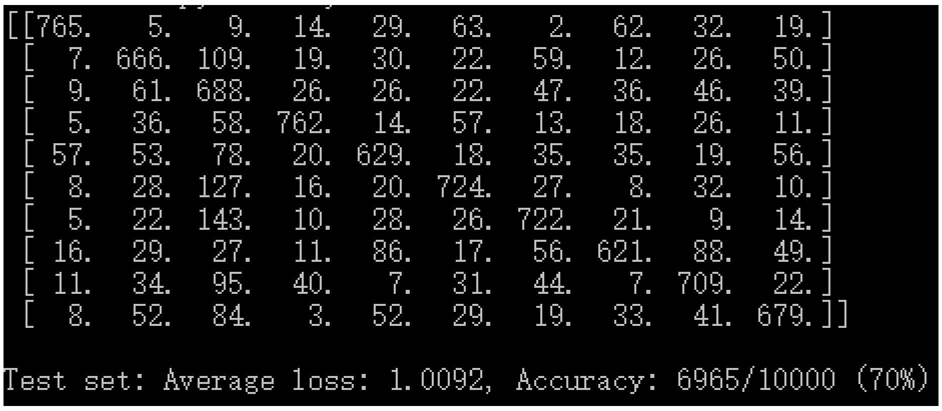
Part 1

Step 1

With a single linear function, the final accuracy achieved is 70% and the confusion matrix is shown below.

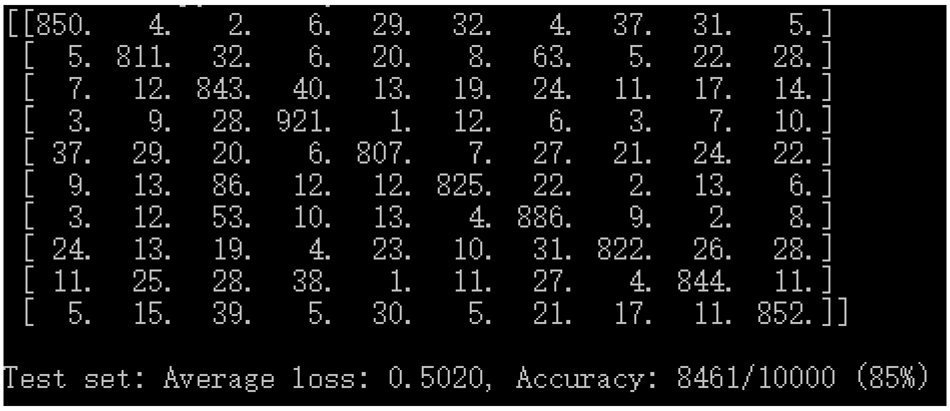


```
[[765.   5.   9.  14.  29.  63.   2.  62.  32.  19.]
 [  7. 666. 109.  19.  30.  22.  59.  12.  26.  50.]
 [  9.  61. 688.  26.  26.  22.  47.  36.  46.  39.]
 [  5.  36.  58. 762.  14.  57.  13.  18.  26.  11.]
 [ 57.  53.  78.  20. 629.  18.  35.  35.  19.  56.]
 [  8.  28. 127.  16.  20. 724.  27.   8.  32.  10.]
 [  5.  22. 143.  10.  28.  26. 722.  21.   9.  14.]
 [ 16.  29.  27.  11.  86.  17.  56. 621.  88.  49.]
 [ 11.  34.  95.  40.   7.  31.  44.   7. 709.  22.]
 [  8.  52.  84.   3.  52.  29.  19.  33.  41. 679.]]

Test set: Average loss: 1.0092, Accuracy: 6965/10000 (70%)
```

Step 2

Different values (multiples of 10) for the No. of hidden nodes were tried. (See below table)
The number of hidden nodes of 150 was used in the code and the final accuracy is between 84% to 85% after several tests. The confusion matrix is shown below.

No. of hidden nodes	Accuracy	No. of hidden nodes	Accuracy
10	68%	20	74%
40	81%	50	81%
60	82%	70	82%
80	83%	90	84%
100	84%	110	83%
120	84%	150	85%
200	85%	300	85%



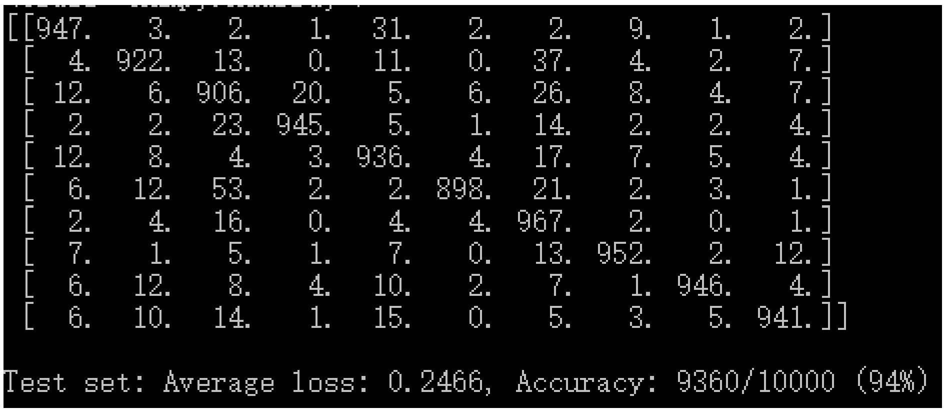
```
[[850.   4.   2.   6.  29.  32.   4.  37.  31.   5.]
 [  5. 811.  32.   6.  20.   8.  63.   5.  22.  28.]
 [  7.  12. 843.  40.  13.  19.  24.  11.  17.  14.]
 [  3.   9.  28. 921.   1.  12.   6.   3.   7.  10.]
 [ 37.  29.  20.   6. 807.   7.  27.  21.  24.  22.]
 [  9.  13.  86.  12.  12. 825.  22.   2.  13.   6.]
 [  3.  12.  53.  10.  13.   4. 886.   9.   2.   8.]
 [ 24.  13.  19.   4.  23.  10.  31. 822.  26.  28.]
 [ 11.  25.  28.  38.   1.  11.  27.   4. 844.  11.]
 [  5.  15.  39.   5.  30.   5.  21.  17.  11. 852.]]

Test set: Average loss: 0.5020, Accuracy: 8461/10000 (85%)
```

Step 3

The structure and hyperparameters used are shown below table.

Operation	Output Shape
Input dimensions	torch.Size([1,1,28,28])
Convolution (5 x 5)	torch.Size([1,24,24,24])
Max pooling (2 x 2)	torch.Size([1,24,12,12])
Convolution (5 x 5)	torch.Size([1,48,8,8])
Max pooling (2 x 2)	torch.Size([1,48,4,4])
Flatten (reshape)	torch.Size([1,48*4*4])
Linear transformation	torch.Size([1,120])
Output Dimension	torch.Size([1,10])

The final accuracy is between 93% to 94% after several tests. The confusion matrix is shown below.



```
[[947.    3.    2.    1.   31.    2.    2.    9.    1.    2.]
 [  4.  922.   13.    0.   11.    0.   37.    4.    2.    7.]
 [ 12.    6.  906.   20.    5.    6.   26.    8.    4.    7.]
 [  2.    2.   23.  945.    5.    1.   14.    2.    2.    4.]
 [ 12.    8.    4.    3.  936.    4.   17.    7.    5.    4.]
 [  6.   12.   53.    2.    2.  898.   21.    2.    3.    1.]
 [  2.    4.   16.    0.    4.    4.  967.    2.    0.    1.]
 [  7.    1.    5.    1.    7.    0.   13.  952.    2.   12.]
 [  6.   12.    8.    4.   10.    2.    7.    1.  946.    4.]
 [  6.   10.   14.    1.   15.    0.    5.    3.    5.  941.]]

Test set: Average loss: 0.2466, Accuracy: 9360/10000 (94%)
```

Step 4

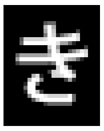
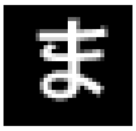
1. After a comparison of the three model, the convolution network has the highest accuracy of 93%. The fully connected 2-layer network has the second highest accuracy of 84% and the single linear function has the lowest accuracy of 70%.

2. Summary of confusion matrix of all the models. More common wrong predictions are included below.

Target	Predicted-NetLin	Predicted-NetFull	Predicted-NetConv
0 = “o”	4	4	
1 = “ki”	2, 3, 4, 8, 9		
2 = “su”	0, 1, 3, 4, 5, 6, 8, 9	1, 5, 6, 9	5
3 = “tsu”	8	2, 8	
4 = “na”	1, 7, 9	9	0
5 = “ ha“	0, 3, 8	0	
6 = “ma”	1, 7	1, 7	1
7 = “ya”	0, 2, 4, 9	0	
8 = “re”	0, 2, 5, 7, 9	0	
9 = “wo”	1, 2, 4, 7		

We can tell that NetLin has significantly more wrong predictions than NetFull and NetConv while NetConv achieves few wrong predictions.

Among all three models, character 2 = “su” is most likely to be mistaken for characters 5 = “ha” and character 6 = “ma” is most likely to be mistaken for the character 1 = “ki”.
For example, between 6 = “ma” and 1 = “ki”, they share certain common features, that’s why they can be misclassified by the models

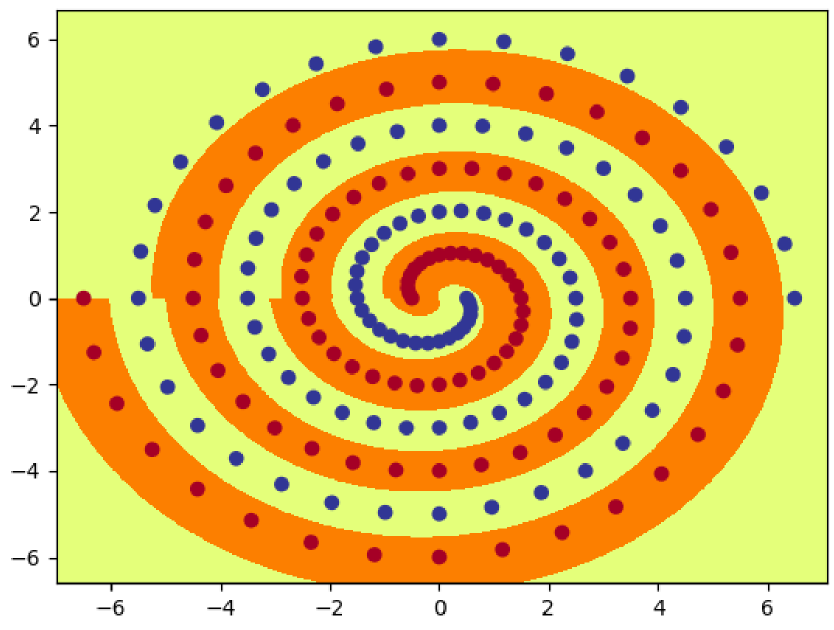

6

1

3. The number of filters used in the convolution network are 24 for the first convolution layer and 48 for the second convolution layer which gives an accuracy between 93% to 94%. Experiment was tried with 36 for the first convolution layer and 72 for the second convolution layer. The increased number of filters achieves an increased accuracy between 94% to 95%, however it also significantly increased the run time with the local machine. Using Max pooling also reduces computational cost while extract the most important features from the image.

Part 2

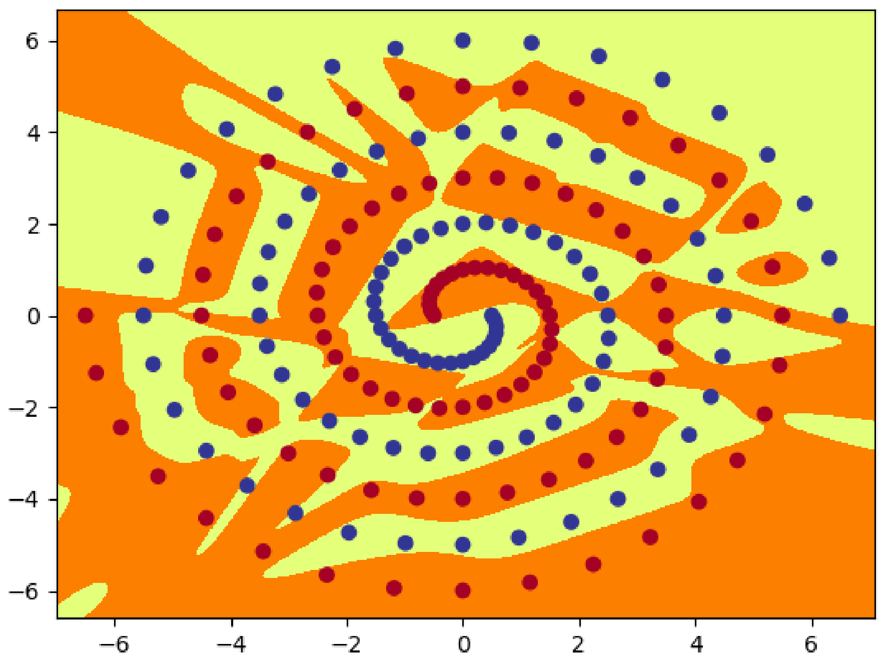
Step2

Minimum No. of hidden nodes is 7 to train data within 20000 epochs, on almost all runs(80%). polar_out.png is shown below.



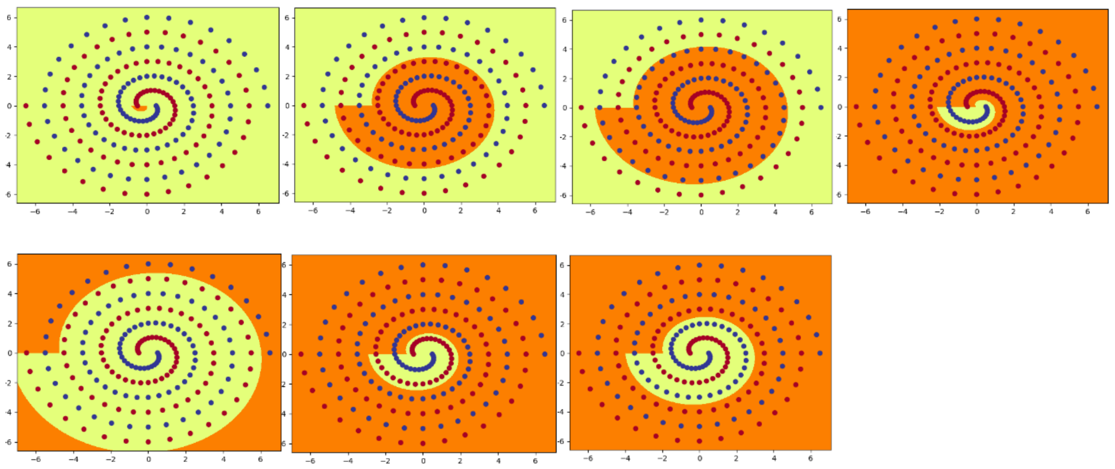
Step 4

No. of hidden nodes used is 16 and initial weight used is 0.25. raw_out.png is shown below.

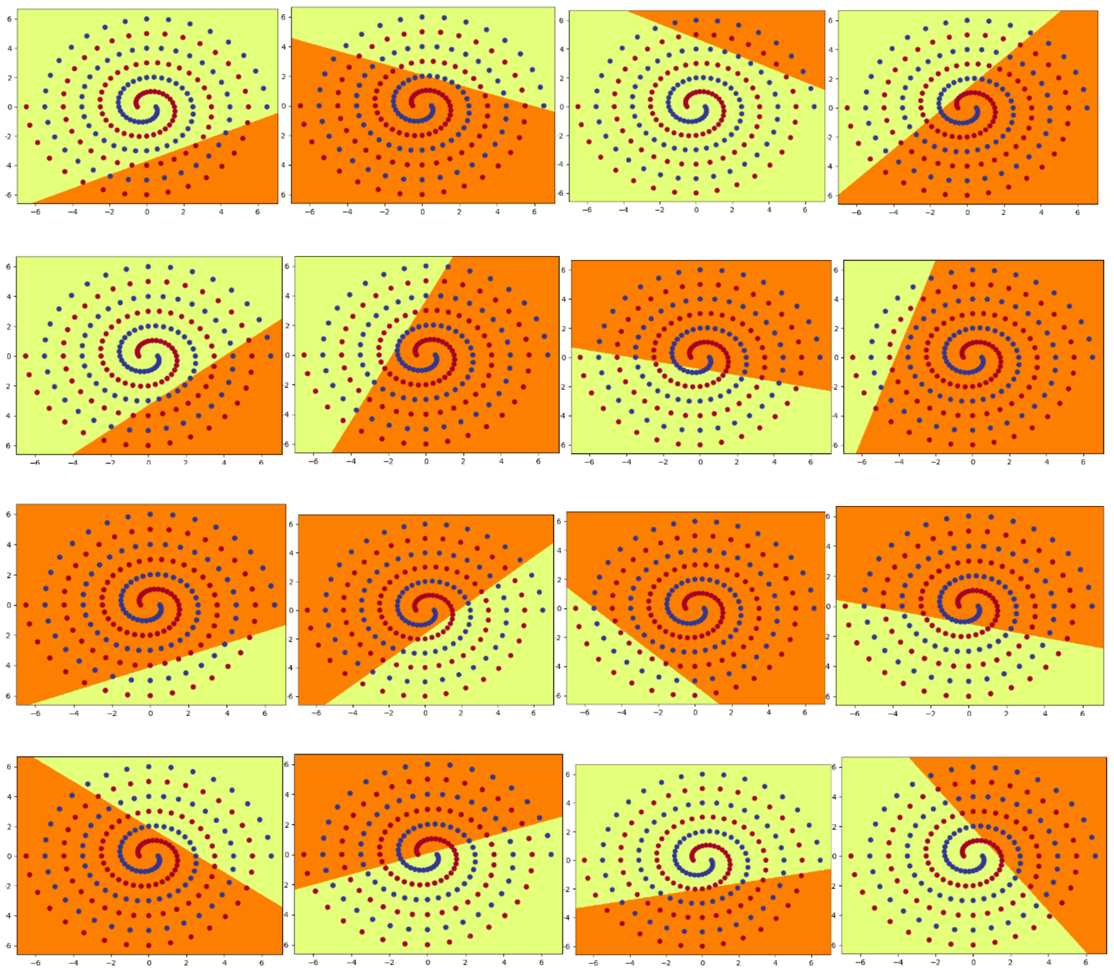


Step 5

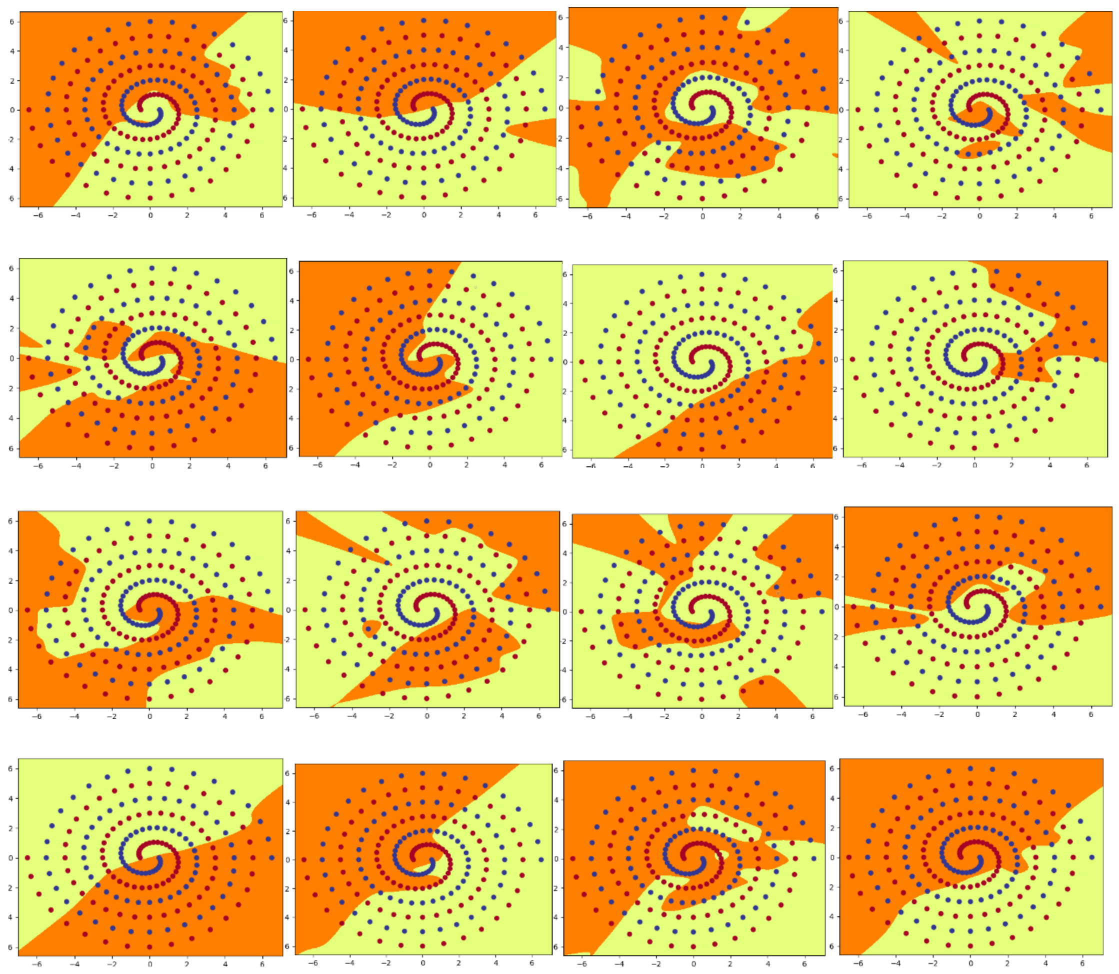
PolarNet plots



RawNet plots for first hidden layer



RawNet plots for second hidden layer



Step 6

1. In PolarNet, the input is converted to polar-coordinates, and it uses only one hidden layer. On the other hand, RawNet uses raw inputs, and it has 2 hidden layers. As a result, fist layer of PolarNet has unique values that generates spiral shaped layers while RawNet generates linearly separated layers.

2. Each layer in our model applies a linear transformation function followed by a squashing nonlinearity function (tang() for hidden layers and sigmoid() for output layer) to previous layer. The hidden layers transform the inputs into something that the output layer can use. The output layer then transforms it into proper scale.

3 In general, weights in a neural network are to set them to be close to zero but not being too small. When initial weight size is set to 0.0001, the neurons in the earlier layers learn much more slowly than neuron in later layers. Gradient gets smaller and smaller and leads to minor weight updates where “vanishing gradient” occurs. When initial weight size is set to 2, the gradient gets much larger in the earlier layers, which gives extremely high weight updates and “exploding gradient” occurs. Both of ““vanishing gradient” and “exploding gradient” makes our network difficult to converge. I ended up with using No. of hidden nodes used =16 and initial weight used = 0.25, which achieve reasonable result.

4. After changing batch size 97 to 194, there is a positive impact on the execution of the program. It gave better performance with respect to time and number of epochs required to train the program.

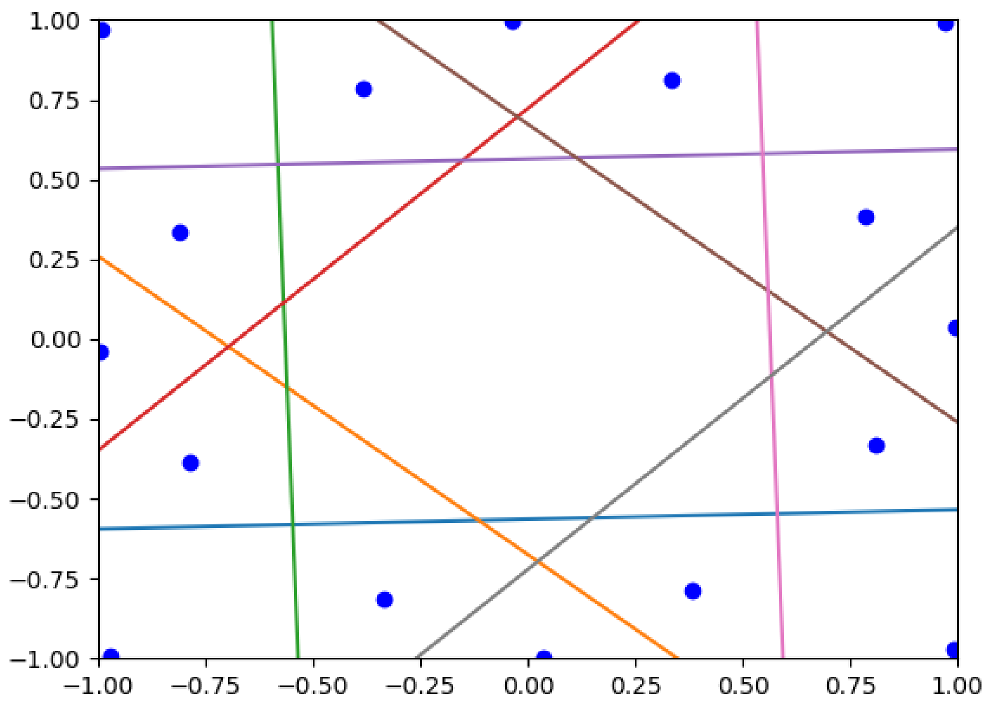
When using SGD instead of Adam, the program was taking longer time to converge with batch size 194, 10 hidden nodes and default learning rate.

After changing the function tanh() to Relu(), the model computes faster and there is no issue of Vanish Gradient problem when Relu() is used.

After adding more layers, it did not have much effect on the output or learning task.

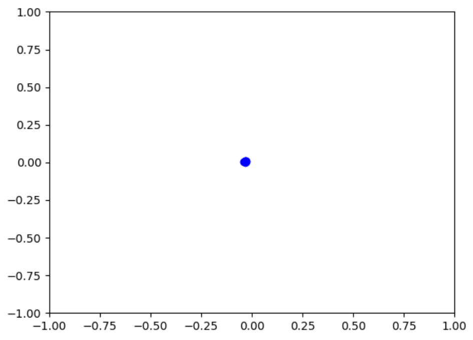
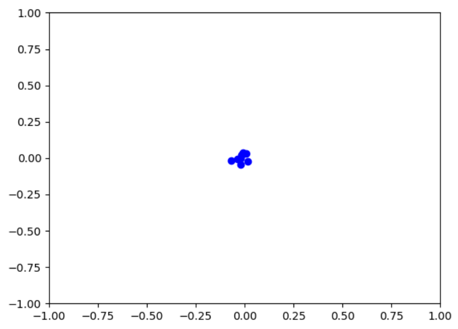
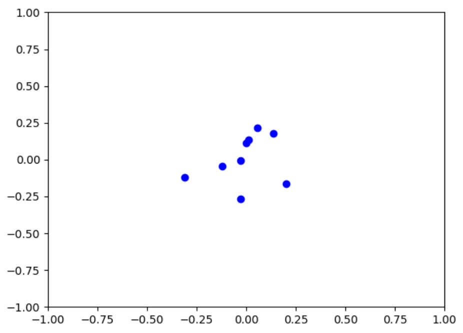
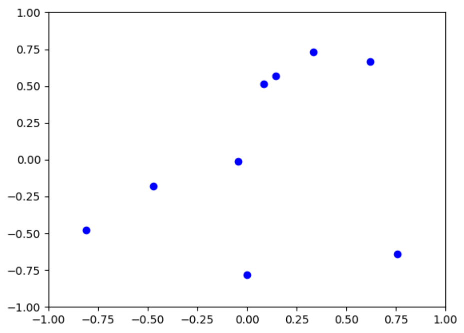
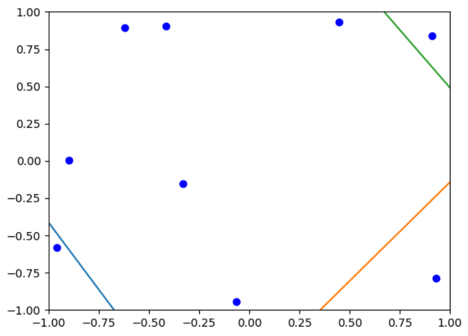
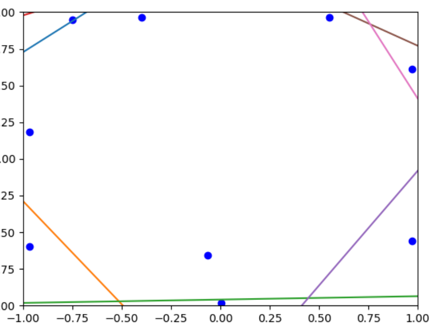
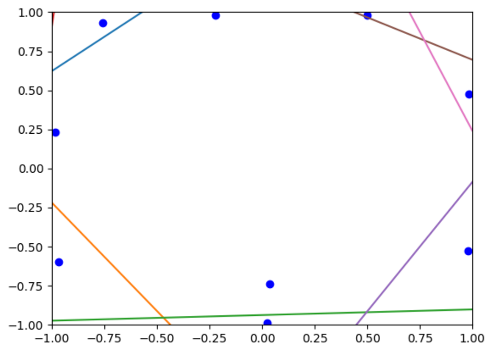
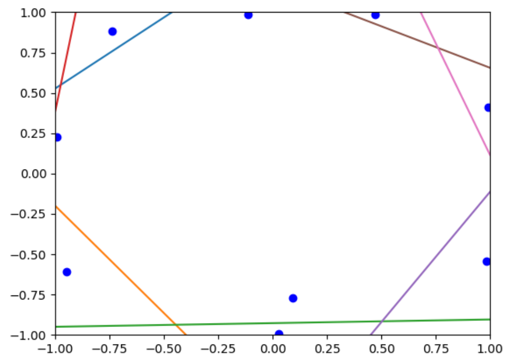
Part 3

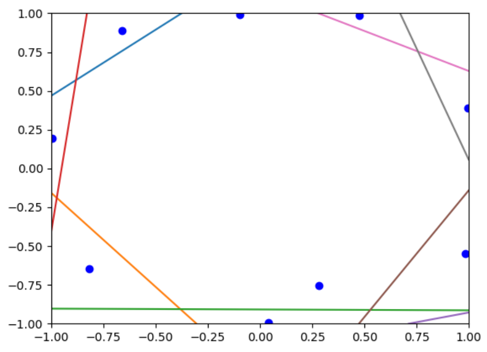
Step 1

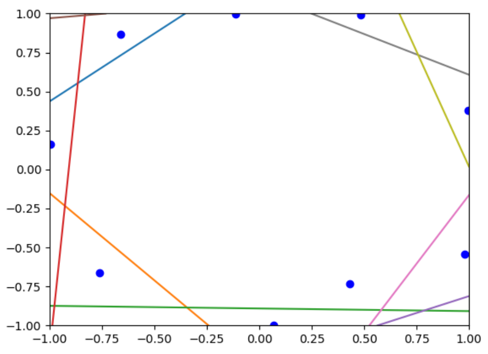


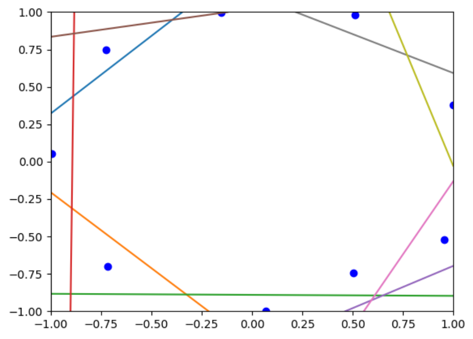
Step 2

Image 1 to 11 (epoch 50 to 3000)

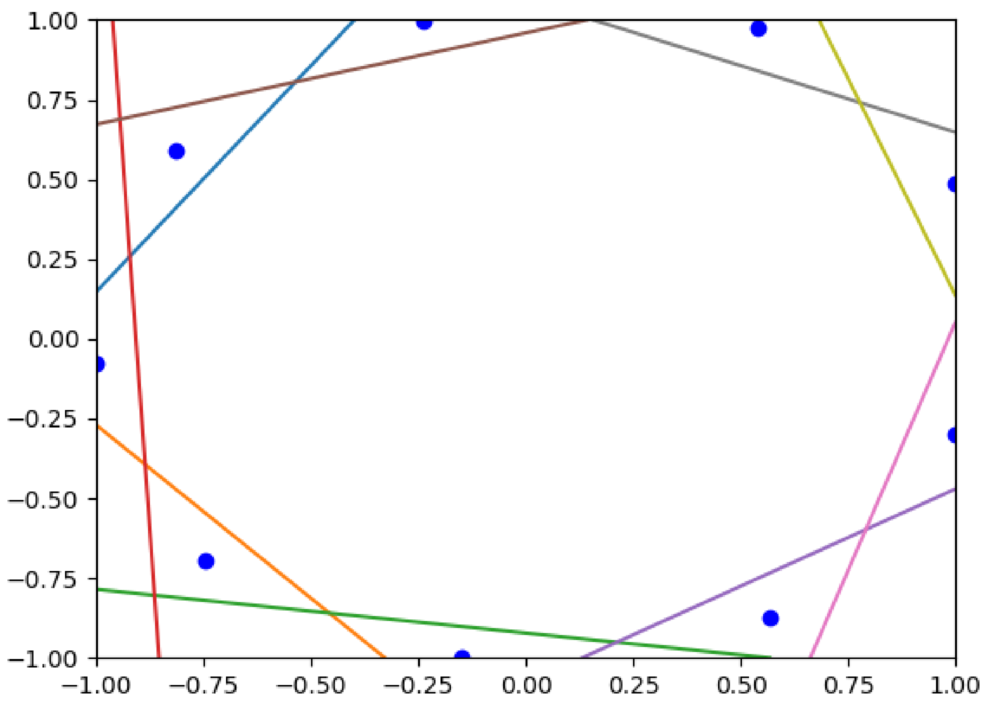











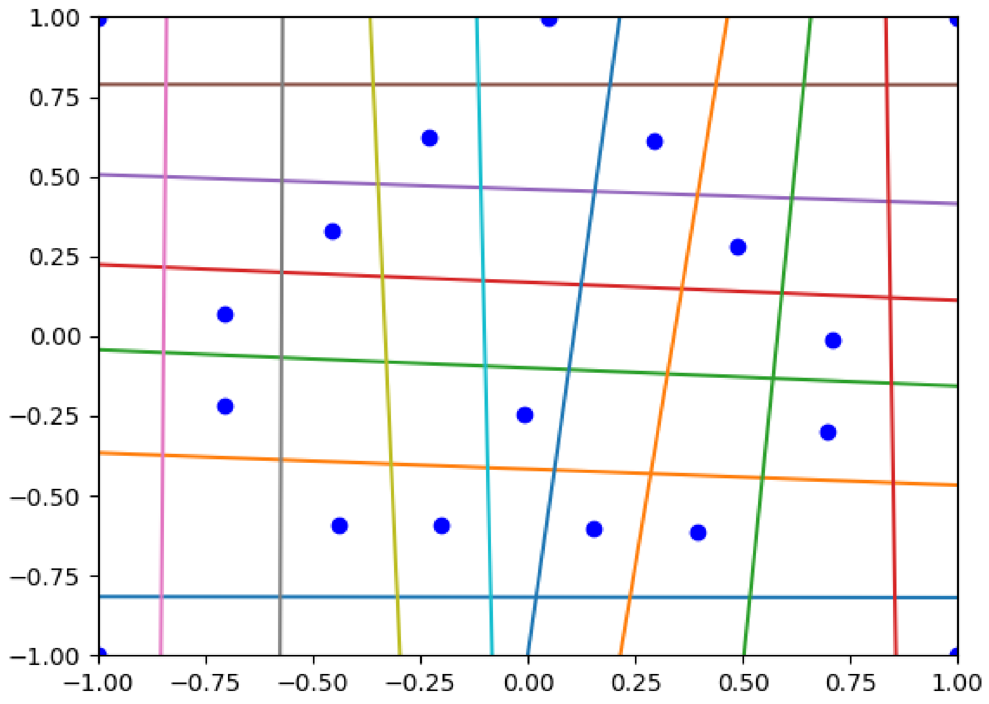




Final image



Step 3



Step 4

Image with tensor target1 (Letter “HD”)

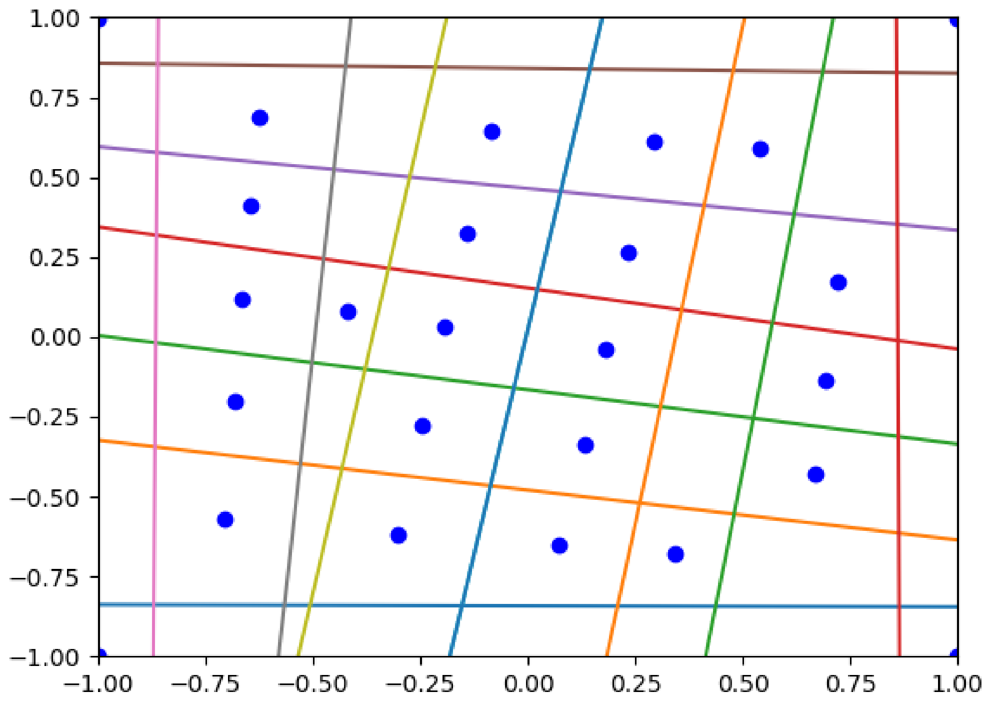
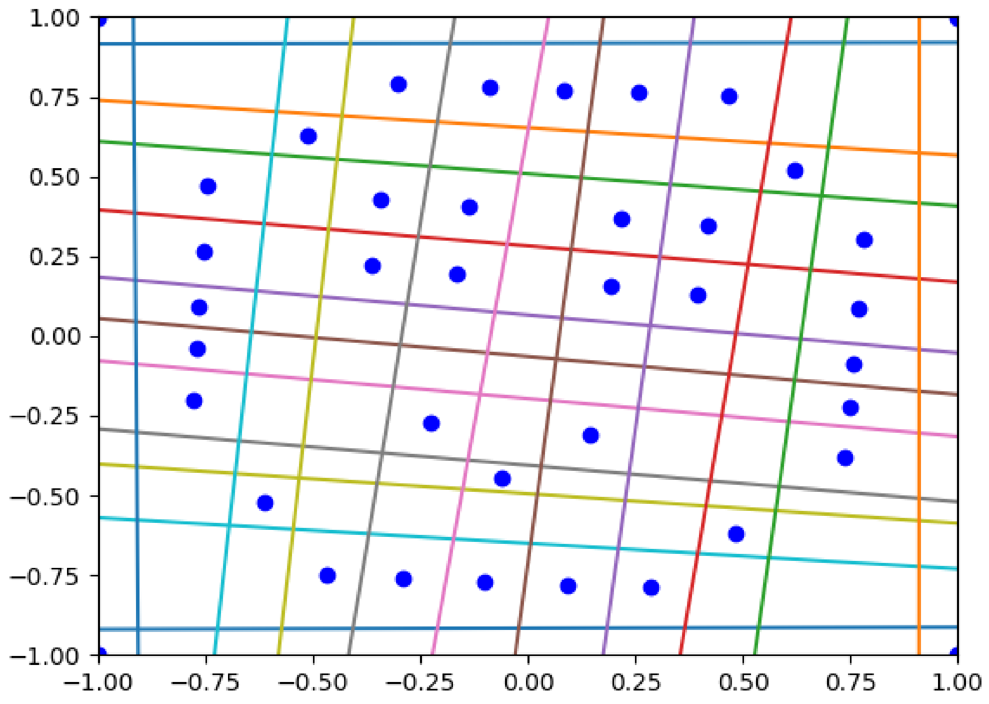


Image with tensor target2(smiley face)


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