UNSW
AUSTRALIA

HW1

COMP9444 Neural Networks and Deep Learning

T2, 2020

z3290805 Pei Wang

University of New South Wales

Faculty of Computer Science and Engineering

July 2020

Part 1: Japanese Character Recognition

1.

```
class NetLin(nn.Module):
        def __init__(self):
                super(NetLin, self).__init__()
                self.fc = nn.Linear(784, 10)

        def forward(self, x):
                x = x.view(x.shape[0], -1)
                output = self.fc(x)
                probs = F.log_softmax(output)
                return probs
```

```
[[766.   7.   8.   4.  59.   8.   5.  16.  11.   8.]
 [  5. 667.  60.  40.  51.  27.  21.  29.  36.  51.]
 [ 10. 107. 688.  57.  79. 119. 144.  25.  95.  86.]
 [ 12.  18.  27. 755.  22.  17.  10.  12.  38.   3.]
 [ 29.  31.  27.  16. 623.  19.  26.  86.   8.  54.]
 [ 64.  23.  20.  59.  21. 728.  25.  17.  30.  32.]
 [  2.  57.  46.  13.  33.  29. 723.  53.  45.  19.]
 [ 63.  15.  38.  17.  35.   8.  20. 623.   7.  30.]
 [ 31.  25.  47.  28.  20.  33.  11.  90. 706.  39.]
 [ 18.  50.  39.  11.  57.  12.  15.  49.  24. 678.]]
```

Test set: Average loss: 1.0085, Accuracy: 6957/10000 (70%)

2.

```
class NetFull(nn.Module):
        def __init__(self):
                super(NetFull, self).__init__()
                global hidden_node
                self.fc1 = nn.Linear(784, hidden_node)
                self.fc2 = nn.Linear(hidden_node, 10)

        def forward(self, x):
                x = x.view(x.shape[0], -1)
                to_hidden = F.tanh(self.fc1(x))
                output = self.fc2(to_hidden)
                probs = F.log_softmax(output)
                return probs
```

80 hidden nodes gets an accuracy of 84%. Then the increase slows down. The highest accuracy was 85% with 120 hidden nodes.

```
[[859.   4.   8.   5.  49.  11.   3.  12.   9.   4.]
 [  4. 815.  15.   9.  27.  11.  13.  10.  27.  16.]
 [  3.  39. 838.  23.  19.  71.  50.  18.  24.  35.]
 [  5.   2.  29. 920.   7.   9.   7.   1.  44.   4.]
 [ 25.  19.  15.   1. 795.  10.  16.  30.   3.  25.]
 [ 24.   9.  19.  13.   8. 837.   8.  10.  13.  10.]
 [  4.  59.  25.   6.  34.  24. 888.  32.  32.  26.]
 [ 41.   7.  13.   2.  19.   3.   7. 819.   6.  16.]
 [ 28.  21.  21.   8.  21.  15.   1.  30. 832.  11.]
 [  7.  25.  17.  13.  21.   9.   7.  38.  10. 853.]]

Test set: Average loss: 0.5209, Accuracy: 8456/10000 (85%)
```

3.

```
class NetConv(nn.Module):
        def __init__(self):
                super(NetConv, self).__init__()
                self.conv1 = nn.Conv2d(1, 6, 5)
                self.conv2 = nn.Conv2d(6, 12, 5)
                self.fc1 = nn.Linear(4800, 400)
                self.fc2 = nn.Linear(400, 10)

        def forward(self, x):
                x = F.relu(self.conv1(x))
                x = F.relu(self.conv2(x))
                x = x.view(x.shape[0], 4800)
                x = F.relu(self.fc1(x))
                x = self.fc2(x)
                probs = F.log_softmax(x)
                return probs
```

The parameter I used are as below:

```
--mom 0.9
--lr 0.01
```

I also used two convolutional layers. The first one has 6 channels with kernel size = 5.

```
self.conv1 = nn.Conv2d(1, 6, 5)
self.conv2 = nn.Conv2d(6, 12, 5)
self.fc1 = nn.Linear(4800, 400)
self.fc2 = nn.Linear(400, 10)
```

It reached 94% at the 5th epoch.

4.a The accuracy of the three models are 70%, 85 and 94%. It is believed that the reason of such variation is related to the complexity of each model.

Firstly, the linear model has the lowest accuracy of 70%. While having the simplest structure, its accuracy stopped improving during the training. It is an indication of its overly simplified structure struggling to meet the demand of the task.

Secondly, the Full model reached an accuracy of 83% with 80 hidden nodes. It reached 85% with 120 hidden nodes. The accuracy stopped improving since then all the way to the end of the experiment with 200 hidden nodes. This suggests that having too many nodes does not necessarily produce better output. On the other hand, the additional layer improved the accuracy from 70% to 85%. This suggests that having an additional layer improves the outcome when the task demands a model with more complexity.

Lastly, the convolution model produced the best outcome of 94% consistently after some fine tuning. The fine tuning involves increasing the momentum to 0.9 and increasing the output channel of the convolution layers. This model started with a 90% accuracy at the 1st epoch and reached an accuracy of 94% at the 5th epoch. It was believed that the two convolution layers were the major factor of this accuracy, because it produced 4800 nodes as the output to the fully connected layer. An additional fully connected layer was also used as the 4th layer since it was allowed by the specification. Such structure enhances the depth and capability of the model, making it satisfies the complexity demand of the task. Details of this fine tuning process will be further discussed in 4.c.

b. The confusion matrix of the three models are listed below. Please find better formatted the output of each code section. The number on diagonal line from top left to bottom right indicates the correctly predicted character. Others are incorrectly predicted. It is found the 3rd row "su" す and the 7th row "ma" ま of the predication were most mistaken. "ha" は and "ma" were mistaken as "su". "ki" き and "re" れ were mistaken as "ma". After reading the Hiragana table, it was found that "su" "ma" and "ha" all have a circle in the writing. "ma" "ki" and "re" all two horizontal and one vertical strokes. These common features are likely to be more difficult for a model to differentiate in different characters.

Linear

```
[[766.   7.   8.   4.  59.   8.   5.  16.  11.   8.]
 [  5. 667.  60.  40.  51.  27.  21.  29.  36.  51.]
 [ 10. 107. 688.  57.  79. 119. 144.  25.  95.  86.]
 [ 12.  18.  27. 755.  22.  17.  10.  12.  38.   3.]
 [ 29.  31.  27.  16. 623.  19.  26.  86.   8.  54.]
 [ 64.  23.  20.  59.  21. 728.  25.  17.  30.  32.]
 [  2.  57.  46.  13.  33.  29. 723.  53.  45.  19.]
 [ 63.  15.  38.  17.  35.   8.  20. 623.   7.  30.]
 [ 31.  25.  47.  28.  20.  33.  11.  90. 706.  39.]
 [ 18.  50.  39.  11.  57.  12.  15.  49.  24. 678.]]
```

Test set: Average loss: 1.0085, Accuracy: 6957/10000 (70%)

Full

[[853. 7. 8. 3. 43. 5. 3. 17. 10. 1.]
[5. 818. 13. 9. 36. 11. 14. 12. 28. 20.]
[3. 32. 835. 33. 21. 66. 47. 14. 26. 46.]
[6. 2. 36. 914. 4. 10. 10. 3. 45. 4.]
[26. 23. 12. 1. 799. 14. 17. 27. 2. 29.]
[27. 10. 20. 18. 9. 852. 7. 11. 8. 5.]
[2. 56. 29. 3. 30. 16. 888. 37. 28. 12.]
[41. 4. 10. 2. 18. 2. 4. 819. 3. 21.]
[30. 17. 20. 7. 20. 15. 3. 27. 843. 8.]
[7. 31. 17. 10. 20. 9. 7. 33. 7. 854.]]

Test set: Average loss: 0.5008, Accuracy: 8475/10000 (85%)

Convolution

[[949. 3. 8. 0. 12. 1. 3. 5. 3. 8.]
[3. 929. 3. 2. 3. 2. 2. 2. 11. 7.]
[3. 5. 905. 14. 3. 21. 8. 9. 7. 13.]
[2. 1. 36. 974. 10. 13. 3. 2. 10. 4.]
[25. 4. 8. 0. 935. 4. 9. 2. 10. 5.]
[2. 5. 8. 0. 9. 932. 2. 1. 2. 1.]
[0. 34. 13. 6. 11. 14. 967. 12. 5. 4.]
[10. 2. 4. 2. 4. 1. 1. 953. 1. 2.]
[6. 8. 7. 1. 9. 5. 2. 5. 951. 8.]
[0. 9. 8. 1. 4. 7. 3. 9. 0. 948.]]

Test set: Average loss: 0.3493, Accuracy: 9443/10000 (94%)

c. The parameters adjusted for this task include learning rate, momentum and the output channels of the convolution layers. The structure of the model was also experimented with an additional max pooling layer and fully connected layer.

The optimal learning rate and momentum was found at 0.01 and 0.9 respectively. Which is consistent to the exercise solution released prior to this assignment. The fluctuations within the loss function between epochs was dealt with increased momentum. The default value would struggle to reach 90% accuracy. By setting the momentus high, the model moved much faster to achieve a high accuracy of 90% in the first epoch. The learning rate was kept at 0.01 because a higher value would make the learning process more volatile.

Several articles on number recognition were read as references prior to doing this task. In comparison, Japanese characters are found to be more complex than numerical numbers because they have more strokes. It was also found that the number of output channels in a convolution layer is related to the depth of feature maps. Hence, the number of channels was increased. This resulted in the number of nodes at the first fully connected being at a high 4800. In an effort to make the model

more efficient, the output channels were reduced by half as an experiment, resulting in half the node at 2400. The resulting accuracy would be at 91%. Hence this indicates it is required to have this amount of nodes to meet the demanded complexity of the task. Max pooling was also added to the convolution layer with a similar lowered accuracy at 90% as results. This indicates while max pooling reduces the sample size, the assumption it allows the model to make neglects the differentiating features in characters.

Lastly as previously elaborate in 4.a, an additional fully connected layer was also used as the 4th layer since it was allowed by the specification. It was done because the accuracy would also be lowered to below 90%. It showed that it was not optimal to have 4800 nodes converge to 10 output nodes in one layer.

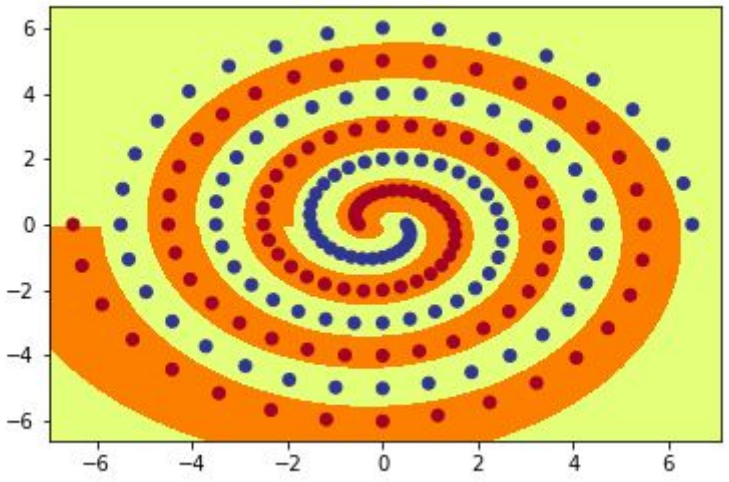
In summary, the complexity of the nn model has to meet the complexity of the task. This is reflected in the parameters and structure set for the model. Generally speaking, more discreet tasks require more nodes and layers and attention to the depth of features.

Part 2: Twin Spirals Task

1.

```
class PolarNet(torch.nn.Module):
        def __init__(self, num_hid):
                super(PolarNet, self).__init__()
                self.fc1 = nn.Linear(2, num_hid)
                self.fc2 = nn.Linear(num_hid, 1)

        def forward(self, input):
                # r=sqrt(x*x + y*y), a=atan2(y,x)
                converted = []
                for x,y in input:
                r= math.sqrt(x*x + y*y)
                a= math.atan2(y,x)
                converted.append([r,a])
                converted = torch.Tensor(converted)
                to_hidden = torch.tanh(self.fc1(converted))
                self.h1 = to_hidden
                output = self.fc2(to_hidden)
                output = F.sigmoid(output)
                return output
```

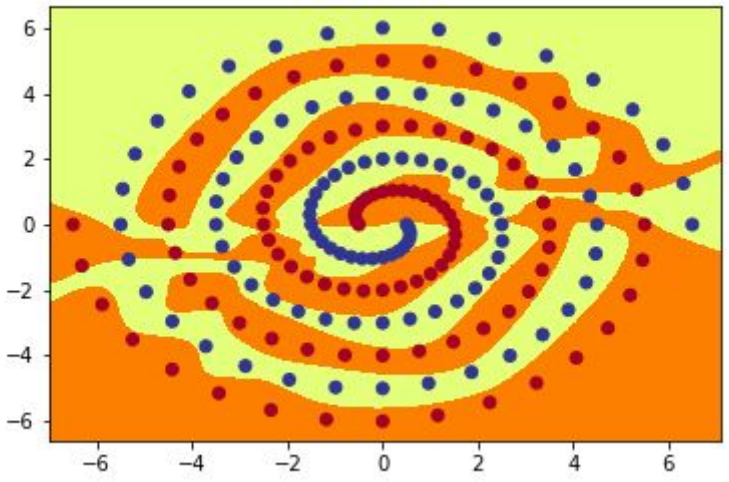
2. The minimum number of hidden nodes required is 8 to train PolarNet to 100% accuracy.



3.

```
class RawNet(torch.nn.Module):
        def __init__(self, num_hid):
        super(RawNet, self).__init__()
        self.fc1 = nn.Linear(2, num_hid)
        self.fc2 = nn.Linear(num_hid, num_hid)
        self.fc3 = nn.Linear(num_hid, 1)

    def forward(self, input):
        to_hidden1 = torch.tanh(self.fc1(input))
        self.h1 = to_hidden1
        to_hidden2 = torch.tanh(self.fc2(to_hidden1))
        self.h2 = to_hidden2
        output = self.fc3(to_hidden2)
        output = F.sigmoid(output)
        return output
```

4. The size of the initial weights is 0.3. Other parameters are kept at default.

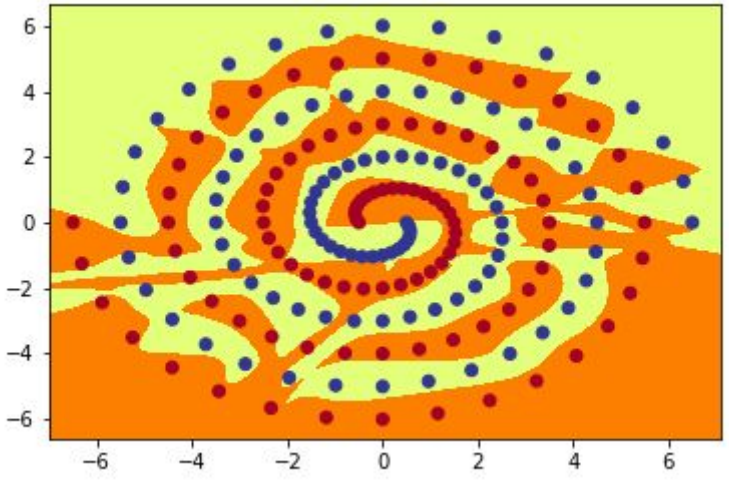


5. One thing worth noticing is that torch.cat was considered but the add operation was used in the end for creating the shortcut. This is done for three reasons: Firstly, despite the shortcut being generated differently compared to the input from the previous layer, it could still serve the purpose of enhancing or adjusting the combined input value before passing into the tanh activation function, which still generates a value between -1 and 1. Secondly, according to the specification, the number of nodes in the hidden layer would have to be set by the command line argument. Torch.cat would either double the number of data per batch to 97*2 = 194 when used without dim, or double the number of nodes in the following hidden layer to 10*2 = 20 with dim = 1. It is considered a violation to the specification. Last but not least, having two sets of tanh output per layer (one to the next layer, another as a shortcut to further layers) complicates the generation of diagrams per node in a later question. Therefore, the benefit of using add operation outweighs the usage of torch.cat in this project. However, it should be used in the future on other occasions.

```
class ShortNet(torch.nn.Module):
        def __init__(self, num_hid):
                super(ShortNet, self).__init__()
                self.fc1 = nn.Linear(2, num_hid)
                self.fc2 = nn.Linear(num_hid, num_hid)
                self.fc3 = nn.Linear(num_hid, 1)
                self.fc4 = nn.Linear(2, 1)

        def forward(self, input):
                input_to_h = torch.tanh(self.fc1(input))
                input_to_out = self.fc4(input)

                h1_to_h2 = torch.tanh(self.fc2(input_to_h)+input_to_h)

                h1_to_out = self.fc3(input_to_h)
                h2_to_output = self.fc3(input_to_h + h1_to_h2)
                output = F.sigmoid(input_to_out+h1_to_out+h2_to_output)
```

```
        self.h1 = input_to_h
        self.h2 = h1_to_h2

        return output
```

6. The command line argument being used are the following:
--init 0.2
--hid 14



7.

```
def graph_hidden(net, layer, node):
        xrange = torch.arange(start=-7,end=7.1,step=0.01,dtype=torch.float32)
        yrange = torch.arange(start=-6.6,end=6.7,step=0.01,dtype=torch.float32)
        xcoord = xrange.repeat(yrange.size()[0])
        ycoord = torch.repeat_interleave(yrange, xrange.size()[0], dim=0)
        grid = torch.cat((xcoord.unsqueeze(1),ycoord.unsqueeze(1)),1)
        with torch.no_grad(): # suppress updating of gradients
        net.eval()        # toggle batch norm, dropout
        output = net(grid)
        net.train() # toggle batch norm, dropout back again
        output = ''
        if layer == 1 :
        output = net.h1
        else:
        output = net.h2
        act_val = output[:,node]
        pred = (act_val >= 0.5).float()
        plt.clf()
        plt.pcolormesh(xrange,yrange,pred.cpu().view(yrange.size()[0],xrange.size()[0]),
cmap='Wistia')
```

Please find diagrams in the appendix of the report.

8a. There are differences in the diagrams produced on the hidden layer and the final output between different models. Within the hidden layer of the polar net, nodes are producing output with curves that fit well to the spiral data in the diagram. The other two models have most straight lines. In the second layer of raw and short net, curves started to become more prevalent. The outcomes are closer to the hidden layer output of polar, but there are still noises in the diagrams.

Networks use activation functions to determine whether a node should fire a signal or not. Additional diagrams were generated to visualize the output of three models(*_act.png found in the appendix of this report) with node ID on x-axis and tanh value on y-axis. After comparing the diagrams of activation function, it was found the faster a model was trained, the more polarized the values in the diagrams were. Polarized in this case means activation values were concentrated to both 1 and -1, which indicated how certain a node was to either fire or not fire the signal. For example, the polar net is the most optimized model. The activation values are highly concentrated around 1 and -1. Although the other two models could also classify the data correctly, their activation functions are more evenly spreaded between -1 and 1. There were also nodes that always fire or not fire, which were not helpful, too. This suggests most nodes in raw and short net were unsure about firing. It was also reflected in the generated spiral diagram (graph_output) which has more sharp edges in comparison. Nodes that did manage to have polarized value were believed to be the one that carried the model through the training. Polar net on the other hand, has more nature and smooth curves.

b. The initial weight set the variance of the normal distribution function used for calculating the loss function. Having the right init value would more likely successfully train the model faster. Having a value too low or too high would lead to slow improvement of accuracy and a lower chance of success. The optimal value was found by trial and error. For example, it was found setting init as 0.3, 0.6 and 0.8 could all reach 100% for the Raw model. However, 0.3 took less than 7500 epochs while the other two both took over 15000. After further experimentations, an init of 0.3 turned out to be the most consistent. On the other hand, the short model is best performing with init set as 0.2. This suggests the optimal value varies depending on the model.

c. The polar net model is the most natural model out of the three. It has been optimized with a math formula. It puts a strong emphasis on the importance of fine tuning a model. The learning process could be greatly improved if our understanding of the task as human beings could be translated into logical or mathematical representations. Should guidance as such be available, a neural network could be simplified in structure while enjoying a shorter training time and a more natural result. In this case, the formula specializes in a general purpose network as an expert in recognizing spirals. Vice versa, the absence of the guidance would require additional computational power and more complex structure as compensations for a network to be trained with most likely a less desirable result.

d. Experiments were done. The changes made for the experiment were removed before modifying the net value.

With torch.optim.SGD: it involves the removal of some parameters such as eps and beta. Momentum could be added. The accuracy could reach 75% after some fine tuning in raw.

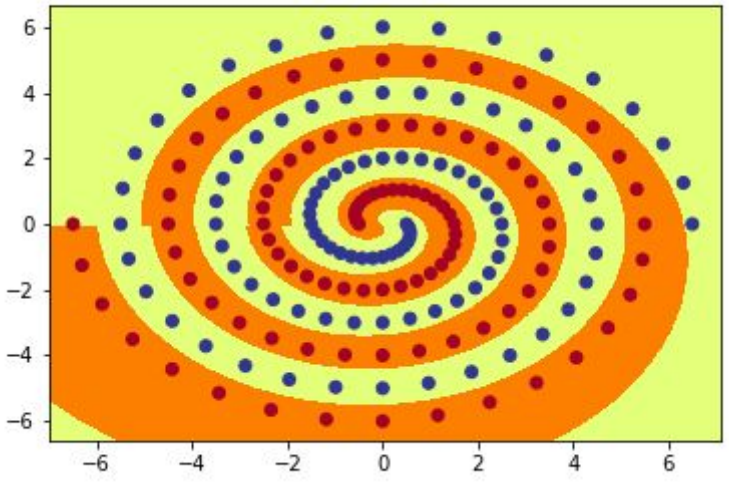
With an additional hidden layer of 10 nodes: In raw net, tt was noticed that if the model did end up reaching 100% of accuracy then it could do it quicker than before. However, there are times when the accuracy suffers from large scale fluctuations. For example, there are times when the accuracy would drop from high 90% to the low 80% for a few (100s) epochs then raised back. Therefore, it was believed that excessive layers would not benefit the training of a model.

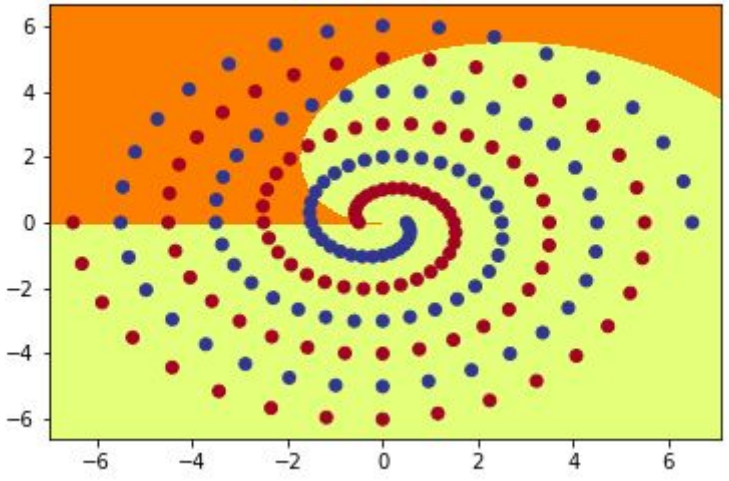
With double the batch size to 194, all three net could be trained in half of the time. For example, the polar net is about 1000 epochs. And short net in between 5000 to 9000 epochs.

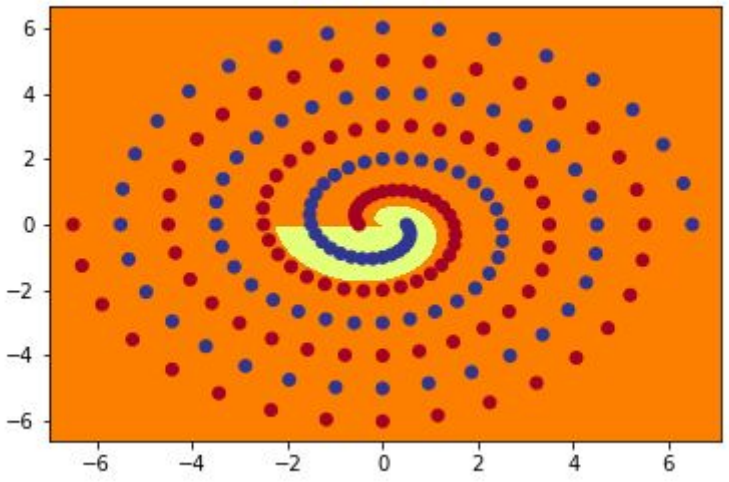
Replacing tanh with relu as the activation function has a negative effect on the polar model that it would take 50% more epochs to train (from 2200 epoch to 3500 epochs). Raw and short net also suffers that their accuracy would drop to around 85% - 95% by the end of the training.

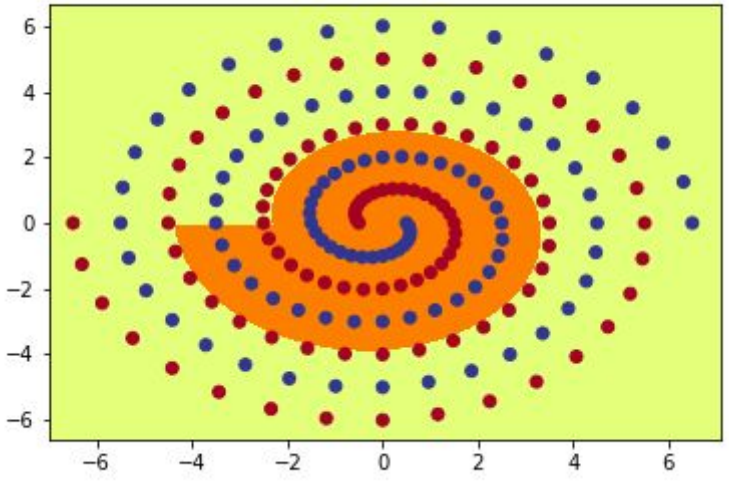
Appendix

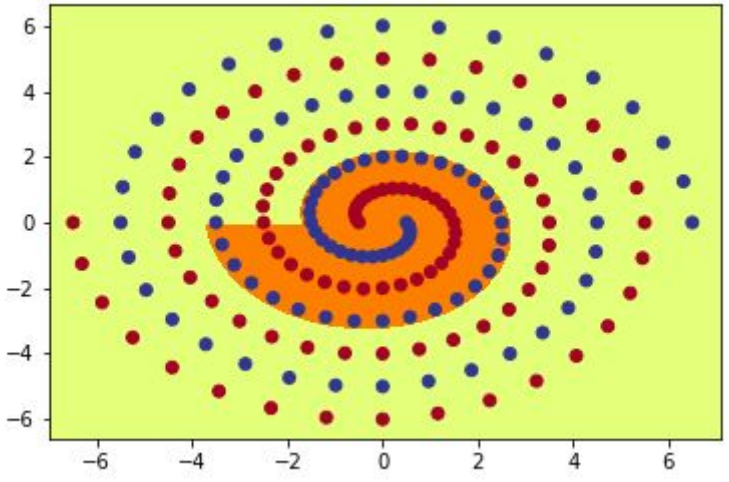
Polar net

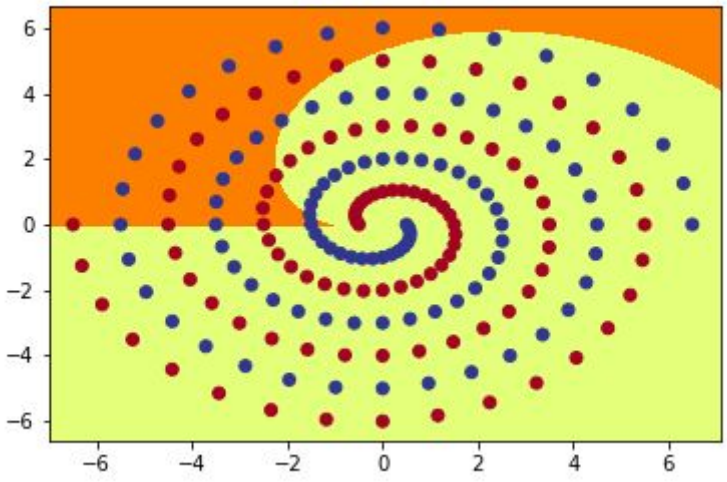


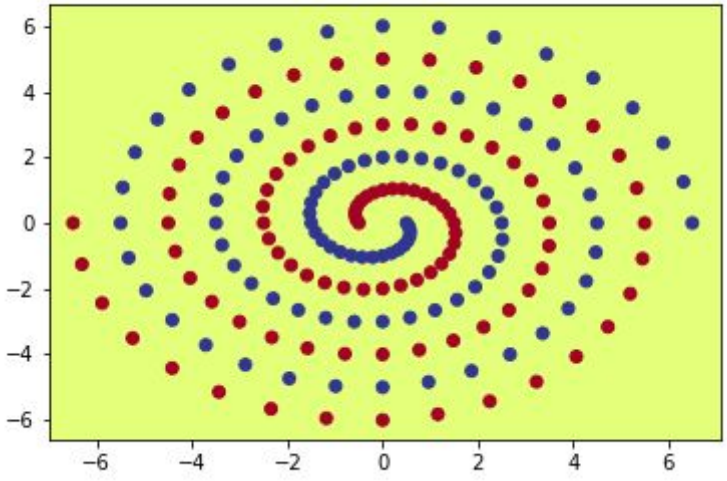


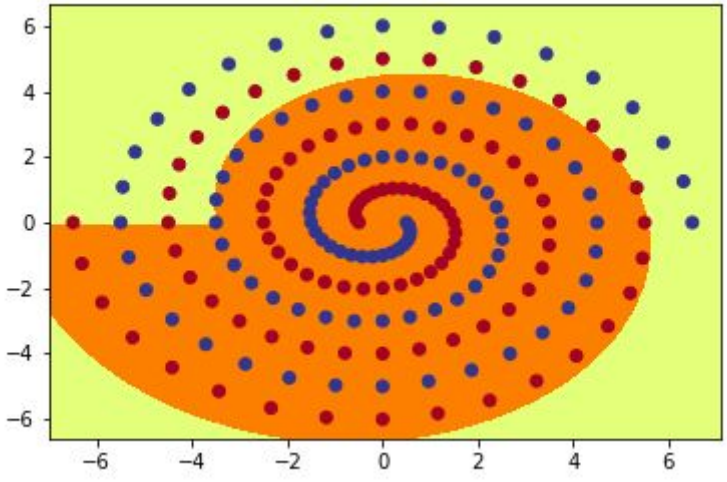

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

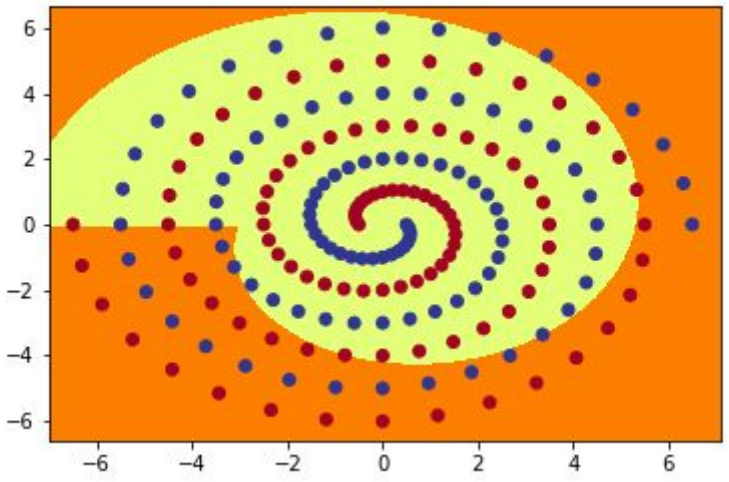

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

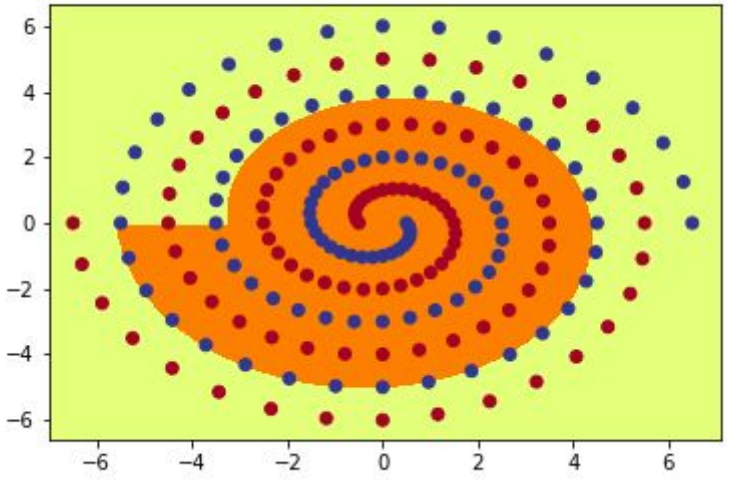

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

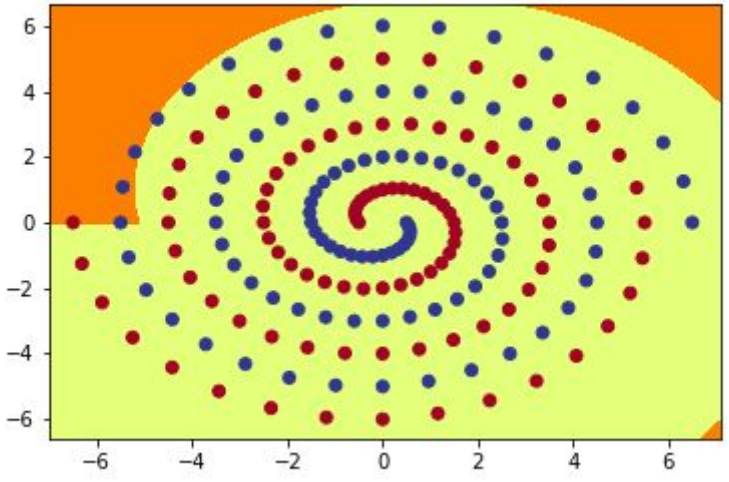

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

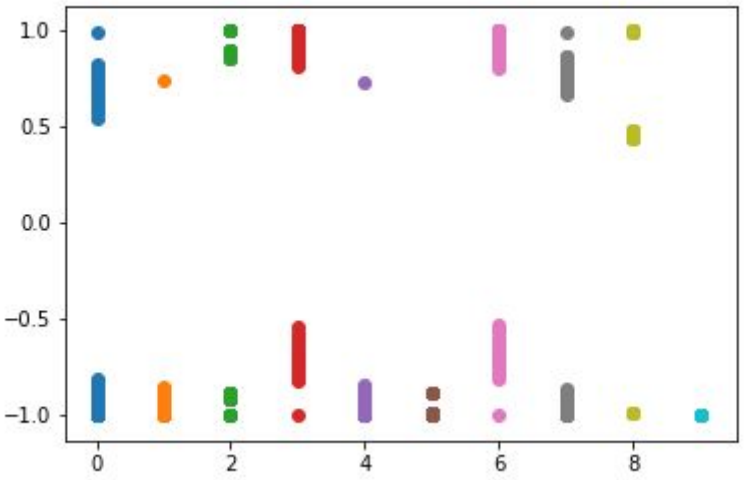

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6



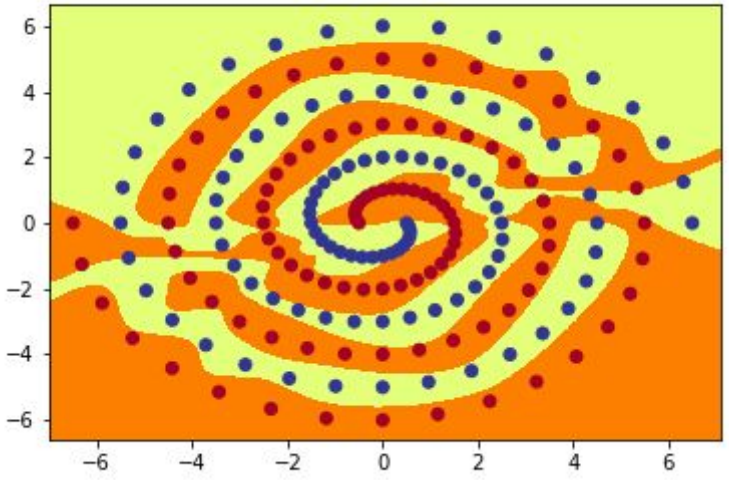


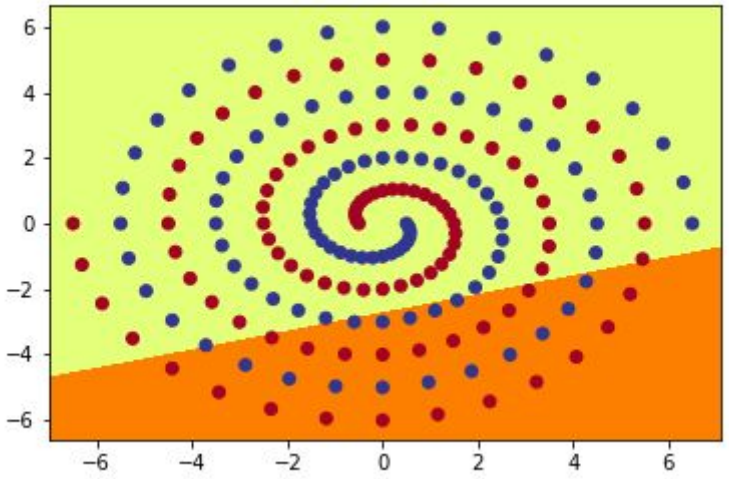


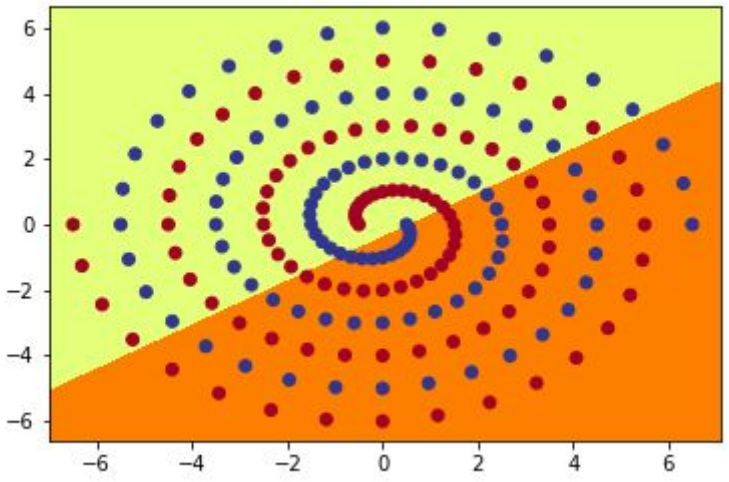
Polar_act (used in the last question):

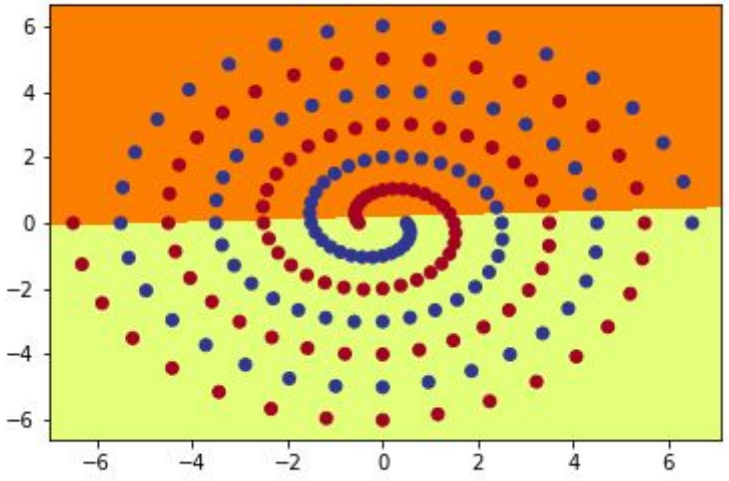


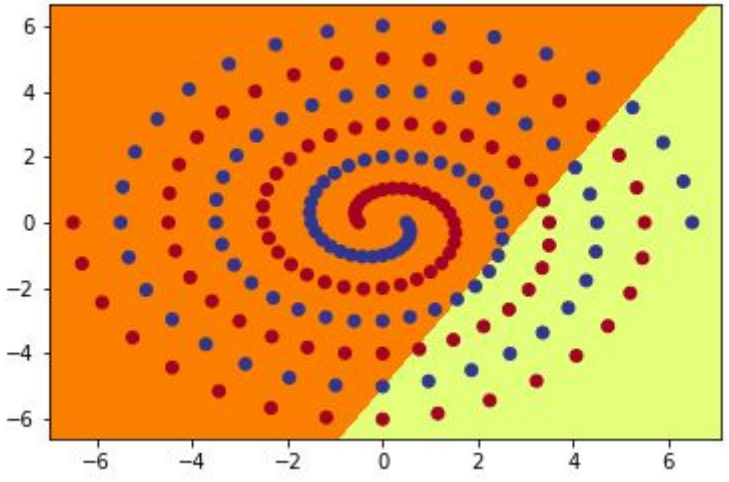
Raw net layer 1:

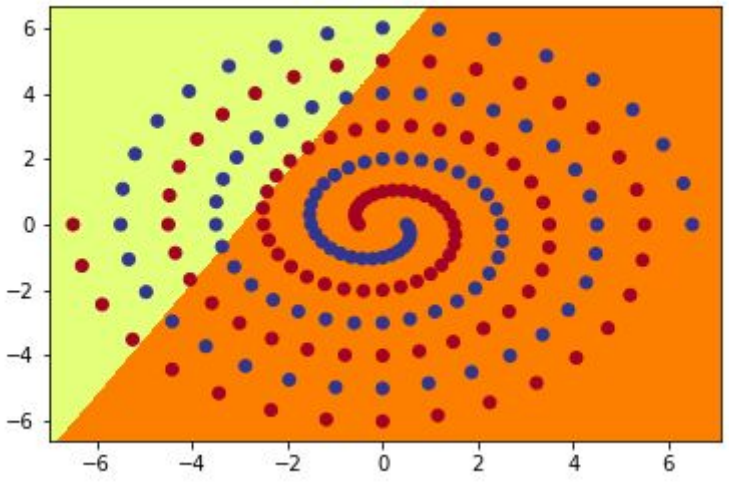


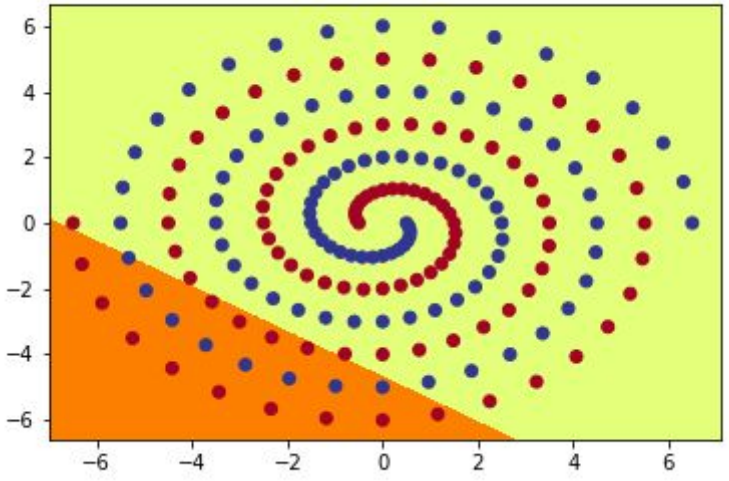


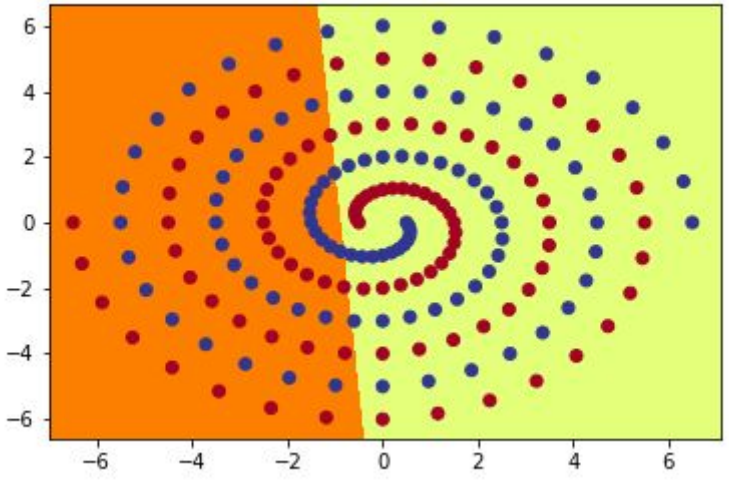

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

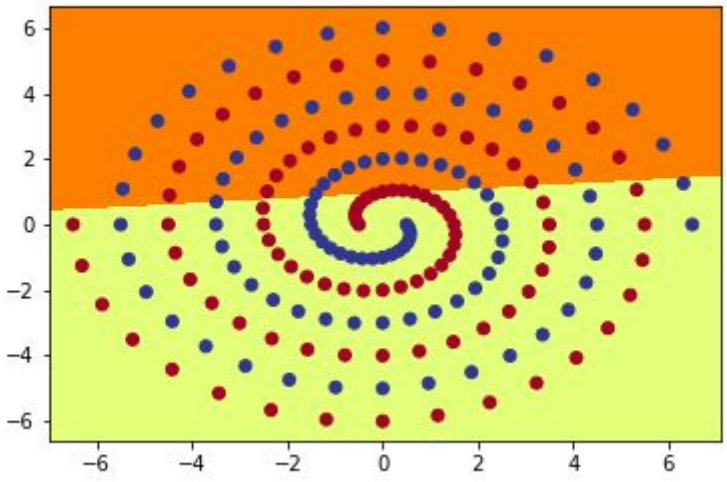

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

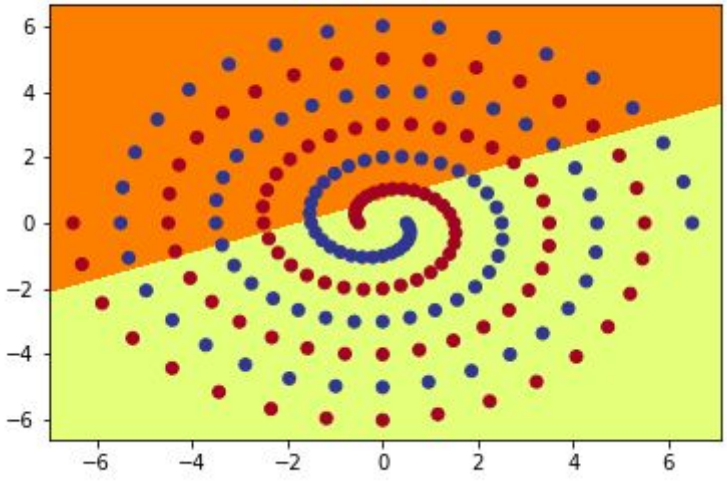

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

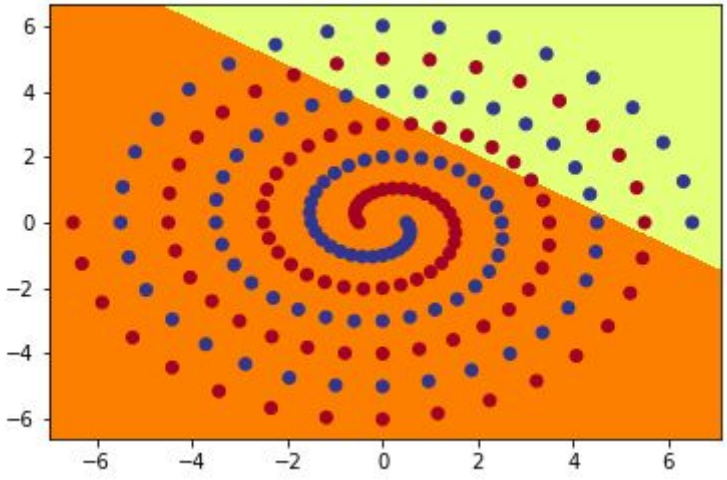

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

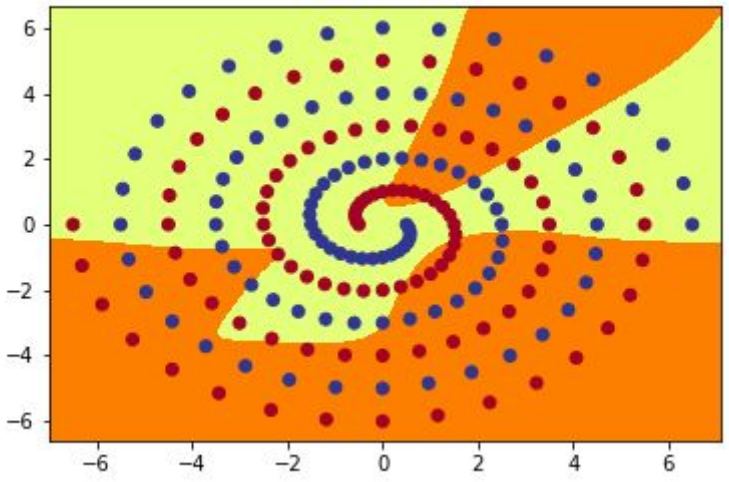

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

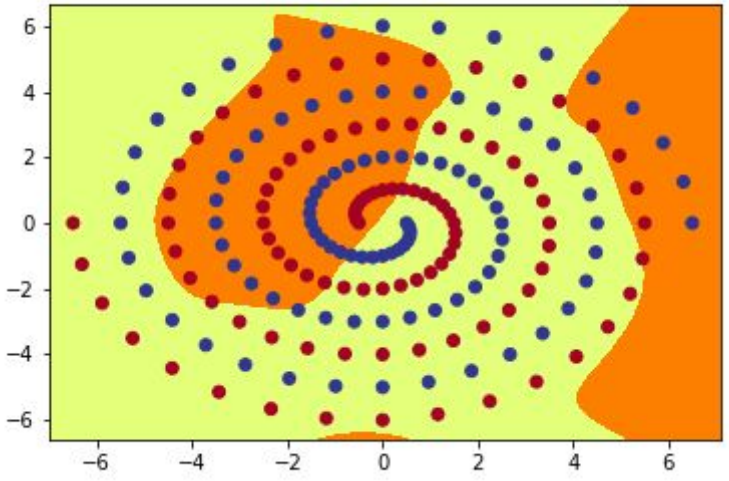


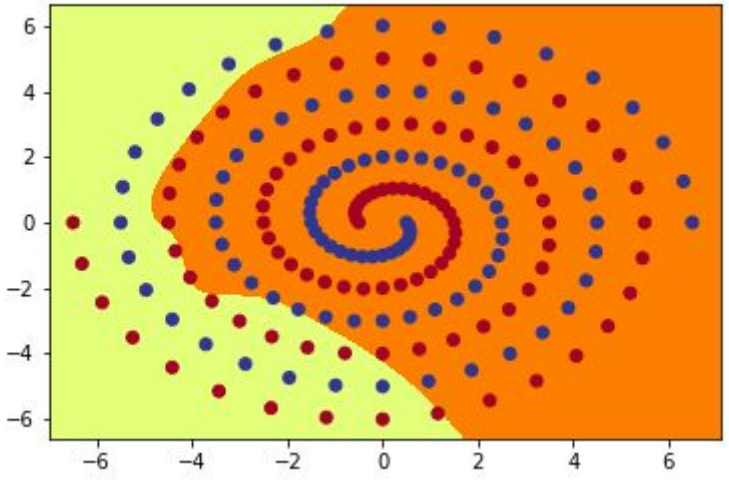


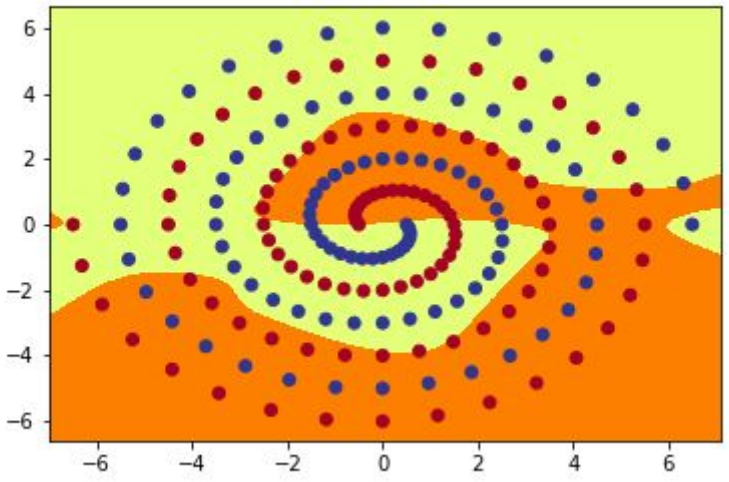


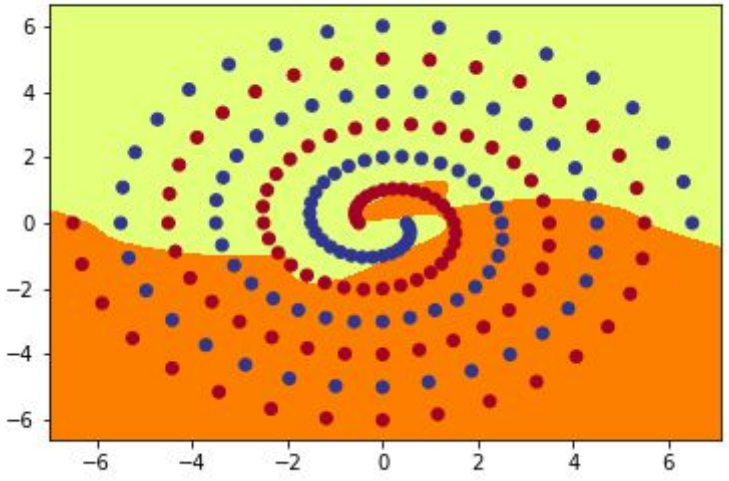
Raw net layer 2:

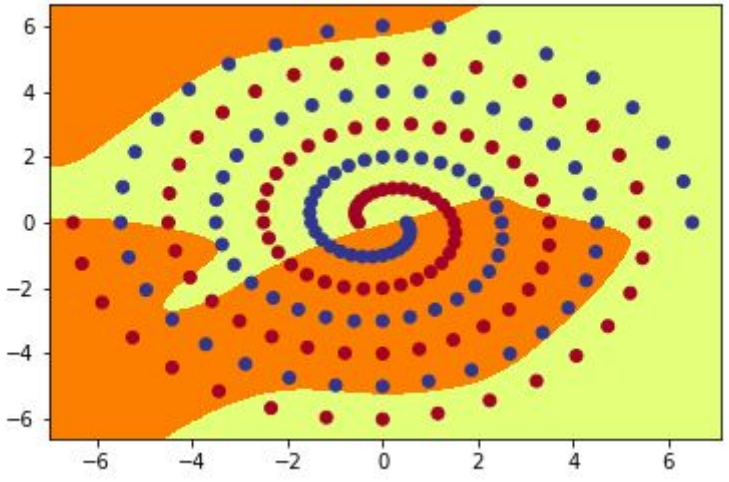

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

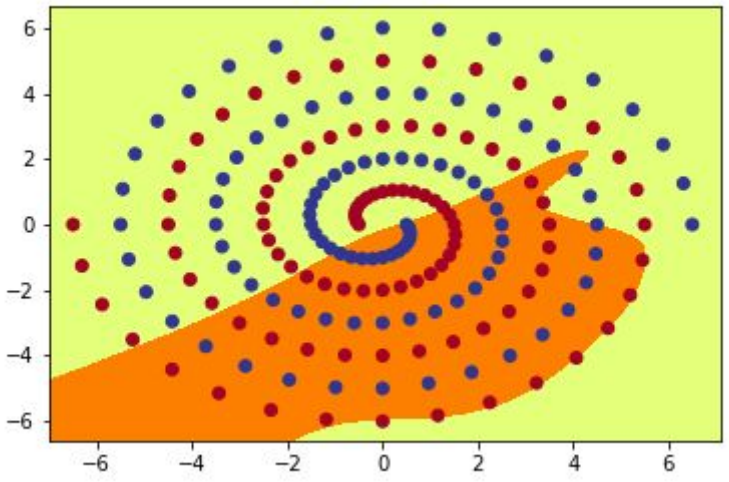

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

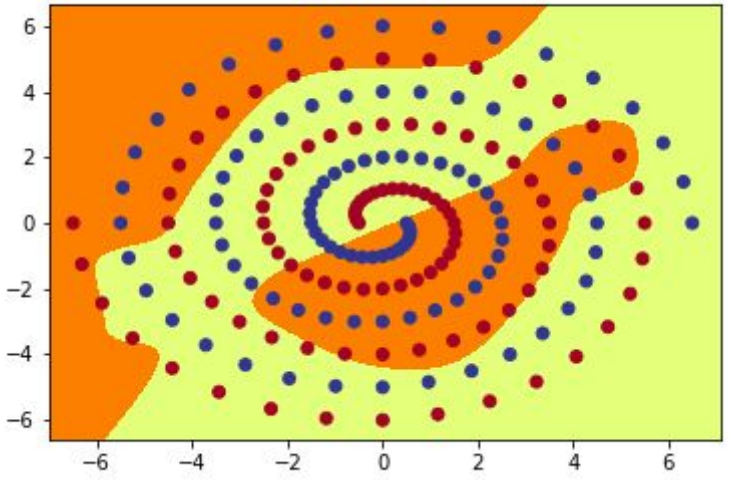

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

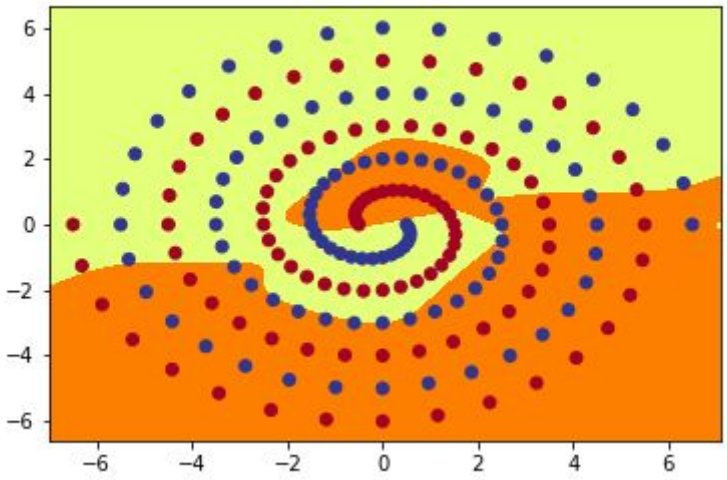

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

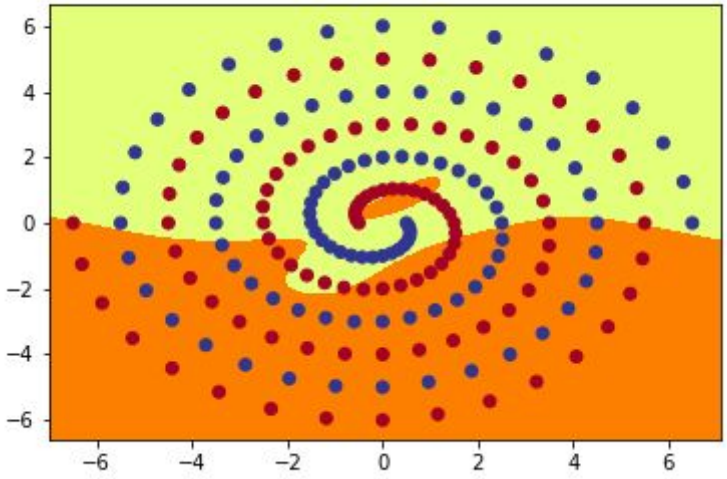

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

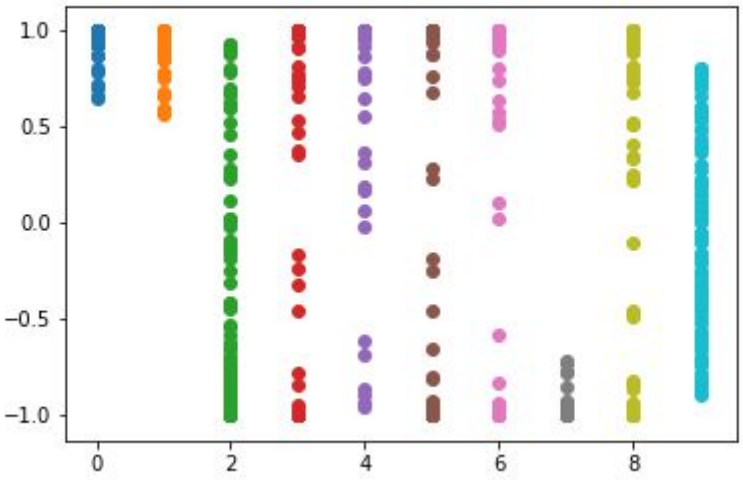

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

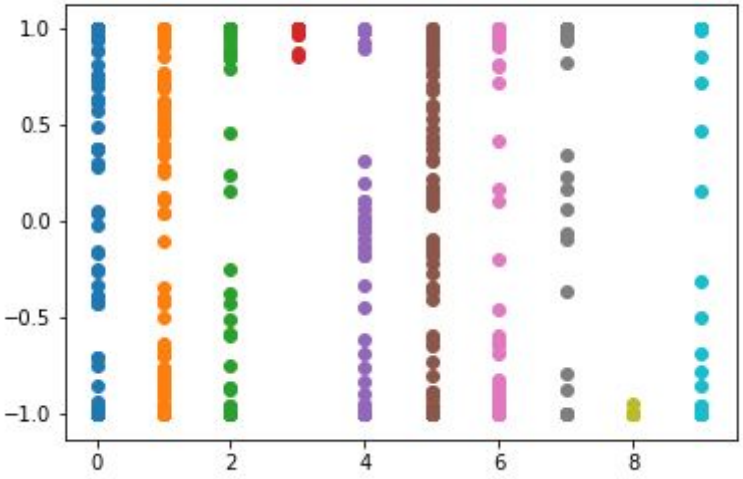

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6



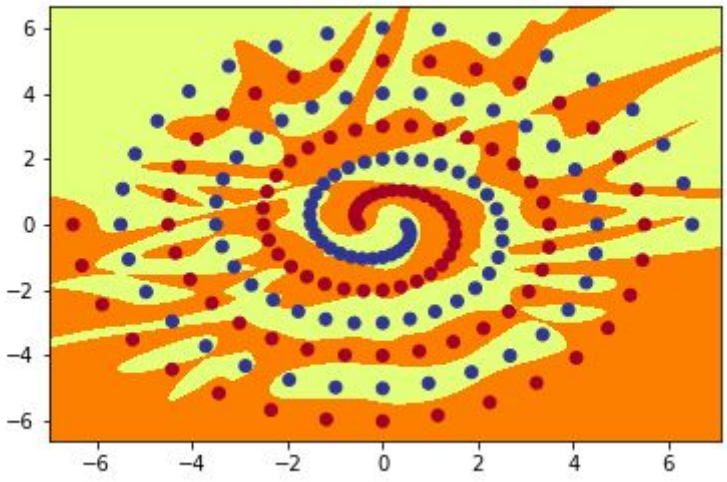


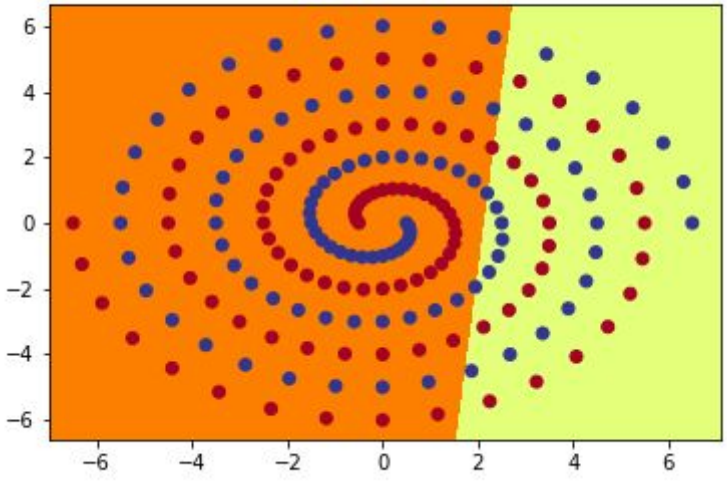
Raw_act layer 1 and 2 (used in the last question):

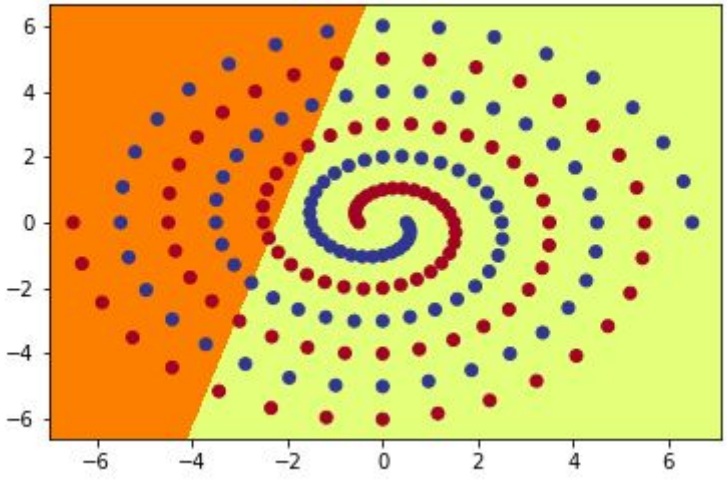


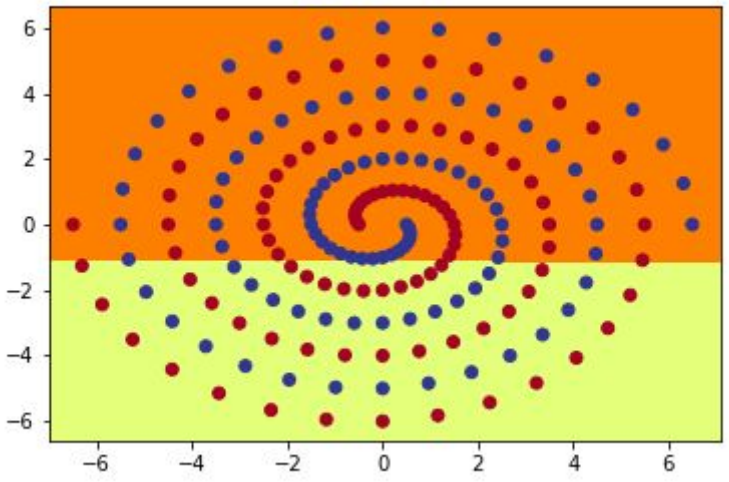


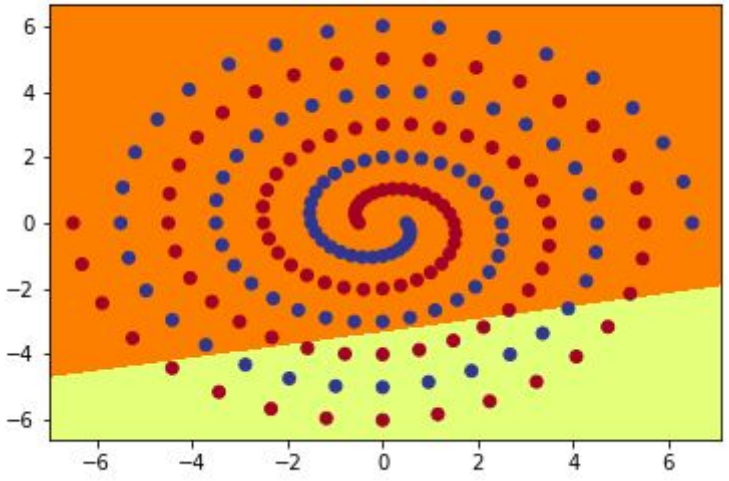
Short net layer 1:

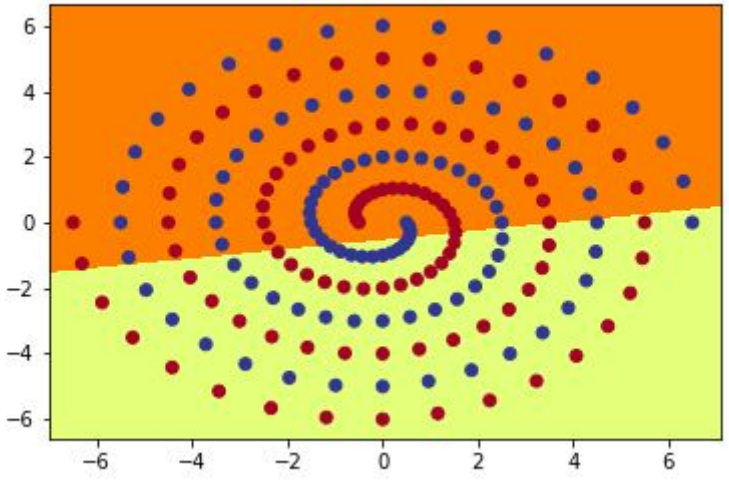

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

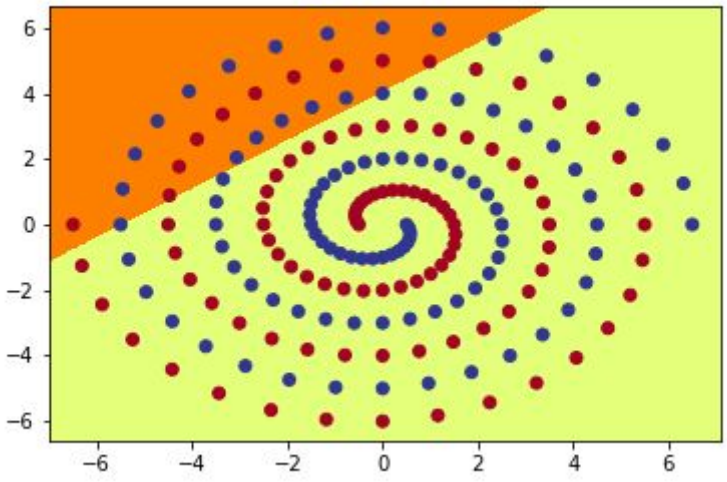

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

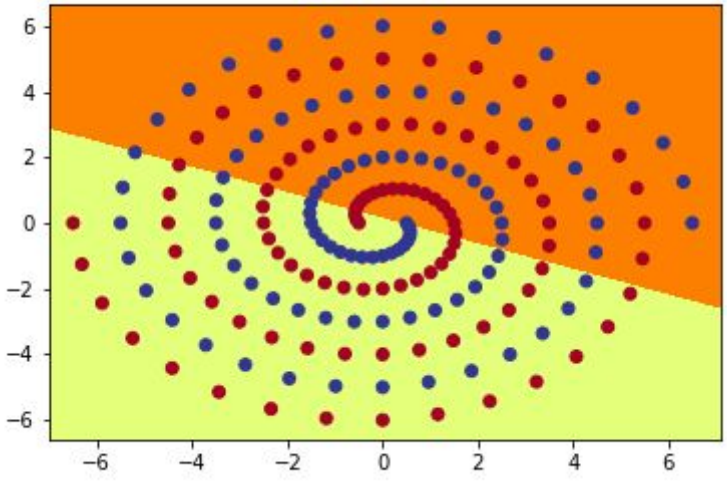

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

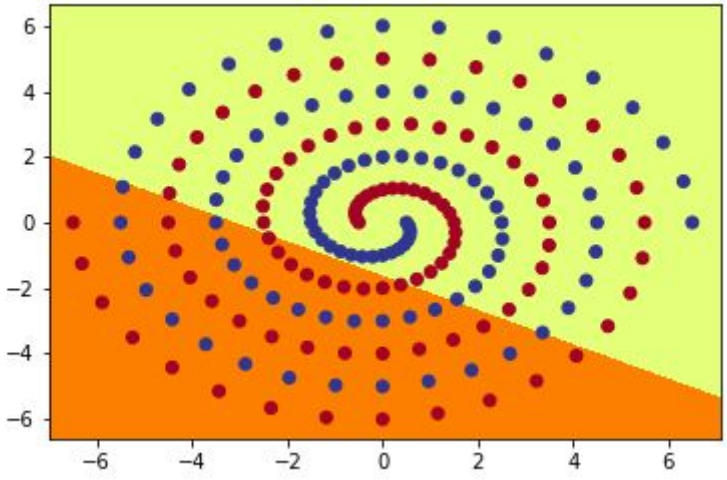

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

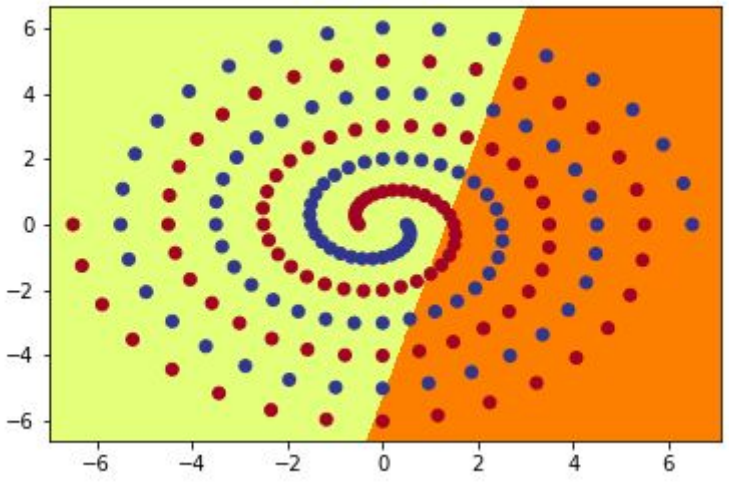

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

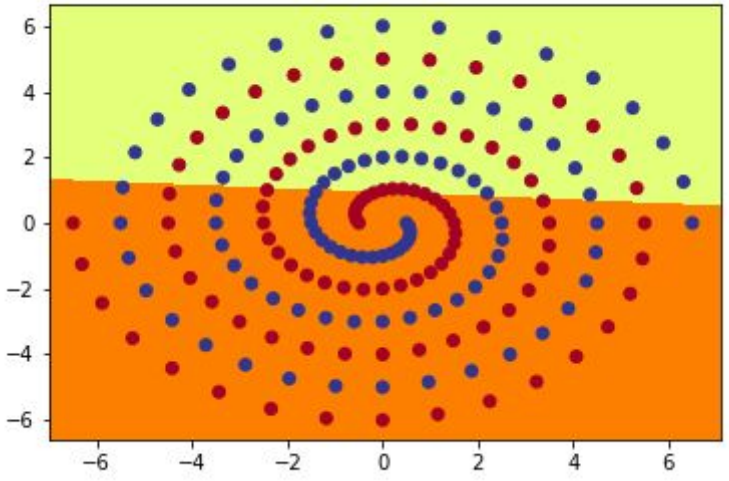

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

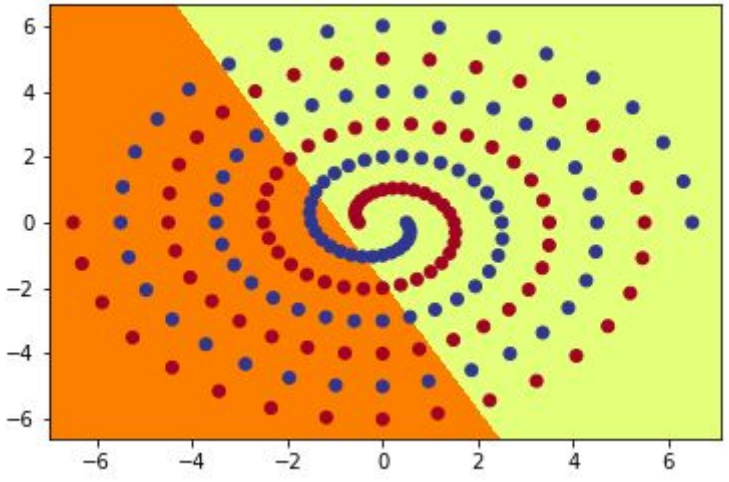

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

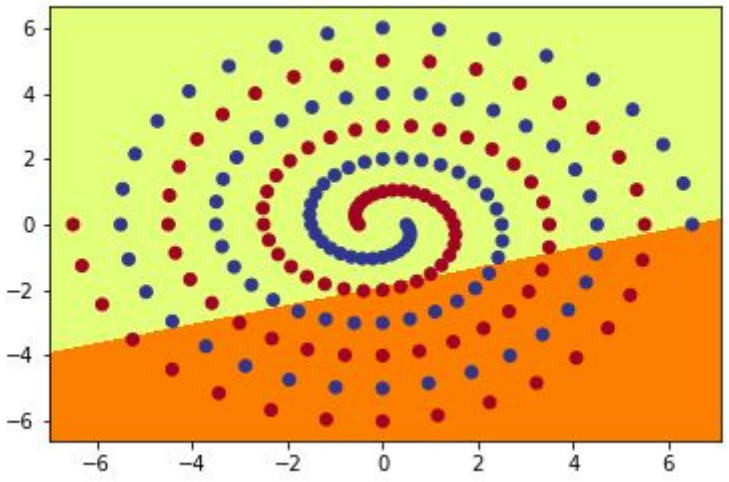

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

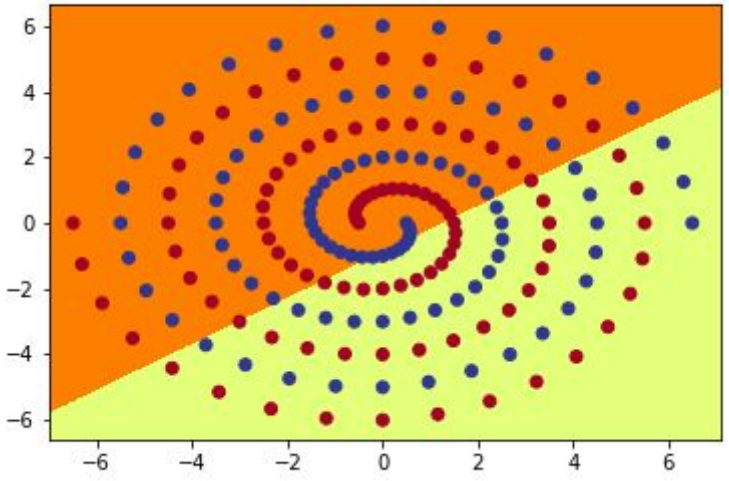

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

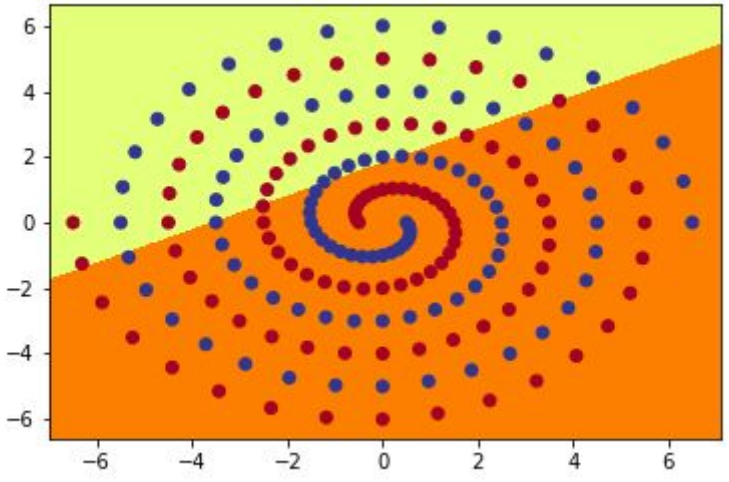

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

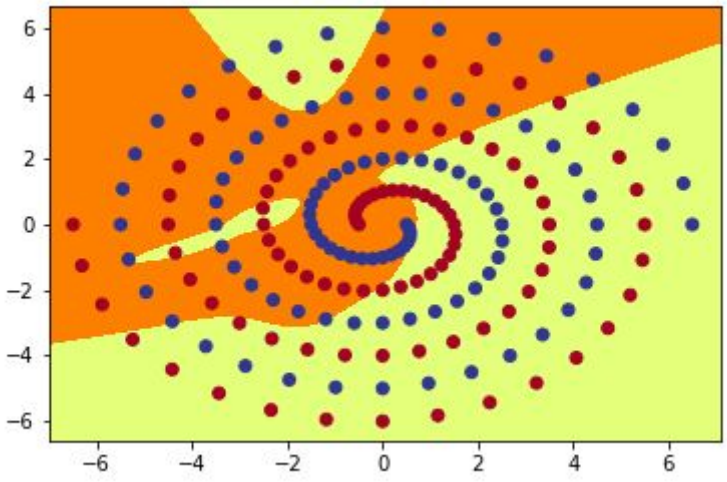

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

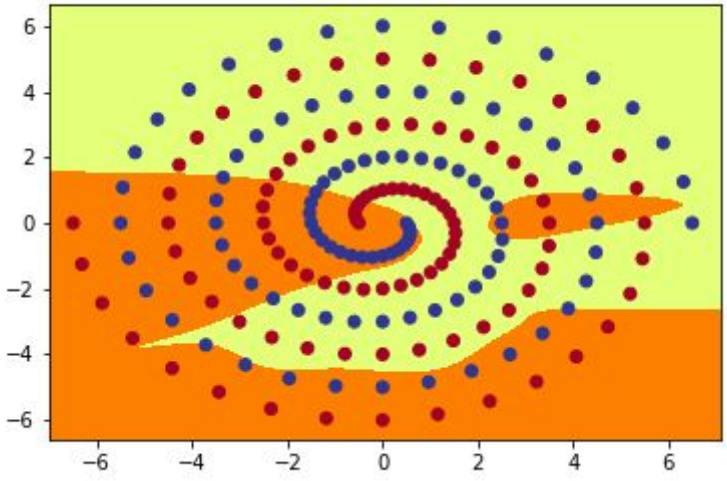


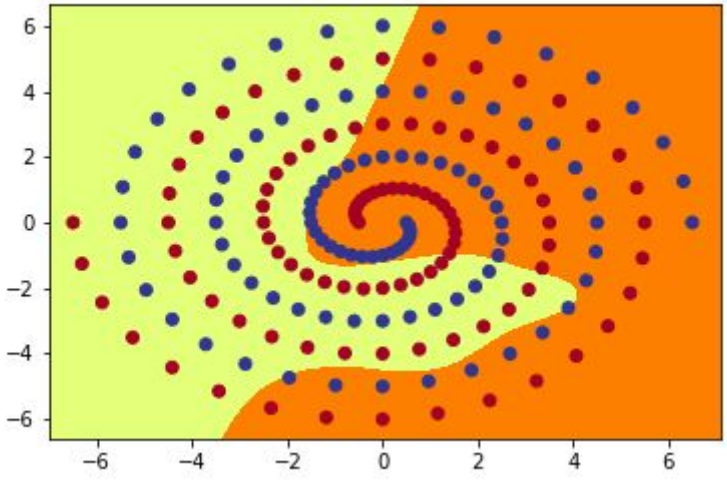


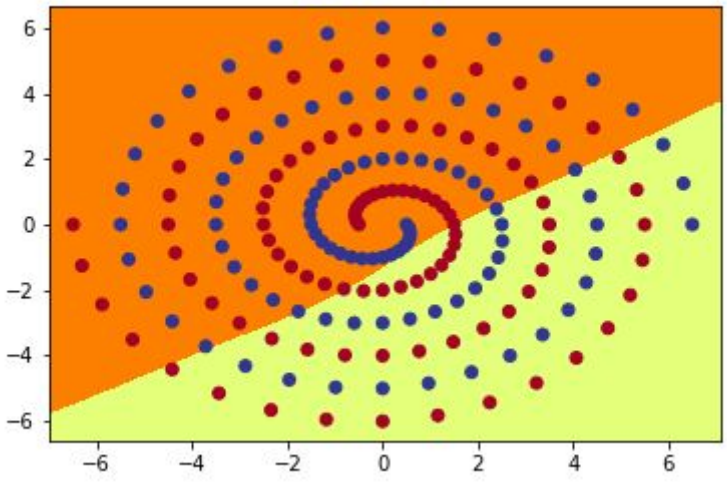


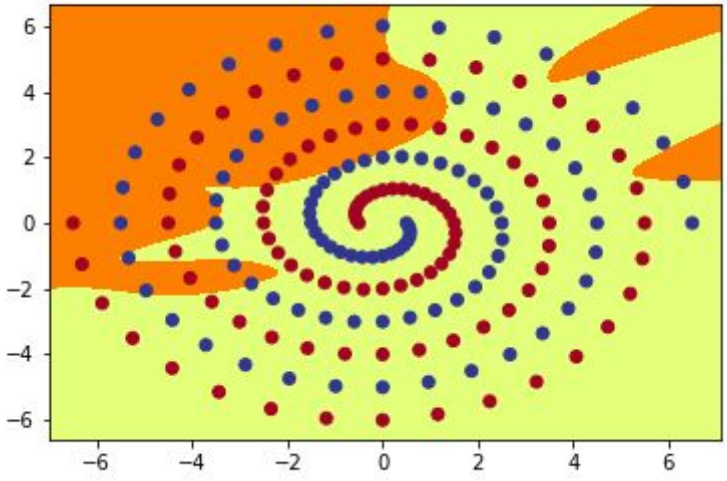
Short Net layer 2:

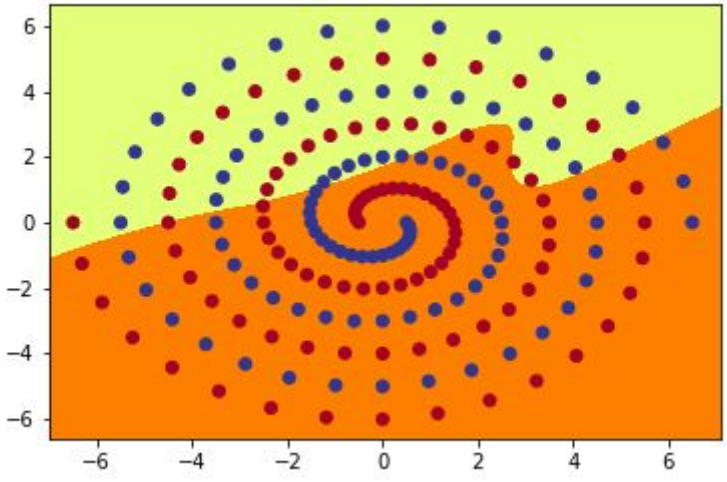

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

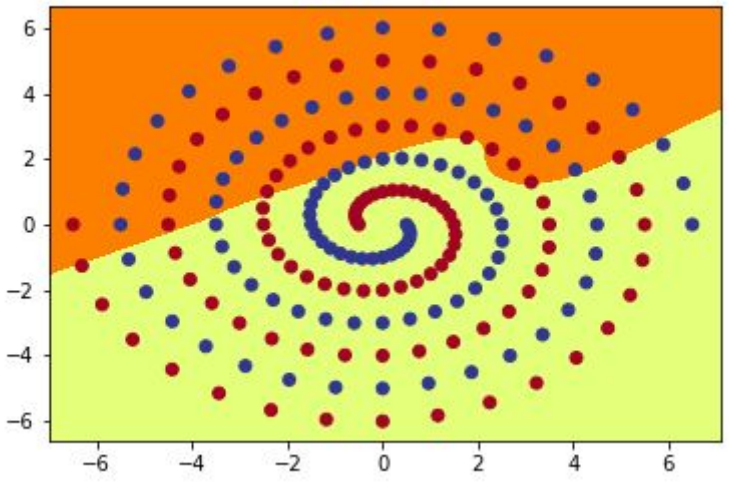

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

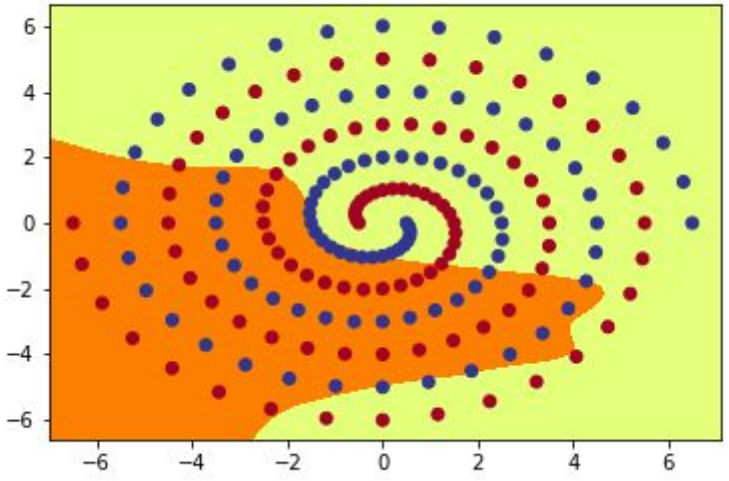

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

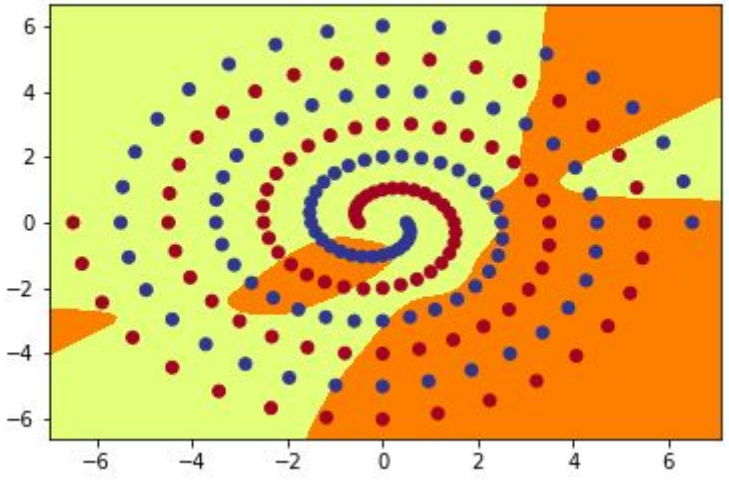

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

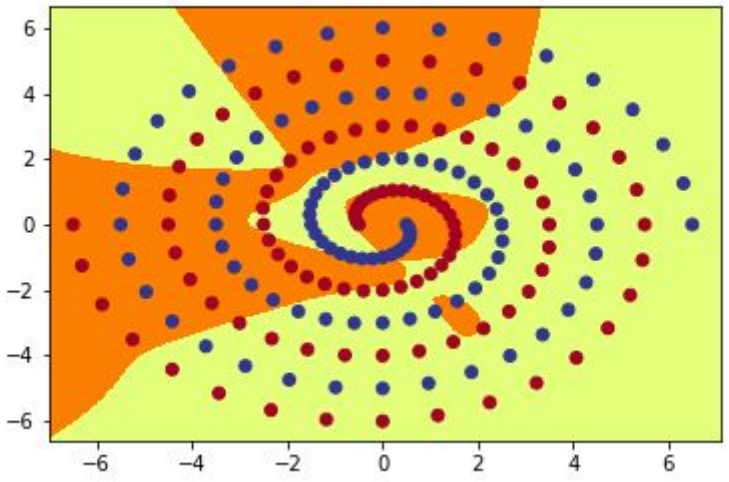

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

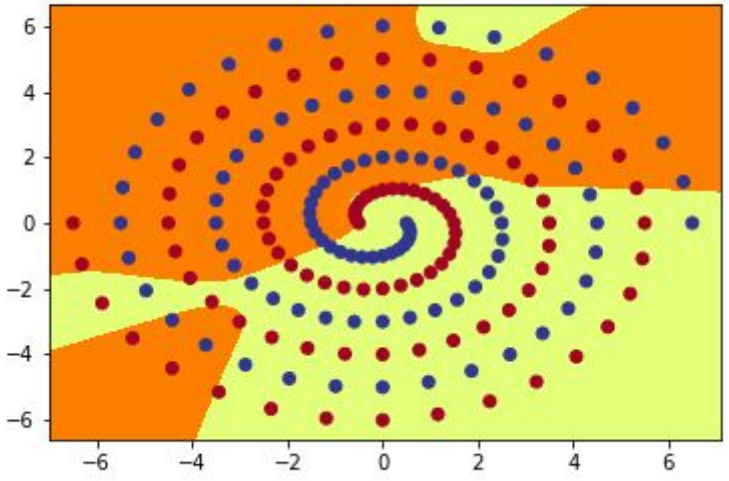

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

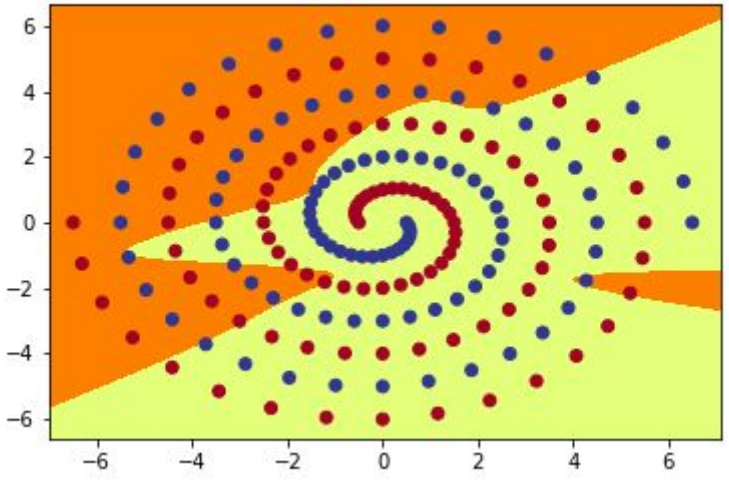

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

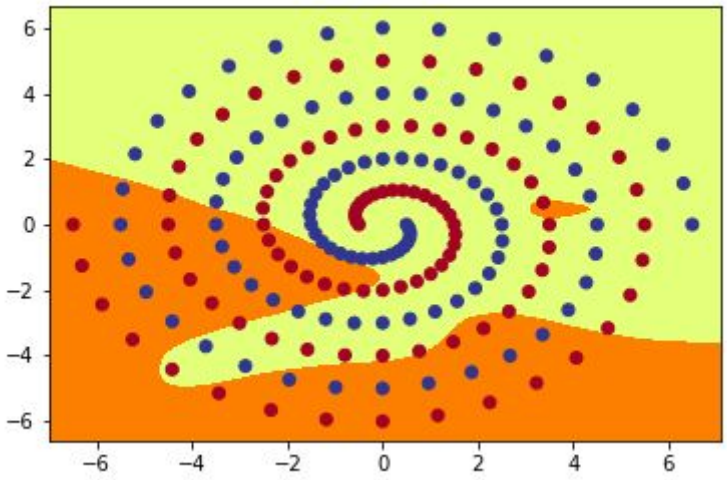

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

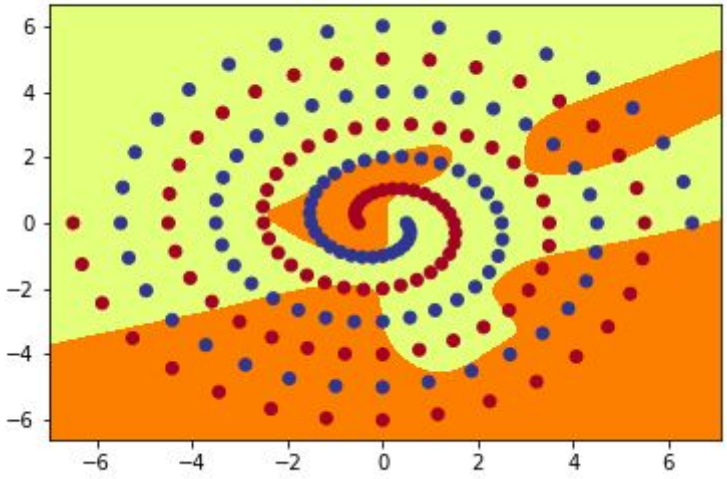

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6


6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

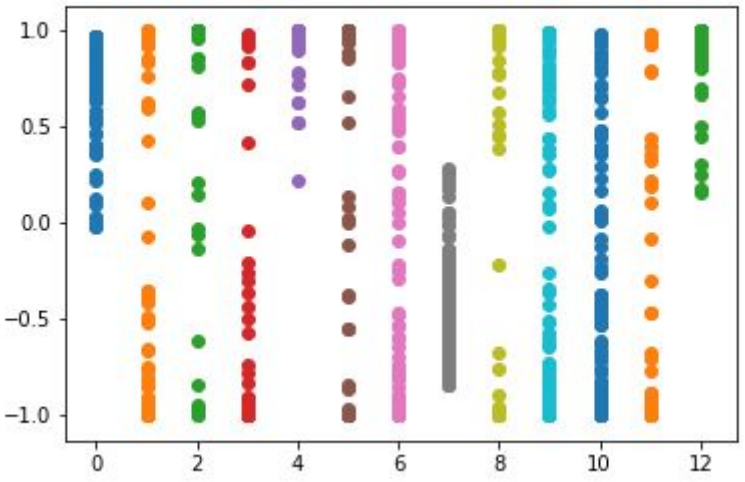

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

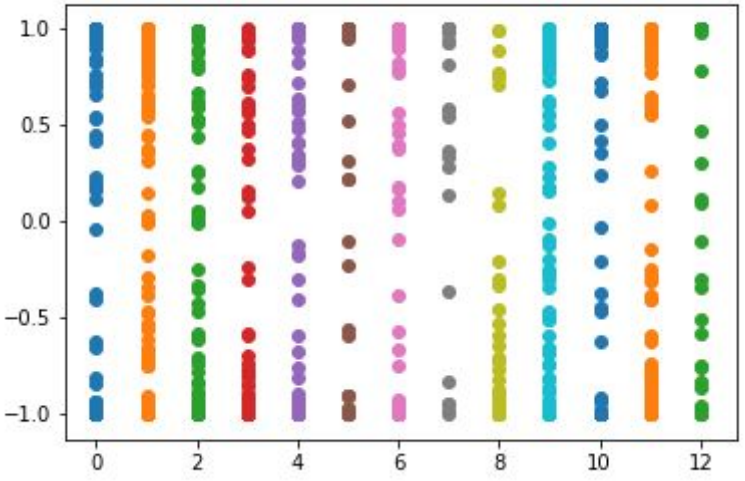

6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6





Short_act layer 1 and 2 (used in the last question):


1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10
12


1.0
0.5
0.0
−0.5
−1.0
0
2
4
6
8
10
12