

# Assignment 1 Report

course: COMP9444

term: 2020T2

full name: Zhaokun Su

zID: z5235878

## Part 1: Japanese Character Recognition

### 1. Model Netlin:

Required: Netlin model computes a linear function and of pixels in the image, followed by log softmax;

After 10 Epoch: the result figure shows below:

Final accuracy: 70%

Confusion matrix are shown below:

```
Train Epoch: 10 [0/60000 (0%)] Loss: 0.821707
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.628484
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.593778
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.597358
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.325932
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.515098
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.663063
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.611595
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.348892
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.662339
<class 'numpy.ndarray'>
[[772.  6.  8.  4. 60.  8.  5. 17. 10.  8.]
 [ 6. 668. 62. 35. 54. 27. 22. 29. 37. 50.]
 [ 9. 109. 691. 60. 79. 121. 147. 28. 92. 87.]
 [13. 18. 25. 761. 20. 17. 10. 11. 41.  3.]
 [28. 29. 26. 14. 623. 19. 25. 82.  8. 53.]
 [61. 22. 22. 58. 20. 727. 25. 16. 30. 32.]
 [ 2. 58. 47. 13. 33. 27. 722. 55. 46. 18.]
 [62. 14. 36. 18. 35.  9. 20. 626.  7. 31.]
 [28. 25. 45. 27. 20. 33. 11. 90. 707. 40.]
 [19. 51. 38. 10. 56. 12. 13. 46. 22. 678.]]

Test set: Average loss: 1.0096, Accuracy: 6975/10000 (70%)
```

### 2. Model Netfull:

Required: Implement a fully connected 2-layer network Netfull, using tanh at the hidden nodes and log softmax at the output node;

Choosing different values for the number of hidden nodes:

When we choose 30 nodes in the hidden layer:

After 10 Epoch: the result figure shows below:

Final accuracy: 78%

Confusion matrix are shown below:

```
Train Epoch: 10 [0/60000 (0%)] Loss: 0.470595
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.461836
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.383045
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.330557
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.157574
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.315698
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.375365
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.414937
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.189628
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.349401
<class 'numpy.ndarray'>
[[809. 6. 7. 63. 19. 5. 12. 16. 13.]
 [ 5. 740. 28. 17. 32. 10. 14. 20. 21. 23.]
 [ 6. 75. 810. 45. 37. 126. 92. 124. 150. 83.]
 [ 6. 7. 25. 852. 12. 13. 6. 17. 46. 1.]
 [ 37. 27. 21. 7. 742. 14. 16. 66. 4. 47.]
 [ 30. 12. 13. 25. 13. 771. 9. 10. 18. 15.]
 [ 4. 67. 32. 12. 27. 35. 834. 59. 36. 27.]
 [ 56. 6. 16. 11. 23. 5. 11. 702. 4. 21.]
 [ 36. 23. 26. 18. 26. 12. 5. 58. 790. 14.]
 [ 11. 37. 22. 6. 25. 5. 8. 42. 15. 766.]]
Test set: Average loss: 0.7070, Accuracy: 7816/10000 (78%)
```

When we choose 50 nodes in the hidden layer:

After 10 Epoch: the result figure shows below:

Final accuracy: 81%

Confusion matrix are shown below:

```
Train Epoch: 10 [0/60000 (0%)] Loss: 0.381416
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.284614
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.294940
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.284867
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.150580
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.269215
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.276085
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.344677
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.181488
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.257798
<class 'numpy.ndarray'>
[[824. 4. 4. 4. 48. 11. 3. 10. 12. 2.]
 [ 6. 784. 17. 11. 40. 12. 11. 20. 17. 19.]
 [ 2. 44. 826. 35. 21. 82. 19. 121. 130. 67.]
 [ 5. 6. 42. 894. 12. 13. 7. 6. 47. 2.]
 [ 37. 24. 14. 2. 759. 12. 14. 38. 9. 33.]
 [ 29. 10. 17. 18. 7. 807. 12. 8. 18. 9.]
 [ 6. 66. 31. 12. 41. 39. 840. 42. 32. 18.]
 [ 46. 8. 16. 5. 16. 5. 10. 785. 5. 20.]
 [ 35. 22. 19. 9. 27. 13. 4. 33. 817. 17.]
 [ 10. 32. 14. 10. 29. 6. 7. 37. 13. 813.]]
Test set: Average loss: 0.6062, Accuracy: 8149/10000 (81%)
```

When we choose 70 nodes in the hidden layer:

After 10 Epoch: the result figure shows below:

Final accuracy: 83%

Confusion matrix are shown below:

```

Train Epoch: 10 [0/60000 (0%)] Loss: 0.307832
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.312092
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.264553
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.238747
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.112723
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.275074
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.240449
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.391764
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.132365
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.323983
<class 'numpy.ndarray'>
[[848.  4.  9.  3. 43.  9.  3. 13.  9.  3.]
 [ 6. 795. 16. 12. 34. 11. 13. 11. 29. 26.]
 [ 1. 31. 827. 37. 16. 77. 51. 11. 20. 46.]
 [ 5. 10. 35. 898.  7. 17.  8.  5. 56.  3.]
 [38. 22. 12.  2. 793.  9. 12. 41.  3. 21.]
 [25. 15. 17. 18. 11. 814.  9. 12. 13. 13.]
 [ 4. 65. 29.  9. 40. 37. 880. 49. 28. 20.]
 [42.  6. 19.  3. 17.  5. 15. 784.  4. 20.]
 [27. 21. 23.  8. 20. 11.  5. 37. 835. 14.]
 [ 4. 31. 13. 10. 19. 10.  4. 37.  3. 834.]]
Test set: Average loss: 0.5508, Accuracy: 8308/10000 (83%)

```

When we choose more than 70 nodes in the hidden layer, we could find that there is little change in final accuracy, so we choose 70 as our best number of hidden nodes.

### 3. Model NetConv:

```

# initially: 1 channel, 24 filters, 5x5 filter size, 2 padding
num_hid = 90
self.conv2d_1 = nn.Conv2d(in_channels=1, out_channels=24, kernel_size=5, stride=1, padding=2, bias=True, padding_mode='zero')
self.conv2d_2 = nn.Conv2d(in_channels=24, out_channels=36, kernel_size=5, stride=1, padding=2, bias=True, padding_mode='zero')
self.max_pool2d = nn.MaxPool2d(kernel_size=2)
self.full_connected_1 = nn.Linear(1764, num_hid, bias=True)
self.full_connected_2 = nn.Linear(num_hid, 10, bias=True)
self.ReLU = nn.ReLU(inplace=False)
self.log_softmax = nn.LogSoftmax(dim=1)

```

When we create our convolutional, we use two different combinations of parameters:

In the first convolutional layer: we use 24 filters and kernel\_size 4 for each filter;

In the second convolutional layer: we use 36 filters and kernel\_size 3 for each filter;

And we add max pooling approach every time after convolutions with max pooling size 2.

```
Test set: Average loss: 0.2333, Accuracy: 9394/10000 (94%)
Train Epoch: 10 [0/60000 (0%)] Loss: 0.029302
Train Epoch: 10 [6400/60000 (11%)] Loss: 0.027045
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.0118214
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.029607
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.064941
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.058670
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.013645
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.079263
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.005141
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.033743
<class 'numpy.ndarray'>
[[959. 3. 11. 2. 18. 3. 3. 4. 3. 6.]
 [ 5. 918. 8. 2. 5. 9. 2. 5. 11. 6.]
 [ 1. 6. 894. 10. 2. 41. 14. 0. 7. 10.]
 [ 1. 0. 35. 973. 7. 8. 2. 0. 5. 1.]
 [20. 15. 7. 0. 943. 7. 7. 5. 10. 12.]
 [ 1. 0. 9. 4. 1. 902. 1. 1. 1. 0.]
 [ 0. 32. 13. 3. 5. 10. 966. 5. 4. 5.]
 [ 8. 10. 11. 2. 7. 6. 4. 965. 2. 2.]
 [ 2. 7. 5. 1. 7.2 5. 0. 3. 952. 7.]
 [ 3. 9. 7. 3. 5. 9. 1. 12. 5. 951.]]
Test set: Average loss: 0.2310, Accuracy: 9423/10000 (94%)
```

We can see that after epoch 8, the accuracy of our model had reached to 94% successfully.

#### 4. Discuss what you have learned from this exercise, including the following points:

##### a. the relative accuracy of the three models,

when  $\text{Netlin} < \text{Netfull} < \text{NetConv}$  we look at these 3 different models: Netlin, Netfull and NetConv.

Model complexity:  $\text{Netlin} < \text{Netfull} < \text{NetConv}$ ;

The highest accuracy:  $\text{Netlin} (70\%) < \text{Netfull} (83\%) < \text{NetConv} (94\%)$

The complexity of networks we established is increasing.

Normally, the more complexity the model has, the higher accuracy it gets.

(1) for linear network Netlin, the structure is not complex enough, which is more likely to be underfitting. And the accuracy is relatively lower.

(2) the second network Netfull has a higher complexity structure than the first one, and there are some improvements in terms of final accuracy.

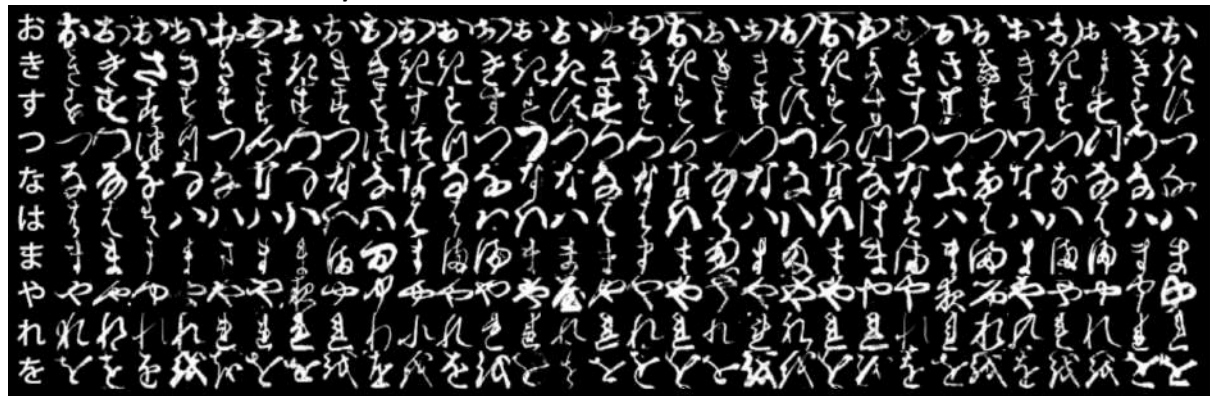
(3) As our input data is of form of image data, the application of convolutional layer is more suitable for processing image-based data. It is mainly because convolutional layer could extract features and insights of behind an image according to the data we obtained. Compared with the other model, training a convolutional model is a better choice for image-based data representations.

**b. the confusion matrix for each model: which characters are most likely to be mistaken for which other characters, and why?**

Firstly, we have a look at our dataset image:

0="o", 1="ki", 2="su", 3="tsu", 4="na",

5="ha", 6="ma", 7="ya", 8="re", 9="wo";



For Netlin:

[[772.	6.	8.	4.	60.	8.	5.	17.	10.	8.]
[ 6.	668.	62.	35.	54.	27.	22.	29.	37.	50.]
[ 9.	109.	691.	60.	79.	<u>121.</u>	<u>147.</u>	28.	92.	87.]
[ 13.	18.	25.	761.	20.	17.	10.	11.	41.	3.]
[ 28.	29.	26.	14.	623.	19.	25.	82.	8.	53.]
[ 61.	22.	22.	58.	20.	727.	25.	16.	30.	32.]
[ 2.	58.	47.	13.	33.	27.	722.	55.	46.	18.]
[ 62.	14.	36.	18.	35.	9.	20.	626.	7.	31.]
[ 28.	25.	45.	27.	20.	33.	11.	<u>90.</u>	707.	40.]
[ 19.	51.	38.	10.	56.	12.	13.	46.	22.	678.]]

The writing format of 3<sup>rd</sup> character ("su") and 7<sup>th</sup> character ("ma") are similar, so it is easier to make a mistake with these two characters.

For Netfull:

[[848.	4.	9.	3.	43.	9.	3.	13.	9.	3.]
[ 6.	795.	16.	12.	34.	11.	13.	11.	29.	26.]
[ 1.	31.	827.	37.	16.	<u>77.</u>	51.	11.	20.	46.]
[ 5.	10.	35.	898.	7.	17.	8.	5.	<u>56.</u>	3.]
[ 38.	22.	12.	2.	793.	9.	12.	41.	3.	21.]
[ 25.	15.	17.	18.	11.	814.	9.	12.	13.	13.]
[ 4.	<u>65.</u>	29.	9.	40.	37.	880.	49.	28.	20.]
[ 42.	6.	19.	3.	17.	5.	15.	784.	4.	20.]
[ 27.	21.	23.	8.	20.	11.	5.	37.	835.	14.]
[ 4.	31.	13.	10.	19.	10.	4.	37.	3.	834.]]

For Netconv:



[959.	3.	11.	2.	18.	3.	3.	4.	3.	6.]
[ 5.	918.	8.	2.	5.	9.	2.	5.	11.	6.]
[ 1.	6.	894.	10.	2.	<u>41.</u>	14.	0.	7.	10.]
[ 1.	0.	35.	973.	7.	8.	2.	0.	5.	1.]
[ <u>20.</u>	<u>15.</u>	7.	0.	943.	7.	7.	5.	10.	12.]
[ 1.	0.	9.	4.	1.	902.	1.	1.	1.	0.]
[ 0.	<u>32.</u>	13.	3.	5.	10.	966.	5.	4.	5.]
[ 8.	10.	11.	2.	7.	6.	4.	965.	2.	2.]
[ 2.	7.	5.	1.	7.	5.	0.	3.	952.	7.]
[ 3.	9.	7.	3.	5.	9.	1.	12.	5.	951.]]

And we can see that 3<sup>rd</sup> character ("su") and 6<sup>th</sup> character ("ha") looks like similar because 6<sup>th</sup> character ("ha") contains "ha" character shape on the right part, maybe this is the reason why it is easier to make a mistake with these two characters.

**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.**

In the convolutional, I have tried two approaches:

One is the original one, we get two convolutional layers without pooling layers after them, and we get a average accuracy of 92%

```
def forward(self, x):
    x = self.conv2d_1(x)
    x = self.ReLU(x)
    x = self.max_pool2d(x)
    x = self.conv2d_2(x)
    x = self.ReLU(x) # somevalues * somevalues * 32
    x = self.max_pool2d(x)
    hidden_input_x = x.view(x.shape[0], -1)
    x = self.full_connected_1(hidden_input_x)
    x = self.ReLU(x)
    x = self.full_connected_2(x)
    output = self.log_softmax(x)
    return output
```

Another one is implemented by add pooling layers after each convolutional layer and two extra convolutional layers, which can get a higher accuracy during training, with a range from 94% to 95%.

Reason for this maybe is our model complexity increases by adding more convolutional layer for our problem and adding max pooling layer seems more likely to fetch right features about our images.

## Part 2: Twin Spirals Task

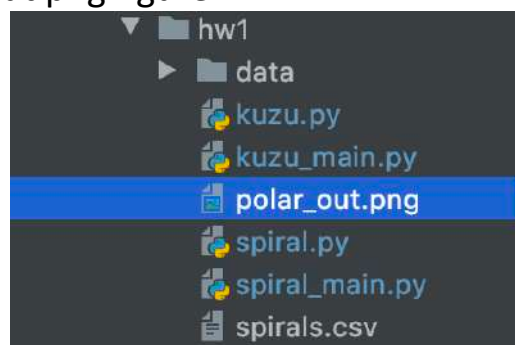
1. code is seen in file spiral.py
2. we run command in terminal:

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net polar --hid 10
```

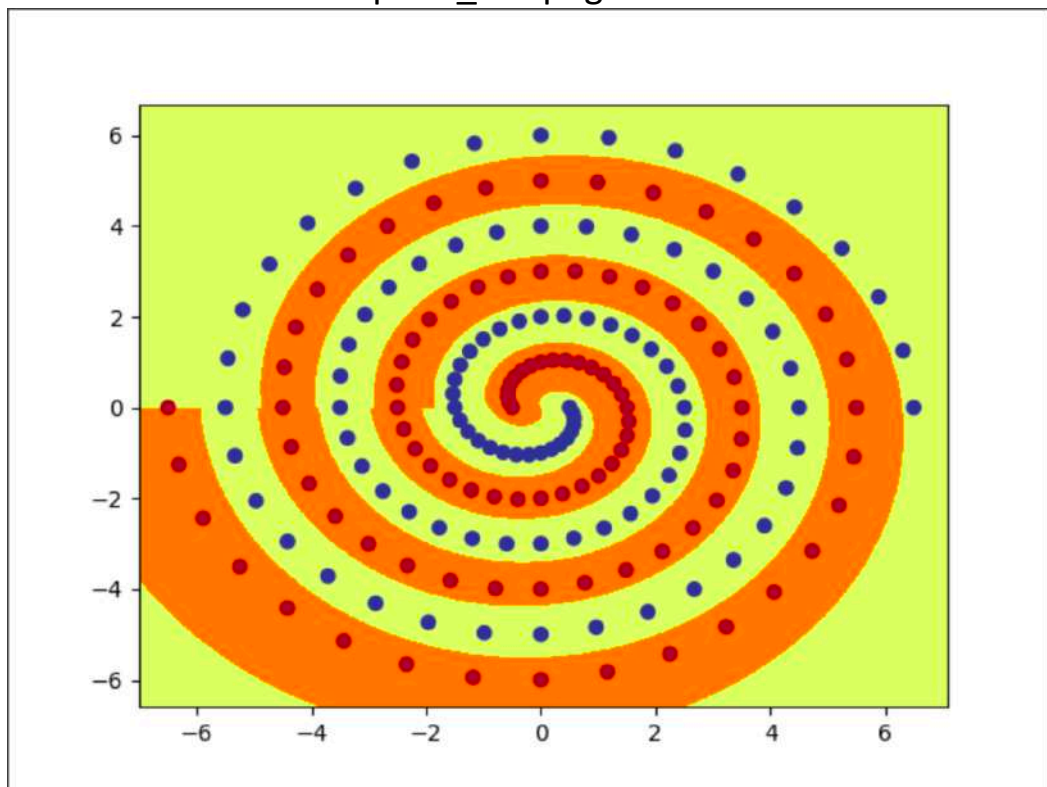
In 2800 epoch, we get 100% accuracy.

```
ep: 1700 loss: 0.0613 acc: 74.23
ep: 1800 loss: 0.0603 acc: 74.23
ep: 1900 loss: 0.0562 acc: 73.71
ep: 2000 loss: 0.0510 acc: 76.80
ep: 2100 loss: 0.0461 acc: 79.38
ep: 2200 loss: 0.0413 acc: 81.44
ep: 2300 loss: 0.0371 acc: 81.96
ep: 2400 loss: 0.0336 acc: 89.18
ep: 2500 loss: 0.0330 acc: 90.72
ep: 2600 loss: 0.0320 acc: 90.72
ep: 2700 loss: 0.0273 acc: 91.75
ep: 2800 loss: 0.0227 acc: 100.00
```

And we get our output png figure:



The result is shown below: polar\_out.png



when we try num\_hid to be 9, 8, 7 and 6 using command:  
python3 spiral\_main.py --net polar --hid "num\_hid" --epochs 20000

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net polar --hid 9 --epochs 20000
```

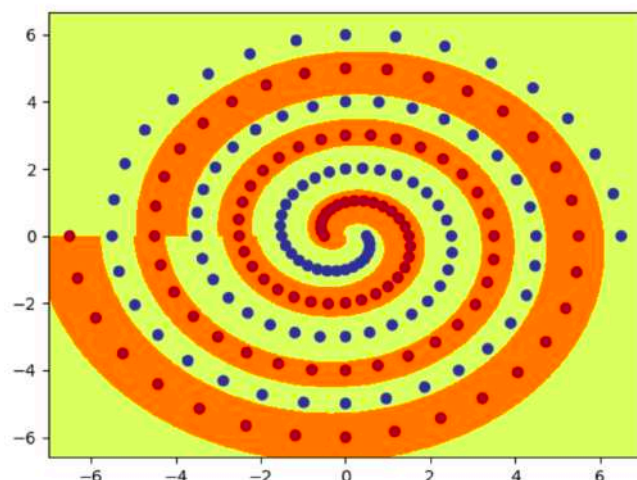
```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net polar --hid 8 --epochs 20000
```

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net polar --hid 7 --epochs 20000
```

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net polar --hid 6 --epochs 20000
```

```
ep: 3000 loss: 0.0261 acc: 67.01
ep: 3100 loss: 0.0235 acc: 67.01
ep: 3200 loss: 0.0213 acc: 67.01
ep: 3300 loss: 0.0194 acc: 67.01
ep: 3400 loss: 0.0178 acc: 67.01
ep: 3500 loss: 0.0164 acc: 67.01
ep: 3600 loss: 0.0152 acc: 67.01
ep: 3700 loss: 0.0142 acc: 67.01
ep: 3800 loss: 0.0133 acc: 67.01
ep: 3900 loss: 0.0125 acc: 67.01
ep: 4000 loss: 0.0118 acc: 67.01
ep: 4100 loss: 0.0111 acc: 67.01
ep: 4200 loss: 0.0106 acc: 67.01
ep: 4300 loss: 0.0101 acc: 67.01
ep: 4400 loss: 0.0097 acc: 67.01
ep: 4500 loss: 0.0093 acc: 67.01
ep: 4600 loss: 0.0090 acc: 67.01
ep: 4700 loss: 0.0087 acc: 67.01
ep: 4800 loss: 0.0084 acc: 67.01
ep: 4900 loss: 0.0082 acc: 67.01
ep: 5000 loss: 0.0080 acc: 67.01
ep: 5100 loss: 0.0078 acc: 67.01
ep: 5200 loss: 0.0077 acc: 67.01
ep: 5300 loss: 0.0076 acc: 67.01
ep: 5400 loss: 0.0075 acc: 67.01
ep: 5500 loss: 0.0075 acc: 67.01
ep: 5600 loss: 0.0074 acc: 67.01
ep: 5700 loss: 0.0074 acc: 67.01
ep: 5800 loss: 0.0075 acc: 67.01
ep: 5900 loss: 0.0078 acc: 67.01
ep: 6000 loss: 0.0758 acc: 100.00
```

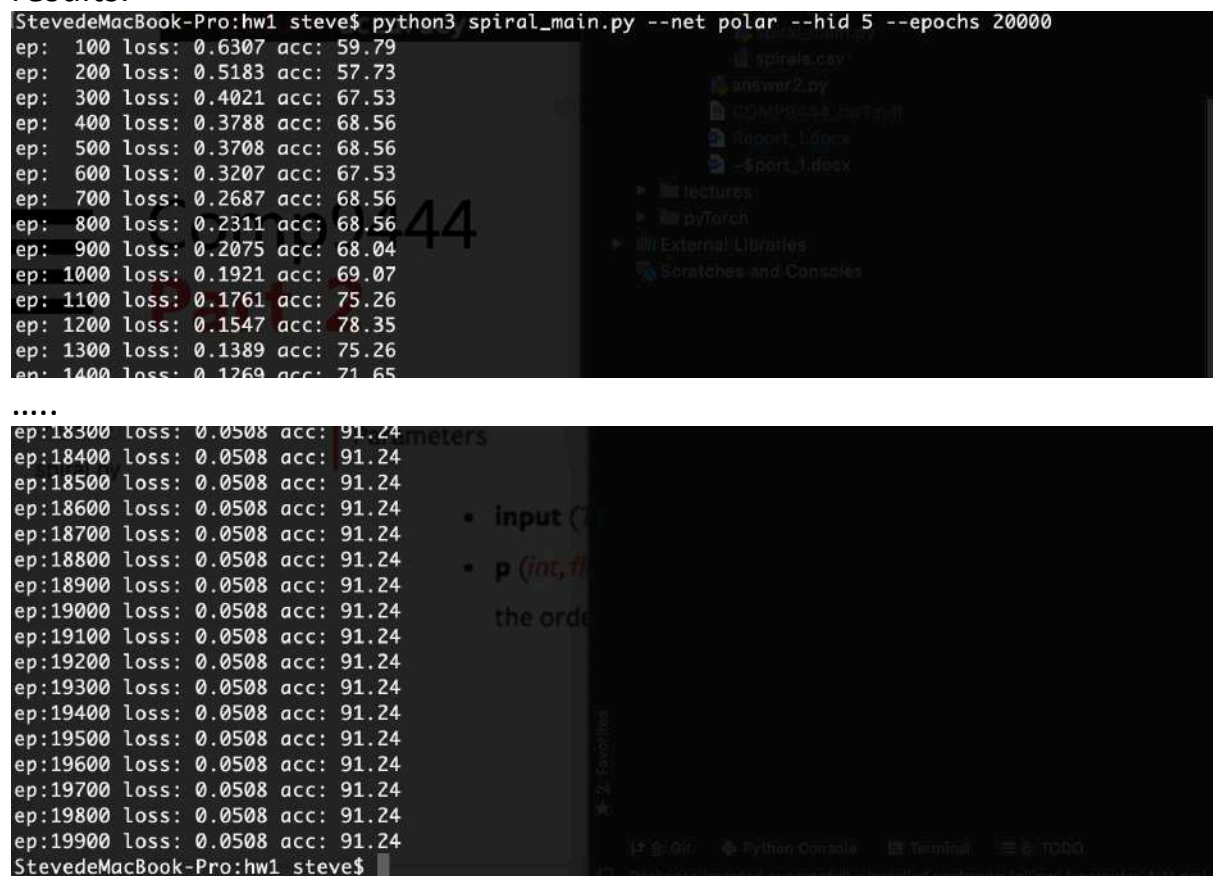
The figure below is figure generated when hid\_num = 6



They all reach to 100% accuracy in the end.



When we try numbers less than 6 for several times, the accuracy cannot reach to 100% within 20000 epochs, e.g. running `--hid = 5`, we get below results.



```

SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net polar --hid 5 --epochs 20000
ep: 100 loss: 0.6307 acc: 59.79
ep: 200 loss: 0.5183 acc: 57.73
ep: 300 loss: 0.4021 acc: 67.53
ep: 400 loss: 0.3788 acc: 68.56
ep: 500 loss: 0.3708 acc: 68.56
ep: 600 loss: 0.3207 acc: 67.53
ep: 700 loss: 0.2687 acc: 68.56
ep: 800 loss: 0.2311 acc: 68.56
ep: 900 loss: 0.2075 acc: 68.04
ep: 1000 loss: 0.1921 acc: 69.07
ep: 1100 loss: 0.1761 acc: 75.26
ep: 1200 loss: 0.1547 acc: 78.35
ep: 1300 loss: 0.1389 acc: 75.26
ep: 1400 loss: 0.1269 acc: 71.65
.....
ep:18300 loss: 0.0508 acc: 91.24
ep:18400 loss: 0.0508 acc: 91.24
ep:18500 loss: 0.0508 acc: 91.24
ep:18600 loss: 0.0508 acc: 91.24
ep:18700 loss: 0.0508 acc: 91.24
ep:18800 loss: 0.0508 acc: 91.24
ep:18900 loss: 0.0508 acc: 91.24
ep:19000 loss: 0.0508 acc: 91.24
ep:19100 loss: 0.0508 acc: 91.24
ep:19200 loss: 0.0508 acc: 91.24
ep:19300 loss: 0.0508 acc: 91.24
ep:19400 loss: 0.0508 acc: 91.24
ep:19500 loss: 0.0508 acc: 91.24
ep:19600 loss: 0.0508 acc: 91.24
ep:19700 loss: 0.0508 acc: 91.24
ep:19800 loss: 0.0508 acc: 91.24
ep:19900 loss: 0.0508 acc: 91.24
SteveMacBook-Pro:hw1 steve$

```

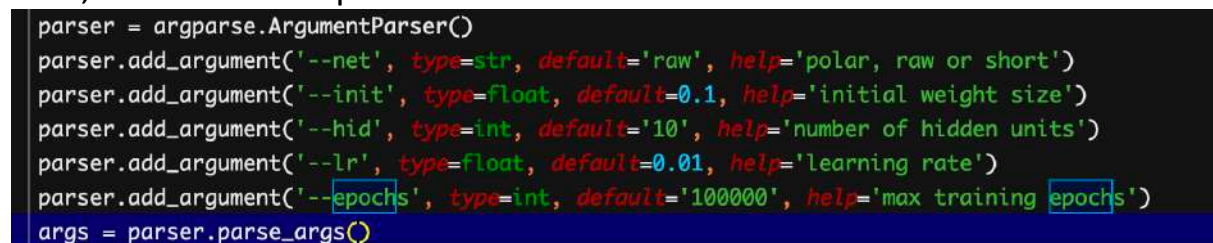
We try a lot of times running, all of them failed. And failure as well when we try `--hid < 5` and the performance is even worse.

As a result, the minimum number we have found is 6.

3. code is seen in file `spiral.py`

4. Initially, we run initial command:

First, we can see all parameter default values:



```

parser = argparse.ArgumentParser()
parser.add_argument('--net', type=str, default='raw', help='polar, raw or short')
parser.add_argument('--init', type=float, default=0.1, help='initial weight size')
parser.add_argument('--hid', type=int, default='10', help='number of hidden units')
parser.add_argument('--lr', type=float, default=0.01, help='learning rate')
parser.add_argument('--epochs', type=int, default='100000', help='max training epochs')
args = parser.parse_args()

```

Then, we run command below:



```

SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw

```

For this particular choice:

We get `--init` value is: 0.1 and `--hid` value is: 10;

We could see in this time, accuracy failed to increase to 100% within 20000 epochs (just to around 50% in 20000 epochs);

```
ep:19000 loss: 0.7128 acc: 50.52
ep:19100 loss: 0.7128 acc: 50.52
ep:19200 loss: 0.7128 acc: 50.52
ep:19300 loss: 0.7128 acc: 50.52
ep:19400 loss: 0.7128 acc: 50.52
ep:19500 loss: 0.7128 acc: 50.52
ep:19600 loss: 0.7128 acc: 50.52
ep:19700 loss: 0.7128 acc: 50.52
ep:19800 loss: 0.7128 acc: 50.52
ep:19900 loss: 0.7128 acc: 50.52
ep:20000 loss: 0.7128 acc: 50.52
ep:20100 loss: 0.7128 acc: 50.52
ep:20200 loss: 0.7128 acc: 50.52
ep:20300 loss: 0.7128 acc: 50.52
ep:20400 loss: 0.7128 acc: 50.52
ep:20500 loss: 0.7128 acc: 50.52
ep:20600 loss: 0.7128 acc: 50.52
ep:20700 loss: 0.7128 acc: 50.52
ep:20800 loss: 0.7128 acc: 50.52
ep:20900 loss: 0.7128 acc: 50.52
ep:21000 loss: 0.7128 acc: 50.52
ep:21100 loss: 0.7128 acc: 50.52
ep:21200 loss: 0.7128 acc: 50.52
```

```
65
66 data = torch.tensor(df.values, dtype=torch.float)
67
68 num_input = data.shape[1] - 1
69
70 full_input = data[:, 0:num_input]
71 full_target = data[:, num_input:num_input + 1]
72
73 train_dataset = torch.utils.data.TensorDataset(
74 train_loader = torch.utils.data.DataLoader(train_dataset)
75
76 # create neural network
77 if args.net == 'polar':
78     net = PolarNet(args.hid)
79 elif args.net == 'short':
80     net = ShortNet(args.hid)
81 else:
82     net = RowNet(args.hid)
```

Now, we try to change `--init` parameter in order to get a better performance.

Firstly, we try to decrease our `--init` value by running such command:

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw --init 0.05 --epochs 20000
```

We get parameters:

`--init: 0.05`

`--epochs: 20000`

`--hid_num: 10 (default)`

```
ep:18900 loss: 0.7128 acc: 50.52
ep:19000 loss: 0.7128 acc: 50.52
ep:19100 loss: 0.7128 acc: 50.52
ep:19200 loss: 0.7128 acc: 50.52
ep:19300 loss: 0.7128 acc: 50.52
ep:19400 loss: 0.7128 acc: 50.52
ep:19500 loss: 0.7128 acc: 50.52
ep:19600 loss: 0.7128 acc: 50.52
ep:19700 loss: 0.7128 acc: 50.52
ep:19800 loss: 0.7128 acc: 50.52
ep:19900 loss: 0.7128 acc: 50.52
```

```
110 plt.savefig('xs_out.png', % args.net)
111
```

The performance is not good within 20000 epochs.

After this trial, we tried to some other group of smaller values of `--init`:

They all get bad results.

`--init: 0.03`

`--epochs: 20000`

`--hid_num: 10 (default)`

Final accuracy in 20000-th epoch: 53.27

`--init: 0.05`

`--epochs: 20000`

`--hid_num: 10 (default)`

Final accuracy in 20000-th epoch: 50.52

--init: 0.08

--epochs: 20000

--hid\_num: 10 (default)

Final accuracy in 20000-th epoch: 51.68

In the next following steps: we tried to increase --init values greater than 0.1:

--init: 0.15

--epochs: 20000

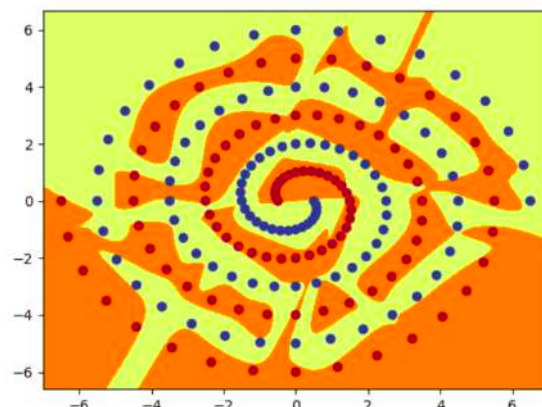
--hid\_num: 10 (default)

Result shown below:

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw --init 0.15 --epochs 20000 m(net.pars
ep: 100 loss: 0.6939 acc: 55.15
ep: 200 loss: 0.6857 acc: 61.34
ep: 300 loss: 0.6844 acc: 59.79
ep: 400 loss: 0.6826 acc: 59.79
ep: 500 loss: 0.6806 acc: 59.79
ep: 600 loss: 0.6780 acc: 59.79
.....
ep: 5800 loss: 0.0358 acc: 99.48
ep: 5900 loss: 0.0355 acc: 99.48
ep: 6000 loss: 0.0352 acc: 99.48
ep: 6100 loss: 0.0349 acc: 99.48
ep: 6200 loss: 0.0347 acc: 99.48
ep: 6300 loss: 0.0345 acc: 99.48
ep: 6400 loss: 0.0343 acc: 99.48
ep: 6500 loss: 0.0336 acc: 99.48
ep: 6600 loss: 0.0320 acc: 99.48
ep: 6700 loss: 0.0304 acc: 99.48
ep: 6800 loss: 0.0288 acc: 100.00
```

In epoch 6800, it reached to 100% acc.

Picture generated:



--init: 0.2

--epochs: 20000

--hid\_num: 10 (default)

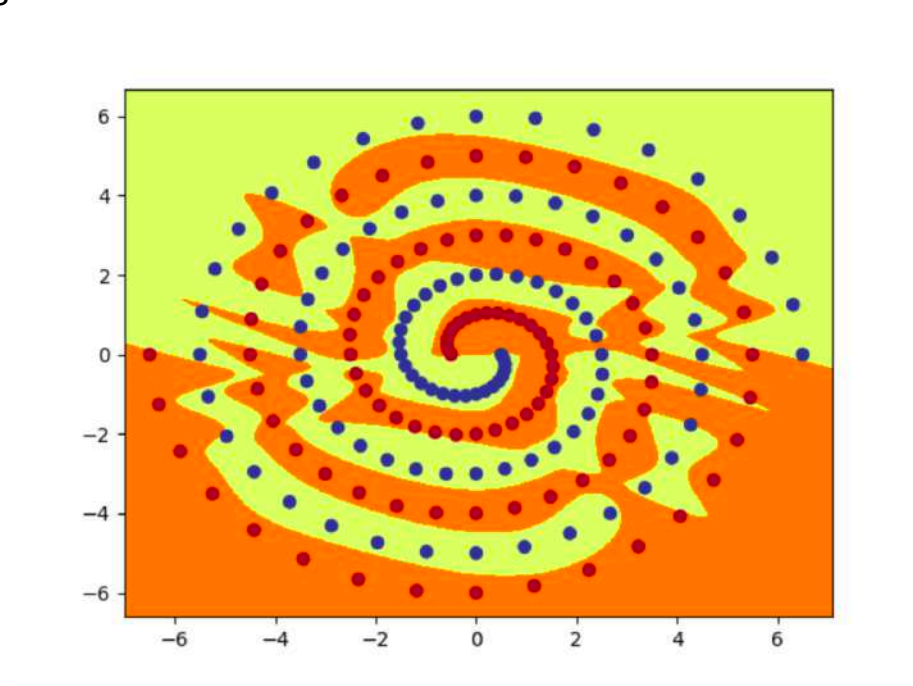
Final accuracy in 20000-th epoch: 53.27

Result shown below:

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw --init 0.2 --epochs 20000
ep: 100 loss: 0.6961 acc: 58.25
ep: 200 loss: 0.6852 acc: 59.79
ep: 300 loss: 0.6798 acc: 60.82
ep: 400 loss: 0.6699 acc: 61.86
ep: 500 loss: 0.6611 acc: 61.34
ep: 600 loss: 0.6531 acc: 59.79
ep: 700 loss: 0.6441 acc: 57.22
ep: 800 loss: 0.6317 acc: 55.15
ep: 900 loss: 0.6157 acc: 55.67
ep: 1000 loss: 0.5761 acc: 58.76
ep: 1100 loss: 0.4996 acc: 58.25
ep: 1200 loss: 0.4235 acc: 64.43
ep: 1300 loss: 0.3505 acc: 64.95
.....
ep:13200 loss: 0.0143 acc: 99.48
ep:13300 loss: 0.0142 acc: 99.48
ep:13400 loss: 0.0142 acc: 99.48
ep:13500 loss: 0.0140 acc: 99.48
ep:13600 loss: 0.0140 acc: 99.48
ep:13700 loss: 0.0139 acc: 99.48
ep:13800 loss: 0.0139 acc: 99.48
ep:13900 loss: 0.0139 acc: 99.48
ep:14000 loss: 0.0138 acc: 99.48
ep:14100 loss: 0.0138 acc: 99.48
ep:14200 loss: 0.0138 acc: 99.48
ep:14300 loss: 0.0138 acc: 99.48
ep:14400 loss: 0.0135 acc: 99.48
ep:14500 loss: 0.0134 acc: 100.00
```

In epoch14500, it reached to 100% acc.

Picture generated:





Until now we know that increasing --init value slightly may give us a better performance of accuracy. Finally, we choose 0.15 as our good --init value for our model.

5. code is seen in file spiral.py

6. As I tried different init value several times, I found it training is the fastest (we get 100% in epochs 2200) when init value is 0.16:

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net short --hid 10 --epochs 20000 --init 0.16
ep: 100 loss: 0.6822 acc: 58.76
ep: 200 loss: 0.6635 acc: 63.92
ep: 300 loss: 0.6402 acc: 64.95
ep: 400 loss: 0.6132 acc: 57.22
ep: 500 loss: 0.5249 acc: 63.92
ep: 600 loss: 0.3685 acc: 68.56
ep: 700 loss: 0.3112 acc: 71.65
ep: 800 loss: 0.2759 acc: 75.77
ep: 900 loss: 0.2530 acc: 78.35
ep: 1000 loss: 0.2322 acc: 80.93
ep: 1100 loss: 0.2130 acc: 80.93
ep: 1200 loss: 0.1942 acc: 80.93
ep: 1300 loss: 0.1797 acc: 82.99
ep: 1400 loss: 0.1701 acc: 85.57
ep: 1500 loss: 0.1588 acc: 87.11
ep: 1600 loss: 0.1441 acc: 90.72
ep: 1700 loss: 0.1303 acc: 92.27
ep: 1800 loss: 0.1164 acc: 94.85
ep: 1900 loss: 0.1021 acc: 97.42
ep: 2000 loss: 0.0883 acc: 97.42
ep: 2100 loss: 0.0758 acc: 98.97
ep: 2200 loss: 0.0631 acc: 100.00
```

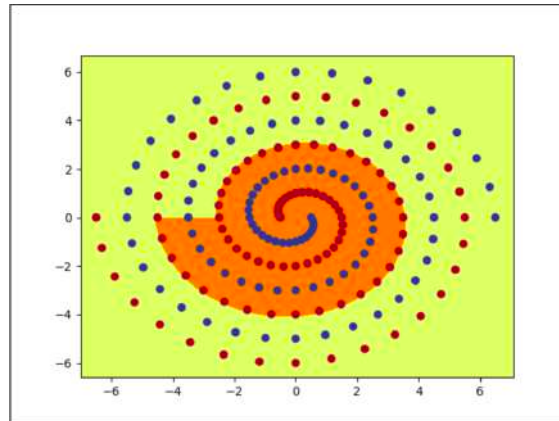
And it turns out the minimum number of hidden nodes per layer is 6. In other word, ShortNet cannot learn to correctly classify all of the training data within 20000 epochs when hidden nodes in each layer is 6.

```
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net short --hid 6 --epochs 20000 --init 0.16
ep: 100 loss: 0.6867 acc: 63.92
ep: 200 loss: 0.6718 acc: 62.89
ep: 300 loss: 0.6606 acc: 64.95
ep: 400 loss: 0.6440 acc: 63.40
ep: 500 loss: 0.6019 acc: 62.37
ep: 600 loss: 0.5171 acc: 65.98
ep: 700 loss: 0.4434 acc: 70.10
ep: 800 loss: 0.3967 acc: 69.59
ep: 900 loss: 0.3726 acc: 71.13
ep: 1000 loss: 0.3585 acc: 73.71
ep: 1100 loss: 0.3478 acc: 76.29
ep: 1200 loss: 0.3372 acc: 78.71
ep: 1300 loss: 0.3265 acc: 81.13
ep: 1400 loss: 0.3158 acc: 83.55
ep: 1500 loss: 0.3051 acc: 85.97
ep: 1600 loss: 0.2944 acc: 88.39
ep: 1700 loss: 0.2837 acc: 90.81
ep: 1800 loss: 0.2730 acc: 93.23
ep: 1900 loss: 0.2623 acc: 95.65
ep: 2000 loss: 0.2516 acc: 98.07
ep: 2100 loss: 0.2409 acc: 100.00
ep: 2200 loss: 0.2302 acc: 100.00
```

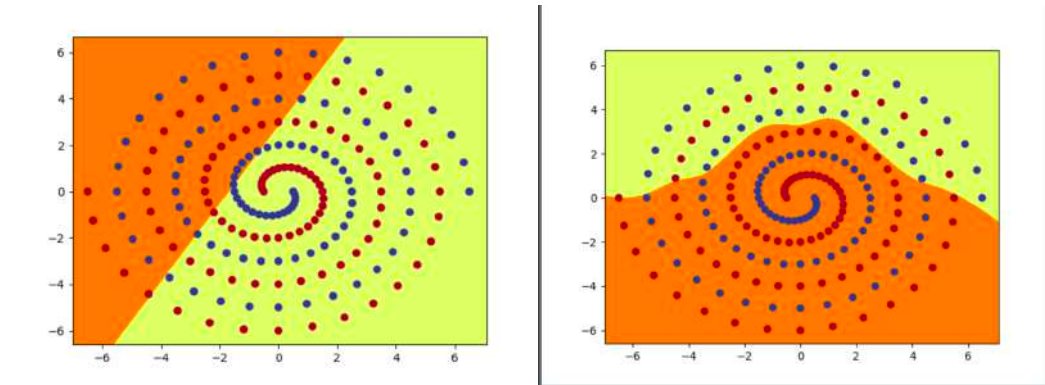
7. we have generated our hidden node pictures:

```
SteveMacBook-Pro:hw1 steve$ ls
__pycache__  polar1_4.png  raw1_1.png  raw1_9.png  raw2_7.png  short1_4.png  short2_2.png  short_out.png
data          polar1_5.png  raw1_2.png  raw2_0.png  raw2_8.png  short1_5.png  short2_3.png  spiral.py
kuzu.py       polar1_6.png  raw1_3.png  raw2_1.png  raw2_9.png  short1_6.png  short2_4.png  spiral_main.py
kuzu_main.py polar1_7.png  raw1_4.png  raw2_2.png  raw_out.png short1_7.png  short2_5.png  spirals.csv
polar1_0.png  polar1_8.png  raw1_5.png  raw2_3.png  short1_0.png short1_8.png  short2_6.png
polar1_1.png  polar1_9.png  raw1_6.png  raw2_4.png  short1_1.png short1_9.png  short2_7.png
polar1_2.png  polar_out.png raw1_7.png  raw2_5.png  short1_2.png short2_0.png  short2_8.png
polar1_3.png  raw1_0.png  raw1_8.png  raw2_6.png  short1_3.png short2_1.png  short2_9.png
```

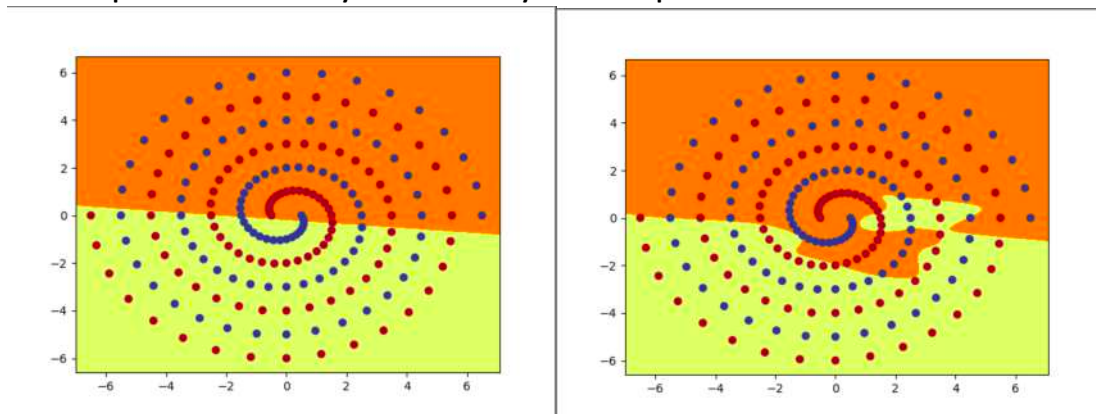
PolarNet part hidden layer1 output:



RawNet part hidden layer1 and layer2 output:

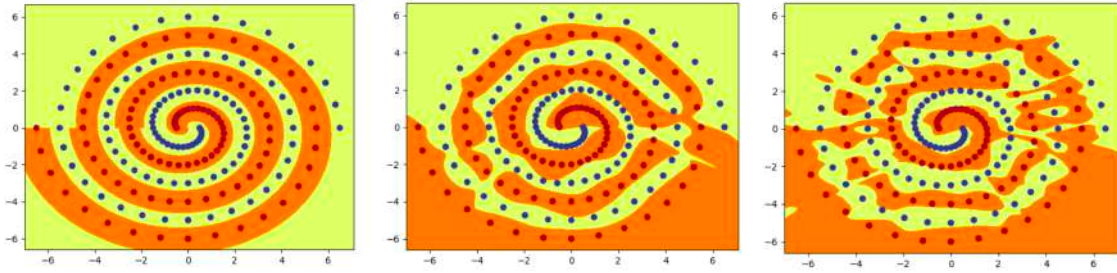


ShortNet part hidden layer1 and layer2 output:



**8. 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 of the three models, and a brief description of how the network uses these functions to achieve the classification**



For polarNet part, there is just one hidden layer, and inputs are all converted to polar coordinate. In this case, we could see that the output in hidden layer looks more like “polar shape”, which is very similar to what we would expect from a classification. And the features filtered out by hidden layer are more in line with the requirements of our classification. A good result similar to a “polar shape” can be finally achieved though the joint action of multiple nodes.

We can only get a linear output from hidden layer 1 for both rawNet part and shortNet part as we use linear model. And we may get some curve-shape boundary in terms of outputs of hidden layer 2. As we can see, raw and short model solve this classification problem mainly based on linear and multivariable non-linear functions. As a result, they do not fit the shape of polar coordinates.

In conclusion, for this two spiral problem, polar model is more suitable using to make decisions than raw and short models.

### **b. the effect of different values for initial weight size on the speed and success of learning, for both RawNet and ShortNet**

In essence, even if choosing the same value of --init parameter, we do some experiments from --init = 0.01 to 0.3 with an increment of 0.02 in both RawNet and ShortNet.

I test a list of values [0.01, 0.05, 0.1(default), 0.15, 0.2, 0.25, 0.3]:

It turns out that each result of training is random because the model initializes weights in a random way at the beginning.

For RawNet:

When we specify epochs 20000, it turns out if we choose --init values less than 0.1, it is relatively hard to train successfully.

```

SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw --hid 10 --epochs 20000 --init 0.05
ep: 100 loss: 0.7123 acc: 50.52
ep: 200 loss: 0.7125 acc: 50.52
ep: 300 loss: 0.7126 acc: 50.52
ep: 400 loss: 0.7127 acc: 50.52
ep: 500 loss: 0.7127 acc: 50.52
ep: 600 loss: 0.7127 acc: 50.52
ep: 700 loss: 0.7127 acc: 50.52
ep: 800 loss: 0.7127 acc: 50.52
ep: 900 loss: 0.7128 acc: 50.52
ep: 1000 loss: 0.7128 acc: 50.52
ep: 1100 loss: 0.7128 acc: 50.52
ep: 1200 loss: 0.7128 acc: 50.52
ep: 1300 loss: 0.7128 acc: 50.52

```

```

.....
ep:19600 loss: 0.7128 acc: 50.52
ep:19700 loss: 0.7128 acc: 50.52
ep:19800 loss: 0.7128 acc: 50.52
ep:19900 loss: 0.7128 acc: 50.52

```

When `--init` value is larger than 0.1, model can be trained to 100% within 20000 epochs although it is still random. Sometimes it trains fast, sometimes it does not. And when init value is close to 0.3, training speed will be a little slightly slow and inaccurate.

```

SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw --hid 10 --epochs 20000 --init 0.27
ep: 100 loss: 0.6749 acc: 61.34
ep: 200 loss: 0.6572 acc: 61.86
ep: 300 loss: 0.6259 acc: 54.64
ep: 400 loss: 0.5982 acc: 53.09
ep: 500 loss: 0.5451 acc: 61.86
ep: 600 loss: 0.4941 acc: 62.89

```

```

.....
ep: 8200 loss: 0.0524 acc: 98.45
ep: 8300 loss: 0.0524 acc: 98.45
ep: 8400 loss: 0.0501 acc: 98.97
ep: 8500 loss: 0.0479 acc: 98.97
ep: 8600 loss: 0.0460 acc: 99.48
ep: 8700 loss: 0.0444 acc: 100.00

```

But it does show that an insight that the init weight matters during our training. It cannot be either too large or too small. Init value 0.13 to 0.16 show the best performance for our model.

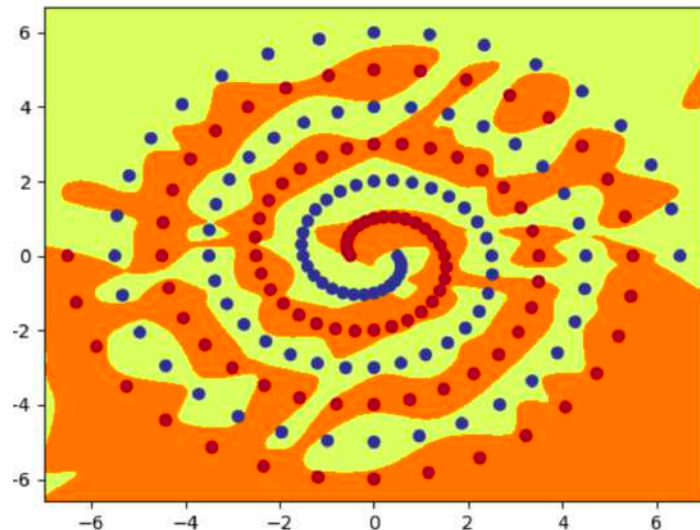
For ShortNet:

The performance is similar to RawNet, the only difference is that the training speed is a little bit faster than RawNet in terms of average training speed.

### c. the relative "naturalness" of the output function computed by the three networks, and the importance of representation for deep learning tasks in general

the "naturalness" of polarNet is much better than rawNet and shortNet. The reason is clear: polarNet use much simpler network structure to achieve a much more satisfied performance (it only uses one hidden layer).





Although rawNet and shortNet do the same effect in the training data set, they use more complex structure to implement that, and the final result is not so satisfied, because our model will perform bad on test set, it cannot predict the right shape for real shape.

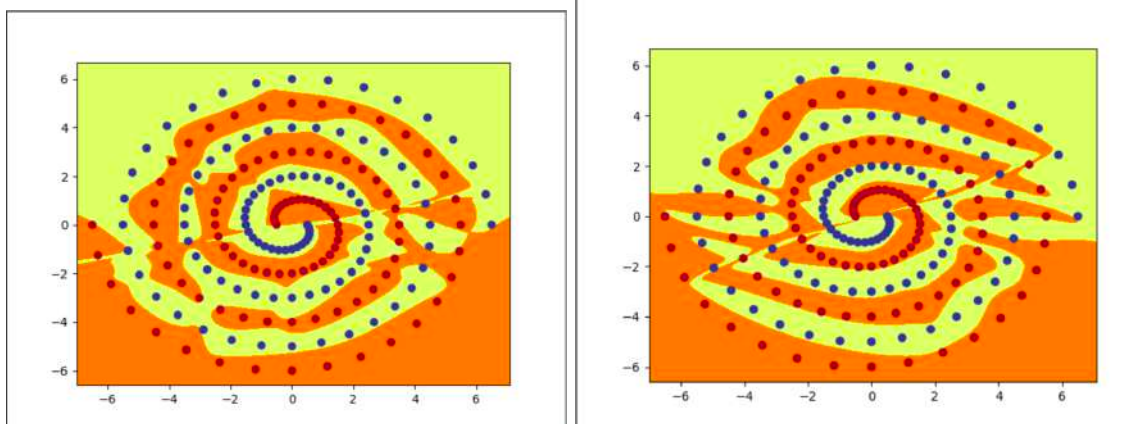
the suitable input data form matters a lot, if we can choose appropriate data form for our model, not only it can simplify our model, but also obtain a better accuracy.

**d. 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.** Firstly I choose rawNet to do some other experiments with some changes since rawNet is relatively a Linear model.

(1) Changing batch size:

```
train_dataset = torch.utils.data.TensorDataset(full_input, full_target)
# train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=97)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=194)
```

Here are figures generated when using batch size 97 and 194:



```

ep: 3300 loss: 0.0772 acc: 95.88
ep: 3400 loss: 0.0752 acc: 96.91
ep: 3500 loss: 0.0732 acc: 96.91
ep: 3600 loss: 0.0696 acc: 96.91
ep: 3700 loss: 0.0658 acc: 97.42
ep: 3800 loss: 0.0627 acc: 98.45
ep: 3900 loss: 0.0578 acc: 98.97
ep: 4000 loss: 0.0273 acc: 100.00
ep: 8800 loss: 0.0181 acc: 98.97
ep: 8900 loss: 0.0179 acc: 98.97
ep: 9000 loss: 0.0175 acc: 98.97
ep: 9100 loss: 0.0172 acc: 98.97
ep: 9200 loss: 0.0169 acc: 98.97
ep: 9300 loss: 0.0165 acc: 98.97
ep: 9400 loss: 0.0162 acc: 98.97
ep: 9500 loss: 0.0159 acc: 98.97
ep: 9600 loss: 0.0155 acc: 100.00

```

And we could see that final output figure do not change a lot, but training speed increased in some extent since our batch size increased.

batch\_size: 194 epoch 4000 reached 100%;

batch\_size: 97 epoch 9600 reached 100%.

(2) using SGD optimizer:

```

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

```

Before we change our optimizer to SGD, we obtain a relative good result.

```

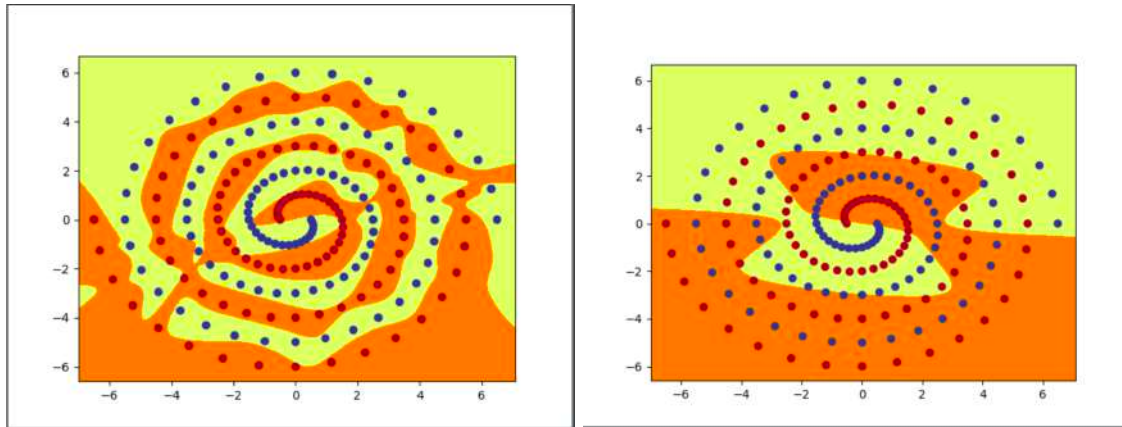
SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw --hid 10 --epochs 20000 --init 0.16
ep: 100 loss: 0.6964 acc: 59.28
ep: 200 loss: 0.6864 acc: 63.92
ep: 300 loss: 0.6856 acc: 60.82
ep: 400 loss: 0.6850 acc: 57.22
ep: 500 loss: 0.6845 acc: 57.73
ep: 600 loss: 0.6841 acc: 57.73
ep: 700 loss: 0.6838 acc: 57.73
ep: 800 loss: 0.6836 acc: 57.73
ep: 900 loss: 0.6834 acc: 57.22
ep: 1000 loss: 0.6833 acc: 56.70
ep: 1100 loss: 0.6831 acc: 57.73
ep: 1200 loss: 0.6831 acc: 56.70
ep: 1300 loss: 0.6828 acc: 57.22
ep: 1400 loss: 0.6827 acc: 57.22

```

```

.....
ep: 6100 loss: 0.1570 acc: 96.59
ep: 6500 loss: 0.1857 acc: 96.39
ep: 6600 loss: 0.1752 acc: 96.91
ep: 6700 loss: 0.1500 acc: 96.91
ep: 6800 loss: 0.1257 acc: 97.42
ep: 6900 loss: 0.0886 acc: 100.00

```



After we change model optimizer to SGD for two spiral problem:  
It turns out our training process faces a very serious difficulty

```

SteveMacBook-Pro:hw1 steve$ python3 spiral_main.py --net raw --hid 10 --epochs 20000 --init 0.16
ep: 100 loss: 0.6936 acc: 48.45
ep: 200 loss: 0.6970 acc: 49.48
ep: 300 loss: 0.6999 acc: 50.00
ep: 400 loss: 0.7024 acc: 50.00
ep: 500 loss: 0.7045 acc: 50.52
ep: 600 loss: 0.7062 acc: 50.00
ep: 700 loss: 0.7075 acc: 50.52
ep: 800 loss: 0.7084 acc: 49.48
ep: 900 loss: 0.7091 acc: 50.00
ep: 1000 loss: 0.7096 acc: 50.52
ep: 1100 loss: 0.7100 acc: 50.52
ep: 1200 loss: 0.7102 acc: 50.52
ep: 1300 loss: 0.7103 acc: 50.52
ep: 1400 loss: 0.7104 acc: 50.52
ep: 1500 loss: 0.7104 acc: 50.52
ep: 1600 loss: 0.7104 acc: 50.52

```

```

.....
ep:19100 loss: 0.6775 acc: 58.25
ep:19200 loss: 0.6774 acc: 58.25
ep:19300 loss: 0.6772 acc: 58.25
ep:19400 loss: 0.6770 acc: 58.25
ep:19500 loss: 0.6768 acc: 58.25
ep:19600 loss: 0.6766 acc: 58.25
ep:19700 loss: 0.6764 acc: 58.25
ep:19800 loss: 0.6762 acc: 58.25
ep:19900 loss: 0.6760 acc: 58.25

```

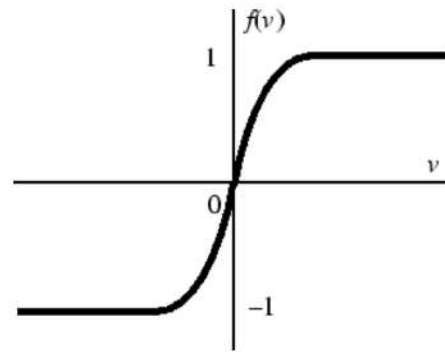
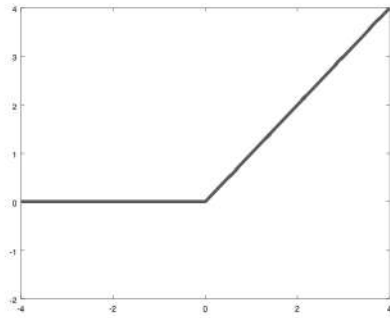
We test many times, but it shows we will get a very bad model for our problem if changing Adam to SGD. Within 20000 epochs, the accuracy is no more than 60% although we do lots of tests.

(3) using relu activation functions:

We change our activate function to ReLU, the performance of our model will be worse as well.

Since we different activation functions have different ranges:

For ReLU: range is: (0, +integer); For Tanh: range is: (-1, 1);



## Rectified Linear Unit (ReLU)

```
def __init__(self, num_hid):
    super(RawNet, self).__init__()
    self.full_connected_1 = nn.Linear(2, num_hid, bias=True) # [2, num_hid]
    self.full_connected_2 = nn.Linear(num_hid, num_hid, bias=True) # [num_hid, num_hid]
    self.full_connected_3 = nn.Linear(num_hid, 1, bias=True) # [num_hid, 1]
    # self.Tanh = nn.Tanh()
    self.ReLU = nn.ReLU()
    self.Sigmoid = nn.Sigmoid()

    """
    input.shape: [64, 2]
    batch_size: 64 coordinate: (x, y)
    """

    def forward(self, input):
        x = self.full_connected_1(input)
        # self.hid1 = self.Tanh(x)
        self.hid1 = self.ReLU(x)
        x = self.full_connected_2(self.hid1)
        # self.hid2 = self.Tanh(x)
        self.hid2 = self.ReLU(x)
        x = self.full_connected_3(self.hid2)
        output = self.Sigmoid(x)

        return output
```

SteveMacBook-Pro:hw1 steve\$ python3 spiral\_main.py --net raw --hid 10 --epochs 20000 --init 0.16

```

ep: 100 loss: 0.6974 acc: 50.52
ep: 200 loss: 0.6977 acc: 51.55
ep: 300 loss: 0.6983 acc: 54.12
ep: 400 loss: 0.6991 acc: 57.22
ep: 500 loss: 0.7001 acc: 55.15
ep: 600 loss: 0.7010 acc: 55.15
ep: 700 loss: 0.7019 acc: 56.19
ep: 800 loss: 0.7028 acc: 56.70
ep: 900 loss: 0.7035 acc: 57.22
ep: 1000 loss: 0.7041 acc: 56.70
ep: 1100 loss: 0.7047 acc: 56.70
ep: 1200 loss: 0.7052 acc: 56.70
ep: 1300 loss: 0.7058 acc: 56.70

.....
ep:19100 loss: 0.6642 acc: 63.92
ep:19200 loss: 0.6640 acc: 63.92
ep:19300 loss: 0.6638 acc: 63.92
ep:19400 loss: 0.6637 acc: 63.92
ep:19500 loss: 0.6635 acc: 63.92
ep:19600 loss: 0.6634 acc: 63.92
ep:19700 loss: 0.6632 acc: 63.92
ep:19800 loss: 0.6630 acc: 63.40
ep:19900 loss: 0.6628 acc: 63.40
  
```

The result shows that our model find it harder to train and will get a worse accuracy at the end if we change our activation function to ReLU() in our model rawNet.

It also gives us an extra information that choosing an appropriate activation function matters when we are training a model.