COMP9444 Neural Networks and Deep Learning

Project 1 - Characters, Spirals and Hidden Unit Dynamics

Name: YU ZHANG

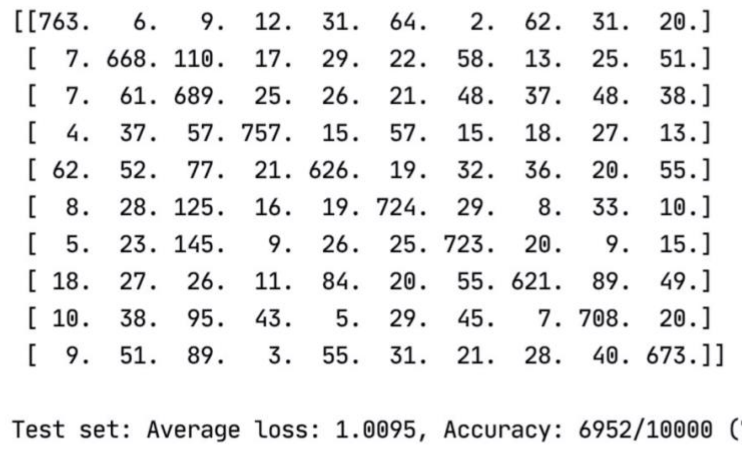
Student ID: z5238743

Part 1: Japanese Character Recognition

Q1. Implement a model NetLin which computes a linear function of the pixels in the image, followed by log softmax:

Answer:

B:

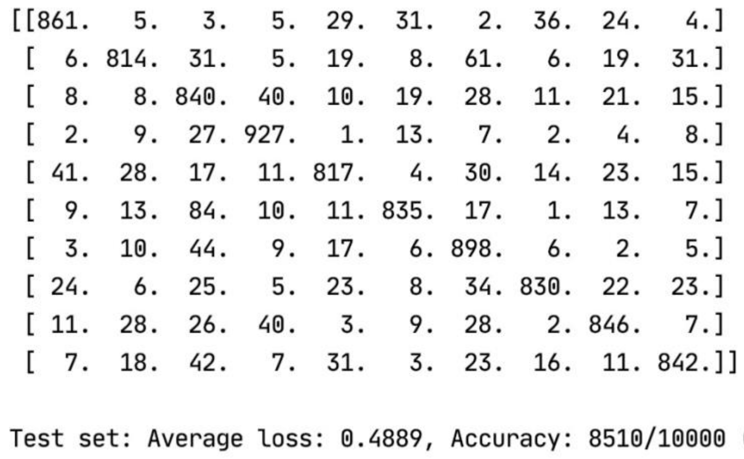


```
[[763.   6.   9.  12.  31.  64.   2.  62.  31.  20.]
 [  7. 668. 110.  17.  29.  22.  58.  13.  25.  51.]
 [  7.  61. 689.  25.  26.  21.  48.  37.  48.  38.]
 [  4.  37.  57. 757.  15.  57.  15.  18.  27.  13.]
 [ 62.  52.  77.  21. 626.  19.  32.  36.  20.  55.]
 [  8.  28. 125.  16.  19. 724.  29.   8.  33.  10.]
 [  5.  23. 145.   9.  26.  25. 723.  20.   9.  15.]
 [ 18.  27.  26.  11.  84.  20.  55. 621.  89.  49.]
 [ 10.  38.  95.  43.   5.  29.  45.   7. 708.  20.]
 [  9.  51.  89.   3.  55.  31.  21.  28.  40. 673.]]

Test set: Average loss: 1.0095, Accuracy: 6952/10000 (70%)
```

Q2. Implement a fully connected 2-layer network NetFull, using tanh at the hidden nodes and log softmax at the output node.

Answer:

the confusion matrix and the accuracy are:



```
[[861.   5.   3.   5.  29.  31.   2.  36.  24.   4.]
 [  6. 814.  31.   5.  19.   8.  61.   6.  19.  31.]
 [  8.   8. 840.  40.  10.  19.  28.  11.  21.  15.]
 [  2.   9.  27. 927.   1.  13.   7.   2.   4.   8.]
 [ 41.  28.  17.  11. 817.   4.  30.  14.  23.  15.]
 [  9.  13.  84.  10.  11. 835.  17.   1.  13.   7.]
 [  3.  10.  44.   9.  17.   6. 898.   6.   2.   5.]
 [ 24.   6.  25.   5.  23.   8.  34. 830.  22.  23.]
 [ 11.  28.  26.  40.   3.   9.  28.   2. 846.   7.]
 [  7.  18.  42.   7.  31.   3.  23.  16.  11. 842.]]

Test set: Average loss: 0.4889, Accuracy: 8510/10000 (85%)
```

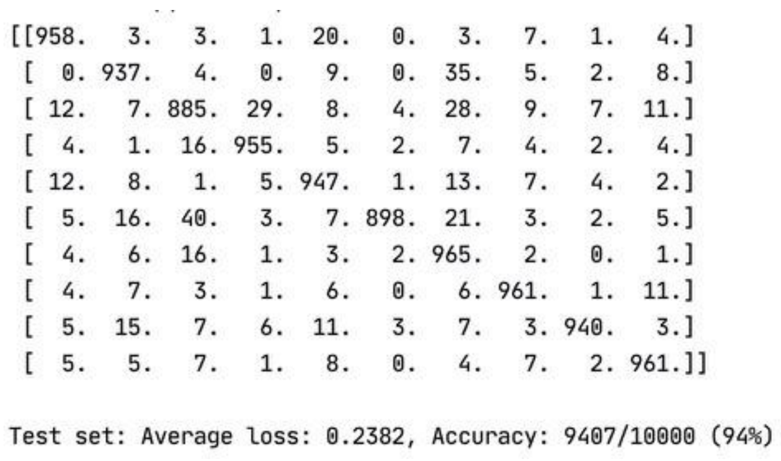
After training different the number of hidden nodes, I get the accuracy of result of different hidden nodes. they are shown below.

The number of hidden nodes	Accuracy
10	67.27%
30	77.83%
60	82.60%
90	83.51%
120	84.45%
150	84.20%
180	84.67%
210	84.82%
240	85.10%
270	84.53%
300	84.69%
330	84.94%
360	85.15%
390	84.93%
420	84.84%
450	84.76%

Q3: Implement a convolutional network called NetConv, with two convolutional layers plus one fully connected layer, all using relu activation function, followed by the output layer, using log softmax. You are free to choose for yourself the number and size of the filters, metaparameter values (learning rate and momentum), and whether to use max pooling or a fully convolutional architecture.

Answer:

the confusion matrix and the accuracy are:



```
[[958.   3.   3.   1.  20.   0.   3.   7.   1.   4.]
 [  0. 937.   4.   0.   9.   0.  35.   5.   2.   8.]
 [ 12.   7. 885.  29.   8.   4.  28.   9.   7.  11.]
 [  4.   1.  16. 955.   5.   2.   7.   4.   2.   4.]
 [ 12.   8.   1.   5. 947.   1.  13.   7.   4.   2.]
 [  5.  16.  40.   3.   7. 898.  21.   3.   2.   5.]
 [  4.   6.  16.   1.   3.   2. 965.   2.   0.   1.]
 [  4.   7.   3.   1.   6.   0.   6. 961.   1.  11.]
 [  5.  15.   7.   6.  11.   3.   7.   3. 940.   3.]
 [  5.   5.   7.   1.   8.   0.   4.   7.   2. 961.]]

Test set: Average loss: 0.2382, Accuracy: 9407/10000 (94%)
```

In the first convolutional network layer, the input_channel is 1, the output_channel is 32 and the kernel_size is 5. In the second convolutional network layer, the input_channel is 32, the output_channel is 64 and the kernel_size is 5.

Q4: Discuss what you have learned from this exercise, including the following points:
A. the relative accuracy of the three models.
B. the confusion matrix for each model: which characters are most likely to be mistaken for which other

characters, and why?
C.you may wish to experiment with other architectures and/or metaparameters for this dataset, and report on your results; the aim of this exercise is not only to achieve high accuracy but also to understand the effect of different choices on the final accuracy.

Answer:

A.:

From all of these 3 neural networks, compared to the result of accuracy, we can clear see that the accuracy of single linear network is lowest than fully_connected network and the convolutional network, fully_connected network has the middle accuracy and the convolutional network has the high accuracy. It can conclude that single linear network and fully_connected network is not suitable for image recognition, however, convolutional network can do better.

In my opinion, single linear network has the lowest and inadequate fitting ability. Although two fully_connected networks have bigger capacity of network than single linear network and can use more complex active function, Besides, it can separate feature into more smaller feature, which means more precise output can be generated. two fully_connected network can't solve image recognition efficient than convolutional network. Compared with single linear network and two fully_connected network, convolutional network has some advantages such as Local connection, weight sharing, pooling and down-sampling. The nodes of the convolutional layer are only connected to some nodes of the previous layer and are only used to learn local features and their weights will reduce and introduce fewer noise. Therefore, the accuracy of convolutional network are highest than others.

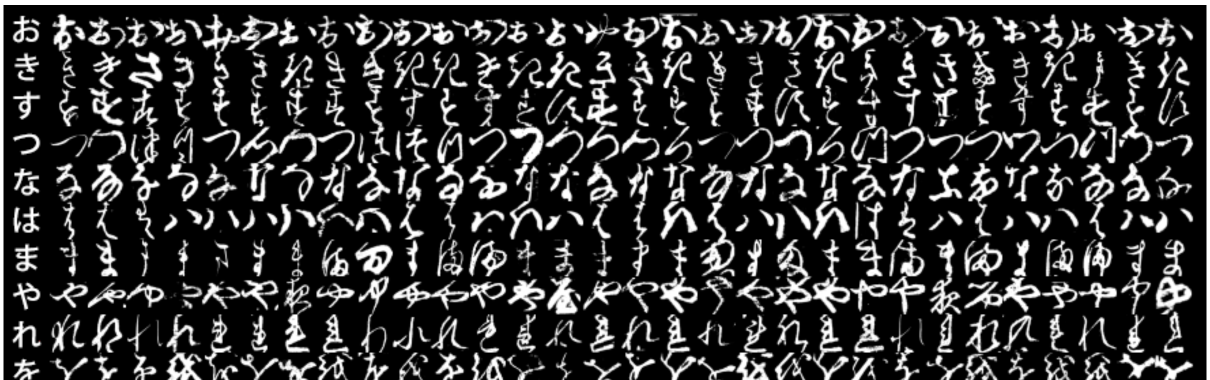
B:

From the 3 networks confusion matrix, the rows of the confusion matrix indicate the target character, while the columns indicate the one chosen by the network.

In Netlin:

According to the confusion matrix, wo can clearly see that 'su' character trained by network have much mistakes than other character trained by network which above 100 times. Because all of these mistakes are the second biggest number than the diagonal in the the confusion matrix. Those numbers are 110, 125, 145 which located in the second row, the fifth row and the sixth row in the third column. Therefore, compared to "ki", "ha" and "ma", "ma" is the most frequently mistaken recognized by "su", although all of them are mistaken recognized by "su".

From the picture shown above, we can see that "o"," ki", "su", "tsu", "na", "ha", "ma", ya", "re", "wo" from up to bottom.



Because NetLin only extracts general features of Japanese characters and it can't go deeper to classify details.

In Netfull:

Compared all the data in the confusion matrix, we can clearly see that 84 is the second largest number except the number on the diagonal in the confusion matrix. We can clearly see that 'su' character trained by network have most mistakes, which located in the sixth row in the third column. This means "ha" is the most frequently mistaken recognized as "su".

Compared to Netlin, the accuracy has improved and two mistaken character has been correct a lot which are "ma" and "ki", From the picture in the analysis of Netlin, we can see Netfull solve the similarity between "su" and "ma" and "ki". With a fully connected layer, the Netfull goes deeper details and make a more precise classification.

In NetConv:

According to the confusion matrix, we can clearly see that 40 is the second largest number except the number on the diagonal in the confusion matrix. We can clearly see that 'su' character trained by network have most mistakes, which located in the sixth row in the third column. This means "ha" is the most frequently mistaken recognized by "su".

Convolutional network has two convolutional layers, The nodes of the convolutional layer are only connected to some nodes of the previous layer and are only used to learn local features and their weights will reduce and introduce fewer noise. If we add more convolutional layers and more hidden nodes, it may become more accuracy in the projects.

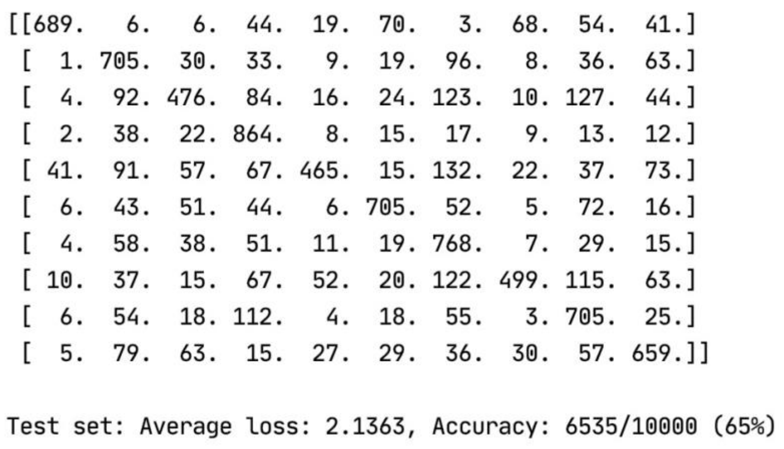
C:

1. Change the metaparameters

1.1 change the learning rate

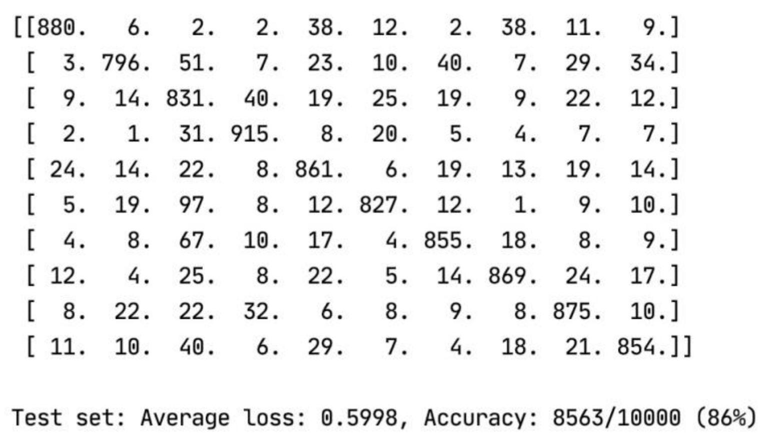
1.1.1   Lr = 0.2

Netlin:



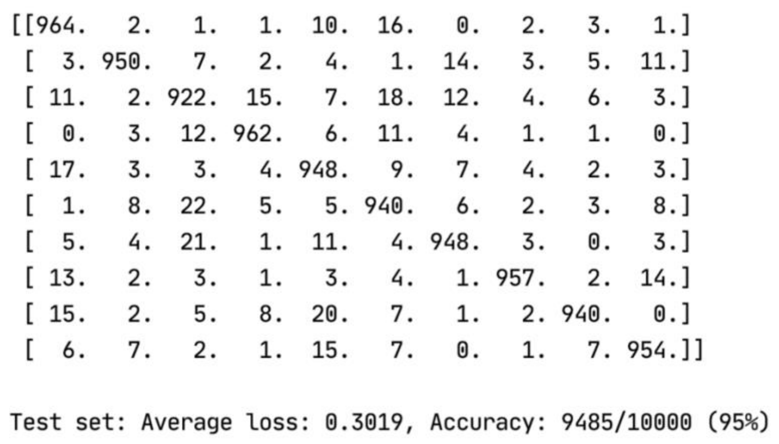
```
[[689.   6.   6.  44.  19.  70.   3.  68.  54.  41.]
 [  1. 705.  30.  33.   9.  19.  96.   8.  36.  63.]
 [  4.  92. 476.  84.  16.  24. 123.  10. 127.  44.]
 [  2.  38.  22. 864.   8.  15.  17.   9.  13.  12.]
 [ 41.  91.  57.  67. 465.  15. 132.  22.  37.  73.]
 [  6.  43.  51.  44.   6. 705.  52.   5.  72.  16.]
 [  4.  58.  38.  51.  11.  19. 768.   7.  29.  15.]
 [ 10.  37.  15.  67.  52.  20. 122. 499. 115.  63.]
 [  6.  54.  18. 112.   4.  18.  55.   3. 705.  25.]
 [  5.  79.  63.  15.  27.  29.  36.  30.  57. 659.]]

Test set: Average loss: 2.1363, Accuracy: 6535/10000 (65%)
```

NetFull:



```
[[880.   6.   2.   2.  38.  12.   2.  38.  11.   9.]
 [  3. 796.  51.   7.  23.  10.  40.   7.  29.  34.]
 [  9.  14. 831.  40.  19.  25.  19.   9.  22.  12.]
 [  2.   1.  31. 915.   8.  20.   5.   4.   7.   7.]
 [ 24.  14.  22.   8. 861.   6.  19.  13.  19.  14.]
 [  5.  19.  97.   8.  12. 827.  12.   1.   9.  10.]
 [  4.   8.  67.  10.  17.   4. 855.  18.   8.   9.]
 [ 12.   4.  25.   8.  22.   5.  14. 869.  24.  17.]
 [  8.  22.  22.  32.   6.   8.   9.   8. 875.  10.]
 [ 11.  10.  40.   6.  29.   7.   4.  18.  21. 854.]]

Test set: Average loss: 0.5998, Accuracy: 8563/10000 (86%)
```

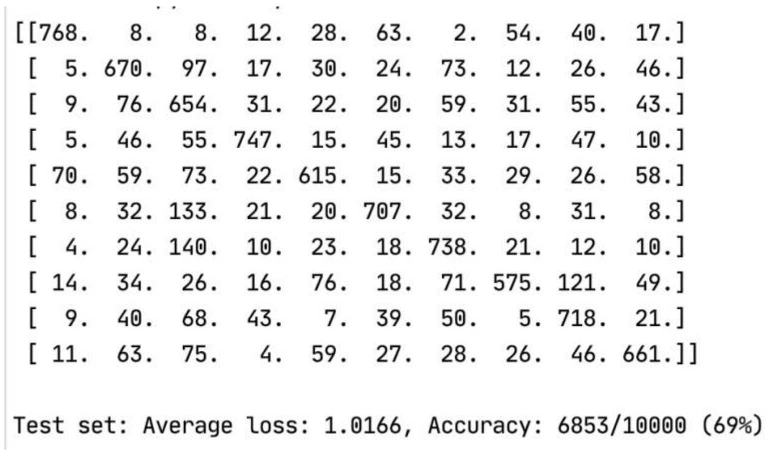
NetConv:



```
[[964.   2.   1.   1.  10.  16.   0.   2.   3.   1.]
 [  3. 950.   7.   2.   4.   1.  14.   3.   5.  11.]
 [ 11.   2. 922.  15.   7.  18.  12.   4.   6.   3.]
 [  0.   3.  12. 962.   6.  11.   4.   1.   1.   0.]
 [ 17.   3.   3.   4. 948.   9.   7.   4.   2.   3.]
 [  1.   8.  22.   5.   5. 940.   6.   2.   3.   8.]
 [  5.   4.  21.   1.  11.   4. 948.   3.   0.   3.]
 [ 13.   2.   3.   1.   3.   4.   1. 957.   2.  14.]
 [ 15.   2.   5.   8.  20.   7.   1.   2. 940.   0.]
 [  6.   7.   2.   1.  15.   7.   0.   1.   7. 954.]]

Test set: Average loss: 0.3019, Accuracy: 9485/10000 (95%)
```

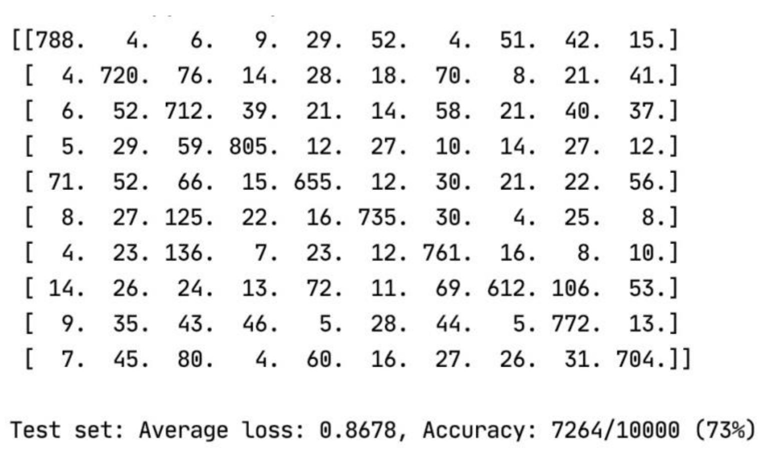
1.1.2. Lr = 0.002

Netlin:



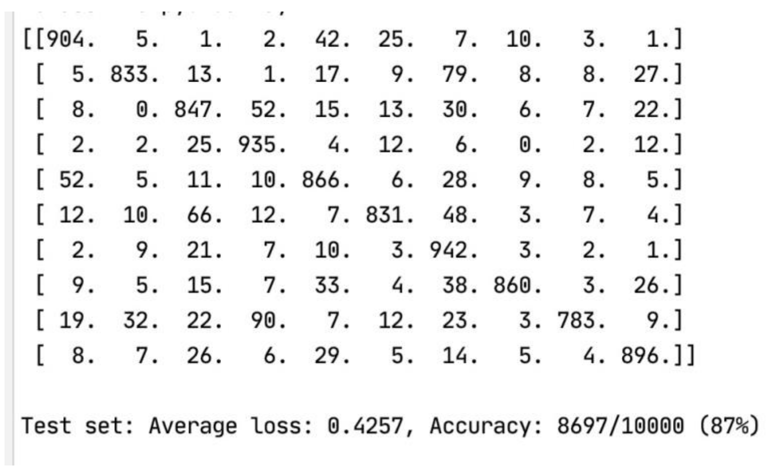
```
[[768.   8.   8.  12.  28.  63.   2.  54.  40.  17.]
 [  5. 670.  97.  17.  30.  24.  73.  12.  26.  46.]
 [  9.  76. 654.  31.  22.  20.  59.  31.  55.  43.]
 [  5.  46.  55. 747.  15.  45.  13.  17.  47.  10.]
 [ 70.  59.  73.  22. 615.  15.  33.  29.  26.  58.]
 [  8.  32. 133.  21.  20. 707.  32.   8.  31.   8.]
 [  4.  24. 140.  10.  23.  18. 738.  21.  12.  10.]
 [ 14.  34.  26.  16.  76.  18.  71. 575. 121.  49.]
 [  9.  40.  68.  43.   7.  39.  50.   5. 718.  21.]
 [ 11.  63.  75.   4.  59.  27.  28.  26.  46. 661.]]

Test set: Average loss: 1.0166, Accuracy: 6853/10000 (69%)
```

Netfull:



```
[[788.   4.   6.   9.  29.  52.   4.  51.  42.  15.]
 [  4. 720.  76.  14.  28.  18.  70.   8.  21.  41.]
 [  6.  52. 712.  39.  21.  14.  58.  21.  40.  37.]
 [  5.  29.  59. 805.  12.  27.  10.  14.  27.  12.]
 [ 71.  52.  66.  15. 655.  12.  30.  21.  22.  56.]
 [  8.  27. 125.  22.  16. 735.  30.   4.  25.   8.]
 [  4.  23. 136.   7.  23.  12. 761.  16.   8.  10.]
 [ 14.  26.  24.  13.  72.  11.  69. 612. 106.  53.]
 [  9.  35.  43.  46.   5.  28.  44.   5. 772.  13.]
 [  7.  45.  80.   4.  60.  16.  27.  26.  31. 704.]]

Test set: Average loss: 0.8678, Accuracy: 7264/10000 (73%)
```

NetConv:



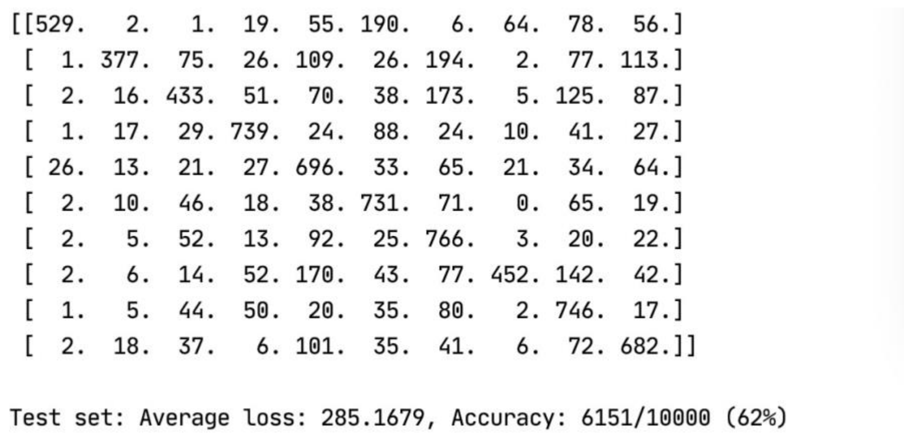
```
[[904.   5.   1.   2.  42.  25.   7.  10.   3.   1.]
 [  5. 833.  13.   1.  17.   9.  79.   8.   8.  27.]
 [  8.   0. 847.  52.  15.  13.  30.   6.   7.  22.]
 [  2.   2.  25. 935.   4.  12.   6.   0.   2.  12.]
 [ 52.   5.  11.  10. 866.   6.  28.   9.   8.   5.]
 [ 12.  10.  66.  12.   7. 831.  48.   3.   7.   4.]
 [  2.   9.  21.   7.  10.   3. 942.   3.   2.   1.]
 [  9.   5.  15.   7.  33.   4.  38. 860.   3.  26.]
 [ 19.  32.  22.  90.   7.  12.  23.   3. 783.   9.]
 [  8.   7.  26.   6.  29.   5.  14.   5.   4. 896.]]

Test set: Average loss: 0.4257, Accuracy: 8697/10000 (87%)
```

From the changing of learning rate, we can clearly see that the bigger learning rate we have, the low accuracy of Netlin we have and the high accuracy of Netfull and NetConv we have.

2.2 change the momentum

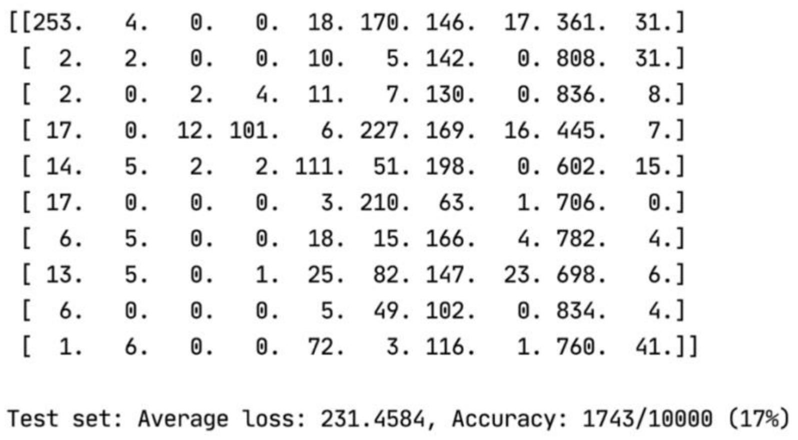
2.2.1 mom = 1.0

Netlin:



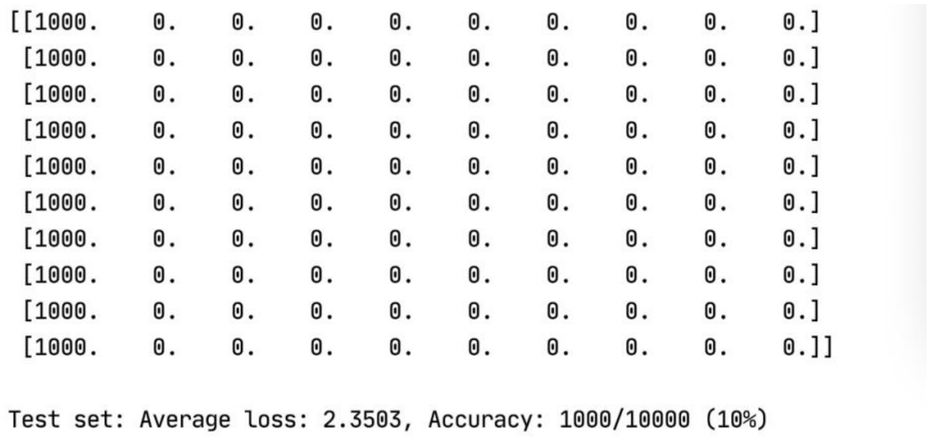
```
[[529.   2.   1.  19.  55. 190.   6.  64.  78.  56.]
 [  1. 377.  75.  26. 109.  26. 194.   2.  77. 113.]
 [  2.  16. 433.  51.  70.  38. 173.   5. 125.  87.]
 [  1.  17.  29. 739.  24.  88.  24.  10.  41.  27.]
 [ 26.  13.  21.  27. 696.  33.  65.  21.  34.  64.]
 [  2.  10.  46.  18.  38. 731.  71.   0.  65.  19.]
 [  2.   5.  52.  13.  92.  25. 766.   3.  20.  22.]
 [  2.   6.  14.  52. 170.  43.  77. 452. 142.  42.]
 [  1.   5.  44.  50.  20.  35.  80.   2. 746.  17.]
 [  2.  18.  37.   6. 101.  35.  41.   6.  72. 682.]]

Test set: Average loss: 285.1679, Accuracy: 6151/10000 (62%)
```

Netfull:



```
[[253.   4.   0.   0.  18. 170. 146.  17. 361.  31.]
 [  2.   2.   0.   0.  10.   5. 142.   0. 808.  31.]
 [  2.   0.   2.   4.  11.   7. 130.   0. 836.   8.]
 [ 17.   0.  12. 101.   6. 227. 169.  16. 445.   7.]
 [ 14.   5.   2.   2. 111.  51. 198.   0. 602.  15.]
 [ 17.   0.   0.   0.   3. 210.  63.   1. 706.   0.]
 [  6.   5.   0.   0.  18.  15. 166.   4. 782.   4.]
 [ 13.   5.   0.   1.  25.  82. 147.  23. 698.   6.]
 [  6.   0.   0.   0.   5.  49. 102.   0. 834.   4.]
 [  1.   6.   0.   0.  72.   3. 116.   1. 760.  41.]]

Test set: Average loss: 231.4584, Accuracy: 1743/10000 (17%)
```

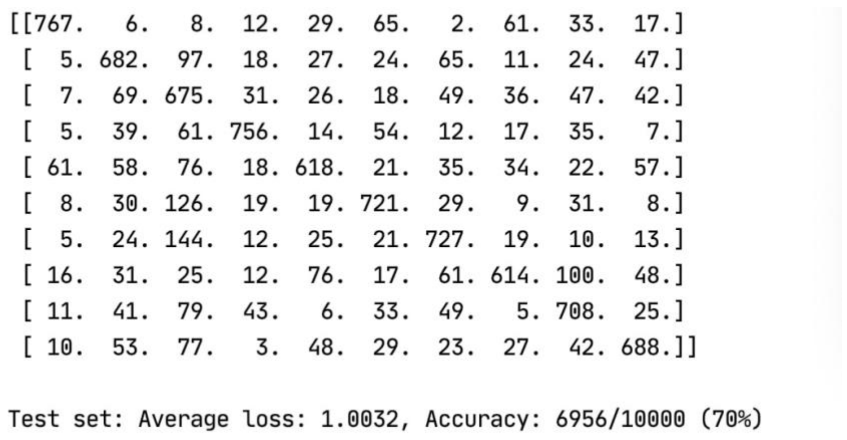
NetConv:



```
[[1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]
 [1000.    0.    0.    0.    0.    0.    0.    0.    0.    0.]]

Test set: Average loss: 2.3503, Accuracy: 1000/10000 (10%)
```

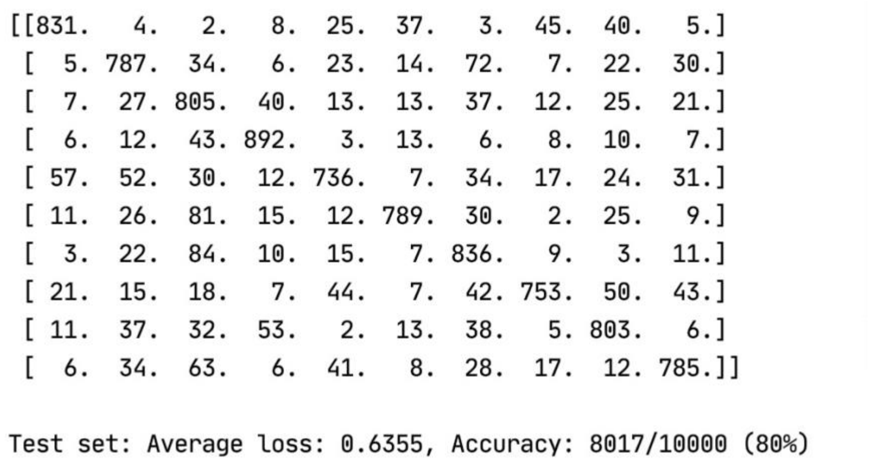
2.2.1 mom = 0.01

Netlin:



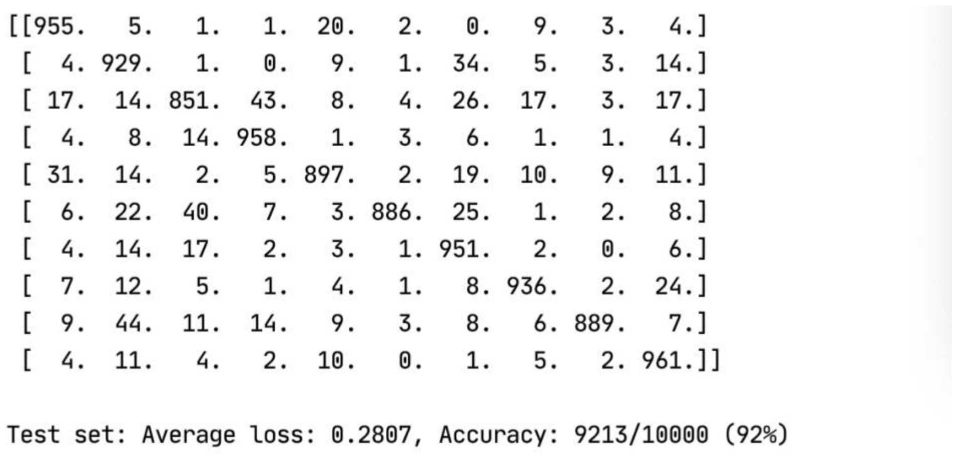
```
[[767.   6.   8.  12.  29.  65.   2.  61.  33.  17.]
 [  5. 682.  97.  18.  27.  24.  65.  11.  24.  47.]
 [  7.  69. 675.  31.  26.  18.  49.  36.  47.  42.]
 [  5.  39.  61. 756.  14.  54.  12.  17.  35.   7.]
 [ 61.  58.  76.  18. 618.  21.  35.  34.  22.  57.]
 [  8.  30. 126.  19.  19. 721.  29.   9.  31.   8.]
 [  5.  24. 144.  12.  25.  21. 727.  19.  10.  13.]
 [ 16.  31.  25.  12.  76.  17.  61. 614. 100.  48.]
 [ 11.  41.  79.  43.   6.  33.  49.   5. 708.  25.]
 [ 10.  53.  77.   3.  48.  29.  23.  27.  42. 688.]]

Test set: Average loss: 1.0032, Accuracy: 6956/10000 (70%)
```

Netfull:



```
[[831.   4.   2.   8.  25.  37.   3.  45.  40.   5.]
 [  5. 787.  34.   6.  23.  14.  72.   7.  22.  30.]
 [  7.  27. 805.  40.  13.  13.  37.  12.  25.  21.]
 [  6.  12.  43. 892.   3.  13.   6.   8.  10.   7.]
 [ 57.  52.  30.  12. 736.   7.  34.  17.  24.  31.]
 [ 11.  26.  81.  15.  12. 789.  30.   2.  25.   9.]
 [  3.  22.  84.  10.  15.   7. 836.   9.   3.  11.]
 [ 21.  15.  18.   7.  44.   7.  42. 753.  50.  43.]
 [ 11.  37.  32.  53.   2.  13.  38.   5. 803.   6.]
 [  6.  34.  63.   6.  41.   8.  28.  17.  12. 785.]]

Test set: Average loss: 0.6355, Accuracy: 8017/10000 (80%)
```

NetConv:



```
[[955.   5.   1.   1.  20.   2.   0.   9.   3.   4.]
 [  4. 929.   1.   0.   9.   1.  34.   5.   3.  14.]
 [ 17.  14. 851.  43.   8.   4.  26.  17.   3.  17.]
 [  4.   8.  14. 958.   1.   3.   6.   1.   1.   4.]
 [ 31.  14.   2.   5. 897.   2.  19.  10.   9.  11.]
 [  6.  22.  40.   7.   3. 886.  25.   1.   2.   8.]
 [  4.  14.  17.   2.   3.   1. 951.   2.   0.   6.]
 [  7.  12.   5.   1.   4.   1.   8. 936.   2.  24.]
 [  9.  44.  11.  14.   9.   3.   8.   6. 889.   7.]
 [  4.  11.   4.   2.  10.   0.   1.   5.   2. 961.]]

Test set: Average loss: 0.2807, Accuracy: 9213/10000 (92%)
```

From the result, we can clearly see that high value of momentum will lead bad accuracy.

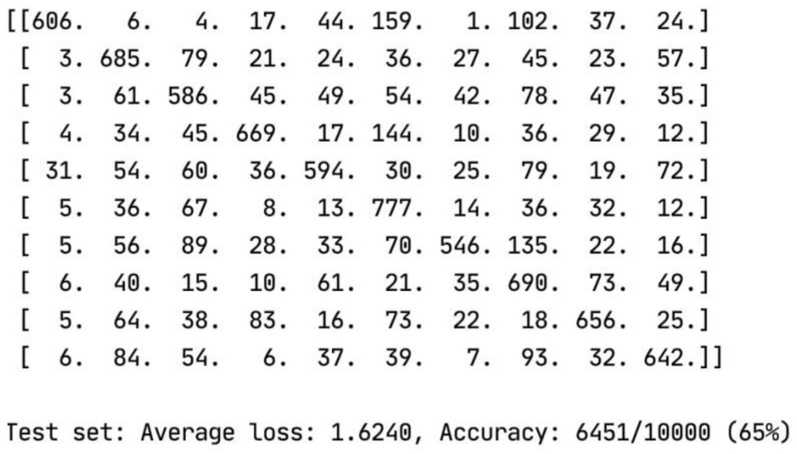
Compared to the momentum value of 1.0 and 0.01, the original default value is 0.5. momentum is to accelerating the SGD and to dampen oscillations. If momentum is comparatively small, the modle's convergence could go slowly because the SGD oscillations. with the higher momentum value, the accuracy will become lower.

2. Change the Structure

I change the optimizer from SGD to Adam.

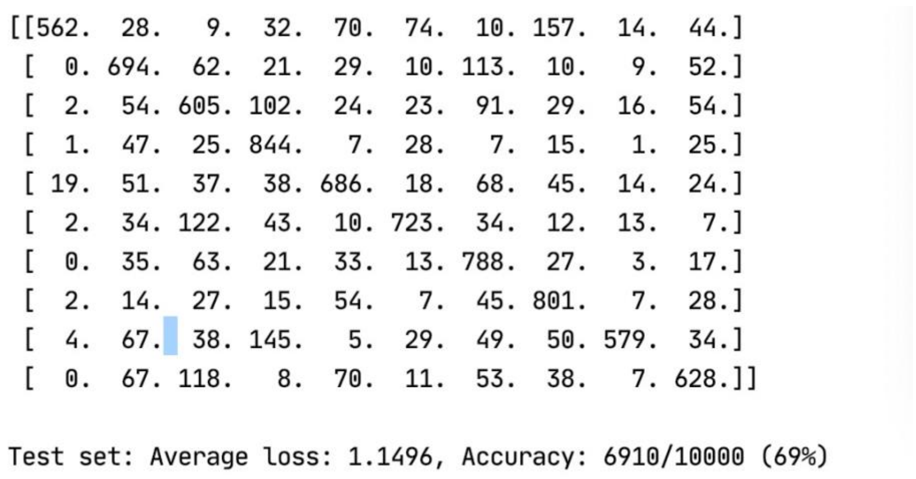
```
optimizer = torch.optim.Adam(net.parameters(),eps=0.000001,lr=args.lr,
                             betas=(0.9,0.999),weight_decay=0.0001)
```

Netlin:



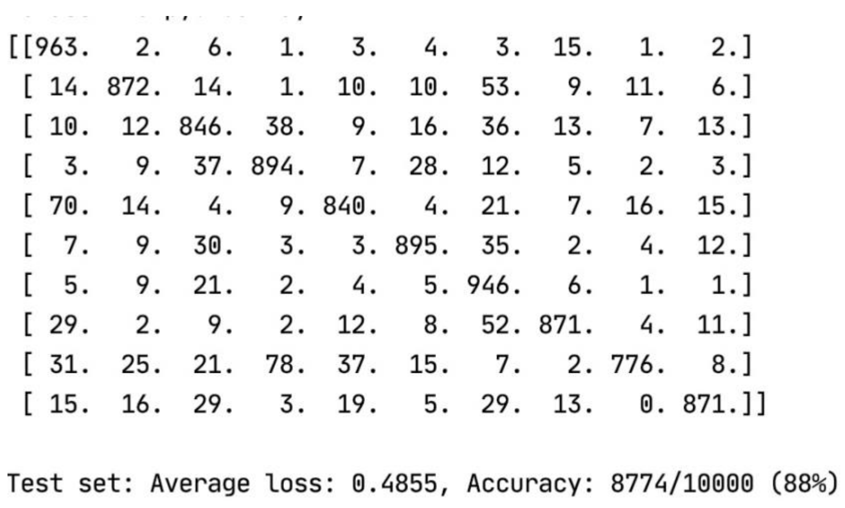
```
[[606.   6.   4.  17.  44. 159.   1. 102.  37.  24.]
 [  3. 685.  79.  21.  24.  36.  27.  45.  23.  57.]
 [  3.  61. 586.  45.  49.  54.  42.  78.  47.  35.]
 [  4.  34.  45. 669.  17. 144.  10.  36.  29.  12.]
 [ 31.  54.  60.  36. 594.  30.  25.  79.  19.  72.]
 [  5.  36.  67.   8.  13. 777.  14.  36.  32.  12.]
 [  5.  56.  89.  28.  33.  70. 546. 135.  22.  16.]
 [  6.  40.  15.  10.  61.  21.  35. 690.  73.  49.]
 [  5.  64.  38.  83.  16.  73.  22.  18. 656.  25.]
 [  6.  84.  54.   6.  37.  39.   7.  93.  32. 642.]]

Test set: Average loss: 1.6240, Accuracy: 6451/10000 (65%)
```

Netfull:



```
[[562.  28.   9.  32.  70.  74.  10. 157.  14.  44.]
 [  0. 694.  62.  21.  29.  10. 113.  10.   9.  52.]
 [  2.  54. 605. 102.  24.  23.  91.  29.  16.  54.]
 [  1.  47.  25. 844.   7.  28.   7.  15.   1.  25.]
 [ 19.  51.  37.  38. 686.  18.  68.  45.  14.  24.]
 [  2.  34. 122.  43.  10. 723.  34.  12.  13.   7.]
 [  0.  35.  63.  21.  33.  13. 788.  27.   3.  17.]
 [  2.  14.  27.  15.  54.   7.  45. 801.   7.  28.]
 [  4.  67.  38. 145.   5.  29.  49.  50. 579.  34.]
 [  0.  67. 118.   8.  70.  11.  53.  38.   7. 628.]]

Test set: Average loss: 1.1496, Accuracy: 6910/10000 (69%)
```

NetConv:



```
[[963.   2.   6.   1.   3.   4.   3.  15.   1.   2.]
 [ 14. 872.  14.   1.  10.  10.  53.   9.  11.   6.]
 [ 10.  12. 846.  38.   9.  16.  36.  13.   7.  13.]
 [  3.   9.  37. 894.   7.  28.  12.   5.   2.   3.]
 [ 70.  14.   4.   9. 840.   4.  21.   7.  16.  15.]
 [  7.   9.  30.   3.   3. 895.  35.   2.   4.  12.]
 [  5.   9.  21.   2.   4.   5. 946.   6.   1.   1.]
 [ 29.   2.   9.   2.  12.   8.  52. 871.   4.  11.]
 [ 31.  25.  21.  78.  37.  15.   7.   2. 776.   8.]
 [ 15.  16.  29.   3.  19.   5.  29.  13.   0. 871.]]

Test set: Average loss: 0.4855, Accuracy: 8774/10000 (88%)
```

From the result, we can see Adam optimizer will reduce the accuracy.

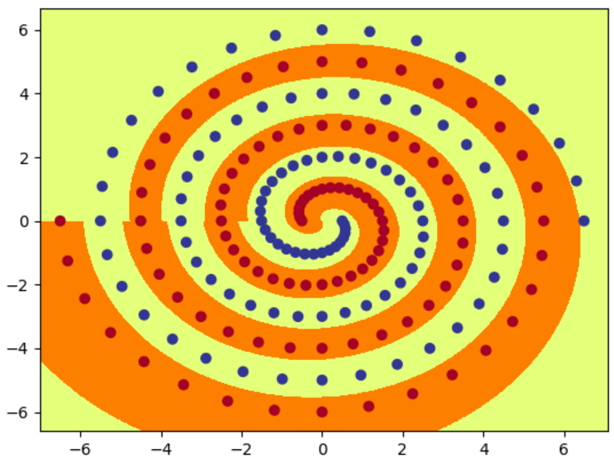
Part 2: Twin Spirals Task

Q1. Provide code for a Pytorch Module called PolarNet which operates as follows: First, the input (x,y) is converted to polar co-ordinates (r,a) with r=sqrt(x*x + y*y), a=atan2(y,x). Next, (r,a) is fed into a fully connected neural network with one hidden layer using tanh activation, followed by a single output using sigmoid activation. The conversion to polar coordinates should be included in your forward() method, so that the Module performs the entire task of conversion followed by network layers.

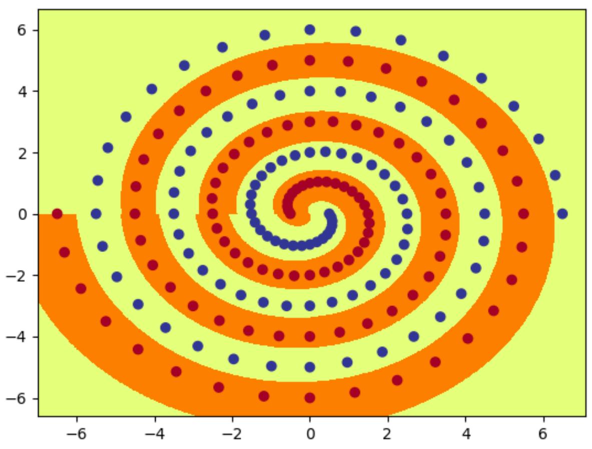
Answer: See the code in spiral.py in detail

Q2. **Try to find the minimum number of hidden nodes required so that this PolarNet learns to correctly classify all of the training data within 20000 epochs, on almost all runs. The `graph_output()` method will generate a picture of the function computed by your PolarNet called `polar_out.png`, which you should include in your report.**

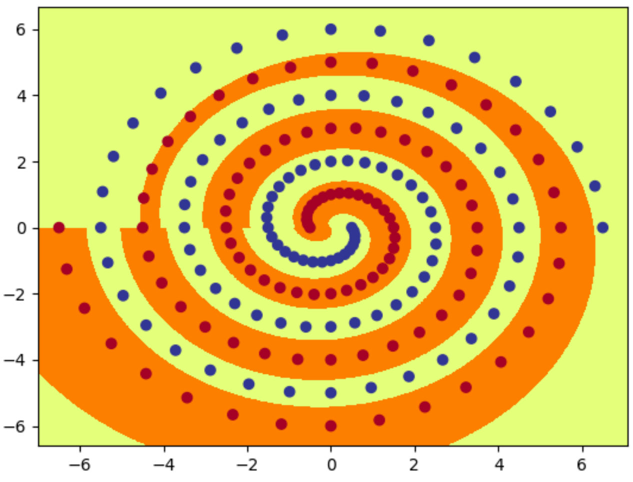
Answer: First, in the Question, the node default is 10 nodes and all 100% runs and finished in 1800 epochs. Below is polar_out.png when number of hidden node is 10.



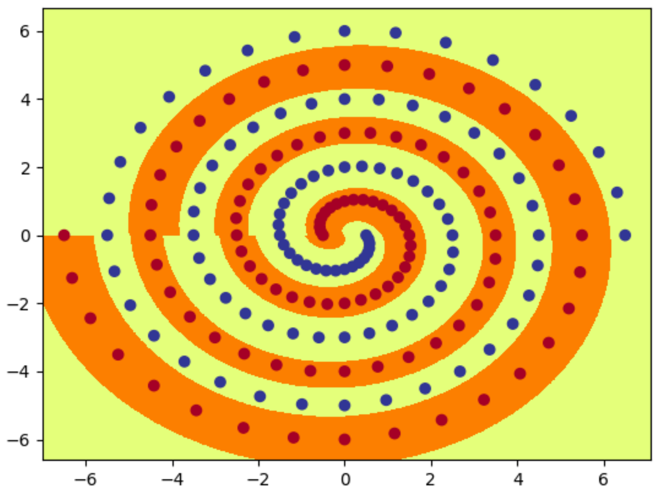
Then I reduced number of hidden nodes by 1 each time, when the number of hidden nodes are 9, all 100% runs and finished in 2400 epochs. Below is polar_out.png when number of hidden node is 9.



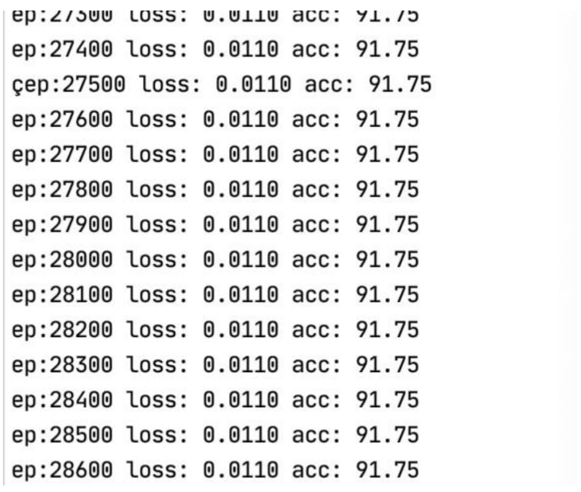
When the number of hidden nodes are 8, all 100% runs and finished in 3300 epochs. buit it has some fluctions in 1900 and 3000. Below is polar_out.png when number of hidden node is 8.



When the number of hidden nodes are 7, all 100% runs and finished in 12900 epochs. buit it has some fluctions in 10100 and 12900. Besides, the increasing accuracy of Polar_net become slowly. Below is polar_out.png when number of hidden node is 7.



When the number of hidden nodes are 6, sometimes all 100% runs and finished in 20000 epochs. but sometime it can't finish in in 20000 epochs. it has the fluctions and the increasing accuracy of Polar_net also become slowly. Below is the picture of epochs when number of hidden node is 6.



```
ep:27300 loss: 0.0110 acc: 91.75
ep:27400 loss: 0.0110 acc: 91.75
çep:27500 loss: 0.0110 acc: 91.75
ep:27600 loss: 0.0110 acc: 91.75
ep:27700 loss: 0.0110 acc: 91.75
ep:27800 loss: 0.0110 acc: 91.75
ep:27900 loss: 0.0110 acc: 91.75
ep:28000 loss: 0.0110 acc: 91.75
ep:28100 loss: 0.0110 acc: 91.75
ep:28200 loss: 0.0110 acc: 91.75
ep:28300 loss: 0.0110 acc: 91.75
ep:28400 loss: 0.0110 acc: 91.75
ep:28500 loss: 0.0110 acc: 91.75
ep:28600 loss: 0.0110 acc: 91.75
```

Therefore, it can conclude that the minimun value of hidden nodes is 7, the polar_out.png is shown above. the nodes can be clearly classified, and the segment area is clean.

Q3. Provide code for a Pytorch Module called RawNet which operates on the raw input (x,y) without converting to polar coordinates. Your network should consist of two fully connected hidden layers with tanh activation, plus the output layer, with sigmoid activation. The two hidden layers should each have the same number of hidden nodes, determined by the parameter num_hid.

Answer: See the code in spiral.py in detail

Q4. Try to choose a value for the number of hidden nodes (--hid) and the size of the initial weights (--init) such that this RawNet learns to correctly classify all of the training data within 20000 epochs, on almost all runs. Include in your report the number of hidden nodes, and the values of any other metaparameters. The graph_output() method will generate a picture of the function computed by your RawNet called raw_out.png, which you should include in your report.

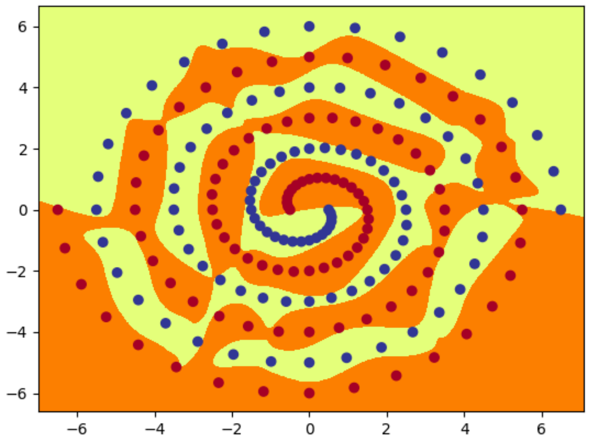
Answer: First I try to fix the number of hidden nodes which is 10. Then I try the default number of initial weight is 0.1. If the accuracy can't finish in 20000 epochs, I will add 0.01 by each time. After I find the best solution of initial weight in the particular number of hidden nodes. I will change the hidden node. then trying to change the initial weight again.

The number of hidden nodes	The number of initial weight	epochs
10	0.1	Above 20000
10	0.11	Above 20000
10	0.12	Above 20000
10	0.121	Above 20000
10	0.122	Above 20000
10	0.125	Above 20000
10	0.126	9400
10	0.13	9100
9	0.1	Above 20000
9	0.126	Above 20000
9	0.13	Above 20000
9	0.14	Above 20000
9	0.15	Above 20000

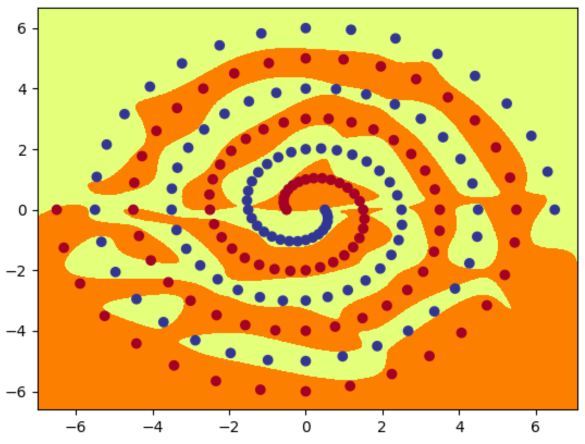
9	0.151	11600
9	0.16	7800
8	0.1	Above 20000
8	0.152	Above 20000
8	0.16	Above 20000
8	0.17	Above 20000
8	0.18	Above 20000
8	0.19	Above 20000
8	0.20	Above 20000
8	0.21	Above 20000
8	0.22	Above 20000
8	0.23	Above 20000
8	0.24	Above 20000
8	0.25	Above 20000

From the table above, I find when the decreasing of hidden nodes, the intital weight is increasing. I can't find the initial weight when the hidden node which is 8.

Below is polar_out.png when number of hidden node is 10 and initial weight is 0.126



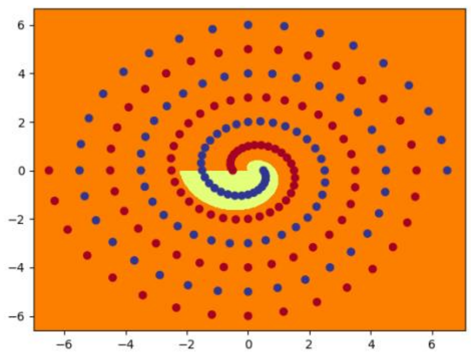
Below is polar_out.png when number of hidden node is 9 and initial weight is 0.151



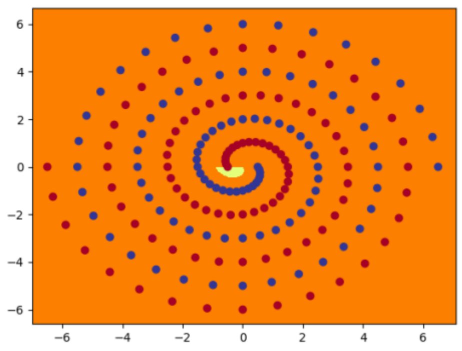
Q5. Using graph_output() as a guide, write a method called graph_hidden(net, layer, node) which plots the activation (after applying the tanh function) of the hidden node with the specified number (node) in the specified layer (1 or 2). (Note: if net is of type PolarNet, graph_output() only needs to behave correctly when layer is 1).

Answer: See the code in spiral.py in detail

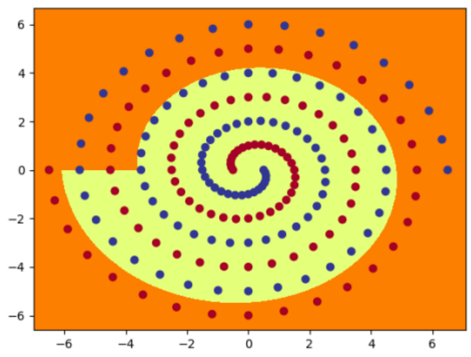
PolarNet:



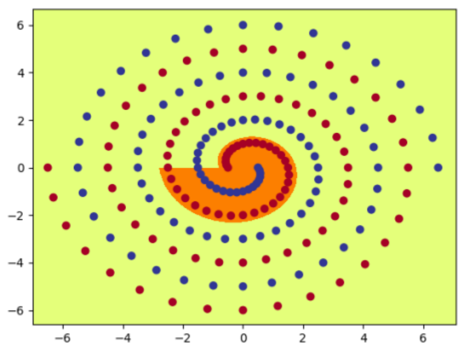
polar1_0.png



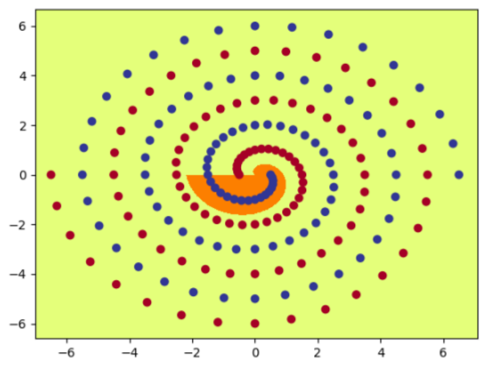
polar1_1png



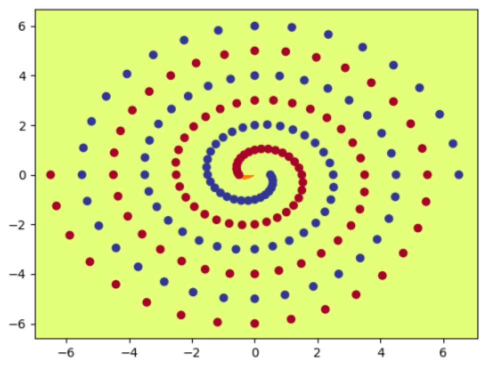
polar1_2.png



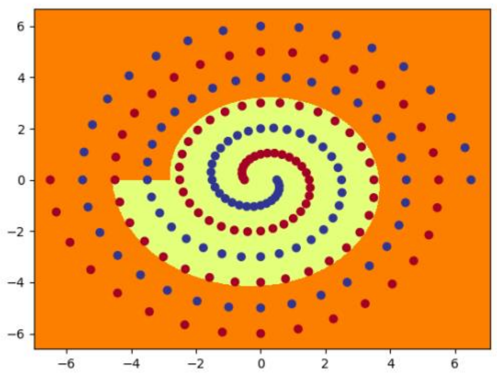
polar1_3.png

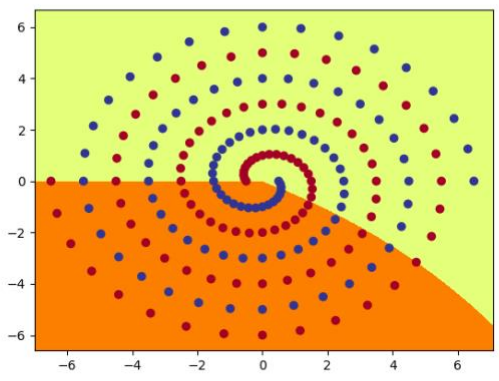


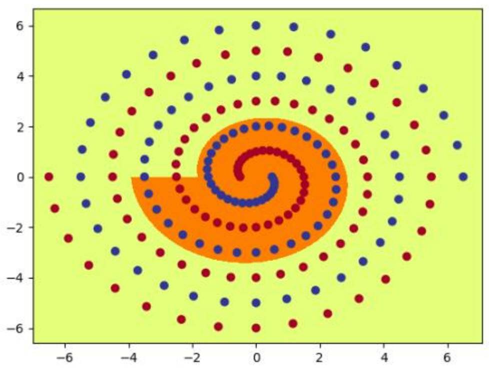
polar1_4.png

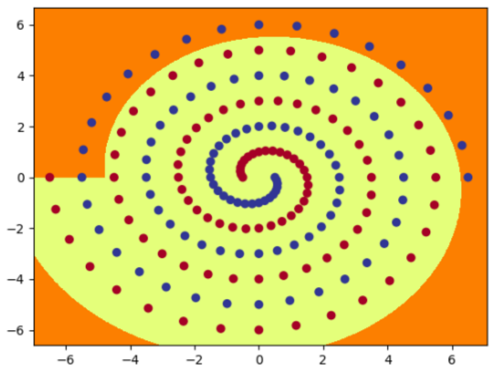


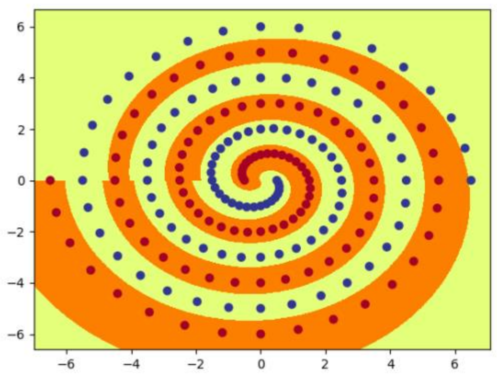
polar1_5.png


polar1_6.png

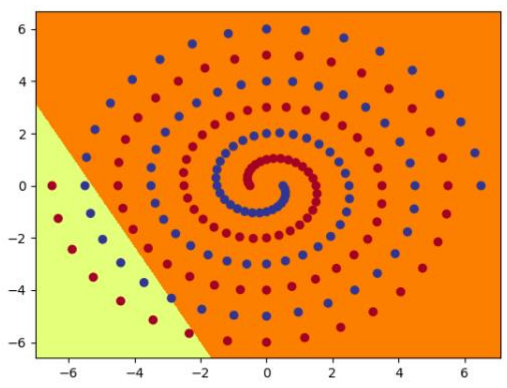

polar1_7.png

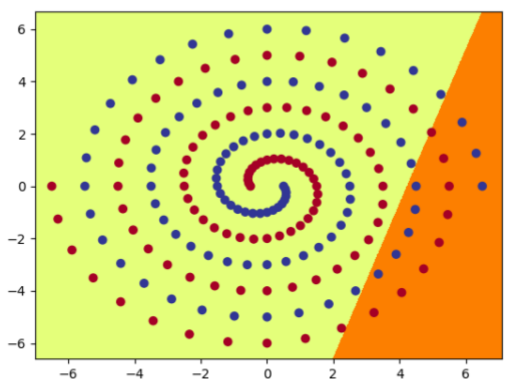

polar1_8.png

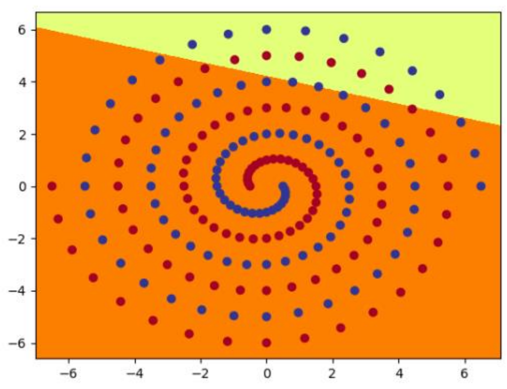

polar1_9.png

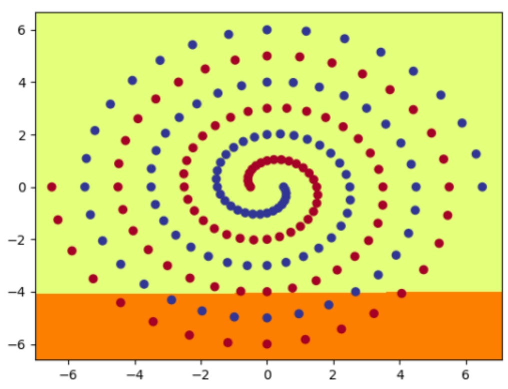

polar1_out.png

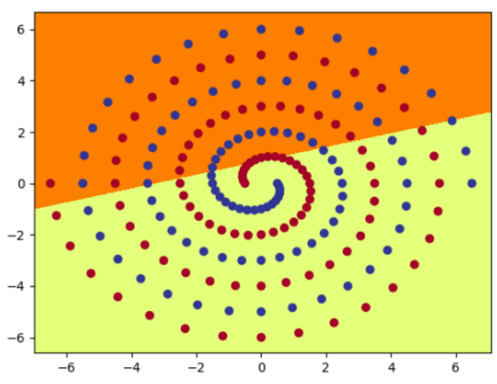
Rawnet: (when the hidden nodes is 10 and the weight is 0.126

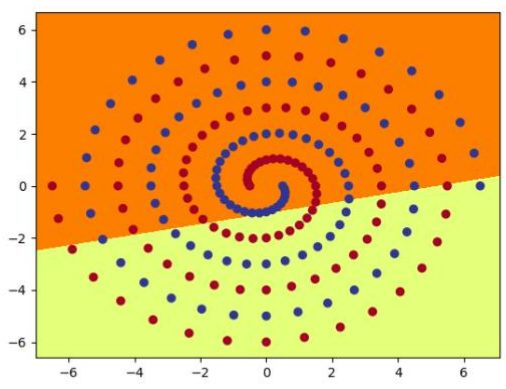

raw1_0.png

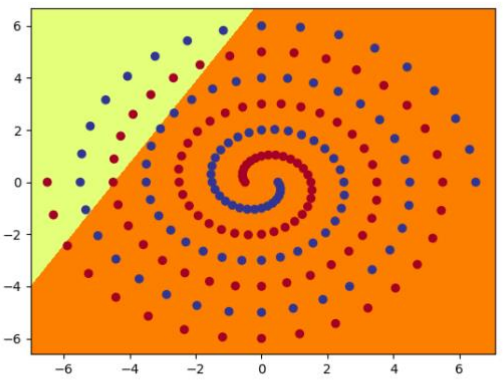

raw1_1png


raw1_2.png

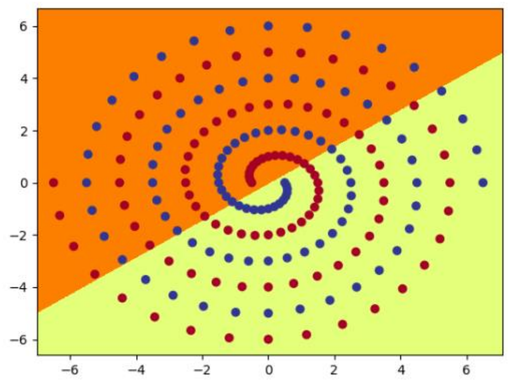

raw1_3.png


raw1_4.png

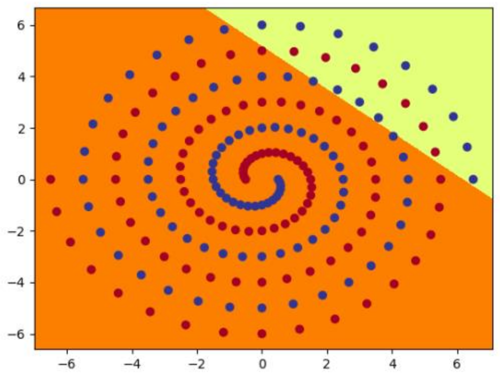

raw1_5.png



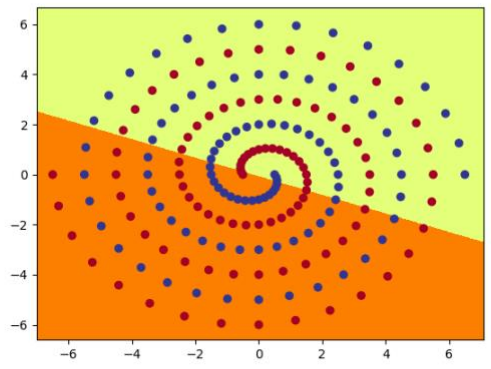
raw1_6.png



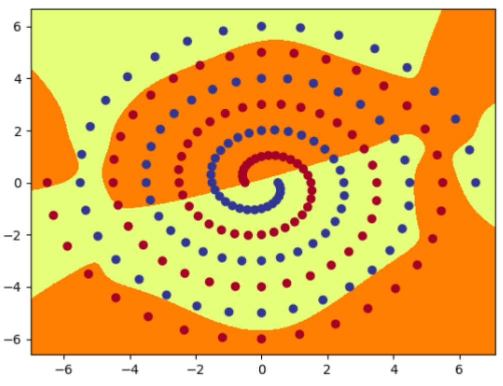
raw1_7.png



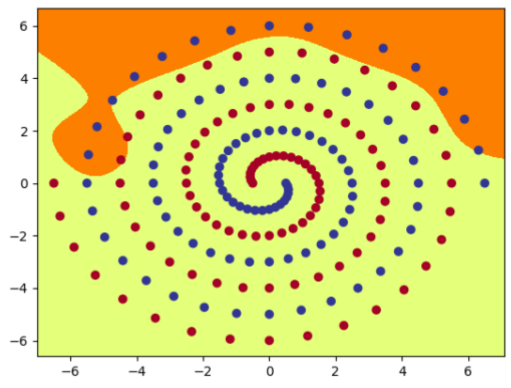
raw1_8.png



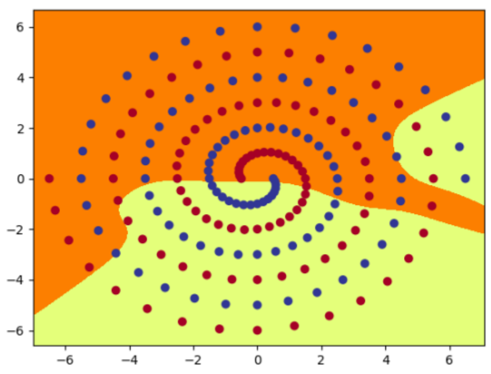
raw1_9.png

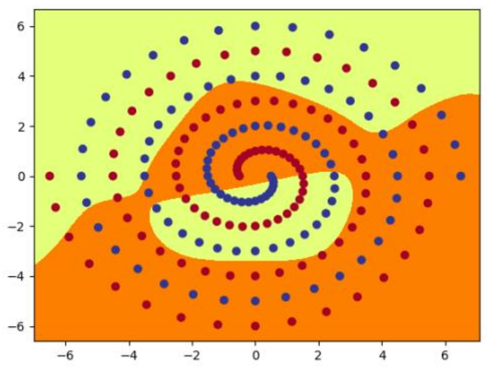


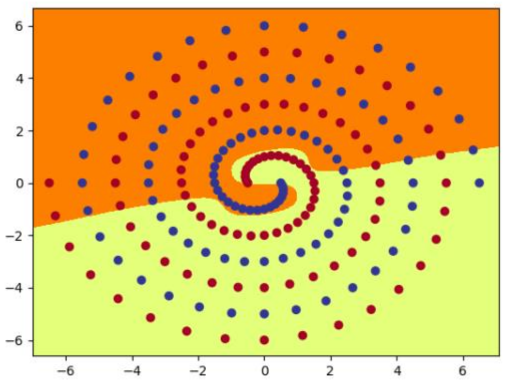
raw2_0.png

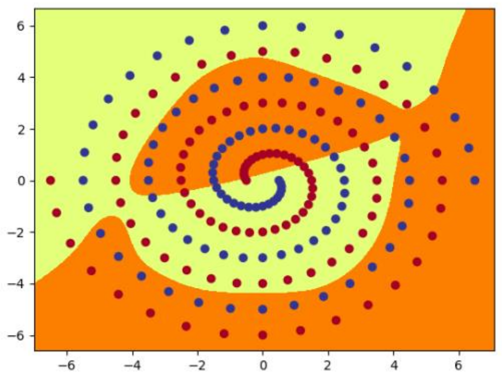


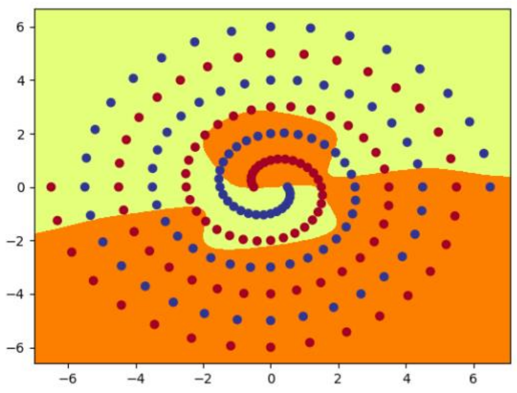
raw2_1.png

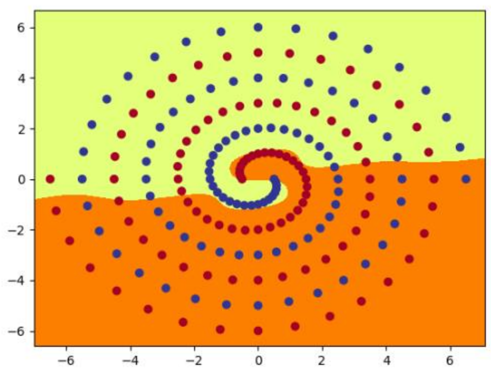

raw2_2.png

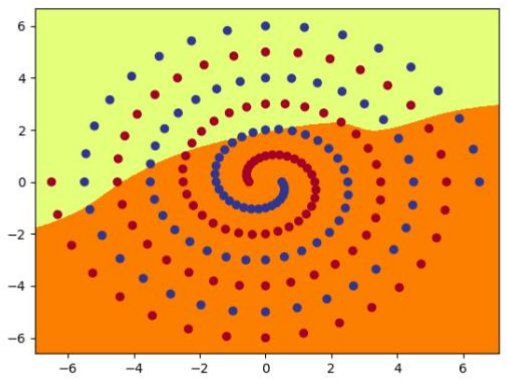

raw2_3.png


raw2_4.png

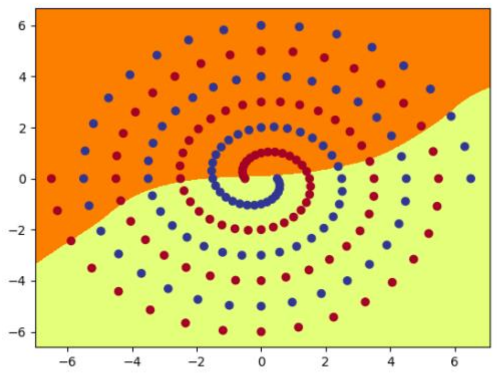

raw2_5.png


raw2_6.png

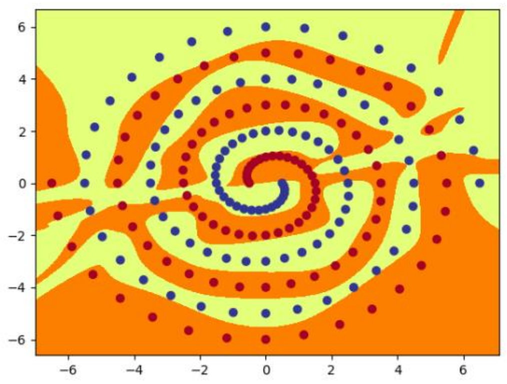

raw2_7.png



raw2_8.png



raw2_9.png



raw _out.png

Q6. Discuss what you have learned from this exercise, including the following points:
A. the qualitative difference between the functions computed by the hidden layer nodes PolarNet and RawNet, and a brief description of how the network uses these functions to achieve the classification
B. the effect of different values for initial weight size on the speed and success of learning for RawNet
C. you may like to also experiment with other changes and comment on the result - for example, changing batch size from 97 to 194, using SGD instead of Adam, changing tanh to relu, adding a third hidden layer, etc.

Answer:

A:

A.a: For PolarNet, function of each node is non-linear. x and y coordinates are first transformed into polar co-ordinates(r,a) and then put the new co-ordinates to a 1-layer fully connected neural network. After this transformed, it become easily to calculation. the hidden layer combine linear function and a tanh function to fit datas. Every hidden nodes can only learn a part of correct decision boundary, the weight make the network learn the complete classification boundary. I use the torch.sqrt function() and torch.atan2 function() to transform(x,y) co-ordinates to(r,a) co-ordinates.

A.b: For RawNet, first hidden layer of nodes give linear function decision boundary and the second hidden layer nodes learn non-linear function. It uses the original input and has two hidden layers and one output layer. An hidden layer is needed to realize more complicated mapping. In the first hidden layer, nodes make linear classification, and data spilt by line in each node. In the second layer, classification take place, but it is irregular in nodes, it always has one or more real data points. I use torch.linear function() and torch.tanh() function to address input(x, y) to num_hid nodes. Every hidden nodes can only learn a part of correct decision boundary, the weight make the network learn the complete classification boundary.

B.

In this part, I just change the “init” weight. I use time function to record the time

1.—init 0.1

When the initial weights are 0.1, RawNet sometimes will finish in 20000 epochs, sometimes will not.

I will do ten times. Consider most situation - the failure: the training accuracy is 50.52%. the time is 58.24 second

2.—init 0.11

When the initial weights are 0.11, RawNet sometimes will finish in 20000 epochs, sometimes will not.

I will do ten times. Consider most situation - the failure: the training accuracy is 92.78%. the time is 60.18 second

3.—init 0.12

When the initial weights are 0.12, RawNet sometimes will finish in 9400 - 3200 epochs.

I will do ten times. Consider most situation – the success: the training accuracy is 100.00%. the time is around 27.76 second to 16.94 second

4.—init 0.13

When the initial weights are 0.13, RawNet sometimes will finish in 13000 – 7800 epochs.

I will do ten times. Consider most situation – the success: the training accuracy is 100.00%. the time is around 39.15 second to 22.65 second

5.—init 0.14

When the initial weights are 0.14, RawNet sometimes will finish in 20000 epochs, sometimes will not.

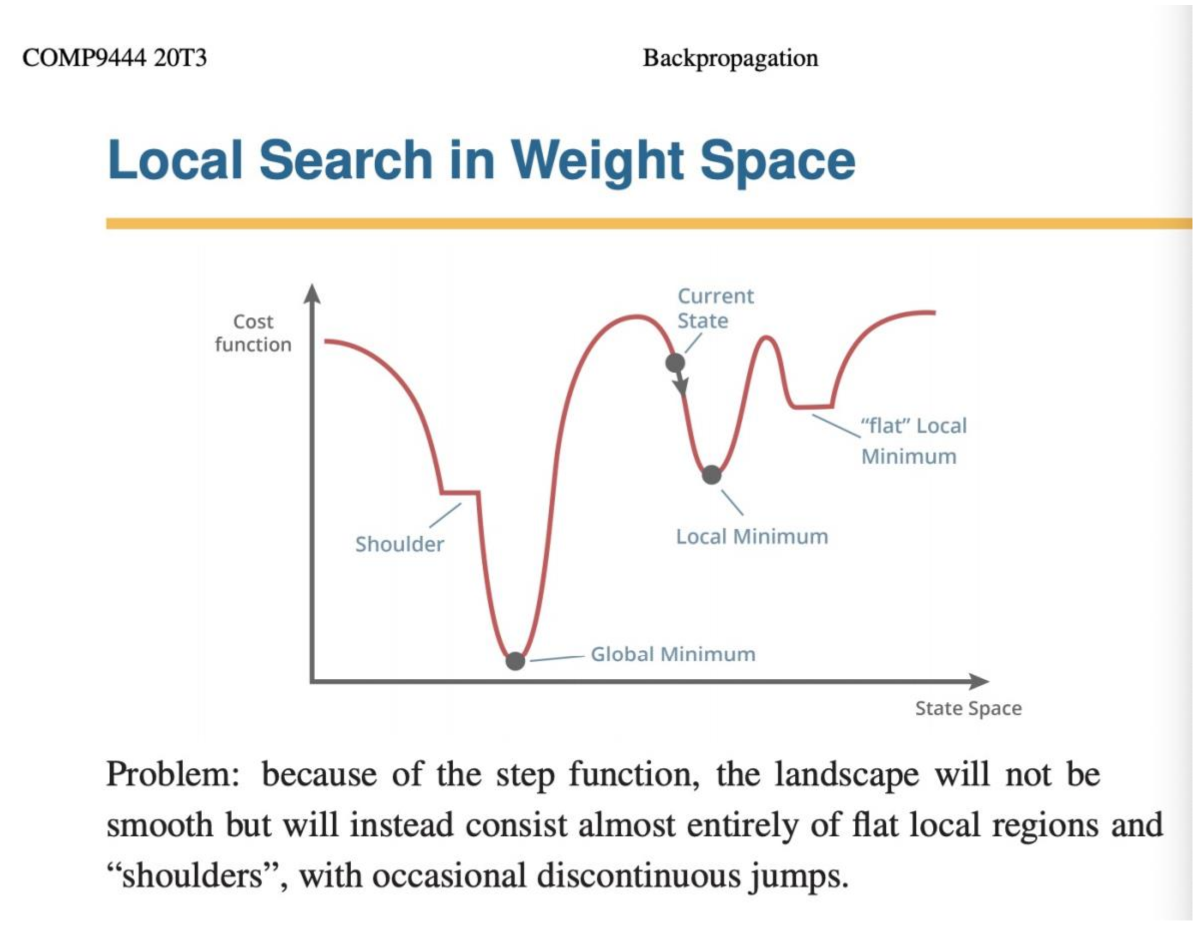
I will do ten times. Consider most situation – the failure: the training accuracy is 97.42%. the time is around 56.05 second

6. –init 0.15

When the initial weights are 0.15, RawNet sometimes will finish in 16400 – 4700 epochs.

I will do ten times. Consider most situation – the succrss: the training accuracy is 100.00%. the time is around 46.03 seconds to 13.30 seconds

Therefore, form the result, we can clearly see if initial weight is too small, the accuracy of these Rawnet will become low. When we increase the initial weight, it may takes some fluctuations to be failure:



COMP9444 20T3 Backpropagation

Local Search in Weight Space

Problem: because of the step function, the landscape will not be smooth but will instead consist almost entirely of flat local regions and "shoulders", with occasional discontinuous jumps.

C.

C.1 changing batch size from 97 to 194

1. PolarNet:

python3 spiral_main.py --net polar --hid 10

it will finish in 2000 epochs. the training accuracy is 100.00%. the time is 3.84 seconds

2. RawNet

python3 spiral_main.py --net raw --hid 10 it will finish in 2500 epochs. the training accuracy is 100.00%. the time is 5.15 seconds

C.2 using SGD instead of Adam

the code is:

```
optimizer = torch.optim.SGD(net.parameters(),lr=args.lr,
                                weight_decay=0.0001)
```

1. PolarNet:

python3 spiral_main.py --net polar --hid 10

it will not finish in 20000 epochs. the training accuracy is 56.70%. the time is 44.43 seconds

2. RawNet

python3 spiral_main.py --net raw --hid 10

it will not finish in 20000 epochs. the training accuracy is 59.79%. the time is 45.18 seconds

C.3 changing tanh to relu

1. PolarNet:

python3 spiral_main.py --net polar --hid 10

it will not finish in 20000 epochs. the training accuracy is 67.53%. the time is 48.89 seconds

2. RawNet:

python3 spiral_main.py -- net raw --hid 10

it will not finish in 20000 epochs. the training accuracy is 77.84%. the time is 54.83seconds

C.4 adding a third hidden layer

In RawNet, I add a hidden layer:

```
self.hid_to_out_xxxx =
nn.Linear(in_features=num_hid,out_features=num_hid,bias=True)  #the
second layer
```

1. PolarNet:

python3 spiral_main.py --net polar --hid 10 it will finish in 1600 epochs. the training accuracy is 100.00%. the time is 4.01 seconds

2. RawNet:

```
python3 spiral_main.py --net raw --hid 10
```

it will finish in 7600 epochs. the training accuracy is 100.00%. the time is 21.52 seconds

Therefore, from the result of changing batch size from 97 to 194, when the batch size is 97, the performance is worse than the one of batch size of 194. with the increasing of batch size will boost up the speed and decrease the epochs. So the batch size can not be small.

From the result of changing the result of using SGD optimizer, when we switch from Adam to SGD, the performance is worse than the Adam. Both of two modles can't finish in 20000 epochs. The reason could be some information lose in hidden layers.

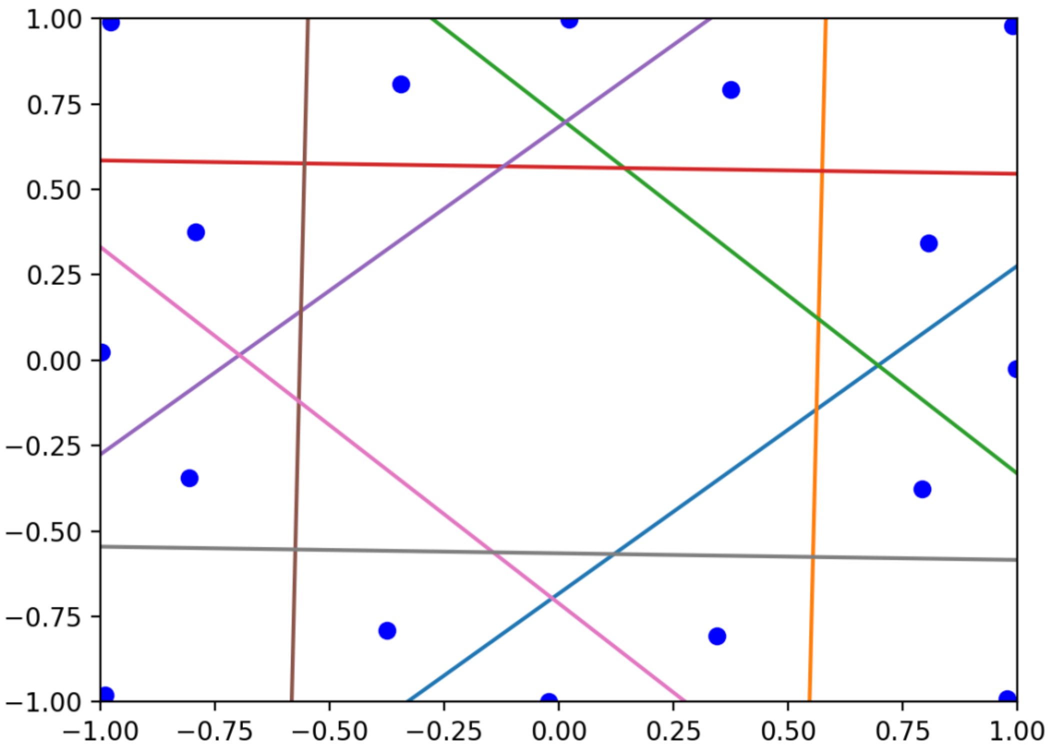
From the result of changing tanh to relu, I find when changing activation function, both of two modles can't finish in 20000 epochs, this maybe because will make some neurons outputs as 0 and these neurons can nerver be activated by the next layer. relu function may cause under-fitting. So it's not good enough for both of two moodles.

From the result of adding a third hidden layer, according to the result. Both of two moodles can finish in 20000 epochs and the time will increase. This may when we add more hidden layers, we need more time to acticvation function and more time to backpropagation.

Part 3: Hidden Unit Dynamics

Q1:Save the final image and include it in your report. Note that target is determined by the tensor star16 in encoder.py, which has 16 rows and 8 columns, indicating that there are 16 inputs and 8 outputs. The inputs use a one-hot encoding and are generated in the form of an identity matrix using torch.eye()

Answer:

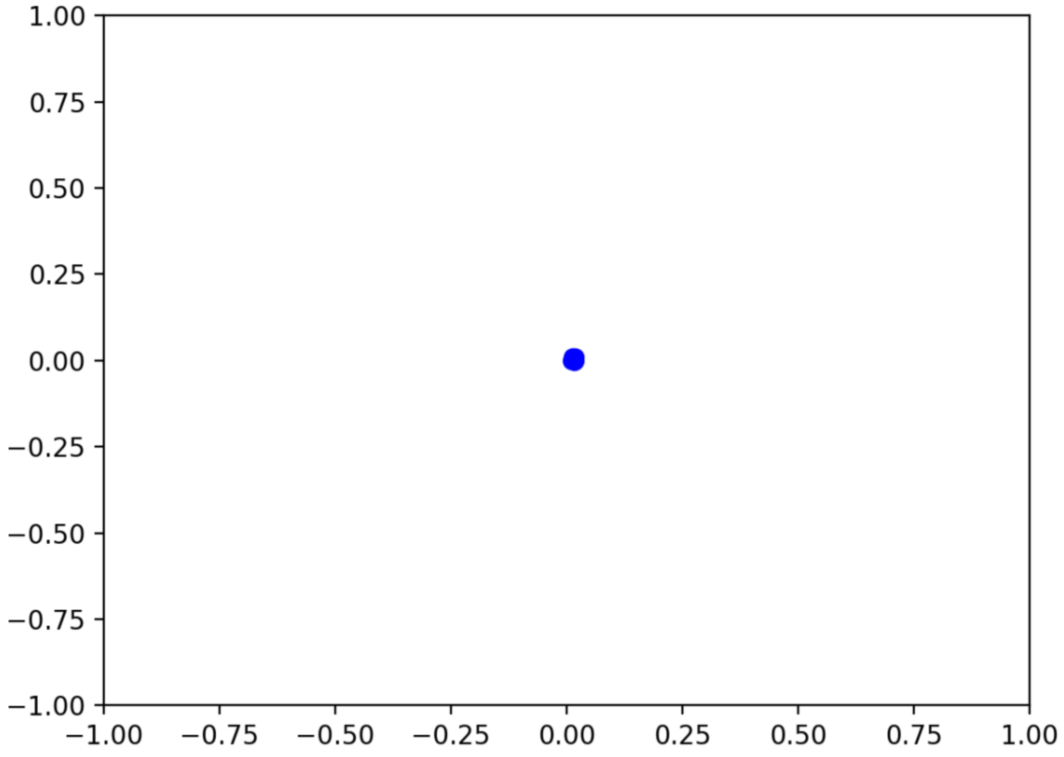


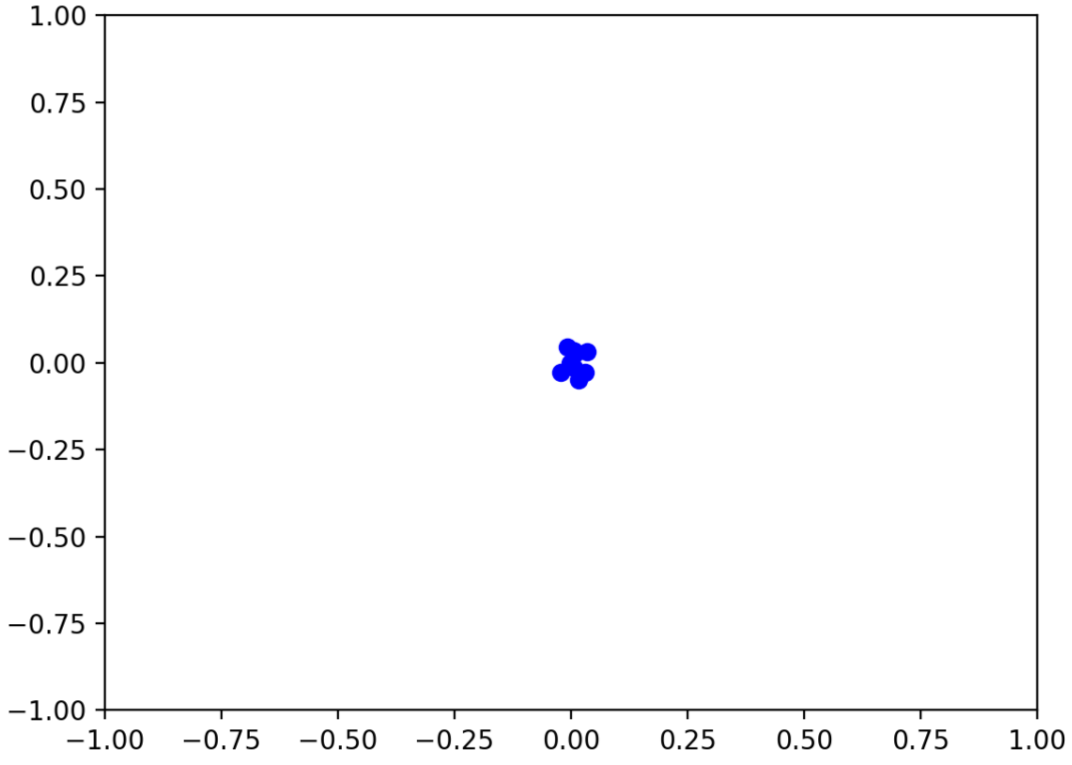
Q2: In this case, the task is a 9-2-9 encoder with both input and target determined by a one-hot encoding. Save the first eleven images generated (epochs 50 to 3000), plus the final image, and include them in your report. Describe in words how the hidden unit activations (dots) and output boundaries (lines) move as the training progresses.

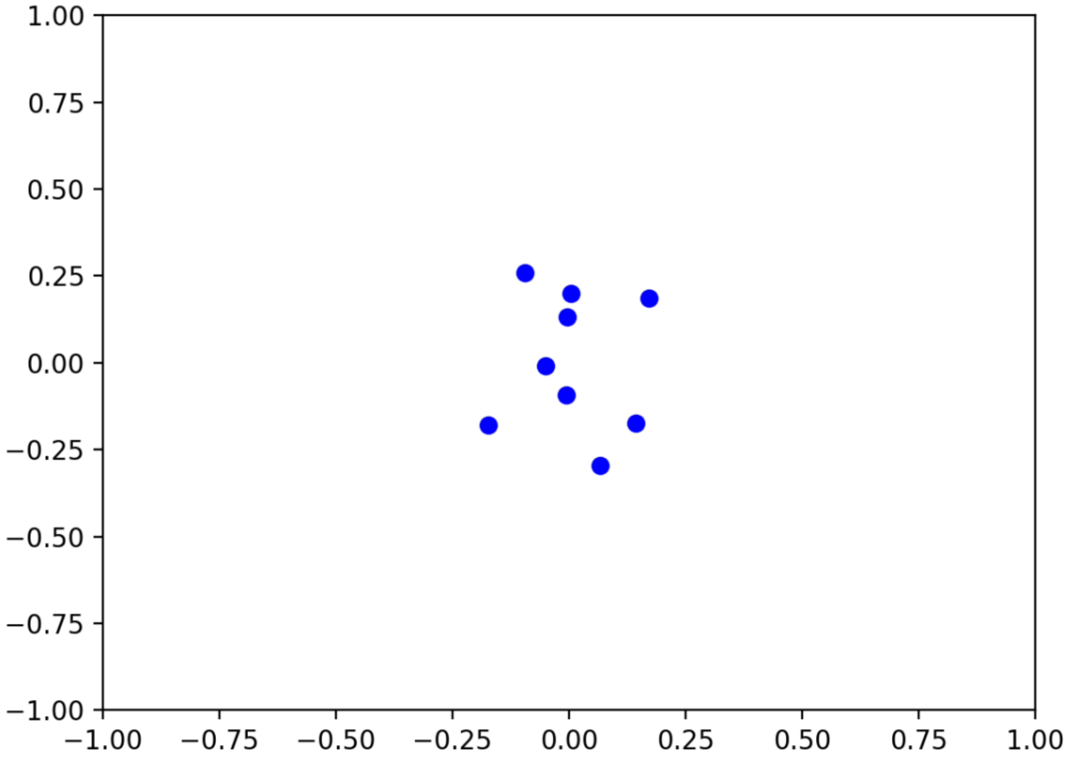
Answer:

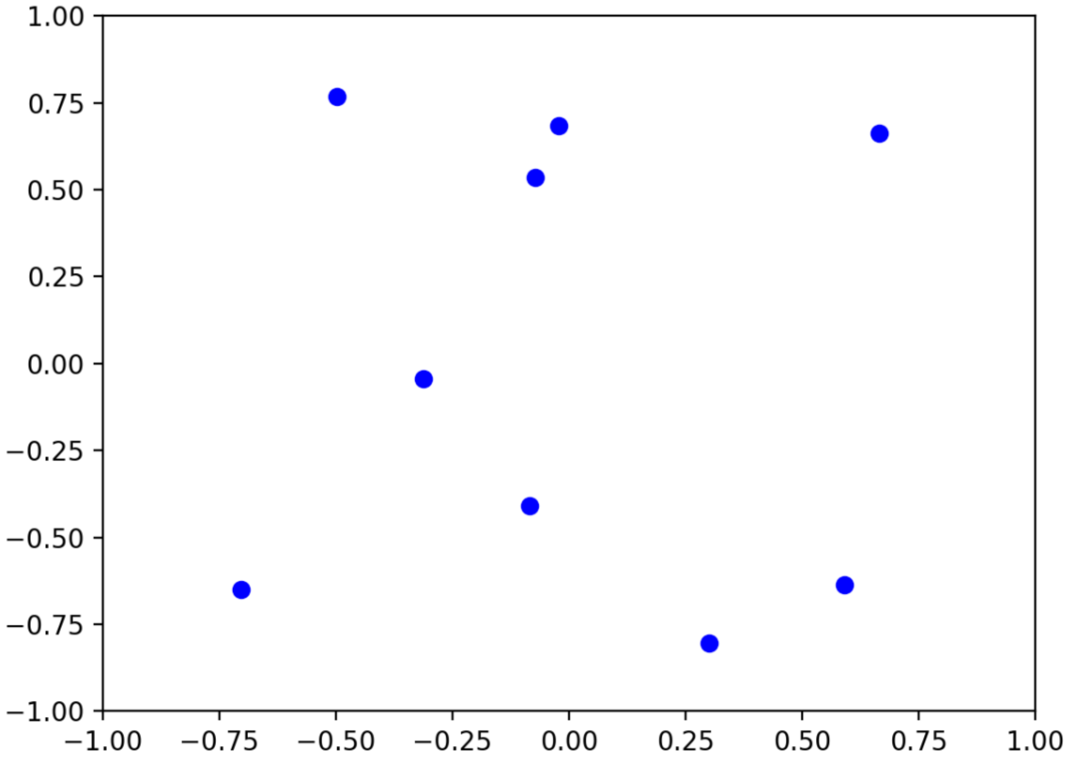
the first eleven images generated (epochs 50 to 3000), plus the final image are shown below.

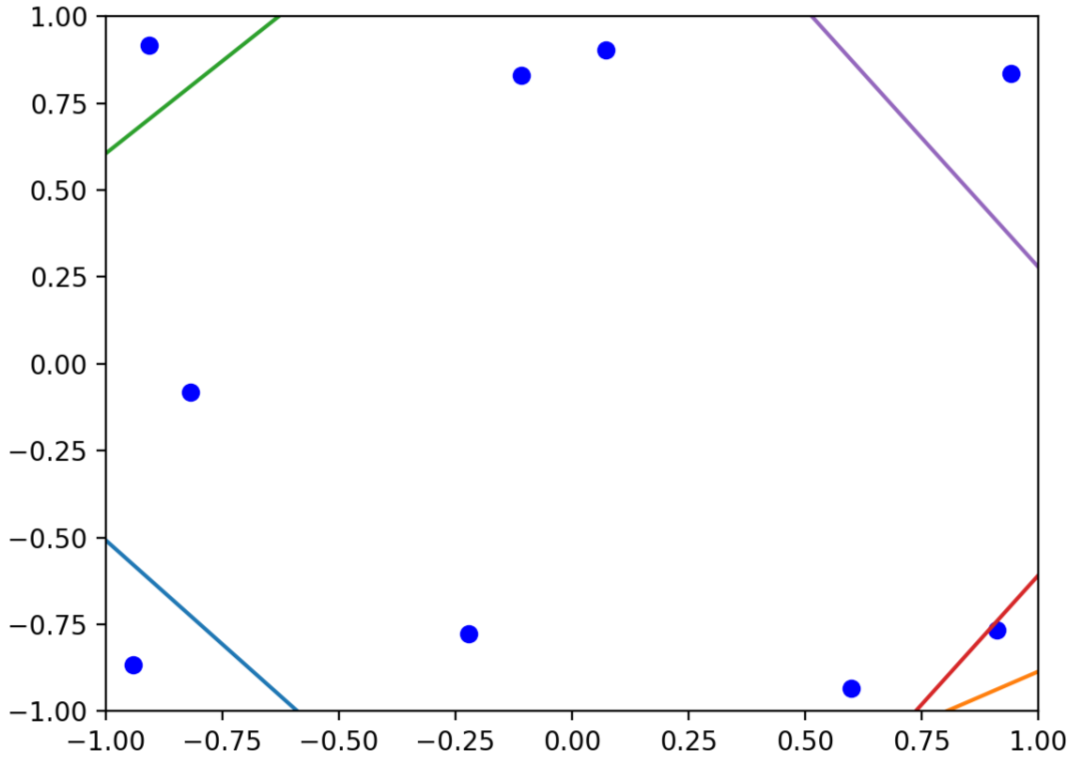
the hidden unit activations are separated by boundaries from the centre of picture to discontinuous, discreted and surrounding of picture.

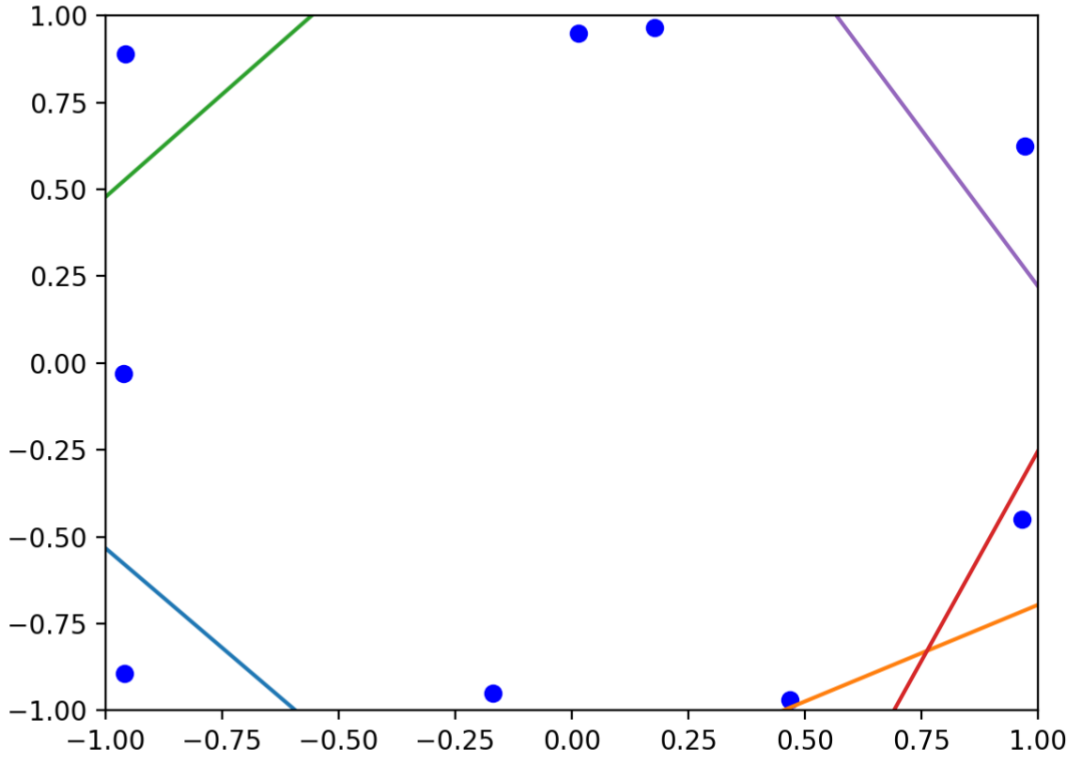

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

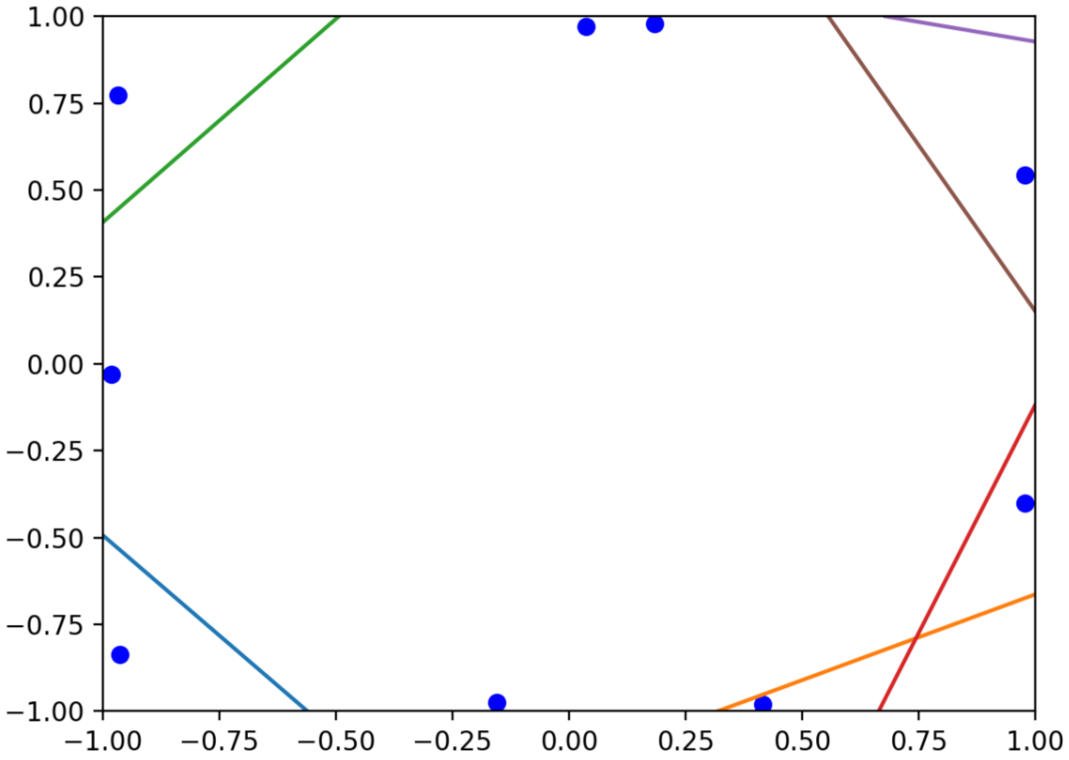

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

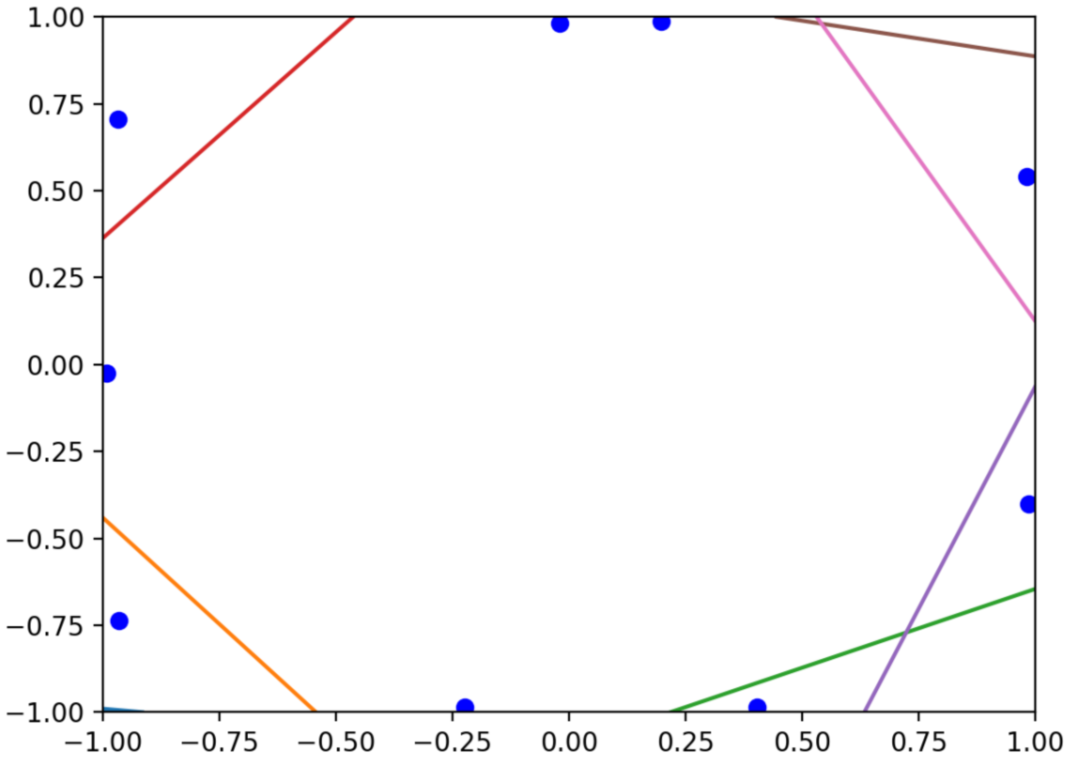

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

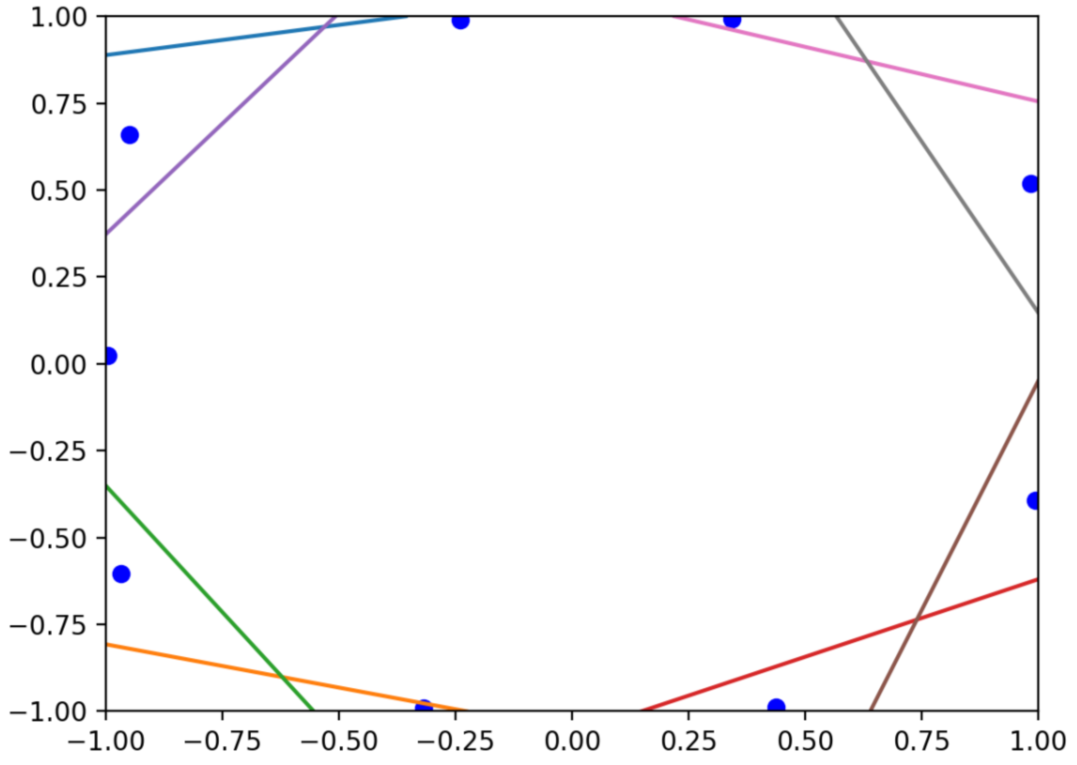

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

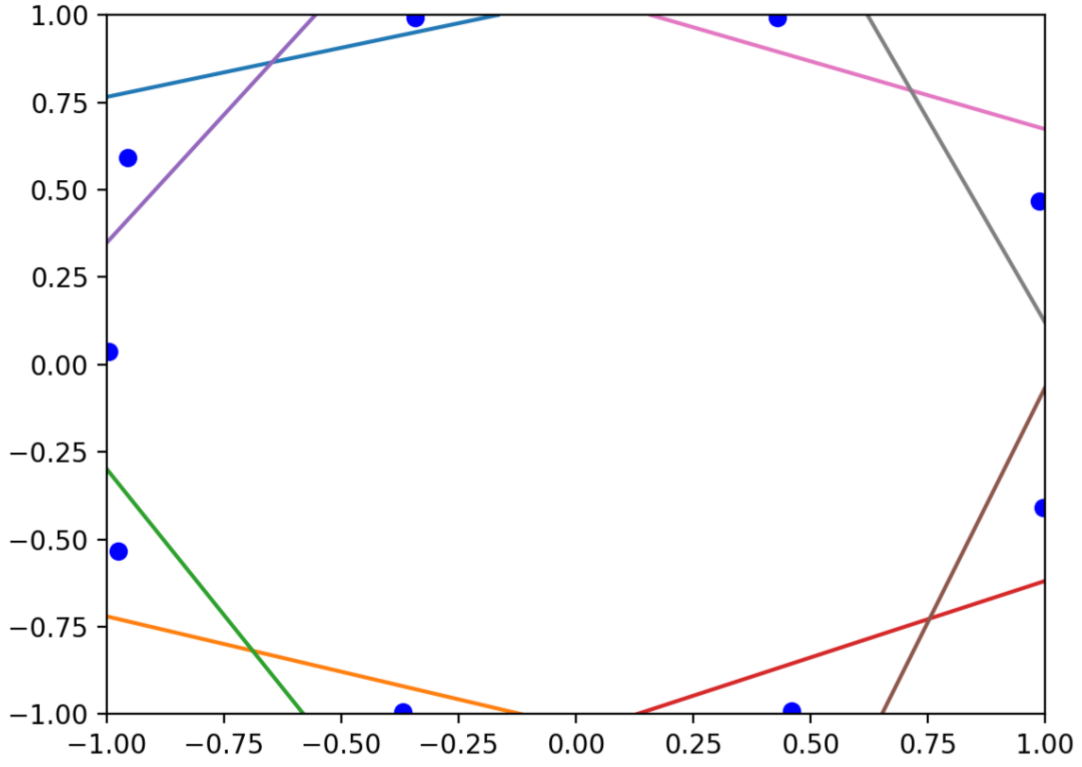

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

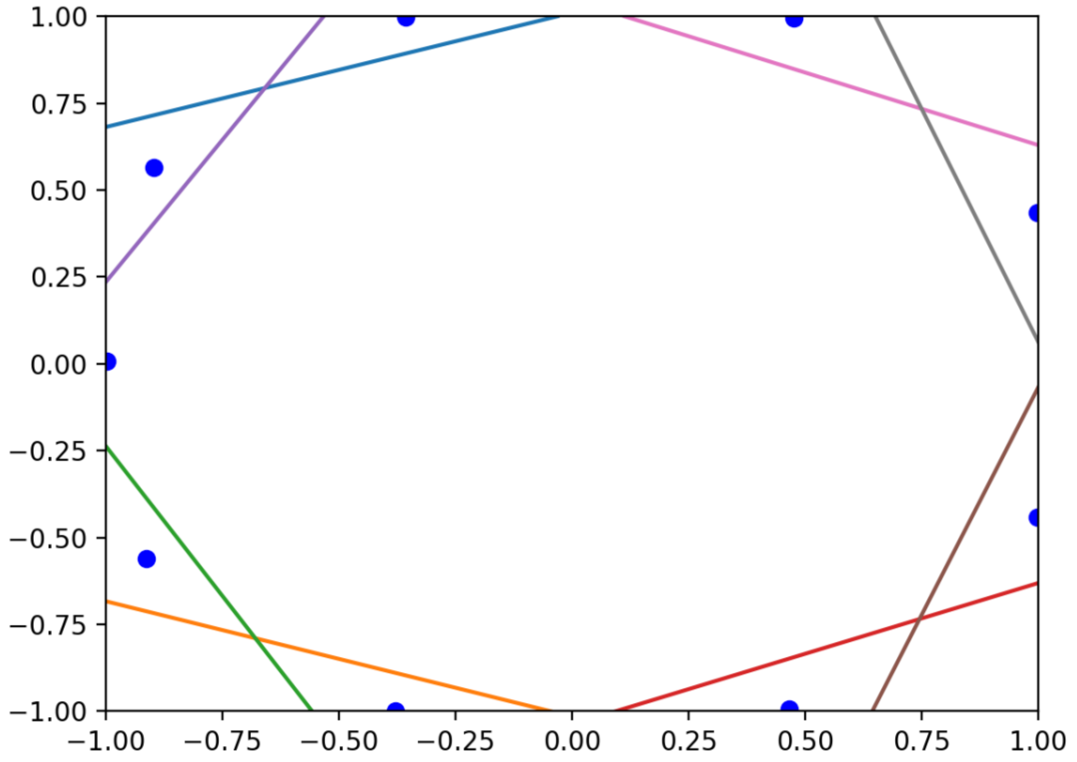

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

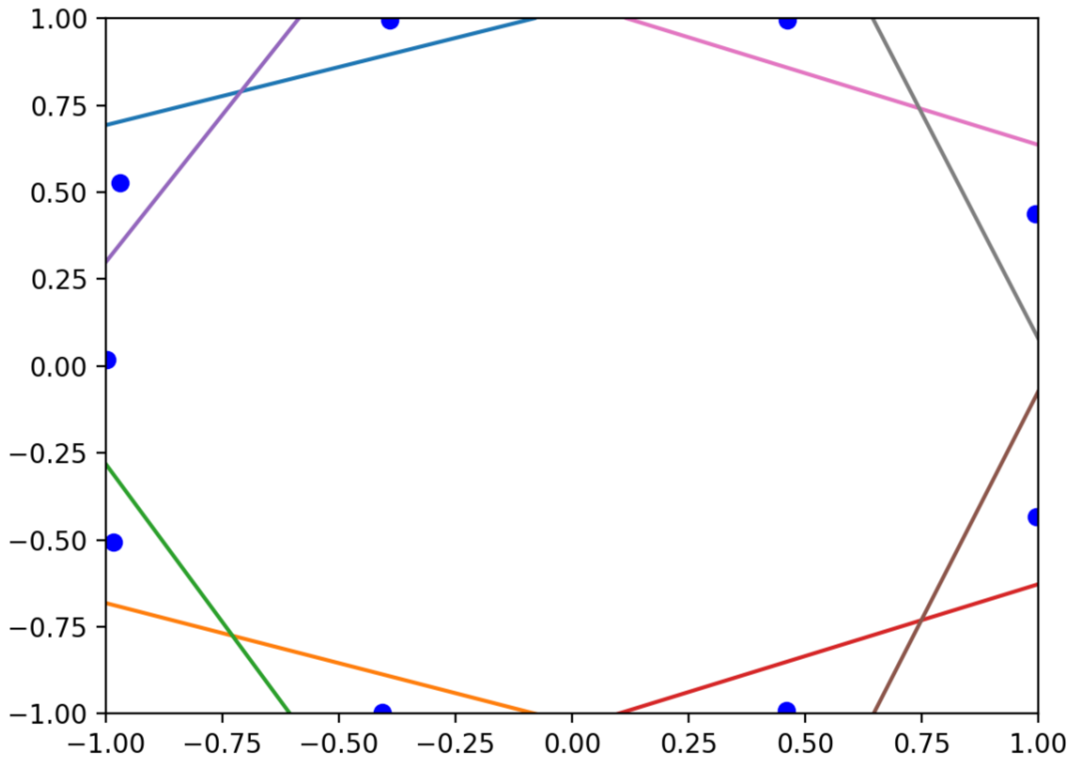

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


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


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

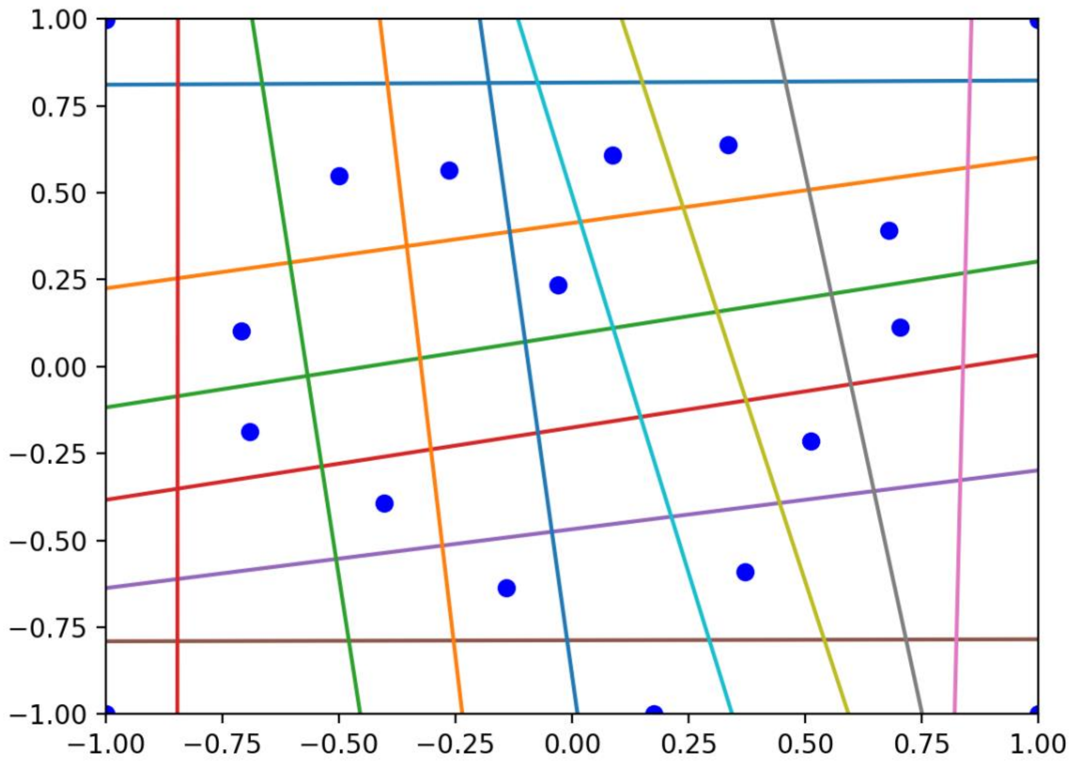

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


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


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

Q3.Create by hand a dataset in the form of a tensor called heart18 in the file encoder.py which, when run with the following command, will produce an image essentially the same as the heart shaped figure shown below (but possibly rotated). Your tensor should have 18 rows and 14 columns. Include the final image in your report, and include the tensor heart18 in your file encoder.py

Answer: See the code in spiral.py in detail and the picture are shown below.

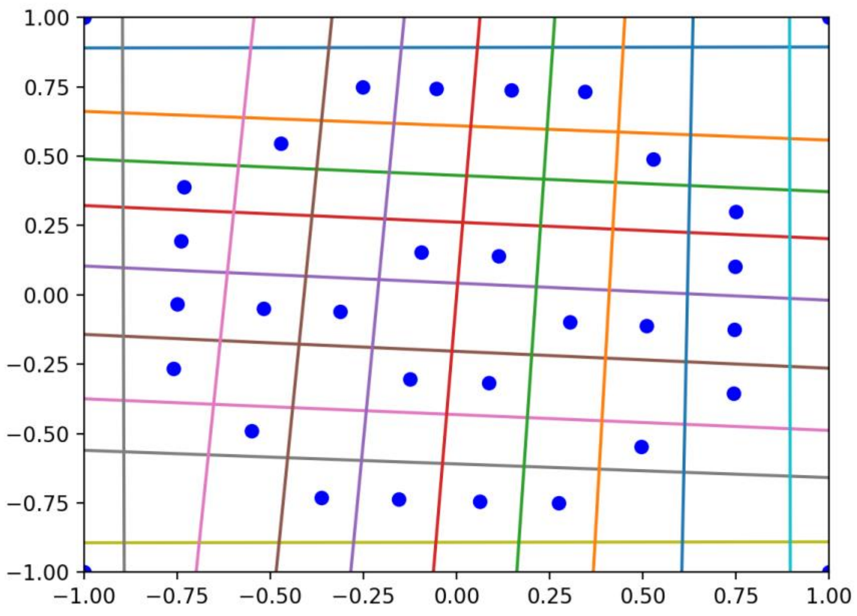


Q4. Create training data in tensors target1 and target2, which will generate two images of your own design, when run with the command python3 encoder_main.py --target=target1 (and similarly for target2). You are free to choose the size of the tensors, and to adjust parameters such as --epochs and --lr in order to achieve successful learning. Marks will be awarded based on creativity and artistic merit. Include the final images in your report, and include the tensors target1 and target2 in your file encoder.py

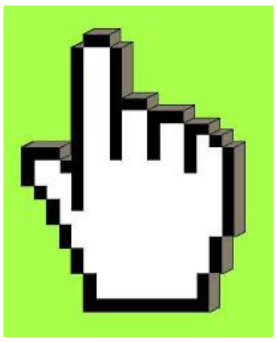
Answer: See the code in spiral.py in detail and the picture are shown below.

First, in target1, I design a pokemon ball, the original picture is

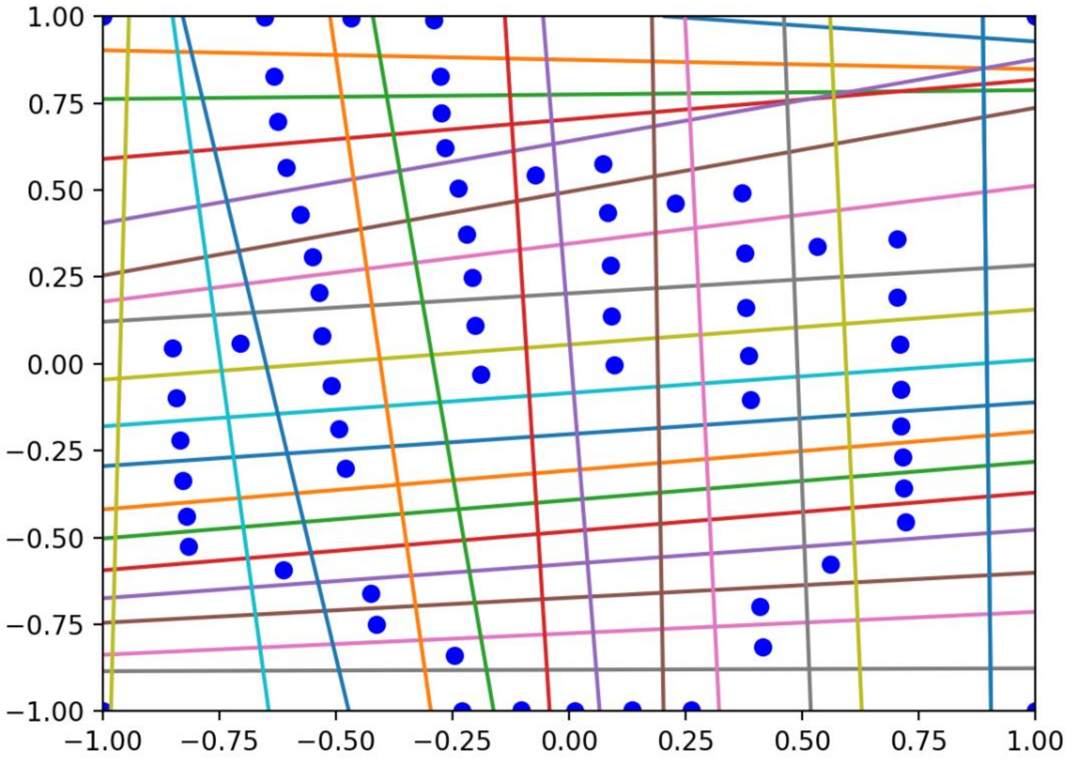
Then the picture used by the network is



Second, in target2, I design a gesture of clicking, the original picture is



Then the picture used by the network is


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