COMP9444 Project1

Part 1

1:

The screen shot for the confusion matrix and final accuracy:

```
[[768.   6.   8.  12.  30.  63.   2.  61.  30.  20.]
 [  6. 669. 108.  18.  27.  24.  59.  13.  26.  50.]
 [  6.  62. 693.  26.  28.  22.  45.  35.  45.  38.]
 [  5.  37.  60. 755.  16.  59.  15.  18.  24.  11.]
 [ 60.  55.  76.  22. 621.  19.  32.  36.  21.  58.]
 [  8.  27. 125.  17.  19. 726.  26.   9.  33.  10.]
 [  5.  22. 147.  10.  26.  26. 719.  22.  10.  13.]
 [ 17.  29.  25.  12.  85.  17.  55. 623.  87.  50.]
 [ 11.  38.  97.  44.   7.  30.  42.   6. 702.  23.]
 [  9.  53.  88.   4.  54.  31.  21.  29.  36. 675.]]

Test set: Average loss: 1.0095, Accuracy: 6951/10000 (70%)
```

2:

I choose the number of hidden nodes to be 200.And it achieves the 84% final accuracy.

```
[[849.   3.   1.   7.  27.  35.   4.  40.  30.   4.]
 [  5. 816.  31.   4.  17.  10.  63.   6.  19.  29.]
 [  8.  10. 857.  42.  10.  13.  21.   8.  17.  14.]
 [  3.   9.  31. 923.   2.  10.   3.   1.   8.  10.]
 [ 43.  31.  23.   6. 808.   8.  32.  17.  18.  14.]
 [  9.  12.  78.  10.  11. 825.  28.   1.  21.   5.]
 [  3.  11.  55.   8.  14.   5. 887.   6.   3.   8.]
 [ 21.  10.  27.   3.  22.   6.  37. 818.  28.  28.]
 [ 10.  25.  35.  45.   2.   7.  32.   3. 834.   7.]
 [  3.  19.  58.   4.  26.   7.  21.  15.  10. 837.]]

Test set: Average loss: 0.5081, Accuracy: 8454/10000 (85%)
```

3:

I use two layer convolutional layers plus two fully connected layers with relu function as activation function after each layer node and log softmax at the output node.I don't change any meta parameter.

Because of the low efficiency of running code on my macbook, I use another windows10 laptop with NVIDIA GPU to run the program.

```
[[952.   2.   2.   0.  26.   5.   0.   7.   2.   4.]
 [  2. 941.   9.   1.   9.   0.  27.   1.   3.   7.]
 [  9.   9. 891.  31.  10.  11.  16.   8.   7.   8.]
 [  1.   3.  14. 953.   2.   8.   9.   4.   2.   4.]
 [ 24.  11.   5.  10. 913.   7.  12.   3.   9.   6.]
 [  6.   8.  34.   7.   3. 913.  19.   3.   3.   4.]
 [  2.   2.  11.   1.   4.   1. 973.   1.   1.   4.]
 [ 14.  11.  10.   0.   3.   2.  12. 932.   4.  12.]
 [  6.  13.   7.   7.   3.   4.   5.   1. 952.   2.]
 [ 14.   4.  13.   6.   6.   2.   4.   2.   9. 940.]]

Test set: Average loss: 0.3263, Accuracy: 9360/10000 (94%)
```

4

a:

The accuracy rate for NetLin after 10 training epochs is around 70%.

The accuracy rate for NetFull after 10 training epochs is around 85%.

The accuracy rate for NetConv after 10 training epochs is around 94%.

Among those three models,the NetConv model achieves the highest accuracy rate on test set(94%),followed by NetFull model,which can reach 85% overall accuracy rate.The NetLin model has the lowest accuracy rate on test set,only 70%.

b:

NetLin:

Looking at confusion matrix for question1,I find ma is most likely to be mistaken for su,because the biggest number except numbers on the diagonal in the confusion matrix is 147,which located at the seventh row and the third column.This means there are 147 cases that seventh row character(ma) is mistaken for the third row character(su).

NetFull:

Ha is most likely to be mistaken for su.

Looking at confusion matrix of q2,the biggest number except diagonal is 78,which is at the sixth row and the third column.

NetConv:

Ha is most likely to be mistaken for su.

Looking at confusion matrix for q3, the biggest number except diagonal is 34,also located at the sixth row and the third column.

c:

From the first three questions, we already know CNN has better performance on recognizing handwritten Hiragana symbols,I want to do a better job on CNN,so the modification for this question is all based on the structure of Q3.

Firstly I don't change any architecture stuff from Q3,I only modify meta parameters,changing optimizer to be Adam, since Adam Optimizer is an extension of SGD, which can replace the classic stochastic gradient descent method to update network weights more effectively.Also I use different learning rate,which are 0.1,0.01,0.001,0.0005,0.0001,respectfully.And I got the following result.

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

Test set: Average loss: 2.3038, Accuracy: 1000/10000 (10%)
```

Lr = 0.1

```
[[847.   5.   7.   1.  40.  13.  15.  28.  31.  13.]
 [  4. 842.  27.   4.  23.   8.  55.   4.  11.  22.]
 [  5.  14. 821.  26.  18.  16.  56.  14.   8.  22.]
 [  3.   4.  20. 893.  15.  16.  40.   0.   4.   5.]
 [ 30.  16.  43.  16. 818.   9.  28.  16.  12.  12.]
 [  4.  13.  67.  14.  11. 832.  31.   2.  12.  14.]
 [  6.   6.  43.   8.  16.   2. 907.   6.   1.   5.]
 [ 11.   8.  29.   7.  23.   4.  61. 827.  15.  15.]
 [ 18.  22.  76.  36.  24.   5.  23.  17. 763.  16.]
 [  5.  16.  38.   3.  27.   4.  30.   4.  11. 862.]]

Test set: Average loss: 0.7937, Accuracy: 8412/10000 (84%)
```

Lr = 0.01

```
[[968.   1.   5.   1.  13.   0.   0.   4.   3.   5.]
 [  4. 951.   9.   0.   7.   2.  14.   0.   2.  11.]
 [ 11.  11. 900.  27.   9.   7.  14.  13.   1.   7.]
 [  1.   2.  15. 950.   4.   7.  10.   6.   3.   2.]
 [ 25.  11.   5.   8. 922.   4.   7.   9.   6.   3.]
 [  8.  18.  46.   8.   4. 888.  11.   2.   7.   8.]
 [  5.  18.  21.   3.  11.   0. 936.   3.   1.   2.]
 [ 19.   3.  13.   2.   8.   6.   2. 927.   6.  14.]
 [ 18.  19.   6.   7.  13.   2.   2.   2. 930.   1.]
 [ 11.   6.  12.   1.  10.   1.   5.  15.   7. 932.]]

Test set: Average loss: 0.5028, Accuracy: 9304/10000 (93%)
```

Lr = 0.001

```
[[968.   1.   3.   0.  14.   2.   0.   7.   4.   1.]
 [  0. 957.   5.   0.   6.   0.  21.   0.   5.   6.]
 [  7.   4. 898.  35.  10.  12.  23.   3.   2.   6.]
 [  1.   0.  11. 971.   2.   4.   4.   4.   1.   2.]
 [ 13.   5.   1.  10. 941.   3.  10.   2.  10.   5.]
 [  4.   3.  28.   7.   3. 931.  12.   0.   6.   6.]
 [  2.   3.   6.   1.   4.   3. 978.   2.   1.   0.]
 [  6.   7.   1.   0.   4.   5.  11. 954.   5.   7.]
 [  4.  12.   4.   2.   3.   5.   3.   2. 965.   0.]
 [  6.  11.   8.   2.   5.   3.   6.   3.   5. 951.]]

Test set: Average loss: 0.3165, Accuracy: 9514/10000 (95%)
```

Lr = 0.0005

```
[[951.   1.   1.   1.  21.   3.   0.  16.   4.   2.]
 [  2. 942.   5.   1.   6.   0.  32.   0.   3.   9.]
 [  9.   6. 918.  21.   6.   6.  17.   3.   5.   9.]
 [  1.   1.  16. 960.   2.   4.   8.   2.   1.   5.]
 [ 13.   7.   4.  10. 931.   0.  15.   8.   7.   5.]
 [  2.   5.  48.  12.   5. 902.  13.   2.   2.   9.]
 [  3.   4.  13.   3.   7.   0. 967.   2.   0.   1.]
 [ 12.   6.   7.   2.   7.   2.   7. 937.   5.  15.]
 [  8.  10.   9.   5.   6.   1.   1.   1. 956.   3.]
 [ 12.  11.   3.   7.   7.   2.   5.   0.   6. 947.]]

Test set: Average loss: 0.2900, Accuracy: 9411/10000 (94%)
```

Lr = 0.0001

From above experiments,we can get when learning rate is 0.0005,the network from Q3 with Adam optimizer has the best accuracy rate within 10 training epochs,above

95%.From the paper Deep Learning for Classical Japanese Literature,the Keras Simple CNN Benchmark has 95.12% accuracy rate on KMNIST set,which is quite similar to my result.

Then I add another max pooling layer after two convolutional layer,with kernel size of 6.I use different learning rate(0.01,0.005,0.001,0.0005) to test.Following figures are results.

```
[[911.   3.   3.   0.  32.  12.   1.  17.  16.   5.]
 [  0. 901.  11.   0.  39.   2.  24.   0.   5.  18.]
 [  4.   9. 824.  72.  39.   7.  21.   5.   8.  11.]
 [  3.   2.   3. 957.  18.   7.   2.   2.   4.   2.]
 [ 23.  10.   1.   5. 906.   5.  15.   6.  11.  18.]
 [  2.  15.  39.  10.  38. 844.  20.   0.   6.  26.]
 [  2.  15.  11.   9.  33.   3. 915.   7.   3.   2.]
 [ 12.  11.  10.   8.  46.   2.  14. 861.   6.  30.]
 [  3.   8.   6.   6.  47.   0.   6.   1. 919.   4.]
 [  4.   4.   4.   3.  51.   2.  11.   2.   2. 917.]]

Test set: Average loss: 0.5651, Accuracy: 8955/10000 (90%)
```

Lr = 0.01

```
[[940.   0.   2.   1.   9.   3.   2.  26.  13.   4.]
 [  1. 933.   9.   0.   2.   4.  38.   6.   2.   5.]
 [  5.   2. 915.  20.  14.  16.  15.   6.   2.   5.]
 [  0.   2.   4. 941.   4.  39.   3.   2.   2.   3.]
 [ 16.  10.   4.   6. 919.   7.   7.   9.  12.  10.]
 [  4.   5.  16.   6.   0. 956.   5.   1.   3.   4.]
 [  0.   1.  15.   5.   6.   5. 956.   8.   1.   3.]
 [  5.   6.   6.   1.   3.   9.   6. 941.   6.  17.]
 [  4.   5.   9.   2.  14.   2.   5.   2. 955.   2.]
 [  6.   7.  11.   2.   4.   1.   4.  11.   3. 951.]]

Test set: Average loss: 0.3137, Accuracy: 9407/10000 (94%)
```

Lr = 0.005

```
[[951.   1.   1.   0.  34.   1.   0.  10.   0.   2.]
 [  0. 959.   2.   0.  11.   0.  16.   1.   4.   7.]
 [  8.  10. 930.  15.   9.   4.   6.   4.   5.   9.]
 [  0.   0.  11. 979.   3.   2.   3.   1.   1.   0.]
 [ 15.   2.   2.   4. 962.   1.   4.   5.   1.   4.]
 [  5.  12.  31.   6.   3. 920.   8.   2.   7.   6.]
 [  1.   5.   8.   0.  11.   0. 971.   2.   0.   2.]
 [ 14.   4.   5.   0.  15.   0.   5. 945.   2.  10.]
 [  5.   9.   3.   1.  33.   0.   5.   1. 941.   2.]
 [  6.   7.   6.   0.   9.   0.   0.   2.   2. 968.]]

Test set: Average loss: 0.2717, Accuracy: 9526/10000 (95%)
```

Lr = 0.001

```
[[949.   3.   2.   2.  20.   7.   0.  11.   1.   5.]
 [  1. 963.   3.   1.   2.   1.  13.   2.   4.  10.]
 [  7.   4. 915.  31.   5.   7.  16.   3.   1.  11.]
 [  0.   1.   8. 986.   0.   1.   1.   1.   1.   1.]
 [ 11.   9.   7.  14. 930.   3.   8.   6.   7.   5.]
 [  1.   8.  27.   7.   2. 933.   2.   3.   1.  16.]
 [  3.  16.  31.   5.   4.   2. 932.   2.   0.   5.]
 [  2.   4.   2.   0.   1.   1.   8. 941.   3.  38.]
 [  0.  10.   6.   9.   5.   1.   0.   0. 963.   6.]
 [  5.   6.   4.   0.   2.   0.   0.   0.   2. 981.]]

Test set: Average loss: 0.2617, Accuracy: 9493/10000 (95%)
```

Lr = 0.0005

From above four tests,when learning rate is 0.001,we get the highest accuracy rate,also 95%,which indicates that adding a max pooling layer doesn't extend the maximum accuracy rate it can reach.
The paper Deep Learning for Classical Japanese Literature shows PreActResNet-18 and its extension model can achieve much higher accuracy,the highest is PreActResNet-18 + Manifold Mixup model,which can reach 98.83% accuracy rate on KMINST set.But due to the hardware and time limitation,I can not realize it on my own.

In conclusion,for image classification problem, CNN architecture much outperforms the traditional linear models,in terms of accuracy rate.Also Tuning meta parameters is quite essential and time-consuming.The evolution of hardware provides us much more computation power that can train much complex model faster.Overall,the tremendous development of neural networks has brought us to a new level of accuracy in classification problems.

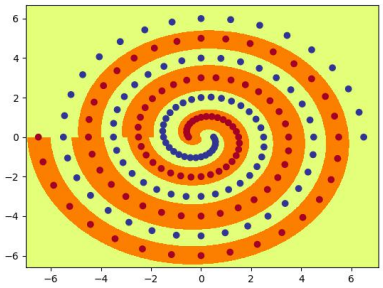
Part2:

2:

The minimum number of hidden node is 7.When I set hidden nodes to be 6, sometimes it will not get 100 accuracy within 20000 epochs.Here are the screen shot.

```
zhengqiwendeMacBook-Pro:hw1 zhengqiwen$ python3 spiral_main.py --net polar --hid 6
ep:24300 loss:  0.0110 acc:  91.75
ep:24400 loss:  0.0110 acc:  91.75
ep:24500 loss:  0.0110 acc:  91.75
ep:24600 loss:  0.0110 acc:  91.75
ep:24700 loss:  0.0110 acc:  91.75
ep:24800 loss:  0.0110 acc:  91.75
```

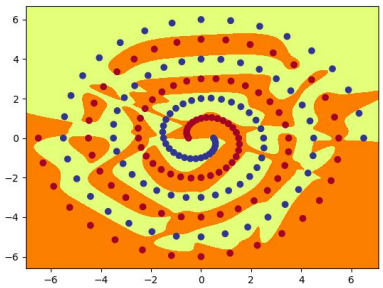
And when I set it to 7,I tried the code for 10 times and each time it will get 100 accuracy within 20000 epochs.Here is the Polar_out image.



Polar_out

4:

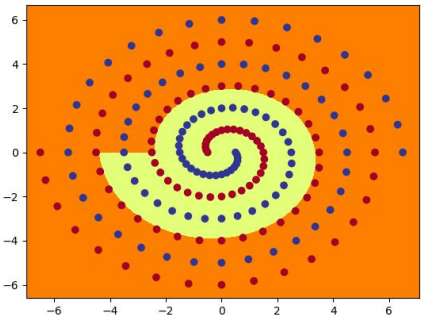
I want to get as low as possible hidden nodes number .I first tried the hidden nodes to be 9,but no matter how I adjust the initial weight, there can always occasionally fail to converge before 20000 epochs.Then I set back the hidden nodes to 10 and after several times try(the data is in Q6 b),I determine the hidden node value to be 10 and make initial weight at 0.16 and let all other meta parameters be default,which can achieve 100 percent accuracy before 20000 epochs on almost all run.



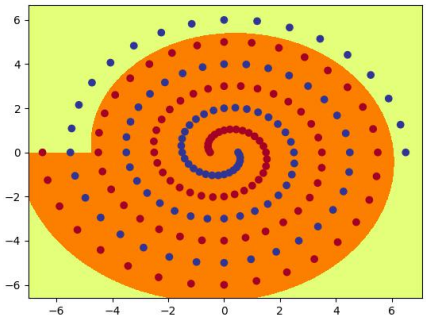
Raw_out

5:

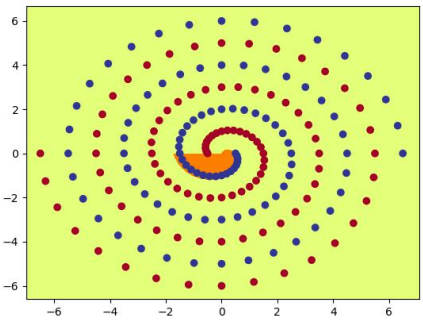
The following pictures are for polar hidden layers.



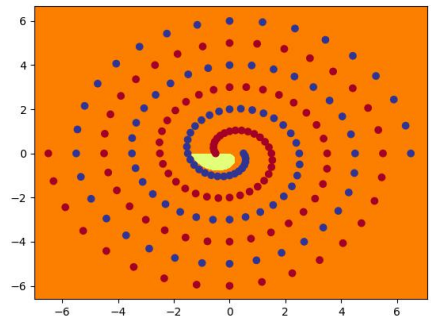
polar1_0



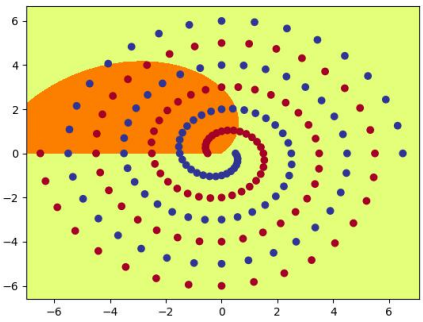
polar1_1



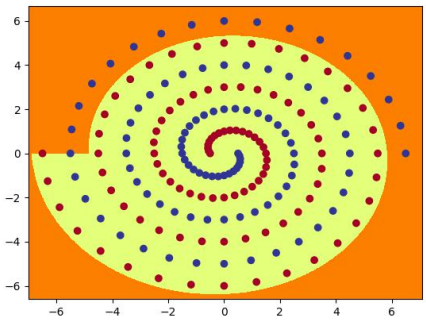
polar1_2



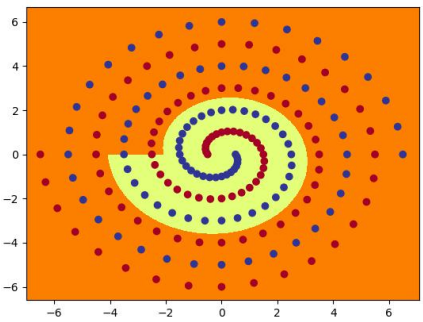
polar1_3



polar1_4

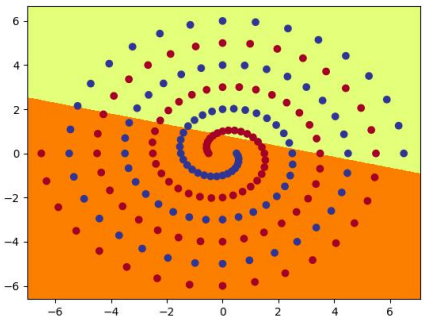


polar1_5

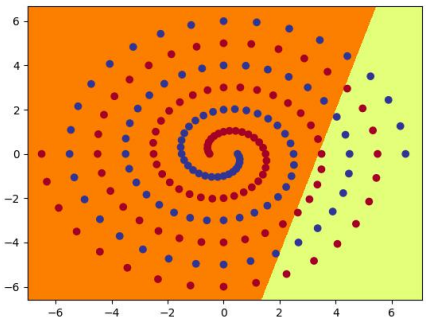


polar1_6

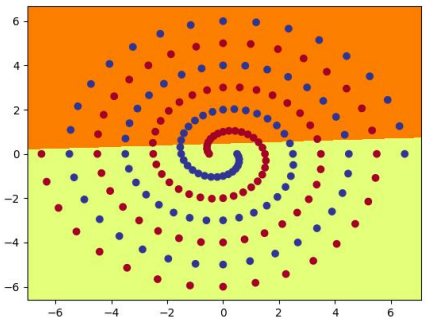
The following pictures are for raw hidden layers.



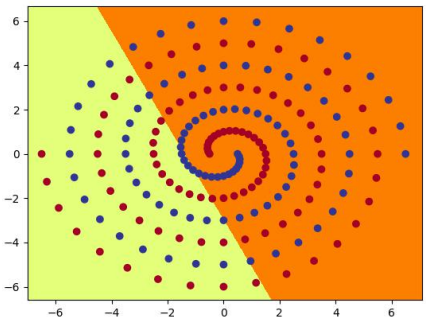
raw1_0



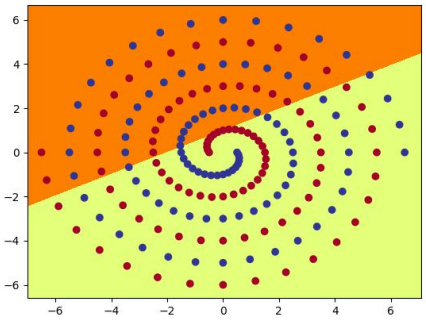
raw1_1



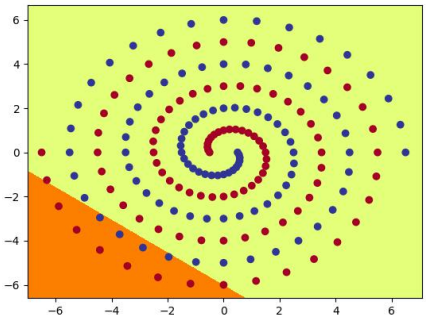
raw1_2



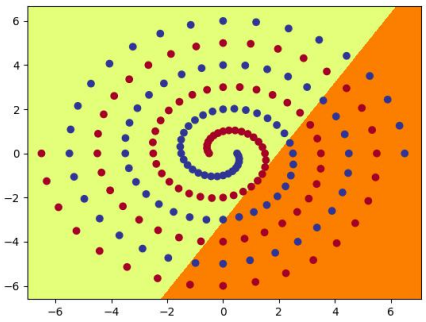
raw1_3



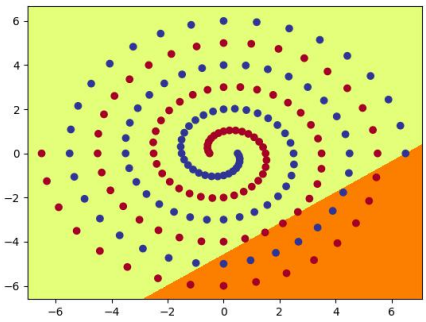
raw1_4



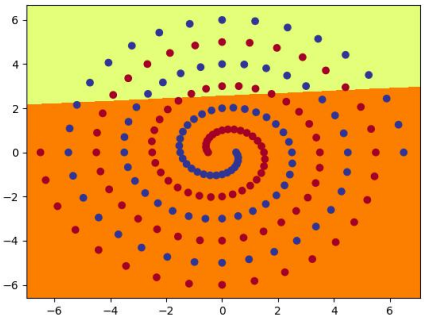
raw1_5



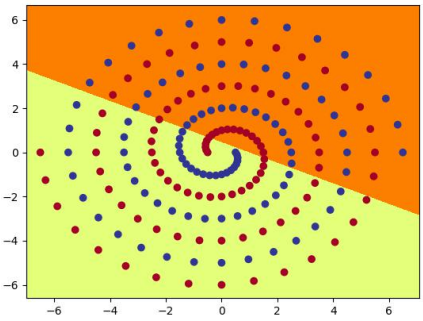
raw1_6



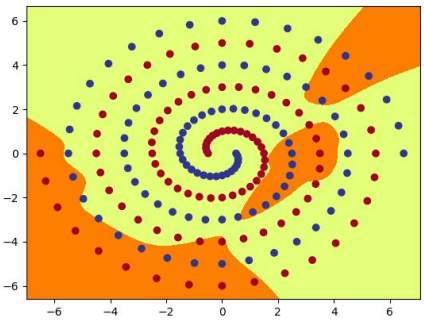
raw1_7



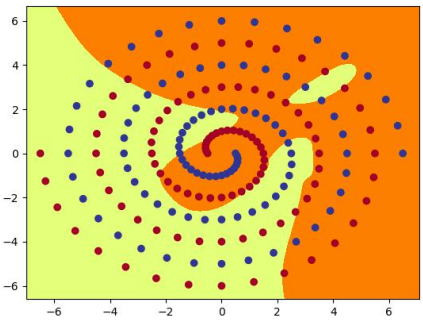
Raw1_8



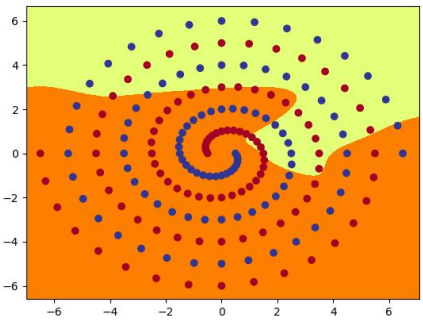
Raw1_9



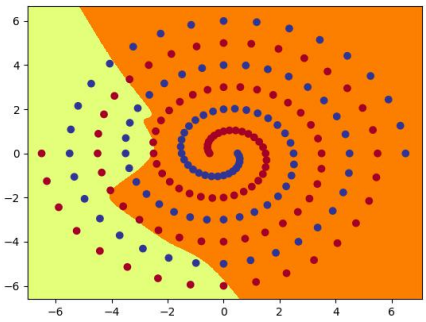
Raw2_0



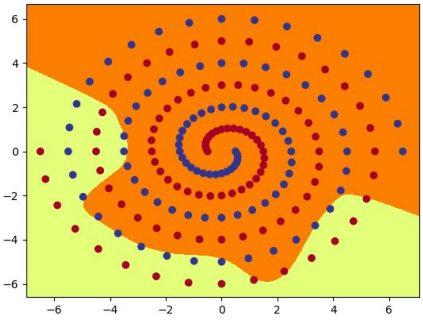
Raw2_1



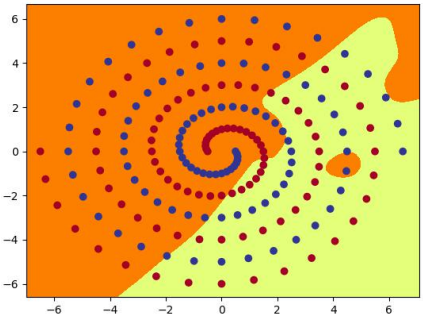
Raw2_2



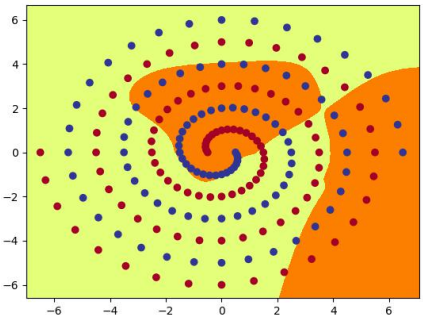
Raw2_3



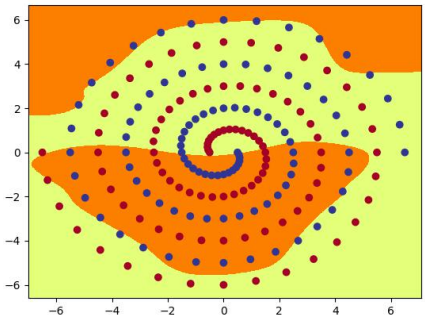
Raw2_4



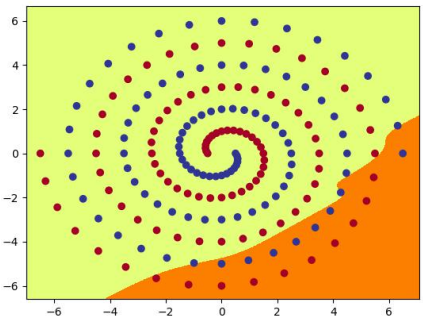
Raw2_5



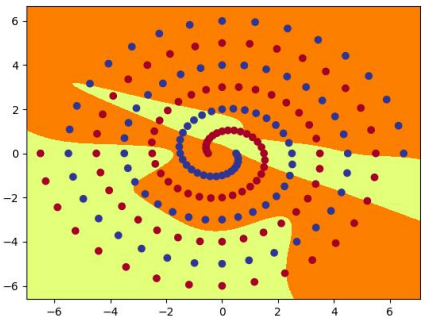
Raw2_6



Raw2_7



Raw2_8



Raw2_9

6:

a:

PolarNet model converts rectangular coordinates to polar coordinates and this nonlinear transformation makes the function learned by model more customized for twin spirals question,whereas RawNet model directly uses linear function to divide regions,which makes it impossible to get 100 accuracy result via one hidden layer.

For PolarNet model, it firstly uses the polar coordinate transformation that make the input variables (x,y) be in the form of (r,a),and the learned weights make the division region of each hidden node be the torus,which is quite suitable for this question and it can learn through one hidden layer with 7 hidden nodes number.From polar_out.png of Q2,the division region is quite symmetric and resonable, it is the result of overlay of 7 hidden nodes output,which we observe by graph_hidden function.

For RawNet model,it uses linear function to divide those dots.From raw_out pictures of Q4,the segmentation area is chaotic,it is the overlay of 10 hidden node from the second hidden layer,where each node is a linear superposition from the first hidden layer.The Q5 raw_0~9 shows the linear division line of 10 hidden node from first hidden layer.So the RawNet is all linear combination that perform poorly for the twin spiral task,compared to PolarNet model.

b:

I tested 10 different initial weight range from 0.1 to 0.2,step 0.01 with default learning rate 0.01 for hidden nodes setting of 10.For each weight I tested for 10 times.The following table is the result epochs to get 100 accuracy rate (> means I manually exit the program and let it end).

weight \ time	1	2	3	4	5	6	7	8	9	10
0.10	>	>	>	>	>	7300	>	>	>	>
0.11	8400	4500	>	4200	6800	5100	>	5800	>	>
0.12	10800	17200	>	>	8900	7100	>	>	9700	>
0.13	6500	>	7200	12800	4500	7500	12900	5100	7300	>
0.14	13800	>	10000	>	>	4600	>	6300	11600	5200
0.15	>	6700	10200	5600	9100	>	>	9100	>	5300
0.16	10100	4500	7300	5900	>	11500	8200	4300	5200	8800
0.17	>	>	>	>	13100	>	>	>	>	>
0.18	4000	>	6800	10200	7800	5900	3000	15900	>	>
0.19	13500	4000	4800	>	5900	>	>	9400	4100	5400
0.20	17000	>	>	>	13700	>	5000	>	11300	10500

From the table above,we can see overall 0.16 initial weight got the best success rate,only 1 failure on all 10 run,and also its speed is quite fast,most of them converge with 10000 epochs,that is why I chose it for Q4.

Q3 is also good,only failed on two run.And the speed among successful time is relatively fast,even compared to 0.16.

0.10 and 0.17 of initial weight got the worst performance,both of them only succeeded once.But despite of most time failure,both of them at least had one time success.

Other initial weights got 5-7 successful times and they are not as correct and efficient as 0.16 and 0.13 at all.

c:

For this question, my modification is based on RawNet.Because I think there can be a lot more improvement space on RawNet model.

Firstly I use the SGD optimizer instead of Adam.And I set initial weight and learning rate with different values.And I got the following tests.(the following all tables > means >20000 epochs)

weight \ lr	0.001	0.01	0.05	0.1	0.3	0.5	1
0.001	>	>	>	>	>	>	>
0.01	>	>	>	>	>	>	>
0.1	>	>	>	>	>	>	>
1	>	>	>	>	>	>	>

We can see from the table no matter how I change the learning rate and initial weight, it won't get satisfactory result.So let me just change back to Adam optimizer.

Then I double the number of batch size,from 97 to 194.This time I record the epochs of getting 100 accuracy rate for 10 times under the initial weight of 0.16 and default learning rate(0.01).

time	1	2	3	4	5	6	7	8	9	10
epochs	1600	2000	4900	1300	>	1600	1300	1900	5900	20000

We can see there is a significant improvement in the efficiency of convergence,because within a certain range, the larger the batch size, the more accurate the determined descending direction, and the smaller the training shock.

So I settle down the 194 batch size and go further more.This time I switch the tanh activation to relu function.I take record of 5 times epochs under 0.16 initial weight and 0.01 learning rate.

time	1	2	3	4	5
epochs	>	>	>	>	>

All those five tests fail to get 100 accuracy in 2000 epochs,so we can conclude Relu function does not have better performance than tanh function for this model. The dead ReLU Problem can be reason for this failure since there is 0 tolerance some weights can be 0.So I switch back to tanh function.

Last step I add the third hidden layer to the RawNet,with in and out number be hidden node number,which is 10.I test 10 times of epochs to get 100 accuracy rate under the initial weight 0.16 and learning rate 0.01.

time	1	2	3	4	5	6	7	8	9	10
epochs	600	9100	9700	3200	>	700	16000	3400	2200	2100

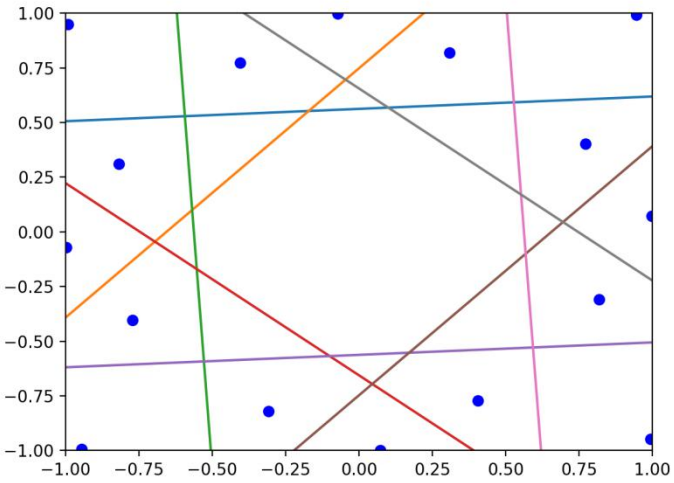
Overall,the three hidden layer RawNet doesn't significantly outperform the original two layer structure,in terms of successful rate and speed,although sometimes it gets very fast convergence(600 and 700 epochs),it can even sometimes fail to converge within 20000 epochs.The reason why it is not better than two-layer structure is maybe that two-layer is sufficient to solve this twin spiral task, while the improvement of adding more layers will become less and less obvious.

In conclusion, what I learned from part2:
Thanks to the framework pytorch , neural network design and realization can be easy.But meta parameters setting can be much more challenging and time-consuming.For this particular question,once we settle down the framework structure,then we need to tune hidden node number,initial weight,learning rate,and also batch size,optimizer,activation function selection to get as higher efficiency and successful rate as possible.This is a really large amount of work I didn't expect.
Also this part beats my common sense that Relu is better to be hidden layer activation function,as well as the deeper the better the neural network can be.It all depends on the

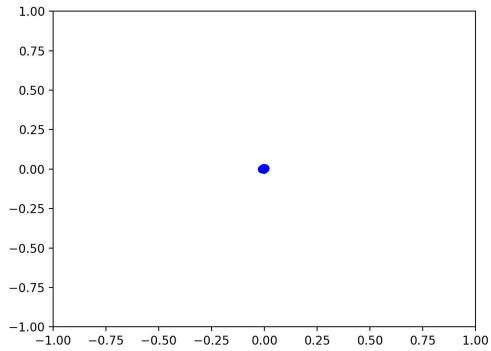
certain question that we need to carefully design and experiment on to solve.
Last but not least,for this particular question,the PolarNet model outperforms the RawNet model from nearly all aspects(especially the difficulty of tuning parameters).So when we decide to tackle a problem by using neural network,the structure of model always matters.

Part3:

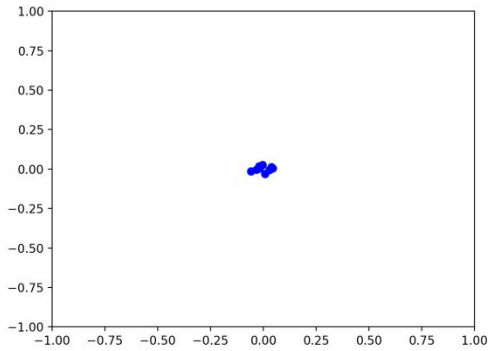
1:
The star16 image is shown as following.



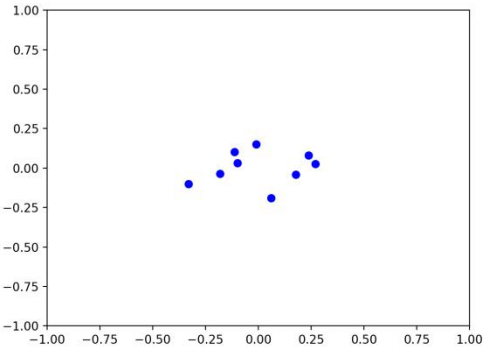
2:
The first eleven images and the final images are shown as following.



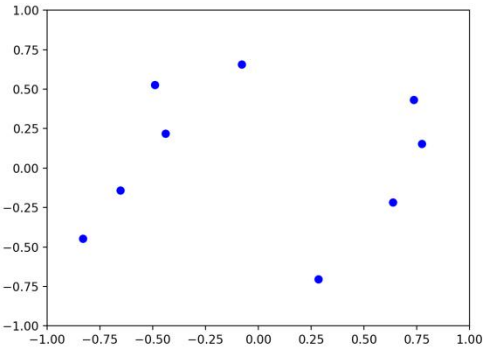
1



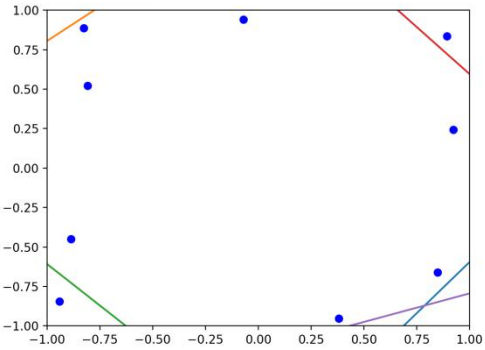
2



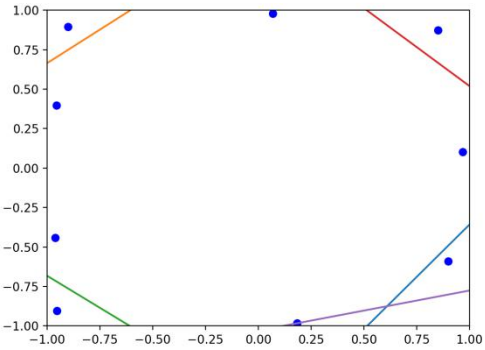
3



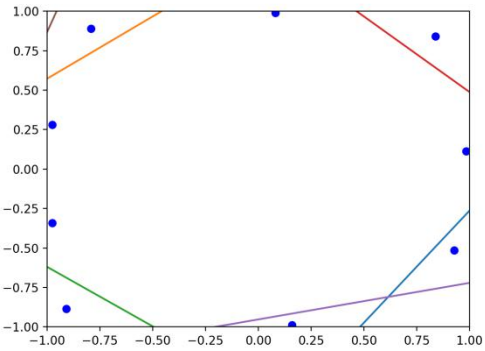
4



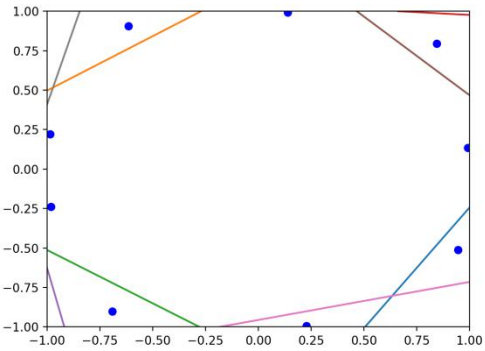
5



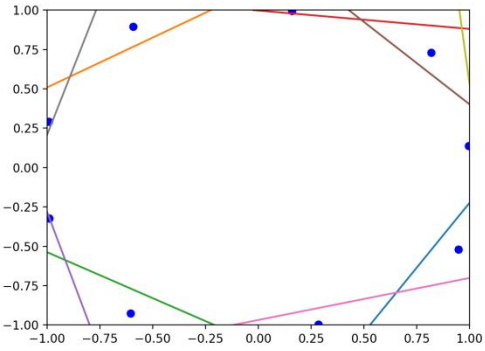
6



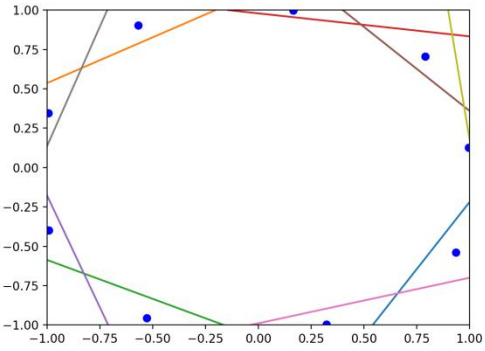
7



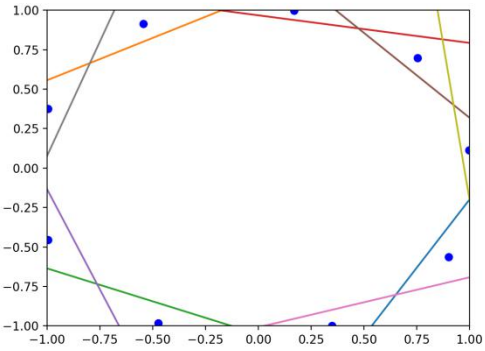
8



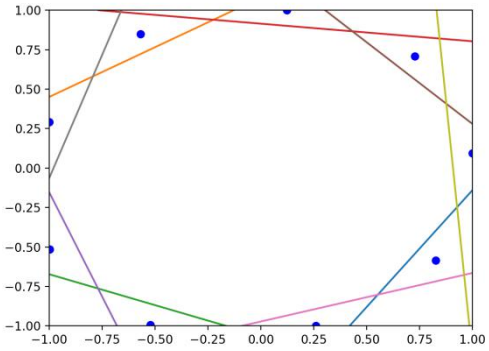
9



10



11

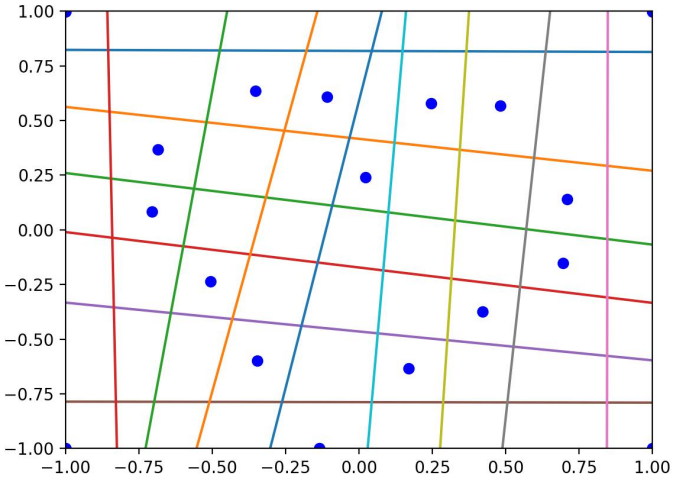


final

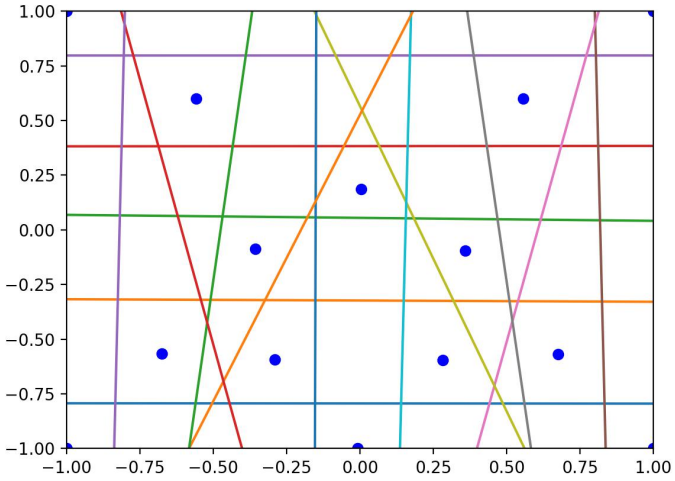
As in progress,the dots(activation units) tend to separate from each other and move from center of picture to the edge.The 5 lines(output boundaries) appears from the fifth figure and more and more lines come out to separate those dots.There are already 9 lines as we wish in figure 9 and the rest of procedures is to adjust these lines to depart dots as perfect as possible(reduce the loss function).And eventually the loss is reduced to 0.02 as we set by default and we get the final image.

3:

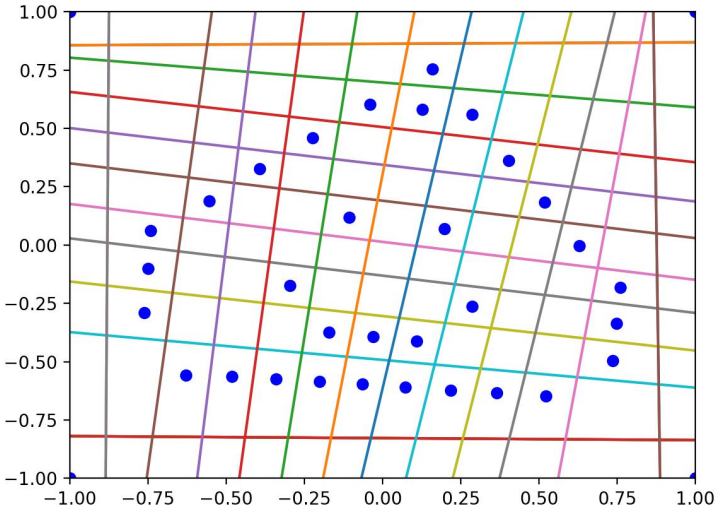
The heart18 image is shown as following.



4:



Target1: I intended to design a 5-pointed star and I set stop argument to be 0.00 and learning rate to be 0.5,after 1000000 epochs training, it reaches 0.002 loss rate and stopped.At least we can figure out the start from the picture.



Target2: I like a Japanese video game called Dragon Quest,in which a cute and weak monster called slime,that is what my second target comes from.I set the learning rate to be 0.8,stop at 0.003 loss rate and I got the above picture,though it is not as cute as the the following picture I found from Internet.