COMP9444 20T2 Project 1

Student Name: Raymond Lu

Student Number: z5277884

Part 1: Japanese Character Recognition

Note:

To reproduce the same results in this project, "*torch.manual_seed(0)*" is added to the "kuzu_main.py" file.

Question 1

The confusion matrix is:

```
[[765.  7.  8.  5. 63.  8.  5. 17. 11.  8.]
 [ 5. 669. 61. 36. 53. 28. 21. 29. 41. 49.]
 [ 7. 110. 692. 60. 79. 125. 146. 28. 97. 90.]
 [ 12. 17. 26. 755. 21. 16. 10. 10. 40.  3.]
 [ 30. 28. 26. 14. 622. 20. 28. 83.  6. 49.]
 [ 66. 23. 20. 60. 17. 727. 25. 15. 30. 32.]
 [ 2. 59. 48. 13. 32. 26. 720. 54. 44. 19.]
 [ 61. 13. 36. 18. 35.  7. 21. 627.  6. 31.]
 [ 32. 25. 45. 27. 20. 33. 10. 88. 702. 39.]
 [ 20. 49. 38. 12. 58. 10. 14. 49. 23. 680.]]
```

The final Accuracy: 6959/10000 (**70%**)

Question 2

After creating a loop to test the results of different hidden layer nodes, the outcome reveals that when the number of hidden layer nodes is **260**, the accuracy reaches the summit.

Hidden Layer Node Number	Accuracy (%)
10	68.49
20	76.08
30	78.59
40	80.62
50	81.71
60	82.08
70	82.5
80	83.07
90	83.3
100	84.24
110	84.49
120	83.87
130	84.27
140	84.37
150	84.72
160	84.76
170	84.16
180	84.18
190	84.31

200	84.65
210	84.85
220	84.74
230	84.82
240	84.73
250	84.93
260	85.06
270	84.66
280	84.83
290	84.72

The confusion matrix is:

```
[[845.  7.  8.  4. 45. 10.  3. 16. 12.  6.]
 [ 3. 815.  9.  7. 26. 12. 11. 14. 30. 21.]
 [ 3. 33. 840. 27. 17. 79. 43. 20. 30. 38.]
 [ 7.  3. 42. 924.  6. 10.  7.  5. 55.  6.]
 [ 28. 16. 11.  2. 812. 11. 16. 18.  3. 30.]
 [ 40. 10. 17. 13.  9. 834.  8. 11.  7.  6.]
 [ 3. 62. 30.  8. 31. 23. 897. 34. 29. 23.]
 [ 35.  6. 12.  1. 17.  1.  5. 832.  3. 16.]
 [ 29. 19. 17.  6. 20. 15.  1. 16. 823. 13.]
 [ 7. 29. 14.  8. 17.  5.  9. 34.  8. 841.]]
```

The final accuracy: 8463/10000 (**85%**).

Question 3

In the first convolutional layer, the *in_channels* is 1, the *out_channels* is 64, and the *kernel_size* is 5. In the second convolutional layer, the *in_channels* is 64, the *out_channels* is 128, and the *kernel_size* is 5.

The confusion matrix is:

```
[[962.  2. 15.  1. 15.  4.  5.  9.  4.  5.]
 [ 4. 948.  6.  1. 11. 16.  8.  3. 16.  6.]
 [ 1.  5. 904. 17.  2. 49. 20.  6. 10.  3.]
 [ 1.  1. 29. 965.  3.  4.  1.  1.  4.  1.]
 [ 17.  9.  6.  4. 938.  3.  6.  3.  7.  5.]
 [ 1.  0.  4.  1.  4. 906.  2.  2.  3.  0.]
 [ 1. 21. 17.  4. 11. 10. 952.  5.  4.  1.]
 [ 8.  2.  6.  3.  6.  3.  2. 951.  2.  2.]
 [ 2.  1.  4.  2.  4.  2.  0.  2. 947.  2.]
 [ 3. 11.  9.  2.  6.  3.  4. 18.  3. 975.]]
```

The final accuracy: 9448/10000 (**94%**).

Question 4

a.

From the outputs of the three models, it could be seen that the convolutional network has the highest accuracy, with the fully connected 2-layer network ranks the second. And the fully connected 1-layer network has the worst performance.

The analysis of the result above is given below.

For the comparison between NetLin and NetFull, as Netfull has two layers, it owns a bigger capacity of network, and can execute more complicated functions such as xor, which cannot be realized by a 1-layer fully connected network. And NetFull can separate features into smaller granularity, which means a more precise output could be generated after computing local features in layers. This could be represented as Japanese characters are divided into smaller proportions for the classification analysis. From the perspective of training data, Japanese character recognition requires complex mappings which are supposed to be realized by the expressive power of multiple layers.

As for the comparison between NetFull and NetConv, it could be explained in the view of the difference between fully connected network and convolutional neural network. Generally speaking, NetConv is more specialized, and efficient than NetFull.

In NetFull, each neuron is connected to every neuron in the previous layer, and each connection has its weight. It makes no assumption about the features in the Kuzushiji dataset. In contrast, in a convolutional layer, each neuron is only connected to a few nearby neurons in the previous layer and the same set of weights is used for every neuron. In this scenario, each neuron is connected to significant neurons in the previous layer and fewer number of connections and weights introduces fewer noises in the training process. Moreover, in the project design, the first convolutional layer has 64 hidden nodes and the second convolutional layer has 128 hidden nodes. This means Japanese characters are segmented into even smaller granularities. As we know that smaller granularity brings higher complexity, the training data is better fitted in NetConv. Therefore, NetConv utilizes the position information better and ignores insignificant data with trivial weights. This is reflected in the result that the accuracy of NetConv could be up to 94%, an ideal level that could recognize Japanese characters outstandingly.

b.

NetLin:

In this model, the characters “き”, “は”, “ま”, “**れ**”, and “を” are most likely mistaken for the character “す”. Also, “や” is most likely mistaken for “な”.

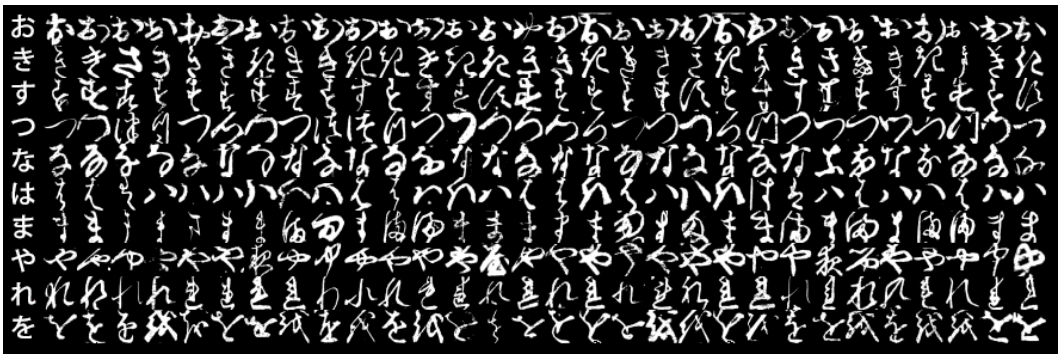

Figure 1. The 10 classes of Kuzushiji-MNIST, with the first column showing each character's modern hiragana counterpart.

From Fig 1, it could be seen that the confusion pairs above look similar in some of their handwriting. The idea here is that NetLin only extracts general features of Japanese characters and it cannot go deeper into finer details.

NetFull:

In this model, there are three significantly confused character pairs, which are: character "は" is still most likely mistaken to "す"; character "れ" is easily mistaken to "つ"; character "き" is likely mistaken to "ま".

In this scenario, compared to NetLin, the outcome gets improved. These two-character pairs have highly similar constitutional parts in their structures. In the first pair, cross "十" part and circle " " part make NetFull mistakenly predict. In the second pair, the curve parts look similar in both characters. In the third pair, the "キ" part and curve "  " parts contribute to the wrong classification.

With an extra fully connected layer, the NetFull can detect finer details but the feature granularity is not good enough to make a precise classification.

NetConv:

In this model, only one pair of significant confusion still exists – the character "は" is still most likely mistaken to "す".

Due to the similar parts introduced above, this pair is still hard to distinguish even we have two convolutional layers. This indicates that a more complex model is required to identify these two characters. However, if we increase the hidden node numbers in the two convolutional layers, the training time will also grow, and long training time is not preferred in this project.

c.

1. Other Structure:

In "kuzu_main.py", we change the optimizer from Stochastic Gradient Descent to Adam Algorithm and do not change other meta-parameters, and the results are given below. The code for Adam Optimizer is: *optimizer = optim.Adam(net.parameters(), lr=args.lr)*.

NetLin:

The confusion matrix is:

```
[[582.  3.  3.  4. 28.  5.  5.  6.  4.  5.]
 [ 6. 675. 58. 28. 49. 33. 46. 37. 58. 79.]
 [ 3. 76. 568. 41. 59. 63. 86. 14. 31. 48.]
 [ 17. 22. 47. 676. 36.  7. 32. 11. 77.  5.]
 [ 45. 26. 50. 16. 597. 13. 36. 61. 15. 36.]
 [170. 37. 58. 143. 33. 780. 67. 20. 71. 39.]
 [ 1. 30. 44.  8. 24. 14. 545. 35. 24.  6.]
```

[111. 45. 79. 39. 82. 37. 138. 691. 18. 99.]
[39. 27. 55. 32. 21. 36. 28. 75. 680. 35.]
[26. 59. 38. 13. 71. 12. 17. 50. 22. 648.]]

The final accuracy: 6442/10000 (**64%**)

NetFull:

The confusion matrix is:

[[717. 2. 4. 4. 25. 5. 2. 9. 19. 3.]
[4. 571. 12. 5. 27. 17. 9. 6. 32. 17.]
[3. 71. 613. 33. 52. 91. 104. 36. 22. 156.]
[13. 19. 121. 880. 25. 33. 46. 21. 87. 24.]
[56. 36. 15. 3. 692. 18. 24. 36. 16. 26.]
[91. 22. 42. 47. 19. 737. 6. 10. 27. 5.]
[15. 135. 77. 4. 58. 52. 768. 67. 23. 23.]
[62. 25. 32. 10. 49. 9. 19. 733. 18. 18.]
[27. 46. 45. 6. 22. 27. 7. 44. 735. 46.]
[12. 73. 39. 8. 31. 11. 15. 38. 21. 682.]]

The final accuracy: 7128/10000 (**71%**).

NetConv:

The confusion matrix is:

[[944. 2. 8. 1. 49. 3. 3. 6. 3. 4.]
[10. 905. 9. 0. 16. 5. 9. 16. 15. 9.]
[2. 4. 836. 11. 2. 42. 22. 42. 9. 11.]
[0. 2. 38. 948. 8. 11. 6. 3. 9. 4.]
[8. 7. 11. 2. 831. 7. 13. 15. 9. 12.]
[6. 1. 5. 16. 1. 870. 0. 1. 3. 1.]
[12. 52. 75. 17. 47. 41. 937. 78. 27. 42.]
[7. 9. 9. 3. 19. 9. 5. 801. 6. 8.]
[7. 10. 0. 2. 19. 7. 3. 13. 913. 3.]
[4. 8. 9. 0. 8. 5. 2. 25. 6. 906.]]

The final accuracy: 8891/10000 (**89%**).

From the result above, it reveals that Adam Algorithm is inferior to Stochastic Gradient Descent in Japanese character recognition. Even though Adam has merit of converging faster, from the outcome, we can see that Adam fails to converge in the optimal solution in this case. In the trade-off between speed and accuracy, SGD is preferred as its speed is slightly slower than Adam.

2. Other Meta-parameters (only one parameter changed from default setting):

2.1 Learning Rate:

lr = 0.1

NetLin:

The confusion matrix is:

[[685. 3. 5. 2. 38. 5. 3. 13. 7. 6.]
[9. 736. 124. 42. 105. 41. 64. 45. 72. 96.]
[5. 20. 450. 20. 28. 41. 31. 13. 17. 28.]
[23. 24. 57. 817. 39. 23. 20. 32. 90. 6.]
[25. 12. 38. 12. 574. 9. 15. 81. 6. 40.]
[106. 29. 45. 49. 25. 776. 42. 30. 34. 41.]
[2. 94. 136. 19. 84. 43. 783. 97. 65. 35.]
[67. 7. 19. 12. 24. 7. 12. 550. 5. 28.]
[44. 18. 74. 12. 19. 38. 16. 86. 672. 38.]
[34. 57. 52. 15. 64. 17. 14. 53. 32. 682.]]

The final accuracy: 6725/10000 (**67%**).

NetFull:

The confusion matrix is:

[[900. 3. 9. 3. 30. 5. 5. 15. 8. 8.]
[7. 872. 19. 4. 29. 16. 19. 7. 24. 10.]
[2. 27. 861. 24. 19. 67. 46. 24. 24. 25.]
[0. 1. 34. 940. 5. 7. 8. 4. 25. 7.]
[23. 9. 7. 2. 834. 5. 11. 8. 2. 13.]
[18. 8. 22. 12. 11. 877. 4. 4. 7. 3.]
[4. 34. 16. 5. 18. 9. 890. 19. 11. 11.]
[31. 4. 8. 2. 15. 1. 7. 882. 1. 11.]
[13. 15. 10. 3. 15. 4. 3. 16. 895. 17.]
[2. 27. 14. 5. 24. 9. 7. 21. 3. 895.]]

The final accuracy: 8846/10000 (**88%**).

NetConv:

The confusion matrix is:

[[960. 2. 12. 0. 16. 1. 1. 5. 2. 4.]
[2. 939. 3. 0. 2. 1. 3. 5. 2. 3.]
[0. 12. 933. 5. 1. 20. 12. 4. 5. 8.]
[2. 0. 14. 982. 10. 2. 1. 1. 5. 1.]
[19. 7. 6. 1. 948. 1. 6. 3. 7. 2.]
[1. 0. 7. 2. 5. 959. 3. 1. 4. 0.]
[0. 27. 10. 6. 2. 8. 973. 6. 1. 0.]
[9. 5. 8. 1. 2. 2. 1. 958. 0. 2.]
[4. 5. 4. 1. 9. 4. 0. 7. 973. 5.]
[3. 3. 3. 2. 5. 2. 0. 10. 1. 975.]]

The final accuracy: 9600/10000 (**96%**).

lr = 0.001

NetLin:

The confusion matrix is:

[[758. 3. 8. 4. 66. 11. 5. 16. 7. 8.]
[6. 657. 72. 55. 53. 40. 24. 36. 35. 73.]
[8. 105. 656. 51. 72. 138. 140. 23. 66. 70.]
[11. 15. 36. 737. 23. 24. 8. 21. 45. 3.]
[31. 34. 21. 14. 622. 21. 28. 93. 8. 63.]
[62. 22. 17. 47. 22. 687. 22. 20. 39. 27.]
[4. 81. 59. 13. 29. 36. 734. 68. 49. 28.]
[56. 10. 26. 15. 26. 9. 16. 537. 4. 24.]
[41. 25. 63. 51. 27. 27. 13. 138. 727. 52.]
[23. 48. 42. 13. 60. 7. 10. 48. 20. 652.]]

The final accuracy: 6767/10000 (**68%**).

NetFull:

The confusion matrix is:

[[776. 6. 8. 5. 75. 9. 3. 16. 8. 5.]
[4. 672. 75. 47. 52. 38. 23. 27. 41. 62.]
[7. 102. 675. 51. 65. 136. 143. 24. 52. 73.]
[10. 16. 35. 761. 21. 25. 8. 17. 38. 5.]
[30. 30. 22. 16. 642. 23. 33. 103. 6. 73.]
[52. 14. 14. 36. 12. 693. 13. 18. 32. 19.]
[4. 79. 60. 14. 24. 30. 742. 73. 49. 23.]
[55. 9. 26. 13. 25. 7. 14. 542. 4. 28.]
[46. 25. 47. 40. 26. 31. 11. 121. 752. 43.]
[16. 47. 38. 17. 58. 8. 10. 59. 18. 669.]]

The final accuracy: 6924/10000 (**69%**).

NetConv:

The confusion matrix is:

[[860. 8. 8. 7. 67. 12. 1. 15. 11. 5.]
[6. 808. 14. 6. 6. 12. 12. 6. 44. 15.]
[4. 21. 827. 45. 17. 72. 45. 17. 37. 42.]
[5. 2. 54. 901. 21. 17. 11. 5. 90. 8.]
[42. 18. 16. 6. 824. 13. 26. 44. 6. 36.]
[45. 20. 12. 16. 6. 816. 7. 8. 24. 7.]
[10. 72. 38. 7. 30. 32. 880. 50. 32. 16.]
[24. 10. 10. 1. 8. 4. 12. 813. 5. 17.]
[2. 7. 7. 4. 15. 15. 2. 7. 745. 7.]
[2. 34. 14. 7. 6. 7. 4. 35. 6. 847.]]

The final accuracy: 8321/10000 (**83%**).

From the result above, it could be seen that for the learning rate, the value of 0.01 is not optimal since when the learning rate is 0.1, the accuracies increase in NetFull and NetConv. This could be interpreted as when the learning rate increases, the model converges to the optimal faster. It could be also reflected in when the learning rate is 0.01, the accuracy is lower than the ones of 0.01.

2.2 Momentum:

mom = 1.0

NetLin:

The confusion matrix is:

```
[[470.  0.  1.  0. 16.  1.  4.  2.  2.  0.]
 [  7. 635. 132.  21.  61.  67.  38.  48.  40. 111.]
 [ 12.  45. 451.  39. 127.  66.  96.  34.  12.  95.]
 [ 49.  21.  33. 680.  34.  22.  7.  20.  31.  4.]
 [ 54.  11.  19.  12. 444.  7.  7.  50.  4.  44.]
 [134.  24.  38.  59.  20. 697.  22.  40.  21.  21.]
 [ 15. 132. 137.  53. 148.  49. 778. 160.  38.  84.]
 [ 54.  2.  3.  5.  17.  3.  4. 377.  1.  8.]
 [151.  95. 159. 115.  62.  80.  39. 231. 846. 119.]
 [ 54.  35.  27.  16.  71.  8.  5.  38.  5. 514.]]
```

The final accuracy: 5892/10000 (**59%**).

NetFull:

The confusion matrix is:

```
[[508.  37.  7.  17.  46.  28.  24.  58.  21.  33.]
 [ 34.  30.  1.  7.  18.  1.  10.  14.  2.  7.]
 [166. 567. 620.  94. 621. 403. 545. 419. 297. 742.]
 [238.  34.  91. 602. 108. 108.  27.  99. 240.  67.]
 [ 3.  1.  0.  1. 53.  0.  1. 22.  0.  1.]
 [ 2.  1.  0.  5.  0. 32.  0.  1.  0.  0.]
 [ 3.  34. 107.  22.  22. 160. 177. 190.  36.  27.]
 [ 17.  6.  8.  7.  4.  95.  20.  42.  3.  1.]
 [ 29. 290. 165. 244. 128. 173. 196. 155. 401. 111.]
 [ 0.  0.  1.  1.  0.  0.  0.  0.  0. 11.]]
```

The final accuracy: 2476/10000 (**25%**).

NetConv:

The confusion matrix is:

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

The final accuracy: 1000/10000 (**10%**).

mom = 0.1

NetLin:

The confusion matrix is:

```
[[764.  5.  7.  4. 59.  7.  4. 14.  9.  9.]
 [ 6. 678. 70. 41. 57. 30. 23. 31. 43. 52.]
 [ 8. 102. 681. 59. 78. 127. 146. 28. 86. 86.]
 [ 12. 18. 29. 754. 18. 20. 12. 13. 45.  4.]
 [ 29. 27. 25. 15. 615. 19. 20. 75.  7. 48.]
 [ 64. 25. 22. 55. 23. 719. 21. 19. 31. 27.]
 [ 2. 63. 47. 11. 33. 29. 727. 54. 45. 23.]
 [ 62. 11. 33. 18. 36.  9. 20. 618.  5. 26.]
 [ 33. 21. 45. 33. 23. 30. 12. 99. 702. 41.]
 [ 20. 50. 41. 10. 58. 10. 15. 49. 27. 684.]]
```

The final accuracy: 6942/10000 (**69%**).

NetFull:

The confusion matrix is:

```
[[825.  4.  8.  4. 60. 11.  4. 19. 10.  5.]
 [ 3. 787. 20.  9. 38. 19. 19. 15. 32. 29.]
 [ 3. 31. 812. 46. 24. 81. 71. 18. 32. 49.]
 [ 7.  4. 42. 896. 10. 14.  7.  7. 56.  5.]
 [ 28. 23. 10.  3. 752. 11. 19. 30.  5. 40.]
 [ 41. 15. 15. 14. 11. 801.  7. 12. 13.  9.]
 [ 3. 72. 36.  7. 35. 27. 854. 48. 36. 28.]
 [ 41.  8. 14.  5. 23.  3.  6. 761.  3. 15.]
 [ 41. 21. 22.  7. 23. 25.  3. 43. 804. 12.]
 [ 8. 35. 21.  9. 24.  8. 10. 47.  9. 808.]]
```

The final accuracy: 8100/10000 (**81%**).

NetConv:

The confusion matrix is:

```
[[953.  3. 10.  3. 18.  4.  1.  8.  4.  5.]
 [ 6. 929.  9.  2.  8. 15.  8.  6. 19.  5.]
 [ 1.  4. 871. 19.  2. 48. 13.  8. 12.  4.]
 [ 1.  0. 45. 956.  3.  4.  1.  1. 10.  3.]
 [ 23. 13. 12.  3. 935.  5.  7.  5. 21.  8.]
 [ 1.  0.  7.  2.  1. 886.  1.  0.  4.  0.]
 [ 2. 36. 31.  6. 14. 28. 967. 11.  7.  2.]
 [ 9.  4.  5.  2.  7.  3.  2. 938.  4.  5.]
 [ 1.  3.  3.  1.  6.  2.  0.  2. 918.  4.]
 [ 3.  8.  7.  6.  6.  5.  0. 21.  1. 964.]]
```

The final accuracy: 9317/10000 (**93%**).

From the result above, it could be seen that when we choose the value of momentum, a big value could lead to a terrible performance of a model. Compared to $mom = 0.1$ and $mom = 1.0$, the original value 0.5 has the best performance. Momentum is introduced to accelerate the SGD and to dampen oscillations. If momentum is comparatively small, the model's convergence could go slowly because the SGD oscillates. However, from the outcome of $mom = 1.0$, with a high momentum value, the model fails to converge to the optimal, and worse still loses a proper capability of classification.

Part 2: Twin Spiral Task

Note:

To reproduce the same results in this project, "*torch.manual_seed(0)*" is added to the "spiral_main.py" file.

Question 2

The minimum value of the hidden node number is **7**. The output picture is given below.

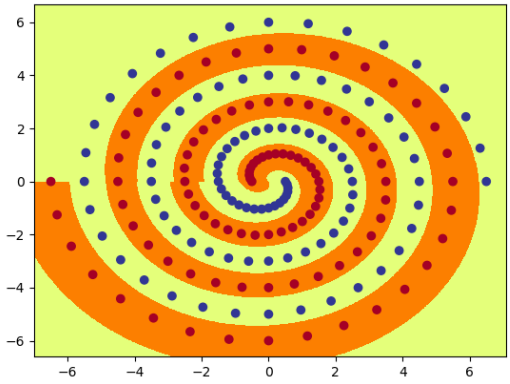

Figure 2. Polar_out.png

From Figure 2, it could be seen that all points are correctly classified, and the segmentation area is clean and neat.

Question 4

The minimum value of the initial weight is **0.121**. In this question, the initial weight is 0.121, the number of hidden nodes in each hidden layer is 10, the learning rate is 0.01, and the max training epoch number is 20000.

The output picture is given below.

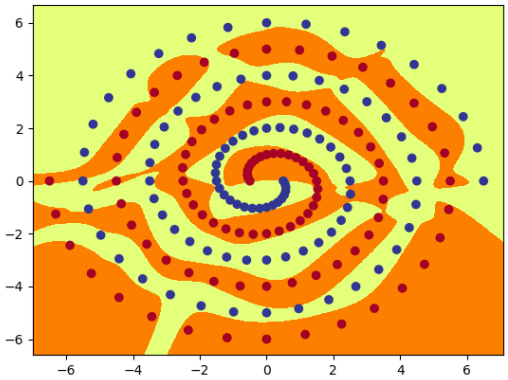

Figure 3. Raw_out.png

Compared to “polar_out.png”, the segmented regions are fragmentary and irregular.

Question 6

To get a good value of initial weight, first we just tune the value of initial weight to find out which initial weight value can have the fewest training epochs.

In the experiments, we find that when the initial weight value is 0.109 and 0.111, the training processes need 700 epochs to finish. When the initial weight value is 0.11, the training epoch is 600. Therefore, 0.11 is a good value for initial weight.

Then, we start tuning the value of hidden layer nodes, and find out the minimum number is 4.

In this question, the initial weight is set to be 0.1, the minimum number of hidden nodes is **4**, the learning rate is 0.01, and the maximum training epoch is 20000.

The output picture is given below.

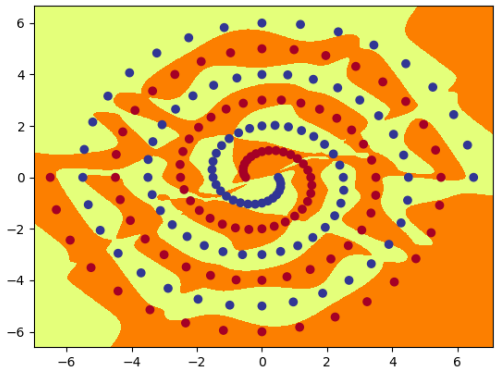

Figure 4. Short_out.png

Compared to “polar_out.png”, the segmented regions are fragmentary and irregular. And the difference between “raw_out.png” and “short_out.png” reveals that in ShortNet, the generation of segmentation areas is not necessarily based on real data points as on the top right-hand side corner of “short_out.png”, an orange region is generated but without any red dots scattering on.

Question 7

PolarNet:

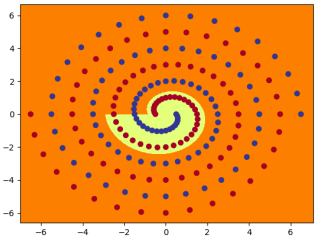

Figure 5. PolarNet Layer 1 Node 0

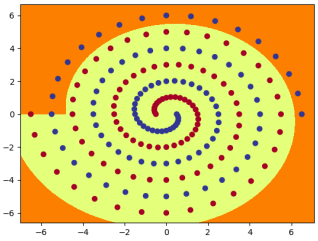

Figure 6. PolarNet Layer 1 Node 1

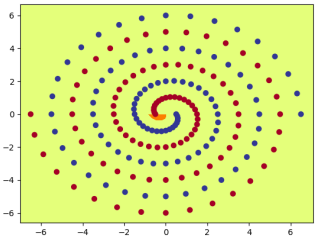

Figure 7. PolarNet Layer 1 Node 2

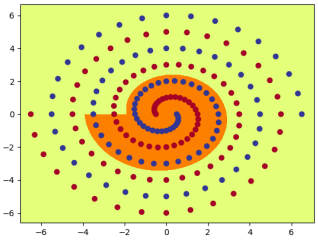

Figure 8. PolarNet Layer 1 Node 3

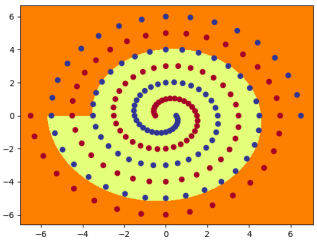

Figure 9. PolarNet Layer 1 Node 4

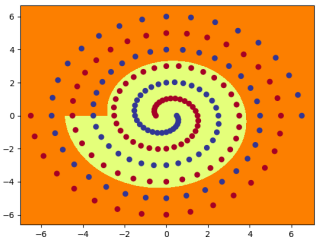

Figure 10. PolarNet Layer 1 Node 5

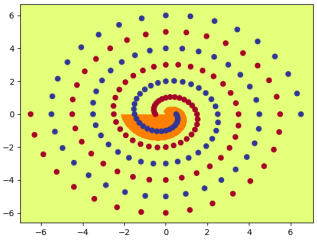

Figure 11. PolarNet Layer 1 Node 6

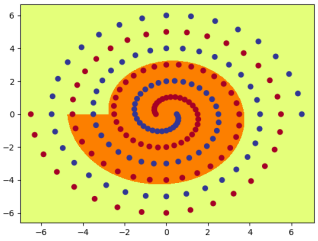

Figure 12. PolarNet Layer 1 Node 7

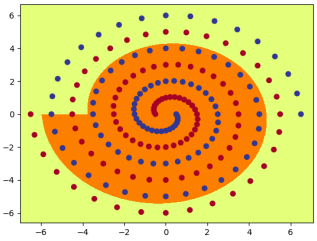

Figure 13. PolarNet Layer 1 Node 8

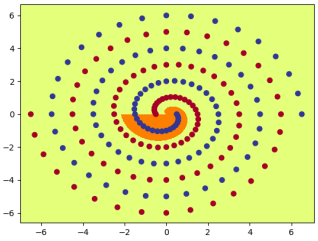

Figure 14. PolarNet Layer 1 Node 9

RawNet:

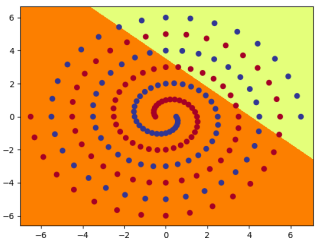

Figure 15. RawNet Layer 1 Node 0

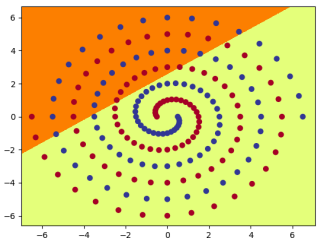

Figure 16. RawNet Layer 1 Node 1

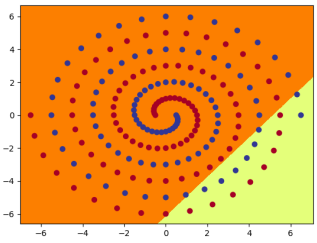

Figure 17. RawNet Layer 1 Node 2

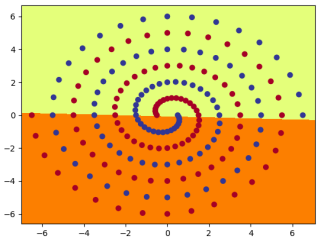

Figure 18. RawNet Layer 1 Node 3

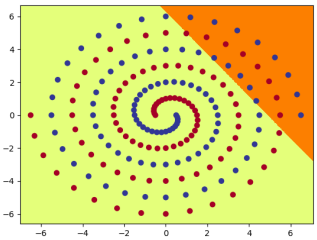

Figure 19. RawNet Layer 1 Node 4

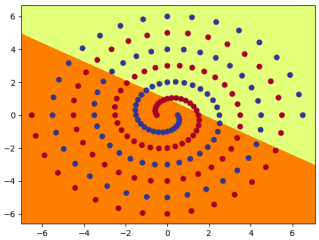

Figure 20. RawNet Layer 1 Node 5

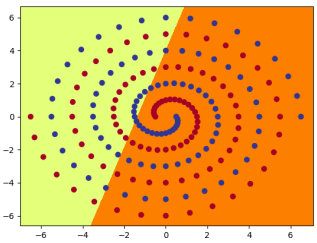

Figure 21. RawNet Layer 1 Node 6

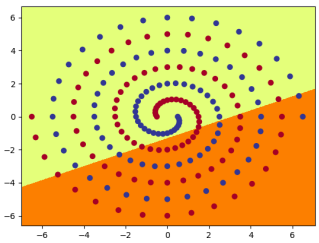

Figure 22. RawNet Layer 1 Node 7

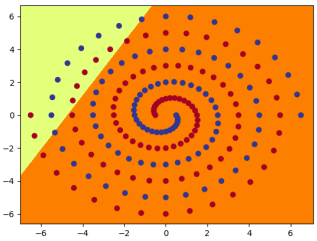

Figure 23. RawNet Layer 1 Node 8

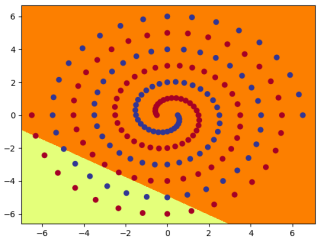

Figure 24. RawNet Layer 1 Node 9

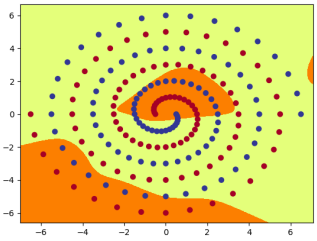

Figure 25. RawNet Layer 2 Node 0

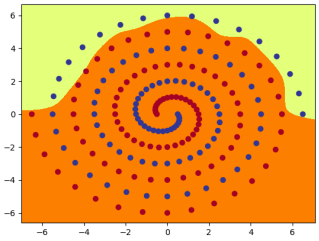

Figure 26. RawNet Layer 2 Node 1

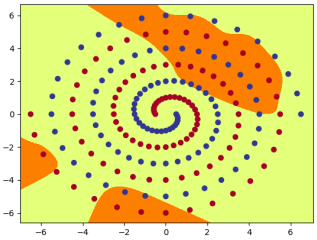

Figure 27. RawNet Layer 2 Node 2

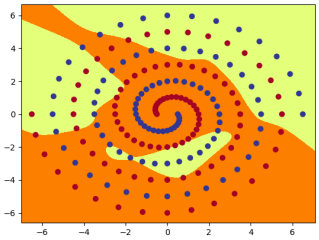

Figure 28. RawNet Layer 2 Node 3

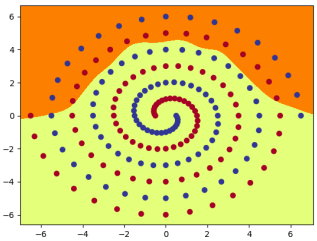

Figure 29. RawNet Layer 2 Node 4

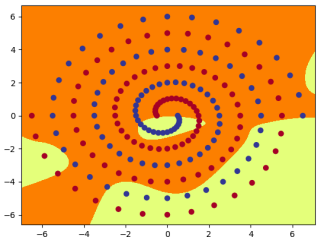

Figure 30. RawNet Layer 2 Node 5

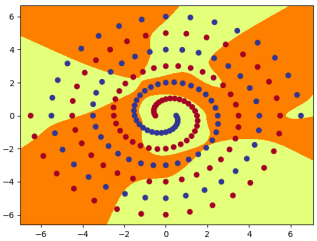

Figure 31. RawNet Layer 2 Node 6

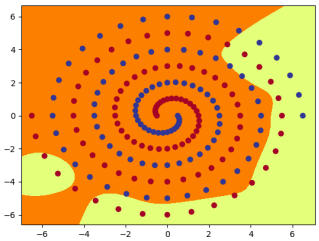

Figure 32. RawNet Layer 2 Node 7

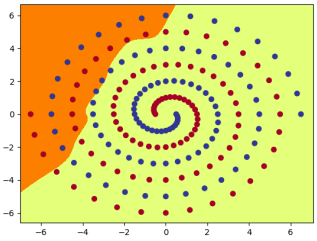

Figure 33. RawNet Layer 2 Node 8

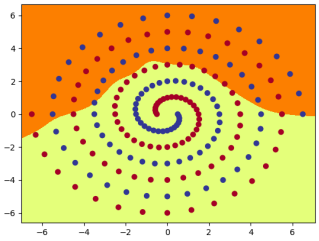

Figure 34. RawNet Layer 2 Node 9

ShortNet:

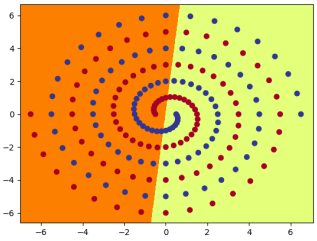

Figure 35. ShortNet Layer 1 Node 0

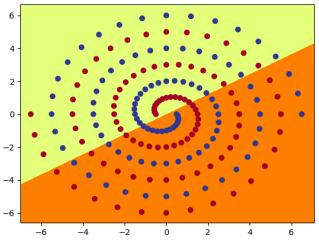

Figure 36. ShortNet Layer 1 Node 1

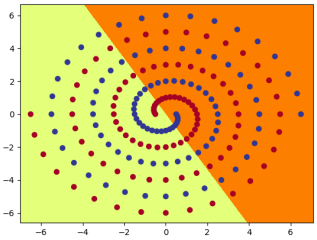

Figure 37. ShortNet Layer 1 Node 2

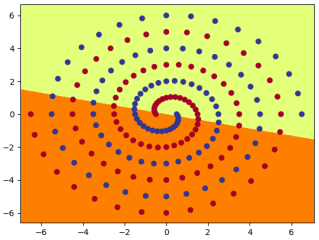

Figure 38. ShortNet Layer 1 Node 3

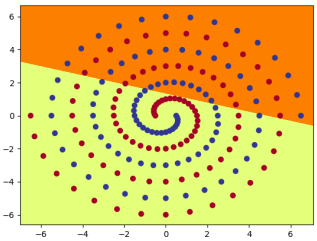

Figure 39. ShortNet Layer 1 Node 4

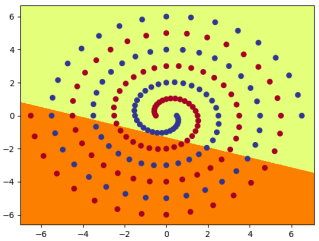

Figure 40. ShortNet Layer 1 Node 5

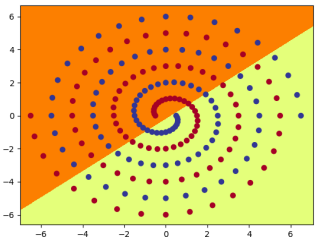

Figure 41. ShortNet Layer 1 Node 6

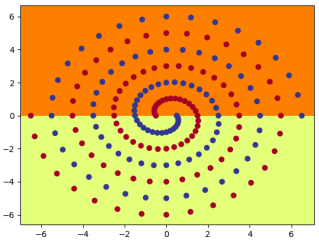

Figure 42. ShortNet Layer 1 Node 7

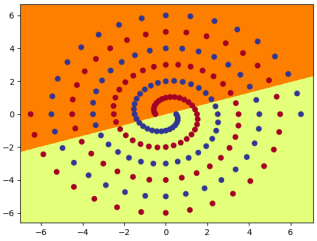

Figure 43. ShortNet Layer 1 Node 8

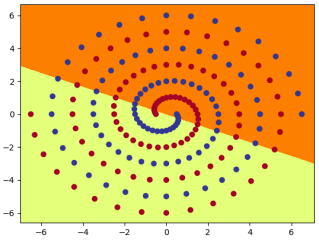

Figure 44. ShortNet Layer 1 Node 9

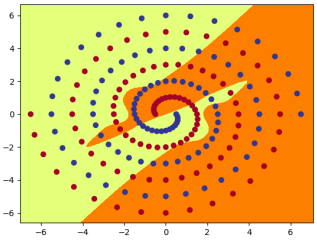

Figure 45. ShortNet Layer 2 Node 0

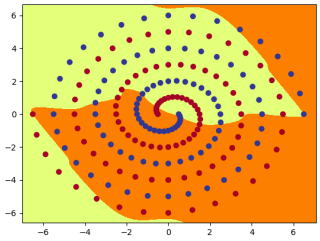

Figure 46. ShortNet Layer 2 Node 1

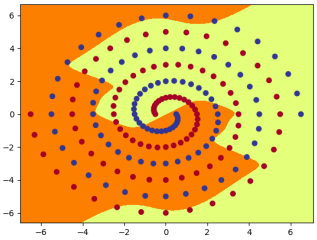

Figure 47. ShortNet Layer 2 Node 2

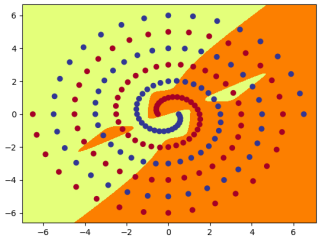

Figure 48. ShortNet Layer 2 Node 3

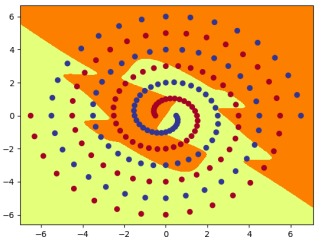

Figure 49. ShortNet Layer 2 Node 4

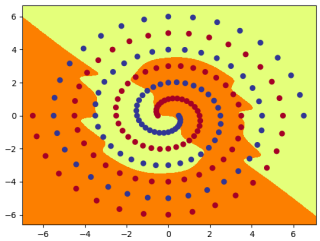

Figure 50. ShortNet Layer 2 Node 5

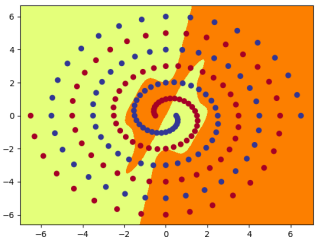

Figure 51. ShortNet Layer 2 Node 6

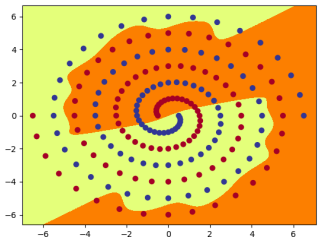

Figure 52. ShortNet Layer 2 Node 7

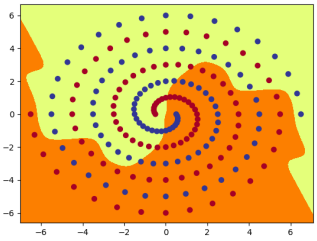

Figure 53. ShortNet Layer 2 Node 8

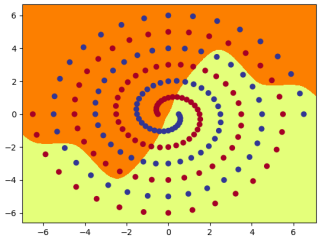

Figure 54. ShortNet Layer 2 Node 9

Question 8

a.

PolarNet:

In PolarNet, the 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 transformed to polar co-ordinates, the calculation becomes simpler, and a hidden layer, namely a combination of a linear function and a $tanh$ function is sufficient to fit the data. From the plots of the hidden layer, it could be seen that for each node, it is responsible for the recognition of a circular curve of points at a certain radius. And the linear combination of each node's result comes out a natural classification. The execution of PolarNet is comparatively fast, for a time of 20.24 seconds with the configuration of [initial weight = 0.1, number of hidden units = 10, learning rate = 0.01, max training epochs = 20000].

From the perspective of function, PolarNet firstly uses *torch.norm()* and *torch.atan2()* to transform (x, y) co-ordinates to (r, a) co-ordinates. After that, *torch.nn.Linear()* is applied to the (r, a) data trying to find out the linear relation and its output is passed to an activation function *torch.tanh()*. Then, the output of $tanh$ function is processed by another *torch.nn.Linear(),* whose result is passed to the final output function *torch.sigmoid()*.

RawNet:

For RawNet, it uses the original input and has two hidden layers and one output layer. An extra hidden layer is needed to realize more complicated mappings such as xor. From the plots of hidden layers, it could be seen that in the first hidden layer, nodes make linear classification, and data points are split by a line in each node. When it comes to the second hidden layer, irregular classification happens in nodes, on account that the model tries to fit the training data in whichever ways. However, for each segmentation area, it always has one or more real data points. The execution time of RawNet is slower than PolarNet because it has one more hidden layer. With the configuration of [initial weight = 0.121, number of hidden units = 10, learning rate = 0.01, max training epochs = 20000], it runs for 60.34 seconds.

From the perspective of function, RawNet uses the *first torch.nn.Linear()* and *torch.tanh()* functions to map the input (x, y) to *num_hid* nodes. Afterwards, another combination of *torch.nn.Linear()* and *torch.tanh()* is applied to map *num_hid* nodes to another *num_hid* nodes in the second hidden nodes. By doing this, the data classification is divided into higher granularity. Finally, the output from the second hidden layer is passed to the output layer, which is composed of a *torch.nn.Linear()* which takes *num_hid* nodes and maps them to one output, and a *torch.sigmoid()* which shows the probability of each prediction class.

ShortNet:

Different from RawNet, though ShortNet also has two hidden layers and one output layer structure, it includes short-cut connections between each pair of layers. This means that to solve the information loss in the degradation in RawNet, residual networks are introduced in ShortNet to optimize the degradation problem and to give a better performance with a benefit of not adding any extra parameters or computational complexity. Observations on the hidden layer plots reveal that in the first hidden layer, nodes have similar functionalities as the RawNet as they all segment data points by a line. Nevertheless, the second hidden layer performs different irregular classification. For the final output, all data points are correctly classified, but it should be noticed that on the top-right hand side corner, there is a segmentation area without any real data points. The reason could be information passed from frontal layers misleads the classification. The execution of ShortNet is the fastest, for a time of 14.42 seconds with the configuration of [initial weight = 0.1, number of hidden units = 10, learning rate = 0.01, max training epochs = 20000].

From the perspective of function, based on the functions in RawNet, there are three extra *torch.nn.Linear()* mapping input layer to the second hidden layer, input layer to output layer and the

first hidden layer to output layer respectively. Furthermore, the *torch.tanh()* at the second hidden layer accepts one more input from the input layer, and the *torch.sigmoid()* gets two more input from input layer and the first layer. Finally, the *torch.sigmoid()* returns the probability of each candidate class.

b.

In this part, only the "init" meta-parameter changes from the default value.

1. RawNet :

--init 0.1:
When the initial weights are 0.1, RawNet fails to fit the training data within 20000 epochs. Eventually, the training accuracy is 53.61%, and the execution time is 72.58 seconds.

--init 0.121:
When the initial weights are 0.121, RawNet starts to get 100% accuracy within 20000 epochs. The execution time is 60.34 seconds.

--init 0.13:
When the initial weights are 0.13, RawNet gets 100% accuracy at epoch 6300. The execution time is 21.07 seconds.

--init 0.14:
When the initial weights are 0.13, RawNet gets 100% accuracy at epoch 9600. The execution time is 34.78 seconds.

--init 0.15:
When the initial weights are 0.15, RawNet fails to fit the training data within 20000 epochs. Eventually, the training accuracy is 97.42%, and the execution time is 70.17 seconds.

In general, RawNet fails to fit the training data completely correct if the initial weights are too small or too big. It seems that the successful initial weight should be near 0.13. Based on the information we get, if initial weights are lower than 0.121, the activations tend to vanish in the process of propagation; and if initial weights are greater 0.14, the activations tend to saturate in the process of propagation.

As for the speed, it could be seen that, the initial weight of 0.13 outperforms the one of 0.14. This indicates that the speed does not increase along with the initial weight. Only if a proper initial weight is chosen, the speed is guaranteed.

2. ShortNet:

--init 0.001:
When the initial weights are 0.001, ShortNet fails to fit the training data within 20000 epochs. Eventually, the training accuracy is 50.52%, and the execution time is 94.04 seconds.

--init 0.01:
When the initial weights are 0.1, ShortNet gets 100% accuracy at epoch 4300. The execution time is 18.44 seconds.

--init 0.05:
When the initial weights are 0.05, ShortNet gets 100% accuracy at epoch 3600. The execution time is 15.75 seconds.

--init 0.1:
When the initial weights are 0.05, ShortNet gets 100% accuracy at epoch 3200. The execution time is 14.42 seconds.

--init 0.15:
When the initial weights are 0.15, ShortNet gets 100% accuracy at epoch 3100. The execution time is 12.91 seconds.

--init 0.2:
When the initial weights are 0.15, ShortNet gets 100% accuracy at epoch 5600. The execution time is 24.11 seconds.

In general, ShortNet has fewer restrictions on initial weight value than RawNet. When the initial weight value is extremely small as 0.001, ShortNet fails to converge. From the value 0.01 to 0.15, as the initial weight value increases, the training epoch number is less to realized 100% accuracy and the training speed increases. However, if the initial weight is too big, the training time gets longer, and it is more likely to fail in convergence.

From the outputs of RawNet and ShortNet, we can know that choosing a decent initial weight value is significant in deep learning. A small or big initial weight value could result in longer training time and a higher probability of convergence failure.

c.

Speaking of relative "naturalness", PolarNet is the best, with RawNet ranking the second and ShortNet playing the worst role.

From Fig 2, we can see that the segmentation of PolarNet corresponds better to human recognition of spiral distribution. And if we insert some red points between the trajectory of existing red dots, they are still classified as red points. The result is also correct for blue point insertion.

However, for the results of RawNet and ShortNet, if we make the insertion, from Fig 55 and Fig 56, it could be seen that these two networks both make the wrong predictions.

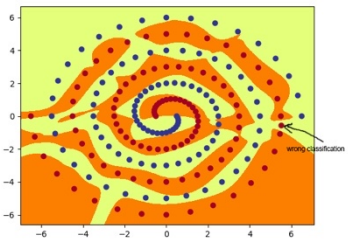

Figure 55. Red Point Insertion to RawNet Model

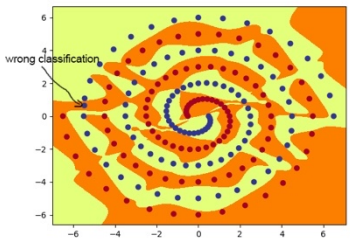

Figure 56. Blue Point Insertion to ShortNet Model

Therefore, RawNet and ShortNet have worse relative "naturalness".

Focusing back to the top-right hand side corner of the ShortNet output, the orange segmentation part is less "natural" as it does not cover any real data points. Hence, RawNet is more natural than ShortNet.

Eventually, from the perspective of representation for deep learning, we can see that it is crucial to understand the essence of raw data. In this question, the data given is scattering on curves and in this

scenario, polar co-ordinates can have a better representation of curve distributions. Hence, after transforming data to polar co-ordinates, we can train a rather simple model with ideal classification. Designing a deep learning model with a good knowledge of the raw data's representation can decrease computation complexity and generate natural segmentation.

d.

1. Changing Batch Size:

Batch Size: 48

--net polar --init 0.1 --hid 10 --lr 0.1 --epochs 20000
PolarNet reaches 100% accuracy at epoch 3200, with execution time of 15.50 seconds.

--net polar --init 0.1 --hid 10 --lr 0.121 --epochs 20000
RawNet fails to reach 100% accuracy. Eventually it has 97.42% accuracy with execution time of 125.92 seconds.

--net short --init 0.1 --hid 10 --lr 0.1 --epochs 20000
ShortNet reaches 100% accuracy at epoch 4500, with execution time of 38.97 seconds.

Batch Size: 194

--net polar --init 0.1 --hid 10 --lr 0.1 --epochs 20000
PolarNet fails to reach 100% accuracy within 20000 epochs. Eventually it has 78.87% accuracy with execution time of 50.85 seconds.

--net polar --init 0.1 --hid 10 --lr 0.121 --epochs 20000
RawNet fails to reach 100% accuracy. Eventually it has 50.52% accuracy with execution of 55.56 seconds.

--net short --init 0.1 --hid 10 --lr 0.1 --epochs 20000
ShortNet reaches 100% accuracy at epoch 1300, with execution of 4.00 seconds.

From the results above, it could be seen that when the batch size is 48, the performance is worse than the one of batch size of 97. When the batch size doubles to 194, the original meta-parameter setting is no longer practical for the model's convergence. Therefore, choosing a proper batch size can not only boost up the convergence speed but also guarantee the success of model construction. The batch size cannot be too small or too big.

2. Using SGD Optimizer:

We replace the Adam optimizer with the following code:

```
optimizer = torch.optim.SGD(net.parameters(),lr=args.lr,momentum=0.9)
```

--net polar --init 0.1 --hid 10 --lr 0.06 --epochs 20000
PolarNet reaches 100% accuracy at epoch 2700, with execution time of 6.93 seconds.

--net raw --init 0.121 --hid 10 --lr 0.06 --epochs 20000
RawNet fails to reach 100% accuracy. Eventually, it has a 70.62% accuracy with execution time of 59.60 seconds.

--net short --init 0.1 --hid 10 --lr 0.06 --epochs 20000
ShortNet reaches 100% accuracy at epoch 1200, with execution time of 4.86 seconds.

When we switch from Adam algoritm to SGD, through the experiments, we find that we need a comparatively big momentum and high learning rate to fit the spiral data. However, after tuning the meta-parameters for the RawNet training for several times, the RawNet still failed to reach 100% accuracy. The reason could be some information is lost when it is passed through hidden layers, since compared to ShortNet, RawNet lacks the direct connections between front layers to the back layers.

3. Using Relu as Activation Function:

We replace "*torch.tanh*" with "*torch.relu*" in the code.

--net polar --init 0.1 --hid 10 --lr 0.1 --epochs 20000
PolarNet fails to reach 100% accuracy within 20000 epochs. Eventually, it has 56.70% accuracy with execution time of 66.87 seconds.

--net polar --init 0.1 --hid 10 --lr 0.121 --epochs 20000
RawNet fails to reach 100% accuracy within 20000 epochs. Eventually, it has 78.87% accuracy with execution of 65.03 seconds.

--net short --init 0.1 --hid 10 --lr 0.1 --epochs 20000
ShortNet fails to reach 100% accuracy within 20000 epochs. Eventually, it has 91.24% accuracy with execution of 89.93 seconds.

When we use $relu$ function instead of $tanh$, the three networks fail to get 100% accuracy. The reason could be $relu$ function cuts off the negative part of the linear function output in each hidden layer and this causes great information loss so that the models fail to converge.

4. Adding A Hidden Layer in RawNet and ShortNet

--net polar --init 0.1 --hid 10 --lr 0.121 --epochs 20000
RawNet realizes 100% accuracy at epoch 6300, with execution time of 23.54 seconds.

--net short --init 0.1 --hid 10 --lr 0.1 --epochs 20000
ShortNet realizes 100% accuracy at epoch 1000, with execution time of 5.59 seconds.

Compared to the two hidden layer structure, when we introduce an extra hidden layer to the RawNet and ShortNet, as the models segment the input data into higher granularities, classification becomes easier so that the training time and the number of epochs decrease.

```
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        # INSERT CODE HERE
        self.l1 = nn.Linear(2,num_hid)
        self.l2 = nn.Linear(num_hid,num_hid)
        self.l3 = nn.Linear(num_hid,num_hid)
        self.l4 = nn.Linear(num_hid,1)

        self.hid1 = None
        self.hid2 = None
        self.hid3 = None

    def forward(self, input):
        inpToHid1 = self.l1(input)
        self.hid1 = torch.tanh(inpToHid1)
        hid1ToHid2 = self.l2(self.hid1)
        self.hid2 = torch.tanh(hid1ToHid2)
        hid2ToHid3 = self.l3(self.hid2)
        self.hid3 = torch.tanh(hid2ToHid3)
        hid3ToOut = self.l4(self.hid3)
        output = torch.sigmoid(hid3ToOut)
        return output
```

Figure 57. 3-hidden-layer RawNet Code

```
class ShortNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(ShortNet, self).__init__()
        # INSERT CODE HERE
        self.lin_inp_hid1 = nn.Linear(2,num_hid) # lin stands for linear
        self.lin_inp_hid2 = nn.Linear(2,num_hid)
        self.lin_inp_hid3 = nn.Linear(2,num_hid)
        self.lin_inp_out  = nn.Linear(2,1)
        self.lin_hid1_hid2 = nn.Linear(num_hid,num_hid)
        self.lin_hid1_hid3 = nn.Linear(num_hid,num_hid)
        self.lin_hid1_out = nn.Linear(num_hid,1)
        self.lin_hid2_hid3 = nn.Linear(num_hid,num_hid)
        self.lin_hid2_out = nn.Linear(num_hid,1)
        self.lin_hid3_out = nn.Linear(num_hid,1)

        self.hid1 = None
        self.hid2 = None
        self.hid3 = None

    def forward(self, input):
        x_inpToHid1 = self.lin_inp_hid1(input)
        x_inpToHid2 = self.lin_inp_hid2(input)
        x_inpToHid3 = self.lin_inp_hid3(input)
        x_inpToOut = self.lin_inp_out(input)
        self.hid1 = torch.tanh(x_inpToHid1)
        x_inputOfHid2 = x_inpToHid2 + self.lin_hid1_hid2(self.hid1)
        self.hid2 = torch.tanh(x_inputOfHid2)

        x_inputOfHid3 = x_inpToHid3 + self.lin_hid1_hid3(self.hid1) + self.lin_hid2_hid3(self.hid2)
        self.hid3 = torch.tanh(x_inputOfHid3)

        x_inputOfOut = x_inpToOut + self.lin_hid1_out(self.hid1) + self.lin_hid2_out(self.hid2) + self.lin_hid3_out(self.hid3)
        output = torch.sigmoid(x_inputOfOut)

        # x_inputOfOut = x_inpToOut + self.lin_hid1_out(self.hid1) + self.lin_hid2_out(self.hid2)
        # output = torch.sigmoid(x_inputOfOut)
        return output
```

Figure 58. 3-hidden-layer ShortNet Code