Assignment 1

Part 1 Question 1 (NetLin)

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.808106
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.629926
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.587550
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.613506
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.322455
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.519625
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.664882
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.608692
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.346079
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.676776
<class 'numpy.ndarray'>
[[767.   5.   9.  13.  30.  66.   2.  61.  30.  17.]
 [  7. 668. 110.  19.  29.  23.  55.  14.  26.  49.]
 [  8.  63. 692.  25.  25.  20.  47.  36.  46.  38.]
 [  6.  36.  64. 753.  16.  56.  13.  18.  27.  11.]
 [ 60.  54.  77.  20. 626.  20.  32.  36.  19.  56.]
 [  8.  27. 125.  17.  20. 723.  28.   9.  33.  10.]
 [  5.  21. 148.  10.  25.  25. 721.  20.  11.  14.]
 [ 18.  29.  27.  10.  83.  18.  53. 623.  90.  49.]
 [ 10.  40.  96.  45.   7.  29.  44.   6. 702.  21.]
 [ 10.  52.  87.   3.  51.  29.  20.  31.  38. 679.]]

Test set: Average loss: 1.0092, Accuracy: 6954/10000 (70%)
```

Question 2 (NetFull)

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.325122
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.284411
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.222842
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.237630
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.123999
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.267251
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.210003
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.369761
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.142478
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.264650
<class 'numpy.ndarray'>
[[852.   5.   3.   7.  29.  26.   5.  41.  27.   5.]
 [  5. 812.  25.   5.  24.  16.  62.   2.  24.  25.]
 [  9.  20. 827.  39.  10.  20.  24.  13.  25.  13.]
 [  2.   8.  29. 917.   0.  14.   7.   7.   8.   8.]
 [ 40.  30.  15.   5. 818.   7.  28.  20.  21.  16.]
 [ 10.  12.  57.   8.  11. 856.  23.   4.  13.   6.]
 [  3.  10.  51.   7.  12.   5. 885.  13.   3.  11.]
 [ 16.  16.  16.   7.  19.  11.  34. 829.  27.  25.]
 [  9.  25.  26.  53.   5.   8.  29.   4. 832.   9.]
 [  5.  21.  42.   4.  30.   8.  24.  12.  15. 839.]]

Test set: Average loss: 0.5091, Accuracy: 8467/10000 (85%)
```

Question 3 (NetConv)

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.037049
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.021285
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.058191
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.056172
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.037495
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.037257
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.018425
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.122596
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.009248
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.036607
<class 'numpy.ndarray'>
[[955.   3.   1.   1.  24.   1.   1.   7.   5.   2.]
 [  2. 914.   9.   0.  10.   0.  41.   6.   4.  14.]
 [ 10.   5. 893.  25.  10.   4.  21.  12.   7.  13.]
 [  3.   0.  18. 954.   2.   3.   9.   5.   1.   5.]
 [ 17.   5.   2.   4. 942.   1.  12.   3.   9.   5.]
 [  5.  15.  44.   5.   6. 881.  23.  12.   4.   5.]
 [  4.   6.  19.   2.   3.   1. 963.   0.   0.   2.]
 [  7.   6.   5.   1.   4.   0.   7. 955.   4.  11.]
 [  4.  15.   9.   4.  12.   2.  10.   3. 938.   3.]
 [  6.   4.   9.   1.   5.   0.   4.   4.   4. 963.]]

Test set: Average loss: 0.2446, Accuracy: 9358/10000 (94%)
```

Question 4

a) Overall, NetLin performed the worst achieving an accuracy of around 70%, with the next being NetFull at 84% accuracy and finally NetConv at 94%.

NetLin utilised a linear function to classify the images. However, this type of network is only accurate when the data is able to be linearly separated into its classes. As we are trying to classify japanese characters written in many different styles, it becomes unlikely that all the data lies perfectly on a linear function.

The most noticeable error in this network was the classification of characters as “su”, especially the “ha” and “ma” characters. The “su” character shares a very similar structure with the “ha” and the “ma” characters; it consists of a horizontal stroke with a vertical stroke that loops around in the middle. This structure is extremely similar to the right part of the “ha” character. Additionally, the “ma” character is almost identical in that it has the same structure but with an additional horizontal stroke.

Looking at the example images, many of the “ha” characters are actually written in the katakana form ハ instead of は. This may have influenced the network’s decision in classifying any hiragana versions of “ha” as “su” as it is quite different from the katakana form.

NetFull consisted of one hidden layer, which used tanh as an activation function and finally a log softmax on the output layer to achieve a one hot tensor. Having a hidden layer with a non-linear activation function allowed Netfull to use backpropogation to adjust weights more meaningfully. In addition, NetFull is able to create a non-linear model to classify the images, and isn’t restricted to a linear approach. This gave NetFull much more learning capacity compared to NetLin, hence it achieved in general a higher accuracy.

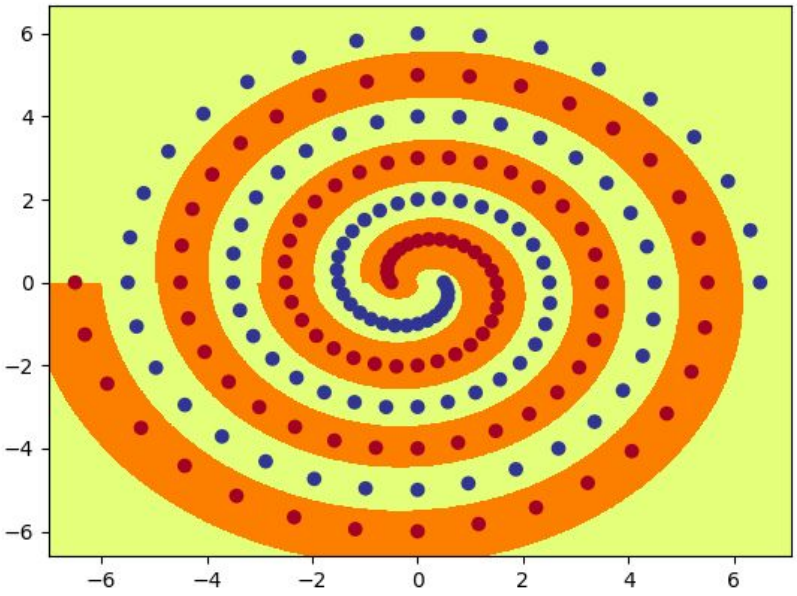
In the case of NetFull, the most common error was the classification of characters as “su” again, contributing to almost 22% of all errors. Another noticeable error was the classification of “ma” as “ki”, most likely due to the two characters sharing a very similar structure which consists of two horizontal strokes followed by a vertical stroke through the middle.

Instead of using tanh as an activation function, I tried using relu instead. In general, it achieved a higher accuracy of 85/86% as opposed to 84% by the 10th epoch. This is because relu allows the network to converge faster. Using a sigmoid as the activation function led to the network learning much more slowly. This is because with extremely large values, the changes made to adjust them are still small as sigmoid will convert the output into a value between 0 and 1 regardless.

NetConv was the most accurate network, achieving an accuracy of 94% by the 10th epoch. This has taught me that convulutional networks tend to be much better at image classification problems, most likely because they are able to identify noticeable patterns through the use of filters. In this task, the network most likely developed filters to identify certain curves, or edges unique to a character.

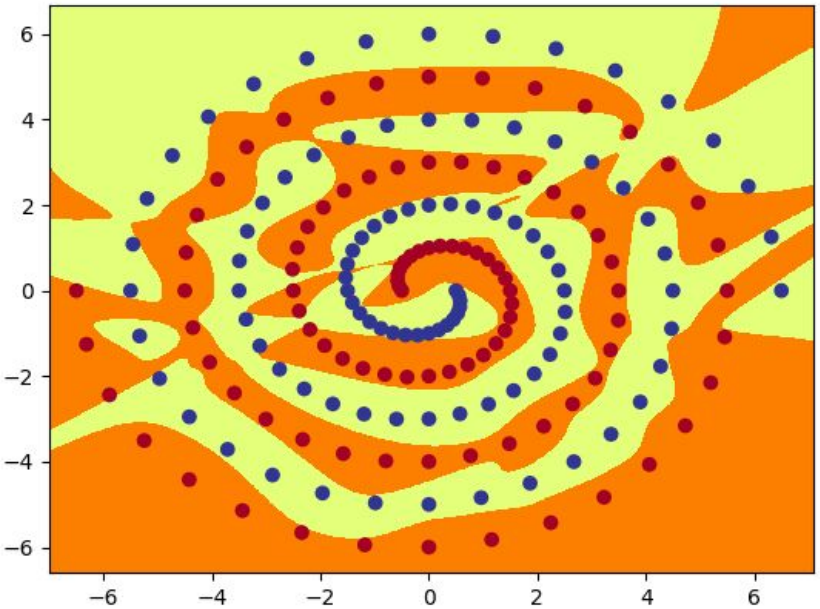
For NetConv, I tested a max pooling architecture and a non max pooling architecture. Surprisingly, their rate of improvement and accuracy were quite similar, with both architectures reaching an accuracy of 94% by the 10th epoch. I was expecting the max pooling architecture to yield higher accuracy as it would work against overfitting. This meant that the model yielded from the non max pooling architecture is quite generalised as well. Increasing the size of the filter led to diminishing returns and eventually began to make accuracy worse. This is most likely because larger filters compressed the image too much making it harder for the network to identify any patterns. Increasing the learning rate allowed the network to learn at a much faster rate up to a certain point. For example, with a learning rate of 0.05, the network reached an accuracy of 94% by the 5th epoch and 96% by the 10th epoch. With a learning rate of 0.2, the network became unstable and at higher values it would fluctuate out of control.

Part 2 Question 2



The lowest number of hidden nodes which consistently reached 100% accuracy in under 20000 epochs was 6.

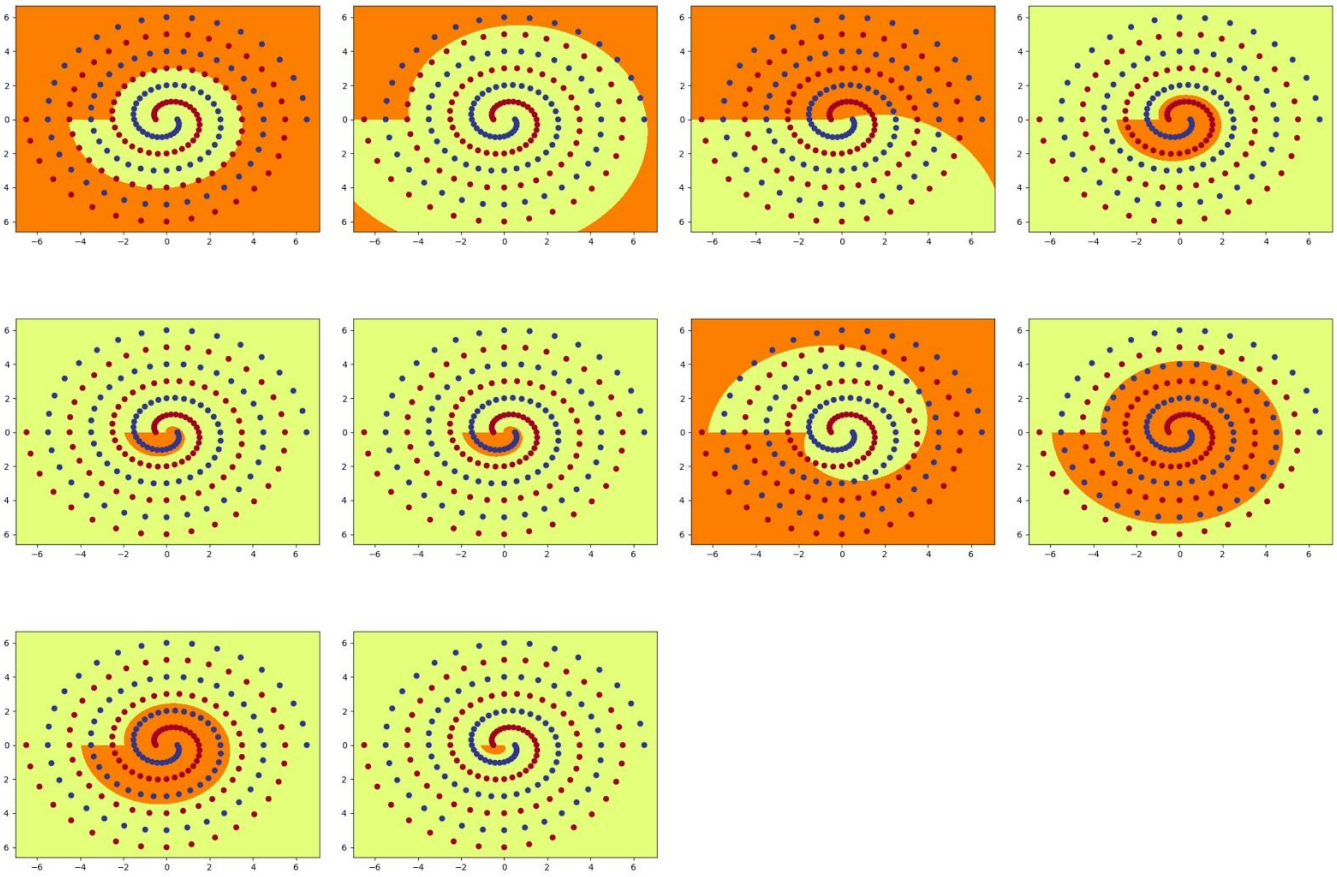
Question 4



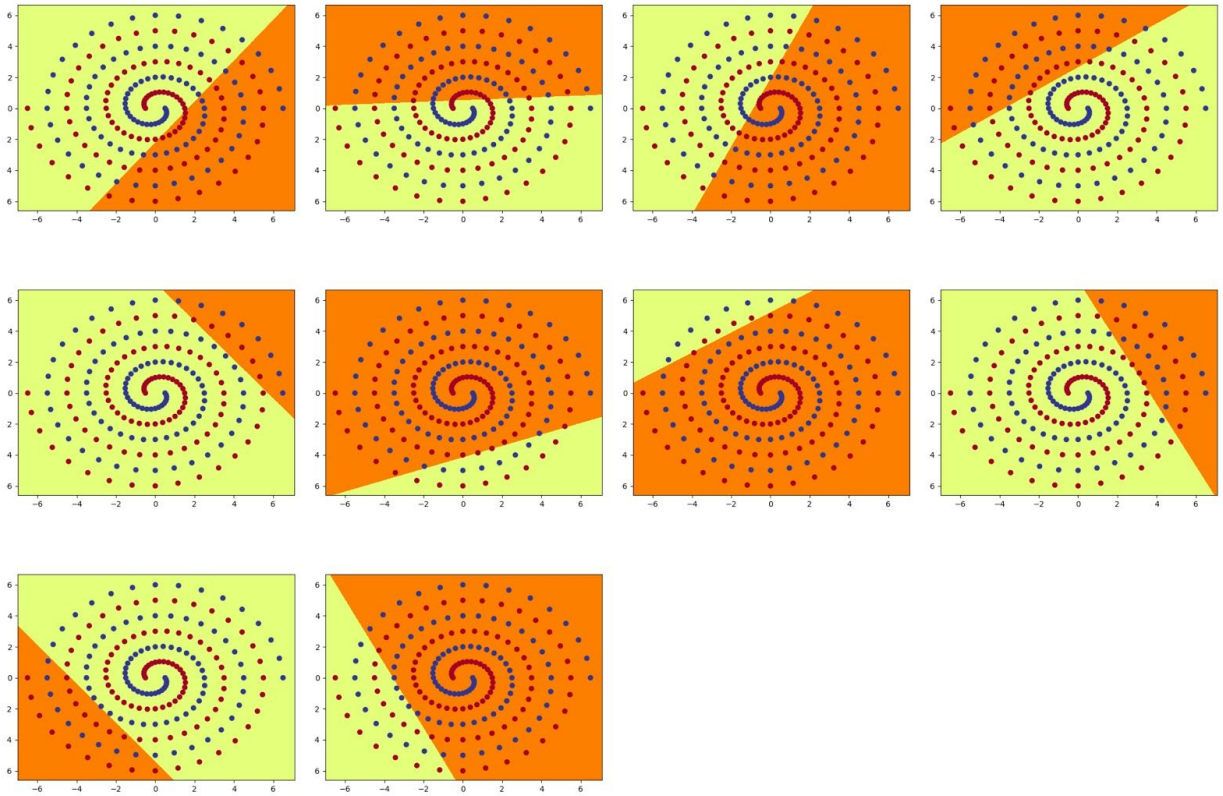
Using 10 hidden nodes and an initial weight of 0.2, the network was able to classify all inputs correctly under 20000 epochs on almost all runs.

Question 5

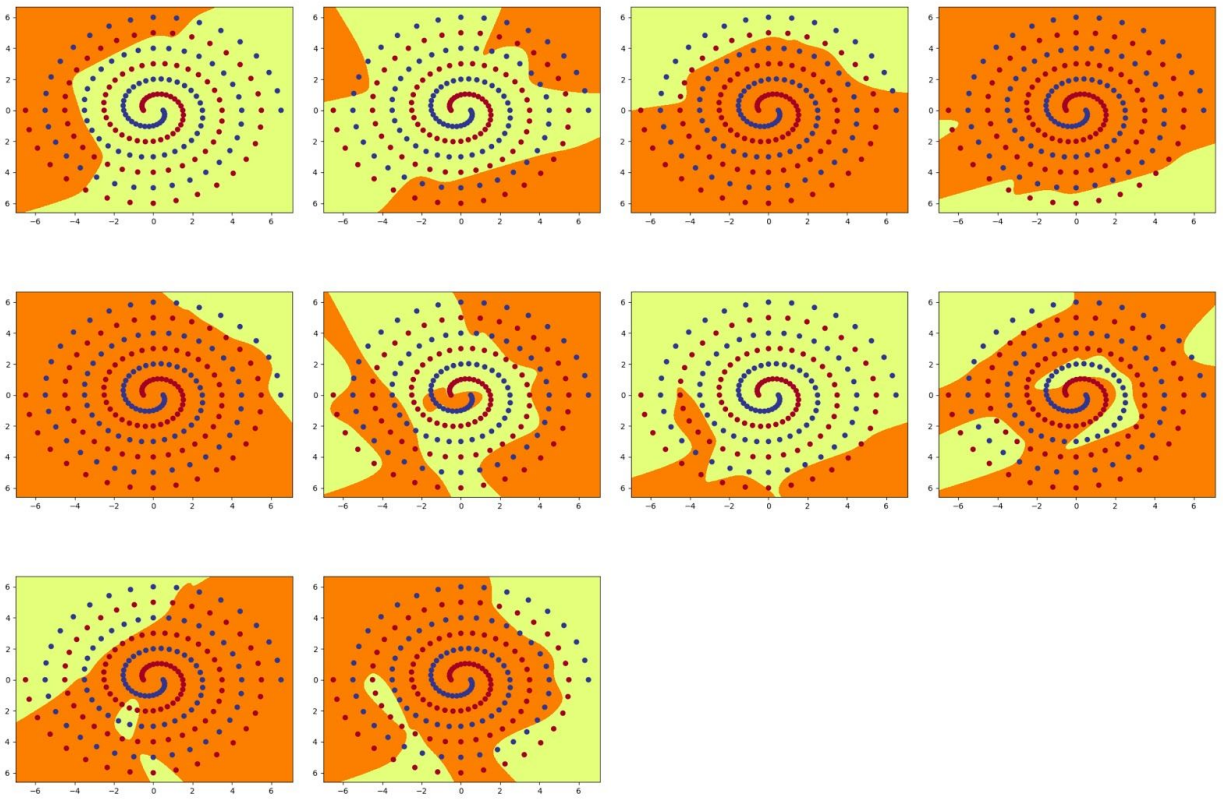
PolarNet (Full Sized Images attached in Appendix)



RawNet Layer 1 (Full Sized Images attached in Appendix)



RawNet Layer 2 (Full Sized Images attached in Appendix)



Question 6

All of PolarNet's hidden nodes learnt a non-linear function which curved around following the shape of the spiral. The shape of many of the functions learnt by the hidden nodes are actually quite similar, but with either inverted outputs, or instead covered a smaller portion of the inputs.

PolarNet uses these functions to identify the relative location of the input. By passing the input through each of these hidden unit functions, PolarNet will be able to distinguish certain features such as whether it is:

- In the top half of the spiral (Hidden Node 3)
- How central the input is (Hidden Nodes 4, 5, 6, 8, 9, 10)
- Whether it is on the outer edges of the spiral (Hidden Nodes 1, 2, 7)

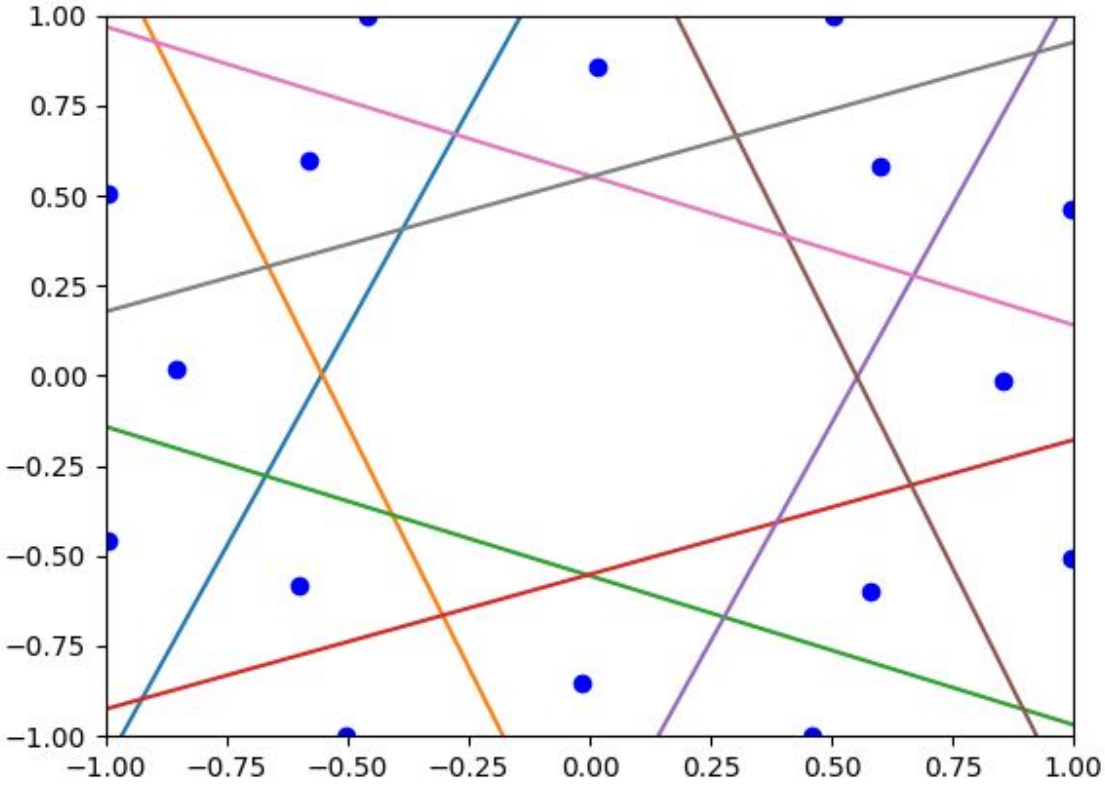
The first layer of hidden nodes for RawNet learnt a linear function, as opposed to the second layer which appears to be less straightforward. Not all hidden nodes seem to have learnt the same function, with some learning a convex or concave function, and others learning a non-linear function without a specific shape.

Since there are two hidden layers in RawNet, the first layer isn't used to classify the input to a specific output. Instead, it has learnt to linearly separate the input into a section of the spiral. This knowledge is passed onto the second layer, which then uses the non-linear functions it has learnt to classify the input.

In relation to initial weight sizes, the network had the most success with an initial weight size of 0.2. Increasing this initial weight size any larger led to decreasing success rates, and often led the network to becoming stuck in a local minimum. The learning speed also slowed down significantly the larger the initial weight size. With an initial weight size between 0.1 and 0.2, the network learned very slowly in the first few thousand epochs, but would then speed up. Sometimes the network succeeded before 10000 epochs, but other times it would become stuck in a local minimum with an accuracy of 99%.

The following changes were conducted keeping initial weight at a size of 0.2. I tried using relu as an activation function instead of tanh, and this led to the network learning at a much slower rate. Adding a third hidden layer however, significantly increased the learning speed, and the network would reach 100% accuracy by an average of 3000 epochs. Increasing the batch size from 97 to 194 caused the network to become unstable. The network would either reach an accuracy of 100% extremely fast (by 2000 epochs), or learn extremely slowly (by 35000 epochs).

Part 3 Question 1



Question 2

Image 1

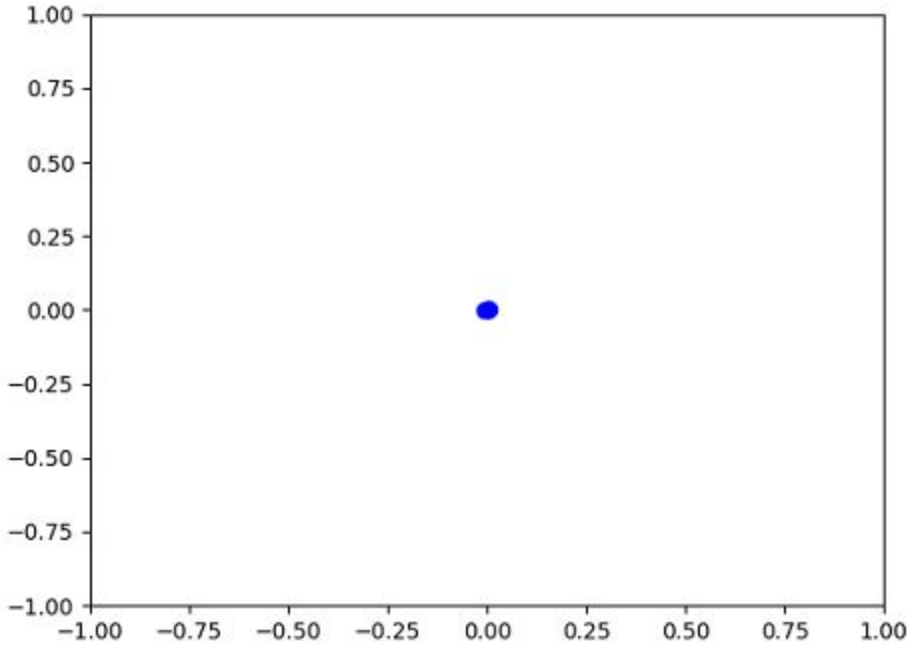


Image 2

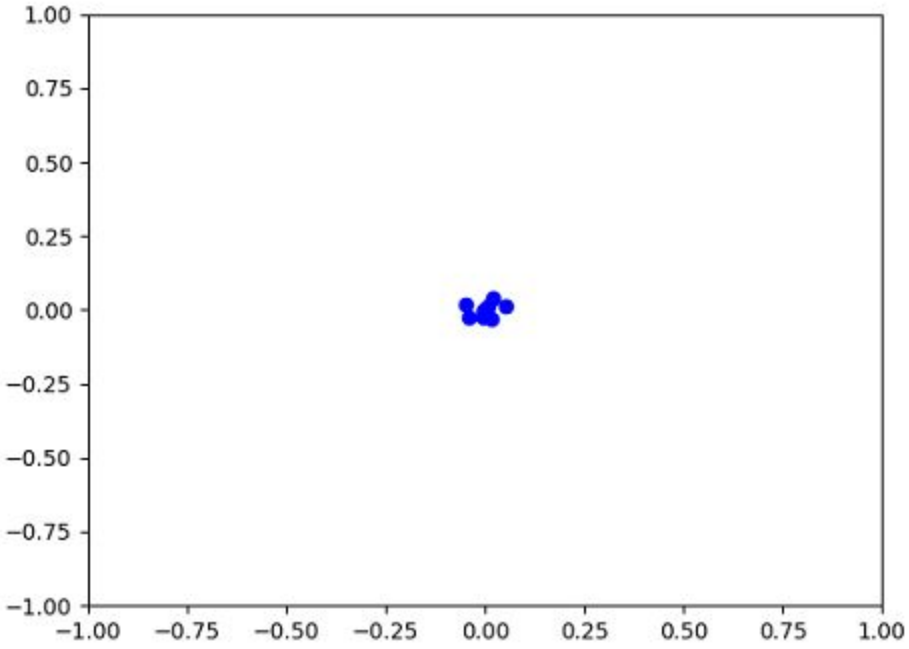


Image 3

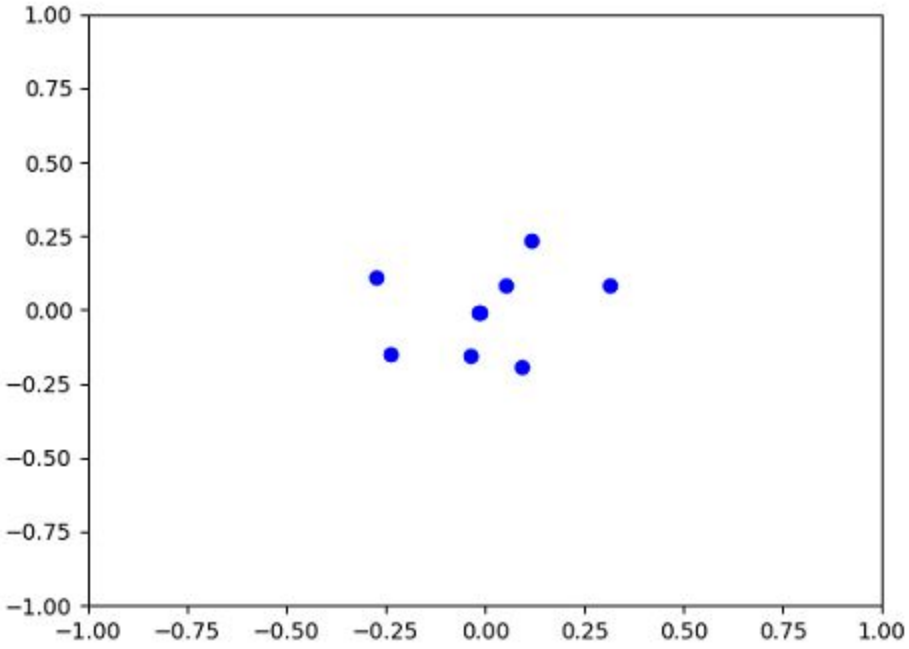


Image 4

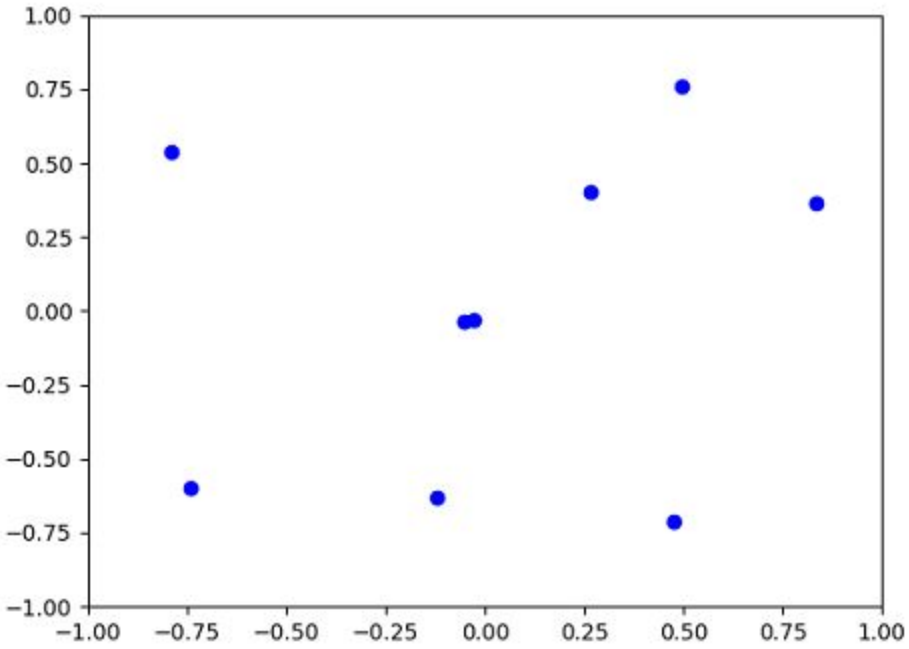


Image 5

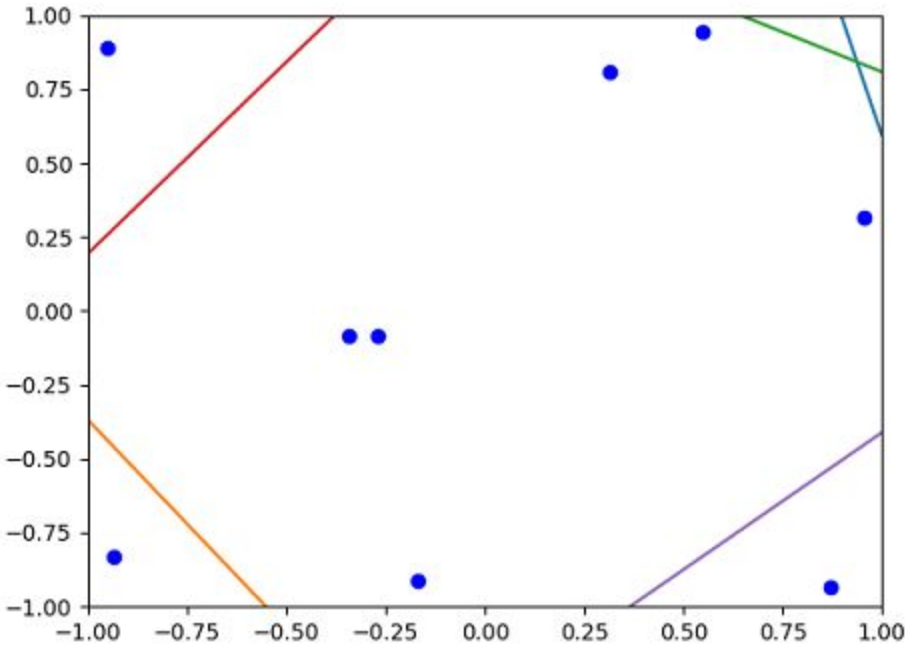


Image 6

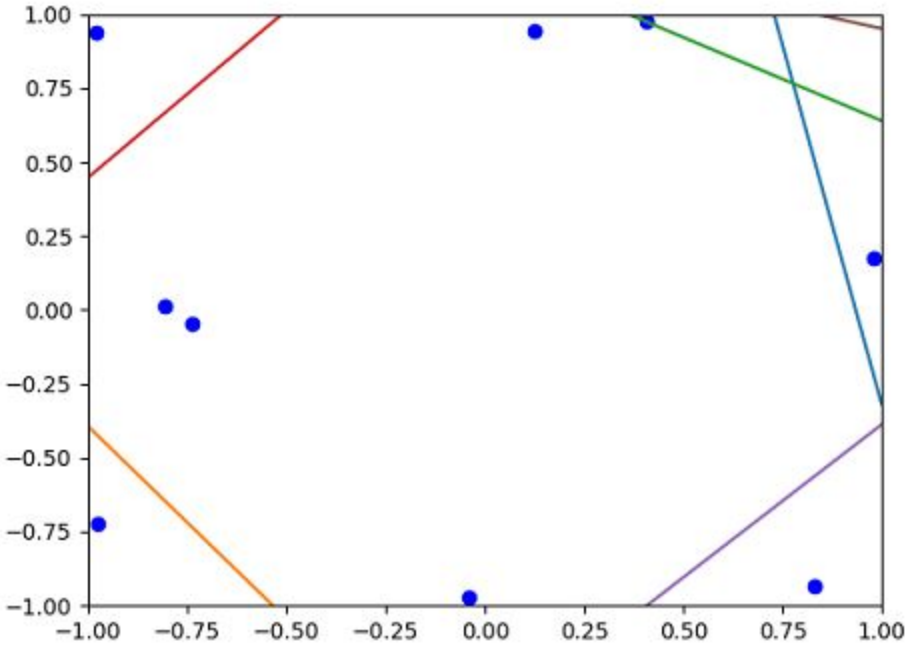


Image 7

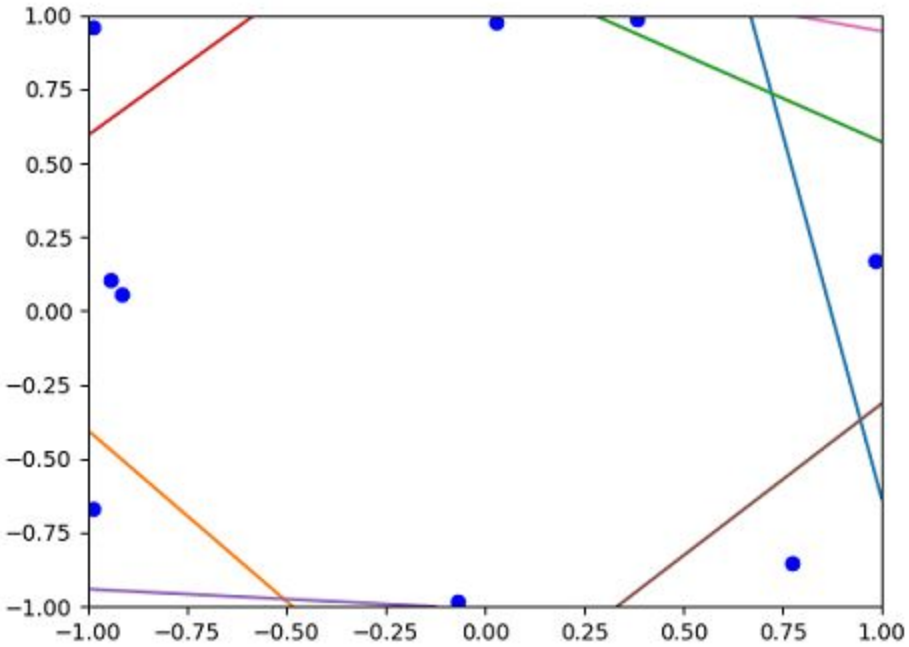


Image 8

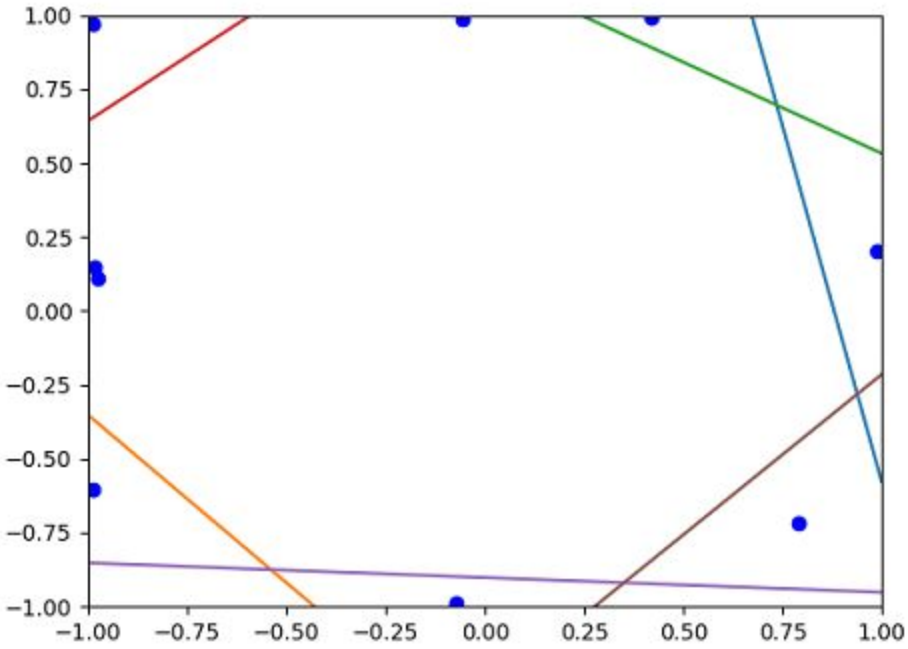


Image 9

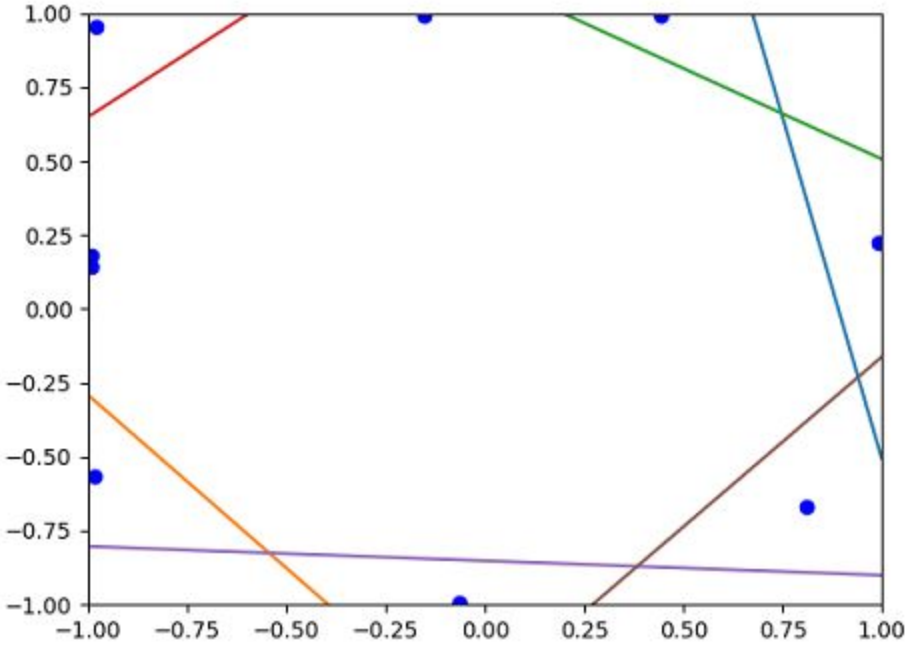


Image 10

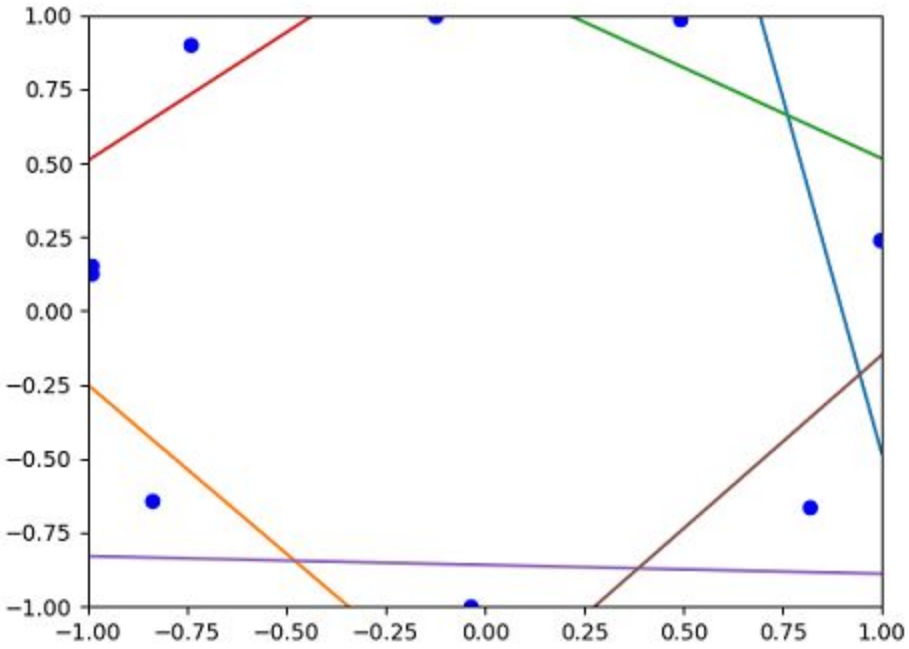
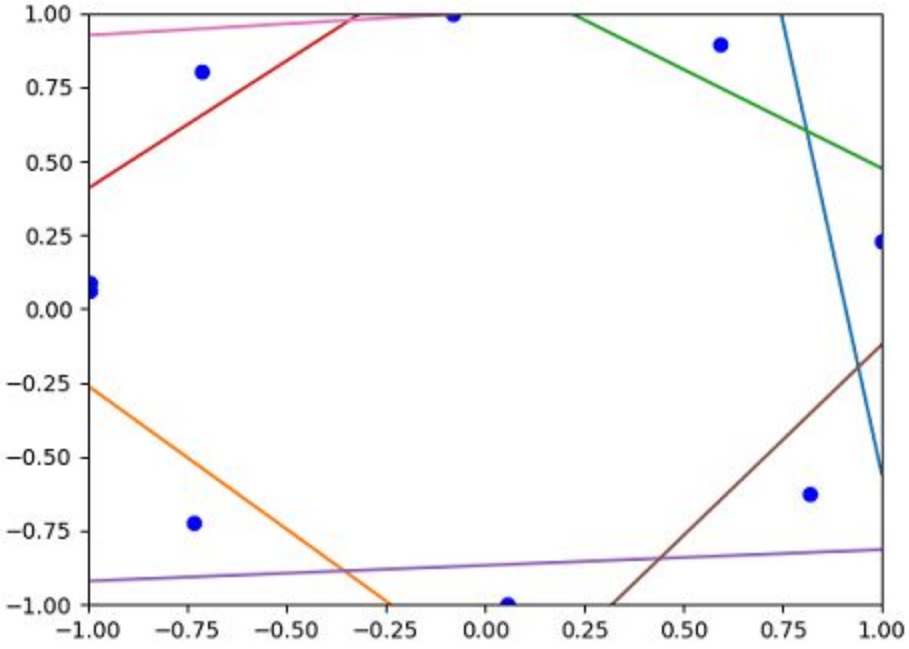
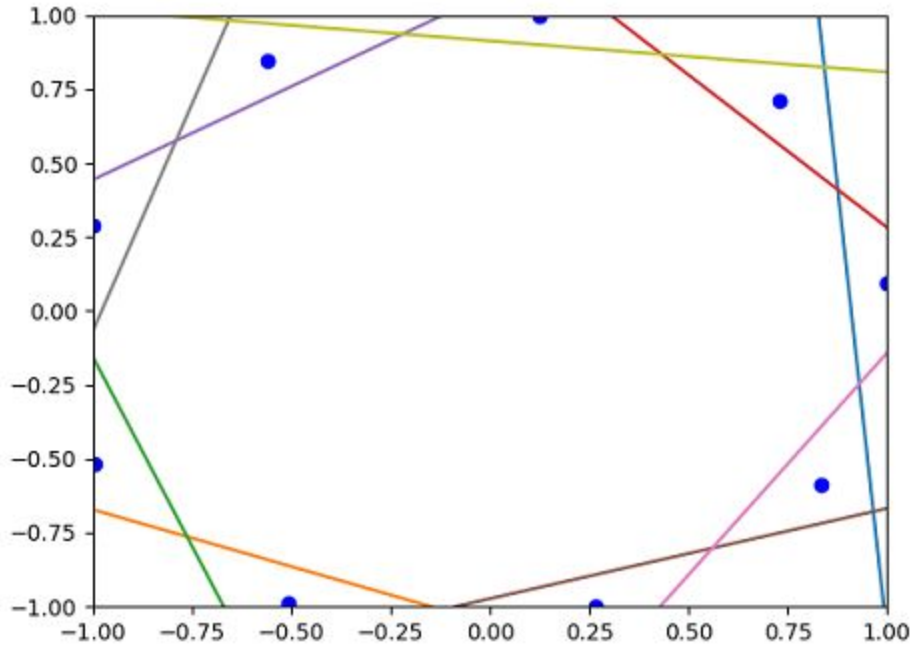


Image 11



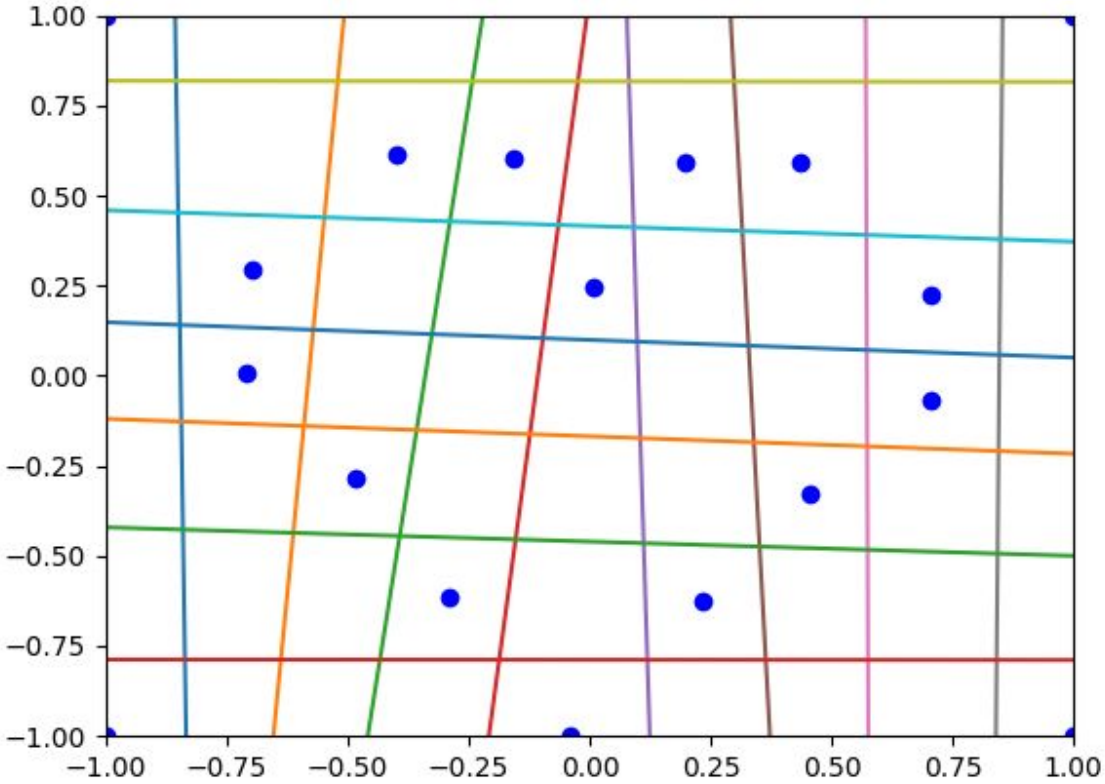
Final Image



Analysis

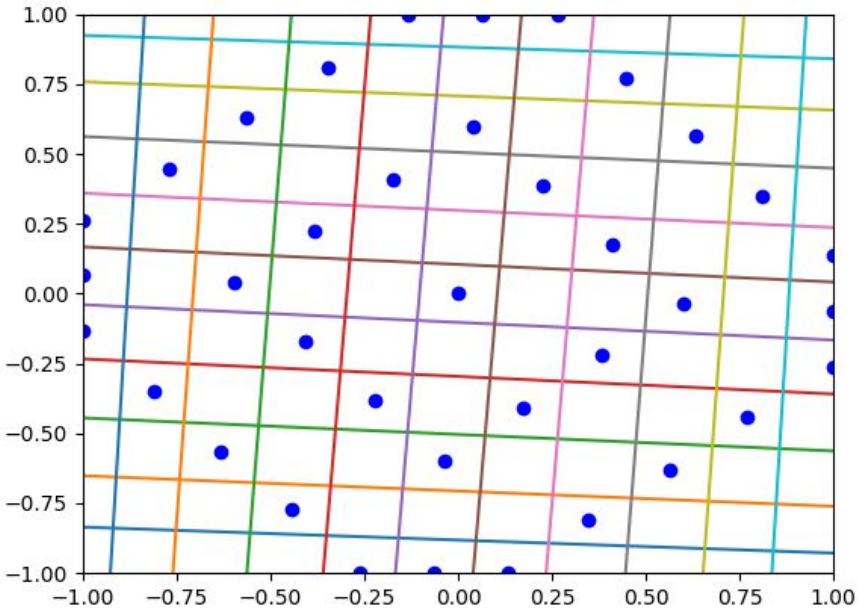
In the beginning, all the hidden unit activations are centred around the middle, with a slight offset for each individual activation. This occurs because the weights are initialised to be around 0.001, and since the hidden layer uses tanh as an activation function, the resulting values are around 0. As the network trains, the dots begin to move towards the outer edges of the hidden unit space at an increasing speed, however there are two dots which stay together and move more slowly. In the end, the dots form a 9 point star, and each boundary line separates a single dot from all the others dots. This makes sense as the input and target output are determined by a one-hot encoding, hence each boundary line must separate a single dot from every other dot.

Question 3

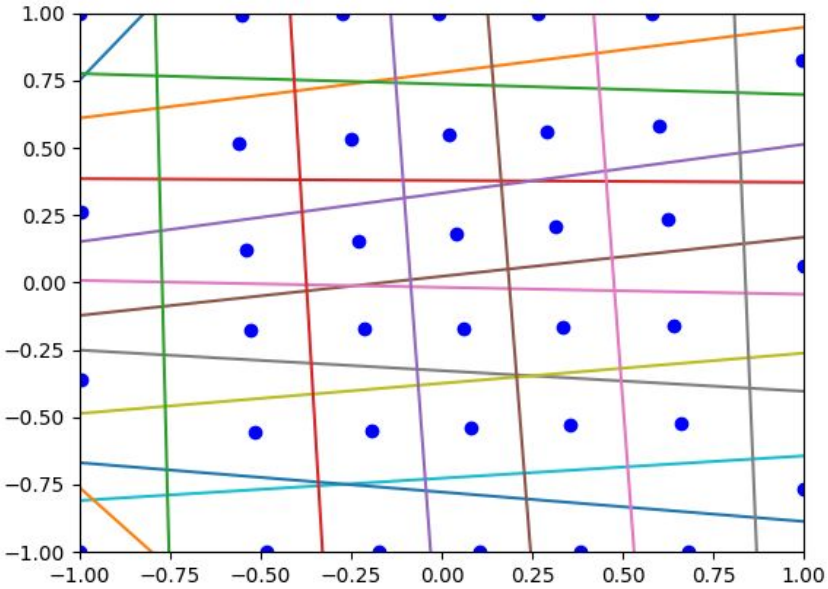


Question 4

Target 1

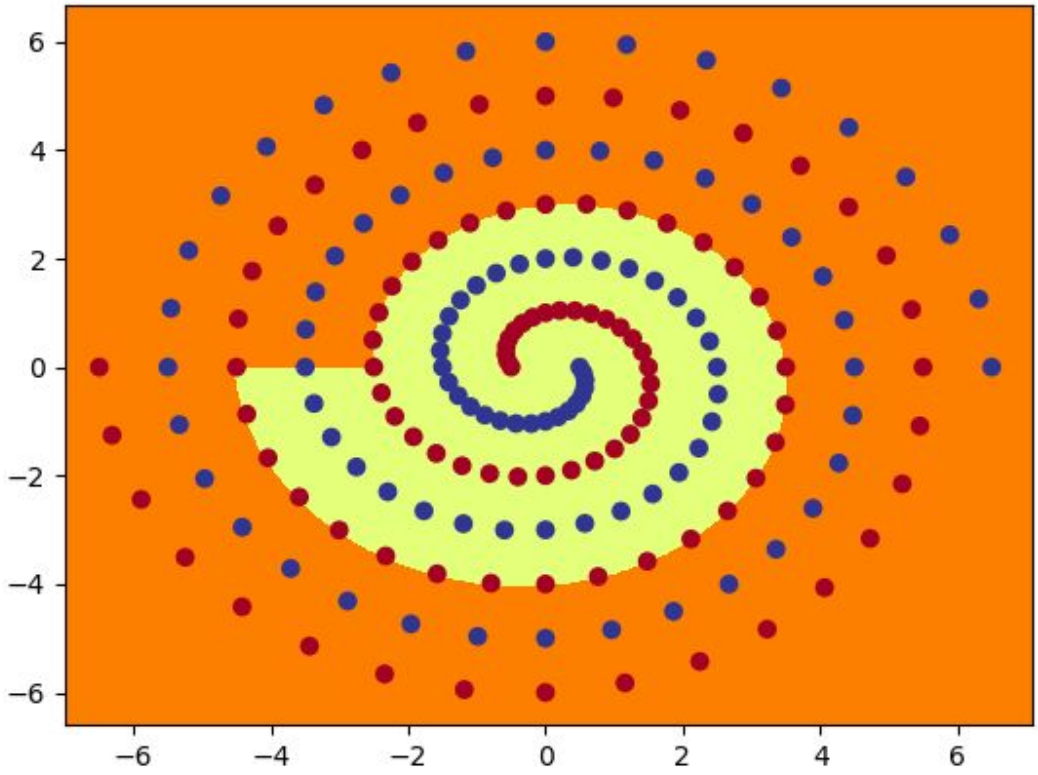


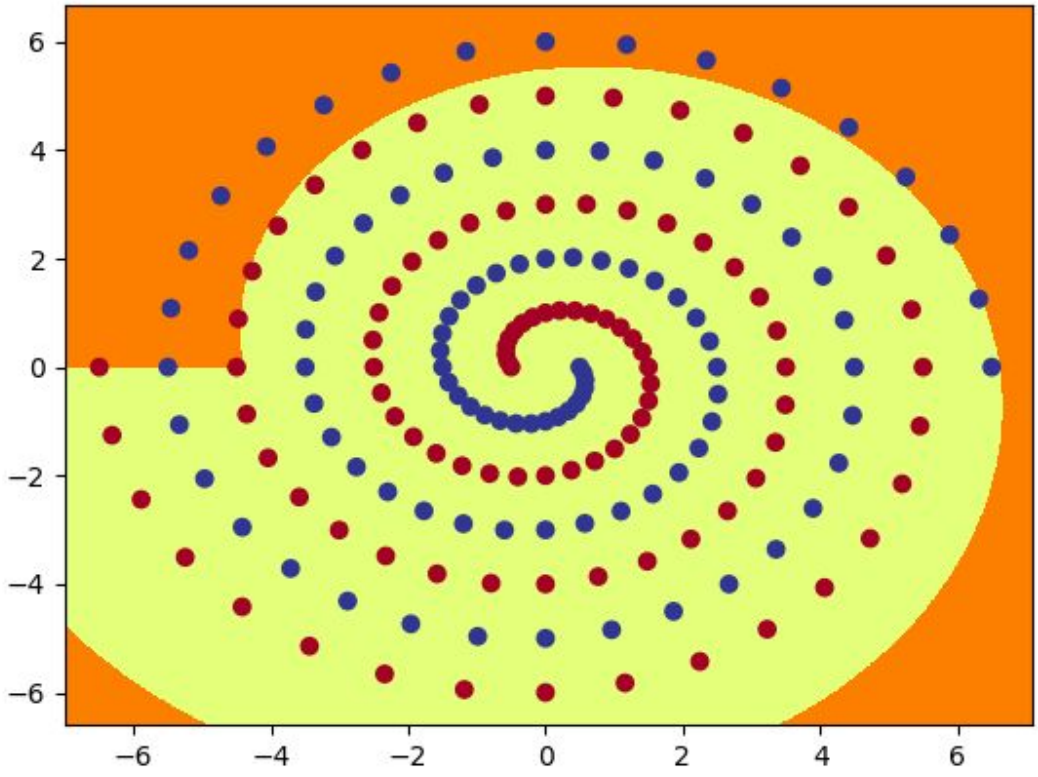
Target 2

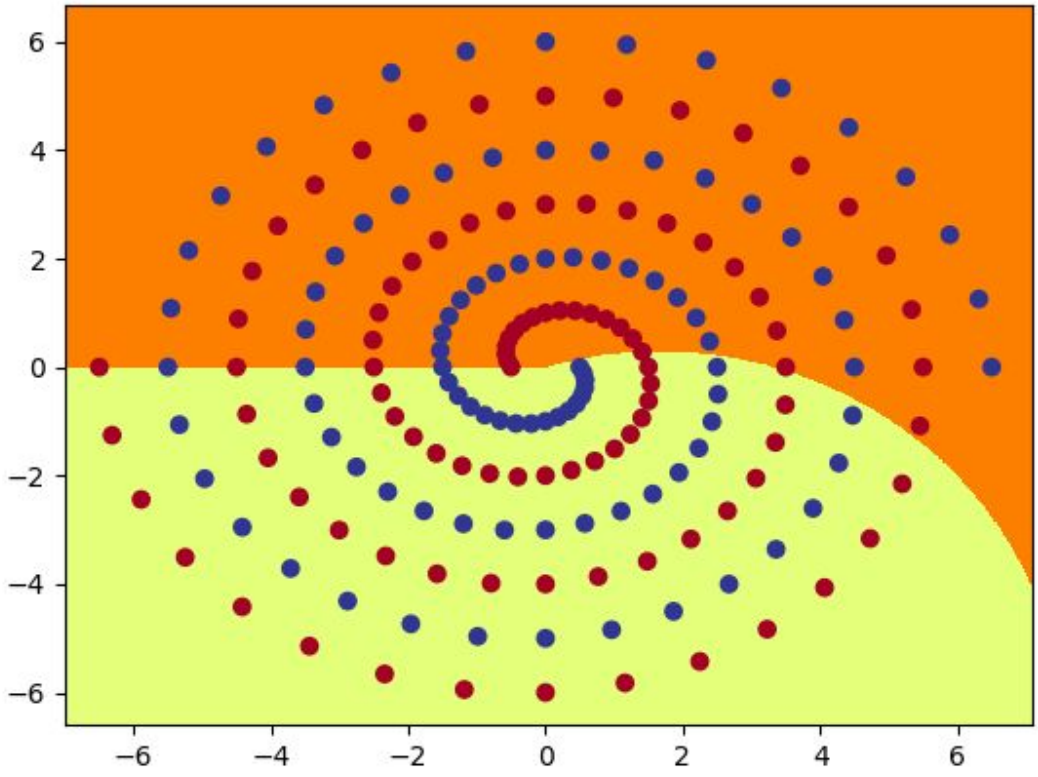

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

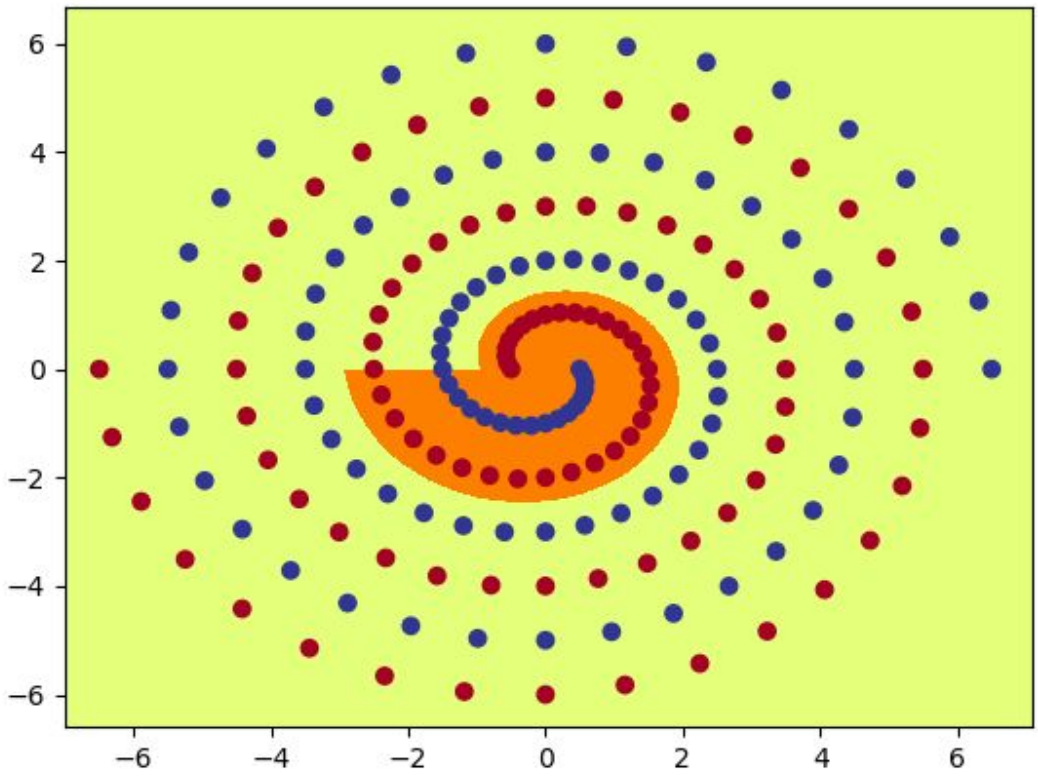
Appendix

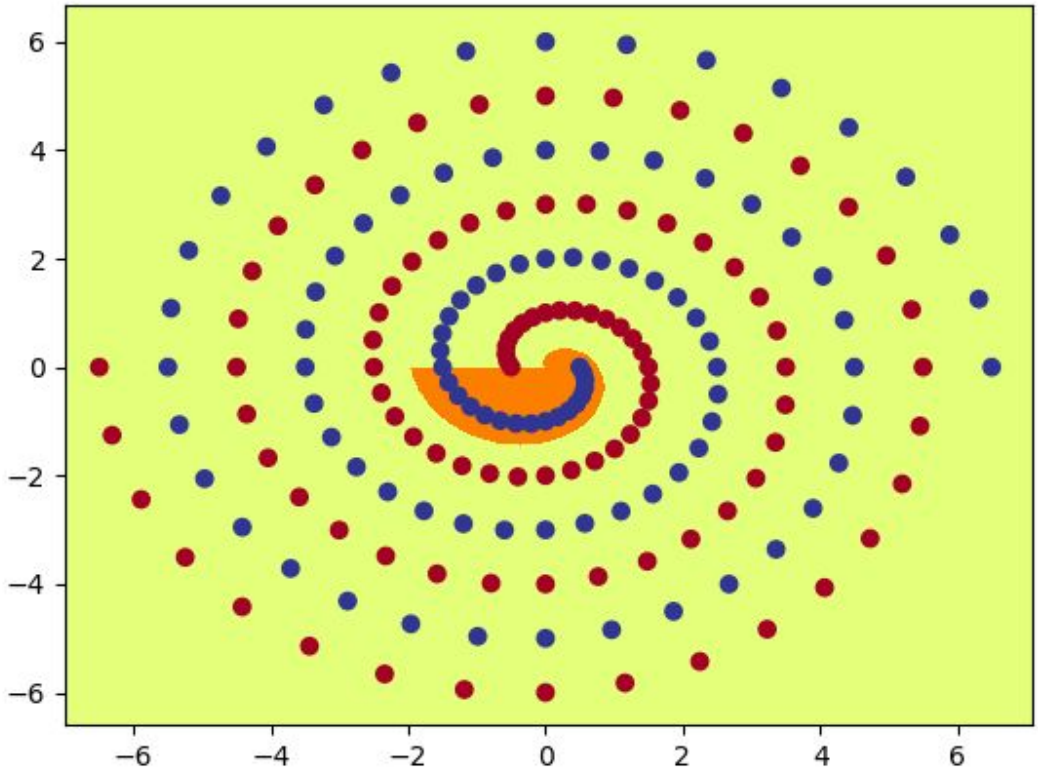
PolarNet

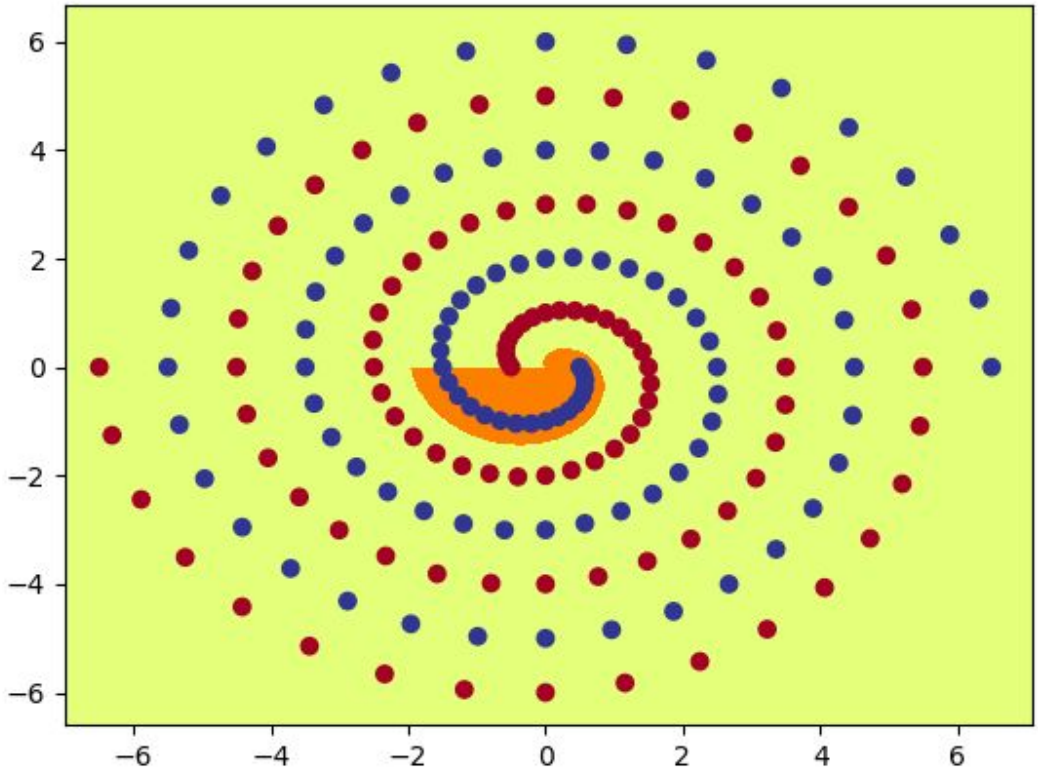


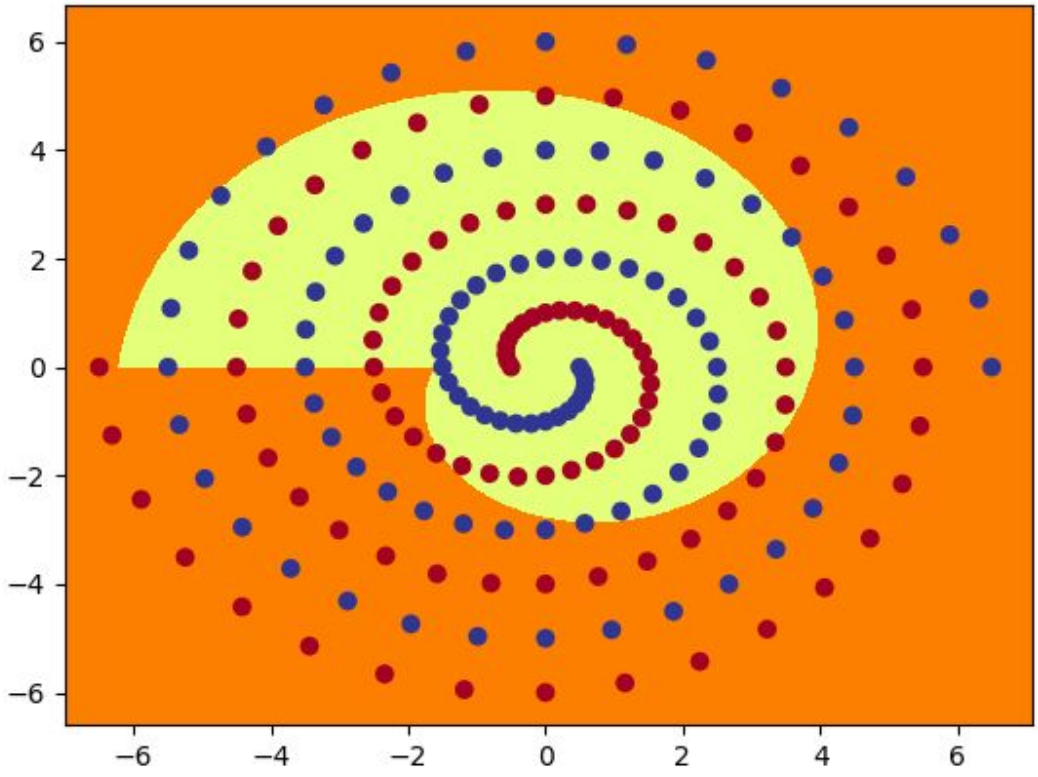

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

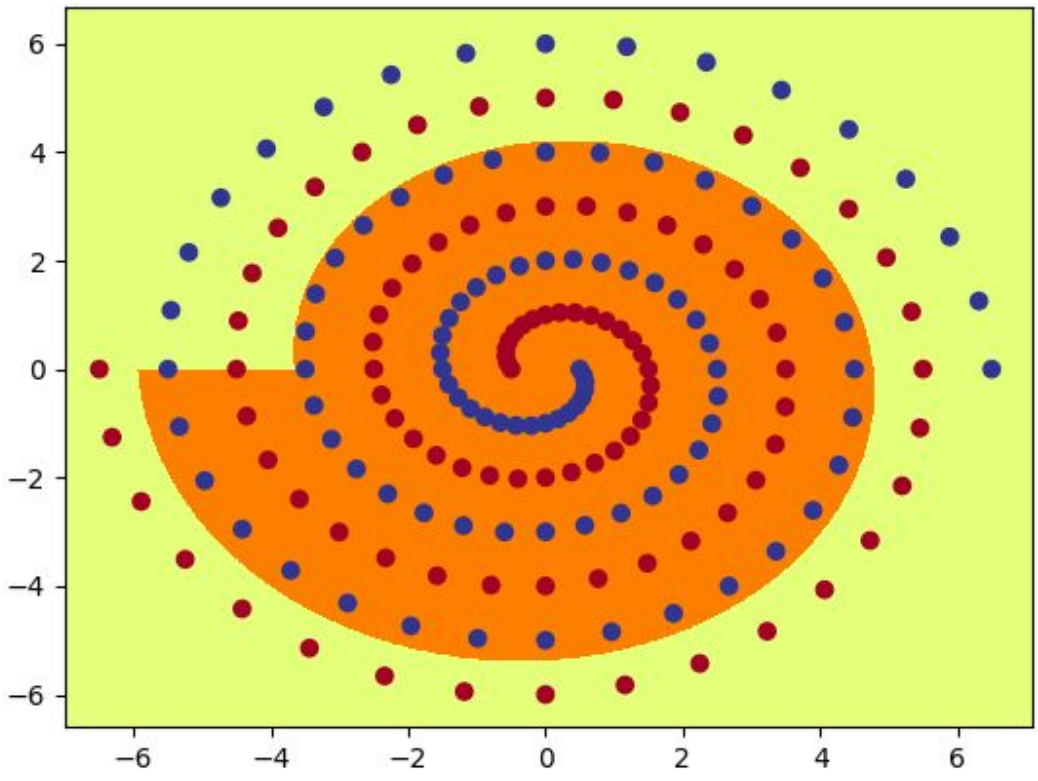

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

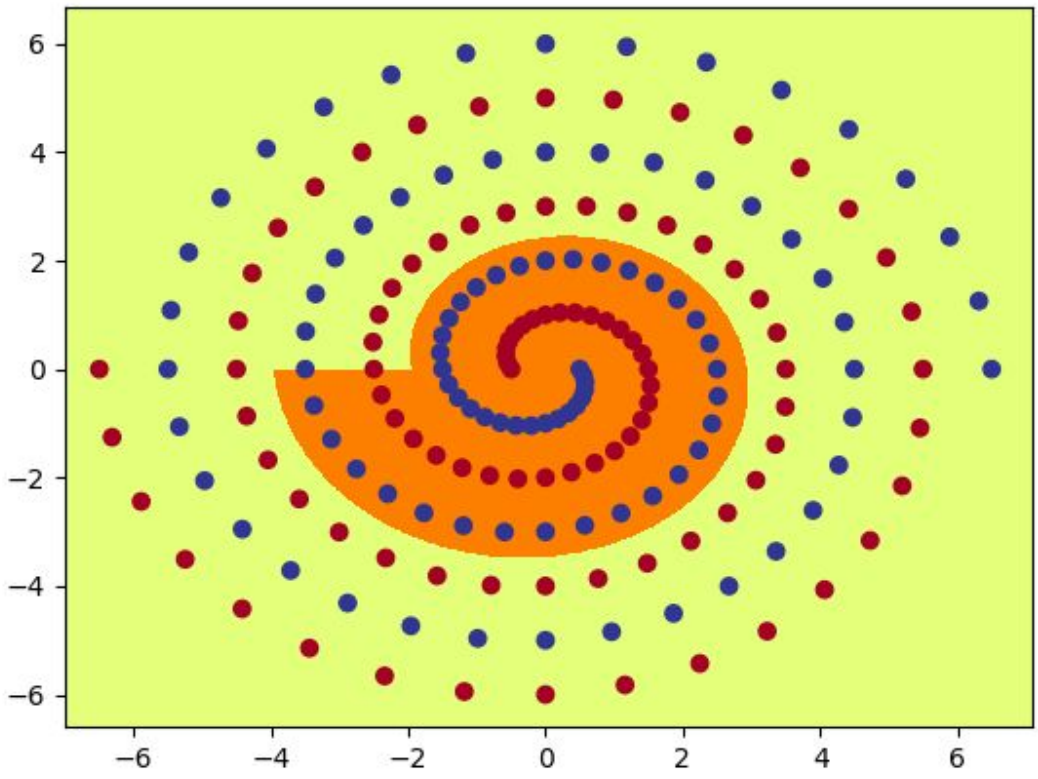

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

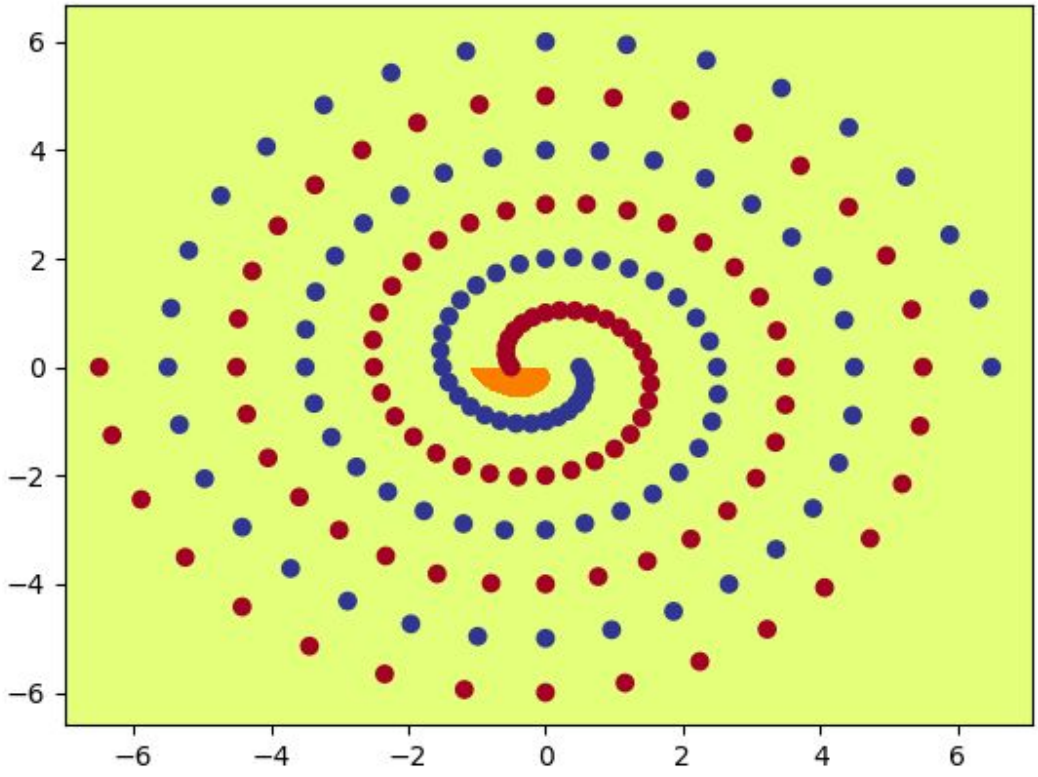

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

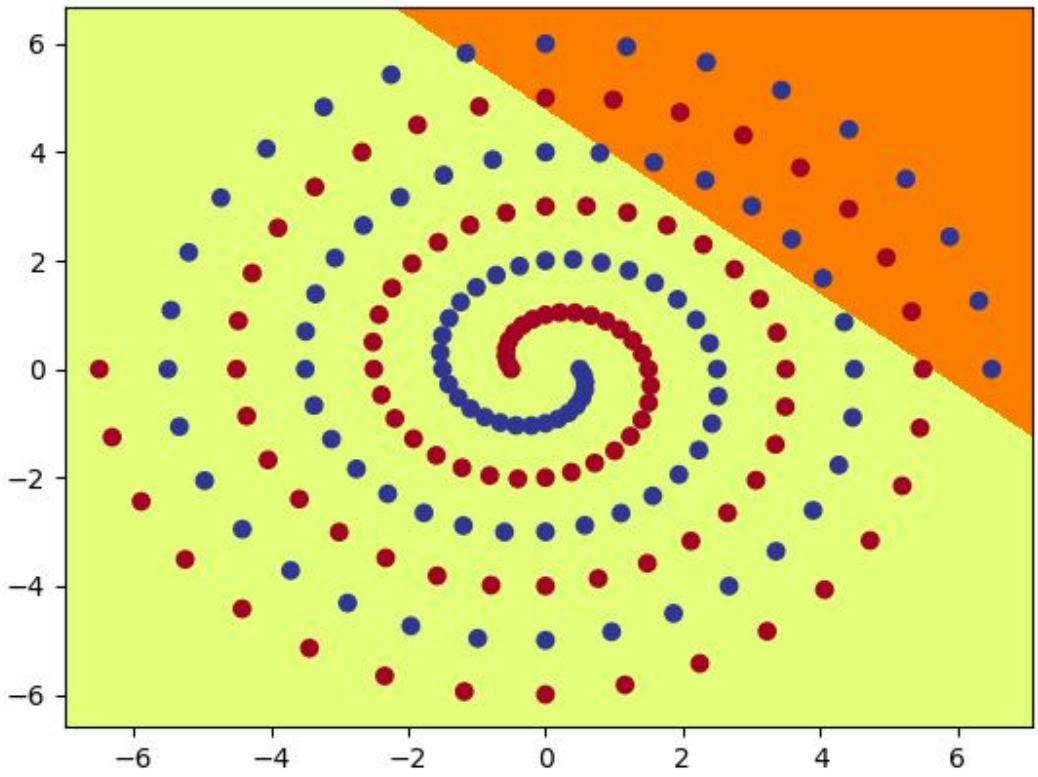

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

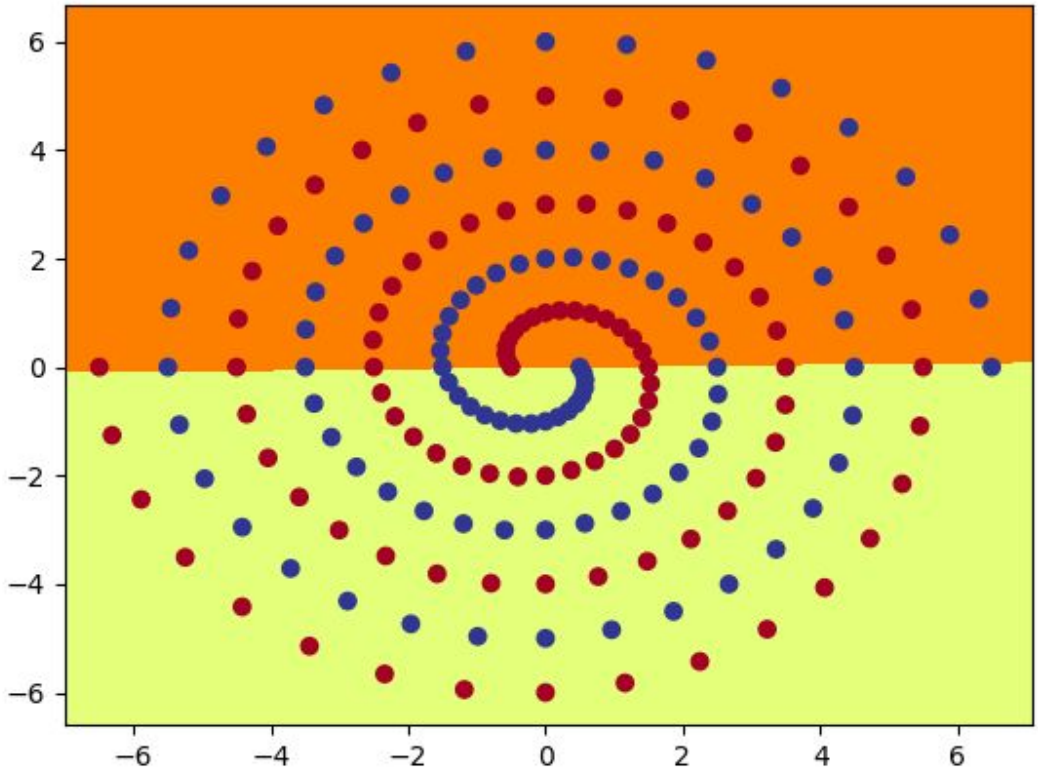

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

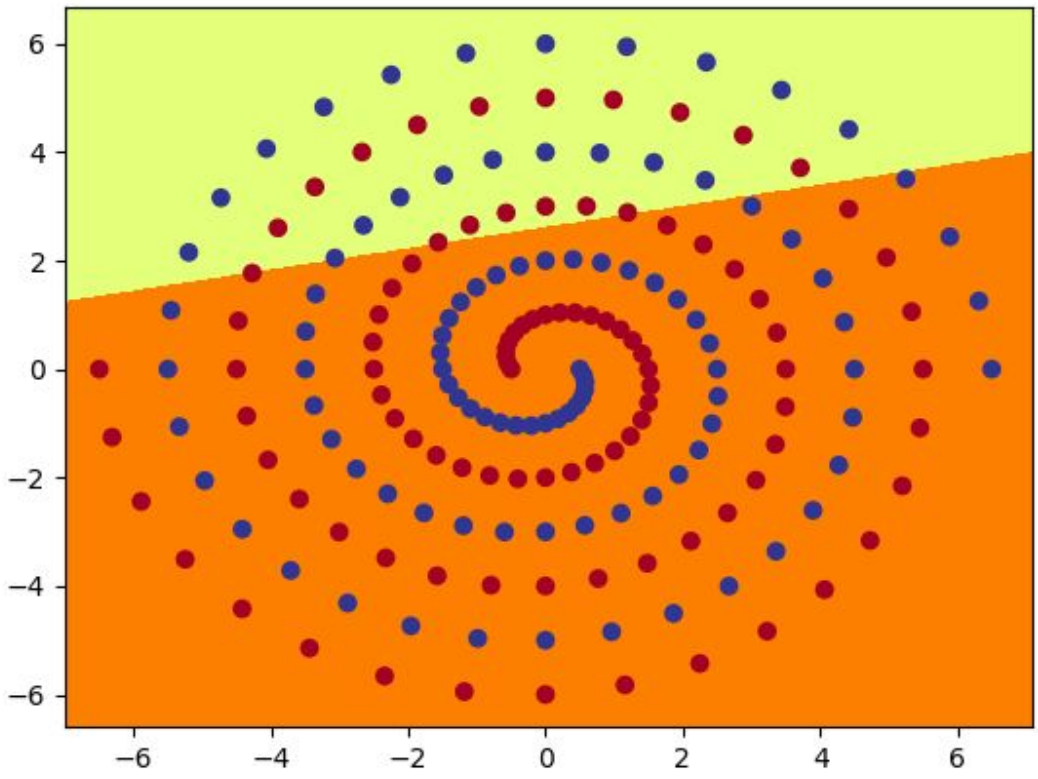

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

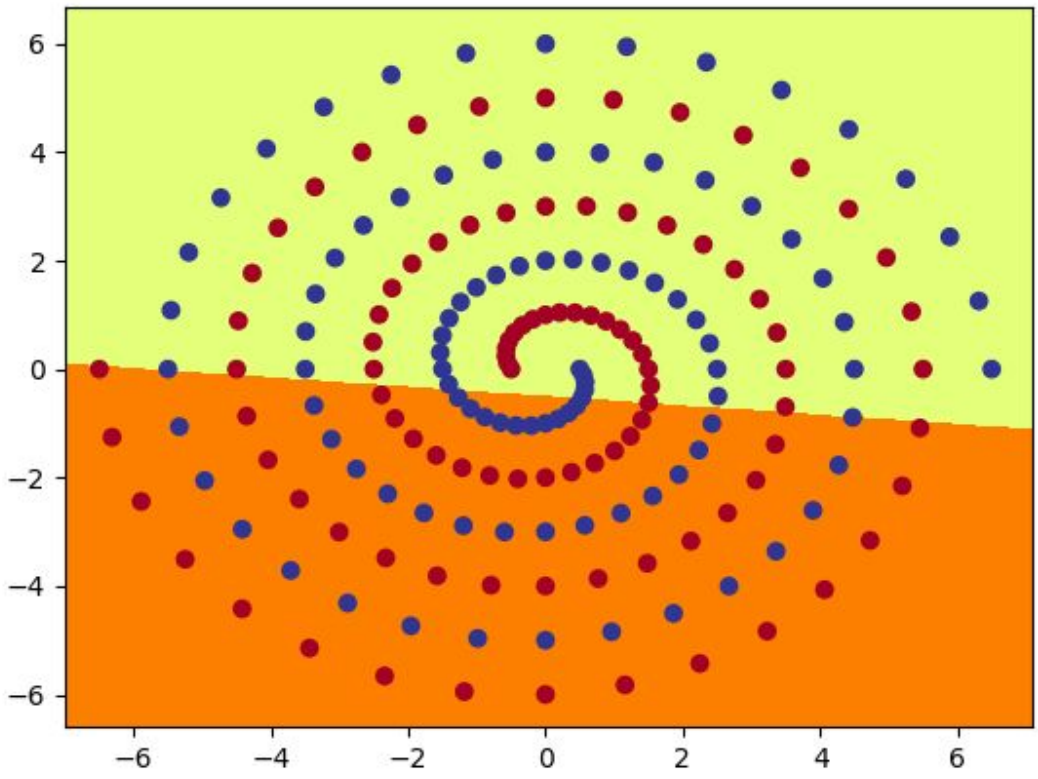

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

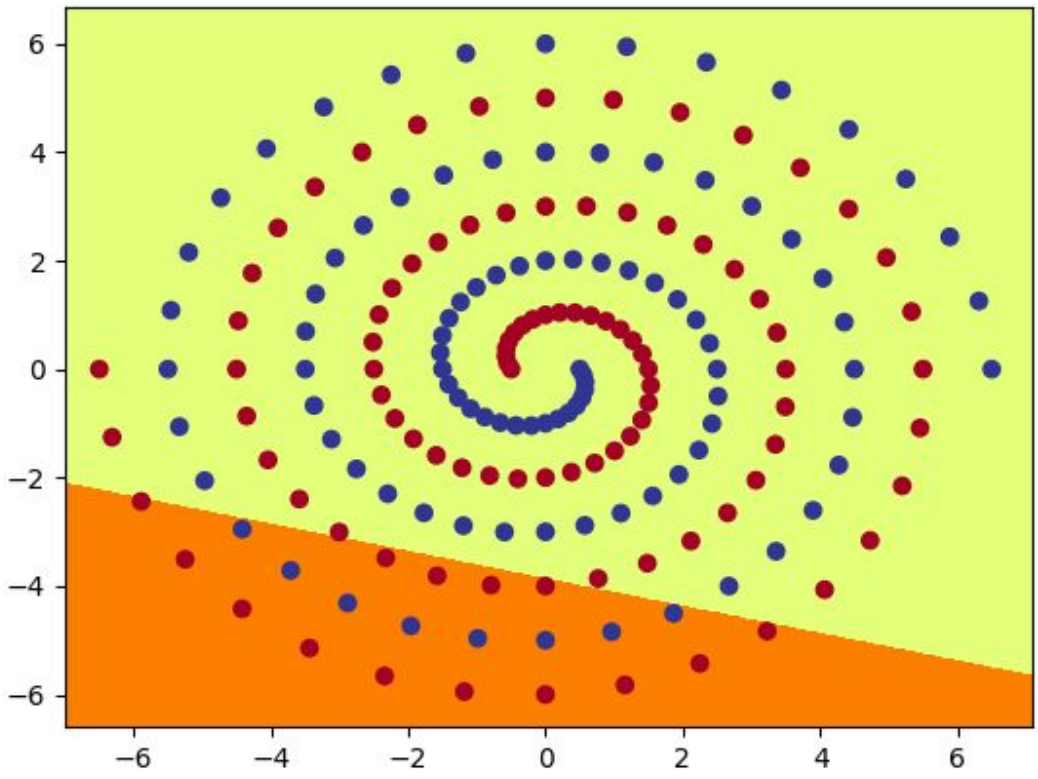
RawNet First Layer

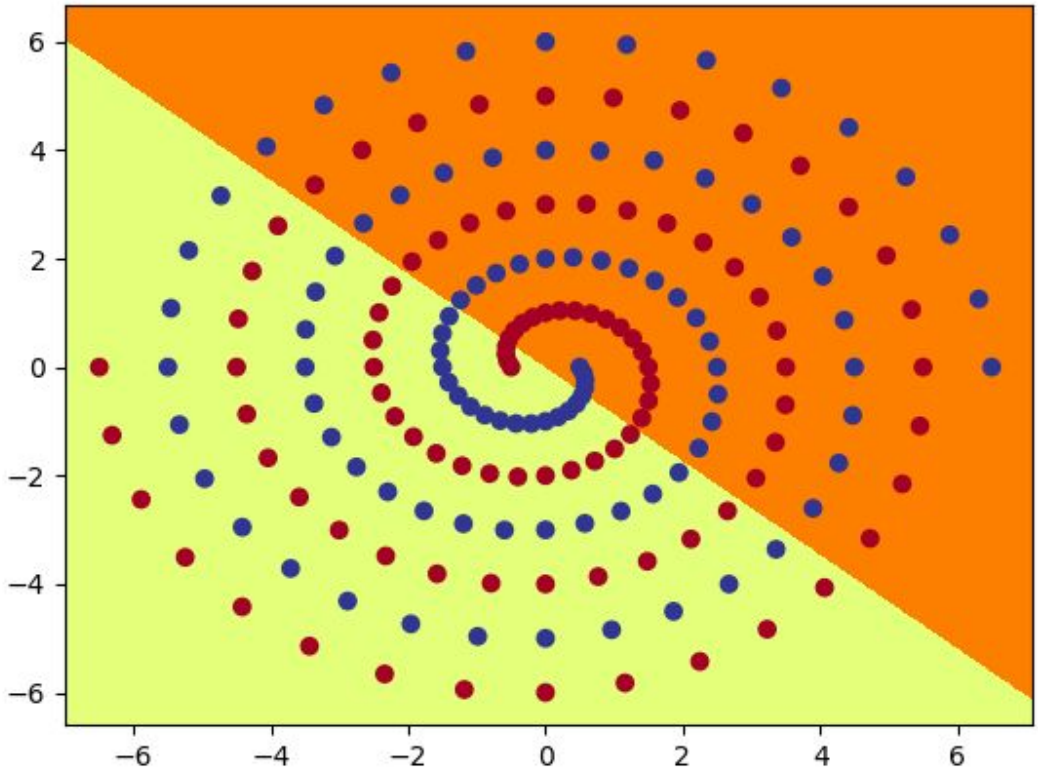


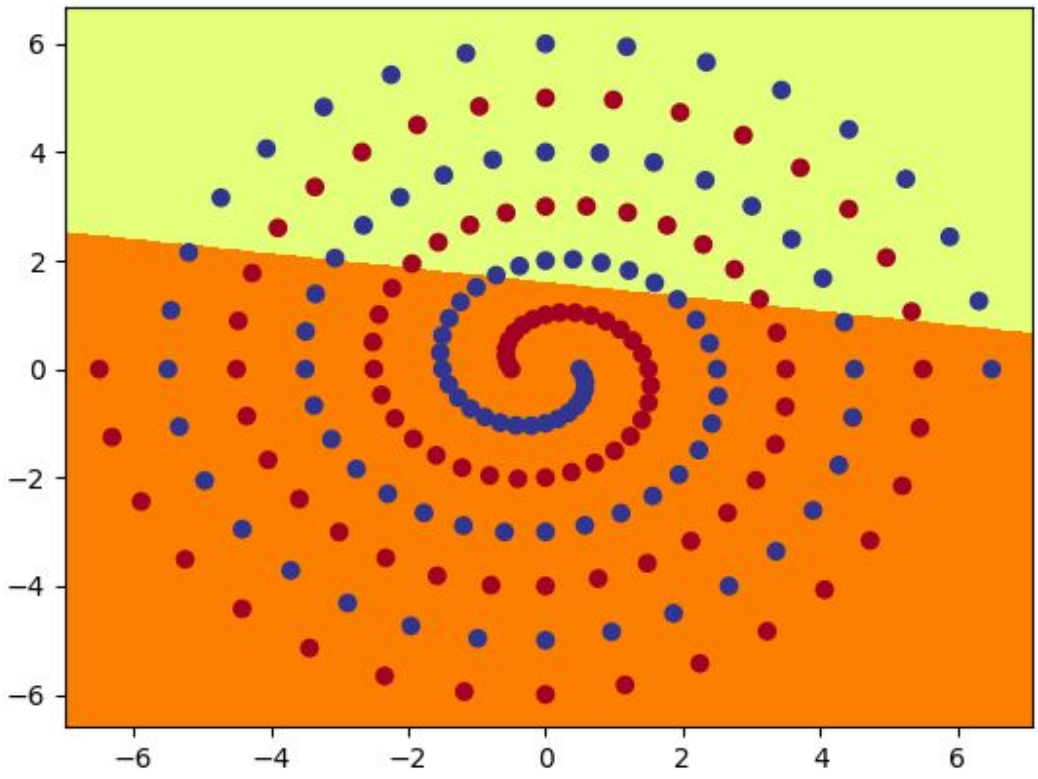

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

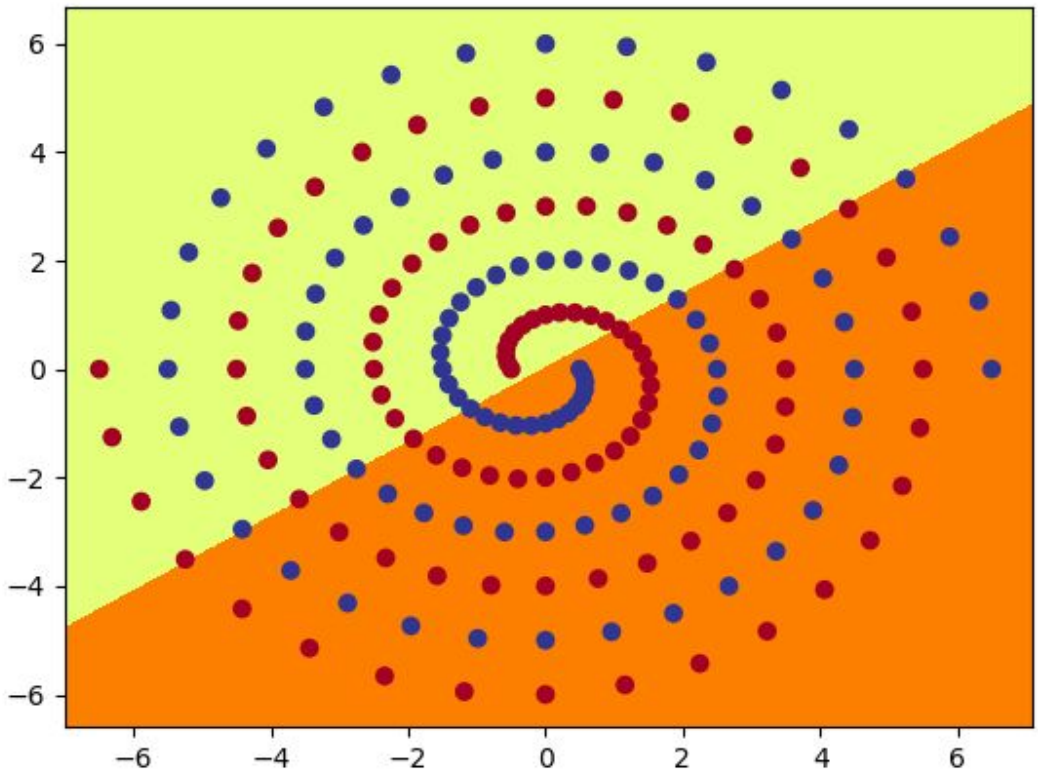

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

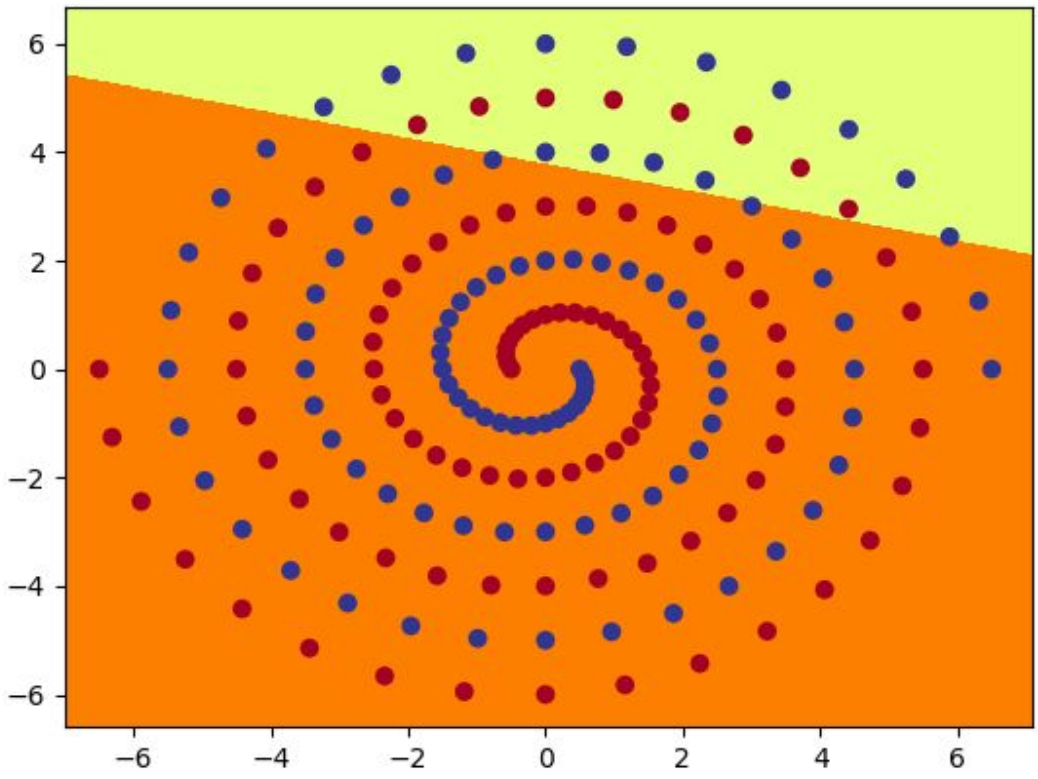

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

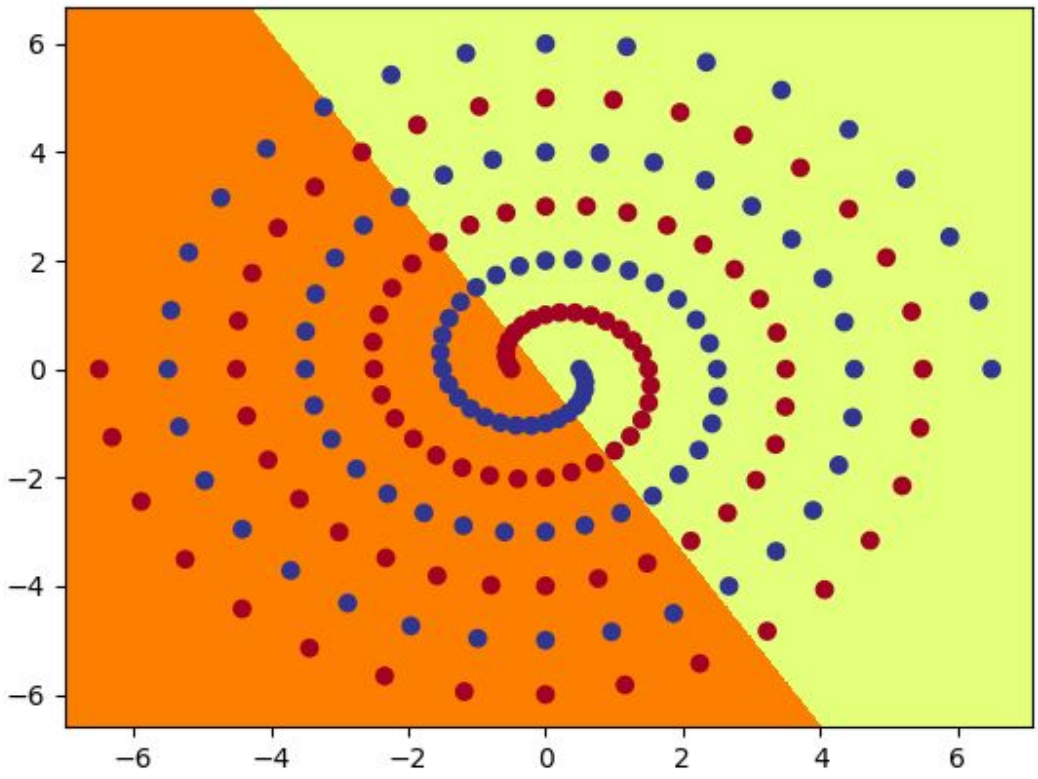

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

RawNet Second Layer