the model, these points move to the edges of this figure gradually. In the fifth picture, I could see that some of the points could be divided with others. In the end, this process finish until nine points could be divided by nine lines.

3. python3 encoder_main.py --target=heart18



4. (1) Android logo
I tried to draw an Android logo as follow. The learning rate is 1.2



(2)Python logo

I tried to draw a python logo. The learning rate is 1.3. The result is “ep51870: loss = 0.0200”.