Dheeraj Viswanadham
(z5204820)

COMP9444 Homework 1 Solutions

Part 1:

1) NetLin Model image of confusion matrix and final accuracy (70%):

10th epoch:

```
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.671114
<class 'numpy.ndarray'>
[[768.   5.   7.  14.  31.  62.   2.  63.  29.  19.]
 [  7. 671. 106.  19.  28.  23.  56.  13.  25.  52.]
 [  8.  61. 693.  27.  25.  20.  46.  37.  46.  37.]
 [  5.  37.  58. 760.  14.  56.  13.  18.  26.  13.]
 [ 62.  51.  81.  22. 621.  17.  33.  37.  21.  55.]
 [  7.  28. 125.  16.  19. 724.  29.   7.  34.  11.]
 [  5.  22. 143.  11.  27.  25. 722.  21.  10.  14.]
 [ 17.  29.  26.  12.  87.  19.  52. 623.  88.  47.]
 [ 10.  40.  93.  41.   7.  28.  45.   6. 707.  23.]
 [  8.  51.  87.   4.  54.  32.  18.  32.  40. 674.]]

Test set: Average loss: 1.0089, Accuracy: 6963/10000 (70%)
```

2) NetFull Model image of confusion matrix and final accuracy (85%):

I experimented with different numbers of hidden nodes and found that when I tried 100 hidden nodes, it yielded an 84% final accuracy and when I tried 190 - 200 hidden nodes I got an 84 - 85% final accuracy.

10th epoch:

With **190** hidden nodes: final accuracy = 84%

```
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.274402
<class 'numpy.ndarray'>
[[851.   5.   1.   6.  32.  30.   3.  40.  28.   4.]
 [  5. 825.  34.   2.  16.   9.  61.   7.  15.  26.]
 [  8.  14. 836.  41.  13.  19.  24.  11.  21.  13.]
 [  3.  11.  32. 912.   3.  14.   4.   2.   8.  11.]
 [ 37.  28.  21.   8. 816.   4.  32.  19.  19.  16.]
 [  8.  16.  80.  10.  16. 819.  21.   1.  20.   9.]
 [  3.  11.  58.   9.  16.   4. 884.   9.   2.   4.]
 [ 15.  15.  18.   4.  19.   8.  36. 834.  19.  32.]
 [  9.  28.  28.  57.   3.   8.  30.   4. 821.  12.]
 [  2.  15.  45.   3.  27.   6.  24.  17.  13. 848.]]

Test set: Average loss: 0.5097, Accuracy: 8446/10000 (84%)
```

Dheeraj Viswanadham
(z5204820)

With **200** hidden nodes: final accuracy = 85%

```
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.282699
<class 'numpy.ndarray'>
[[848.   3.   3.   7.  31.  33.   2.  38.  30.   5.]
 [  6. 817.  30.   5.  18.   8.  60.   5.  16.  35.]
 [  9.  12. 837.  39.  12.  24.  25.  11.  18.  13.]
 [  4.   5.  28. 922.   4.  12.   5.   2.   7.  11.]
 [ 32.  26.  17.   6. 828.  10.  32.  18.  16.  15.]
 [ 11.  13.  80.  11.  11. 828.  25.   2.  12.   7.]
 [  3.  10.  51.  10.  15.   4. 893.   6.   2.   6.]
 [ 23.  10.  18.   2.  21.  12.  29. 837.  17.  31.]
 [ 11.  30.  31.  48.   6.   9.  29.   4. 824.   8.]
 [  3.  19.  49.   7.  30.   4.  22.  13.   6. 847.]]

Test set: Average loss: 0.4994, Accuracy: 8481/10000 (85%)
```

3) Output value calculation of the 2D convolution layer is calculated as follows:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

NetConv Model image of confusion matrix and final accuracy (94%) with 16 output channels in first layer:

```
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.016332
<class 'numpy.ndarray'>
[[964.   4.   2.   0.  21.   1.   0.   5.   0.   3.]
 [  3. 939.   5.   0.  11.   1.  26.   3.   2.  10.]
 [  9.   9. 907.  24.   4.   9.  13.  13.   1.  11.]
 [  1.   1.  13. 970.   1.   4.   4.   3.   0.   3.]
 [ 21.  12.   2.   4. 940.   1.   5.   4.  10.   1.]
 [  6.   9.  36.   6.   6. 899.  15.   5.   5.  13.]
 [  4.   0.  11.   2.   5.   0. 974.   2.   0.   2.]
 [  3.   4.   1.   0.   6.   0.   7. 961.   2.  16.]
 [  4.  25.   7.   4.   8.   1.   8.   1. 937.   5.]
 [  8.   8.  15.   1.  12.   0.   3.   2.   5. 946.]]

Test set: Average loss: 0.2205, Accuracy: 9437/10000 (94%)
```

Dheeraj Viswanadham
(z5204820)

Final NetConv Model image of confusion matrix where we get a **consistent** accuracy of **95%** with 64 output channels in first layer:

```
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.007602
<class 'numpy.ndarray'>
[[966.   1.   0.   0.  21.   1.   0.   7.   1.   3.]
 [  3. 938.   3.   1.   7.   0.  31.   4.   3.  10.]
 [ 13.   9. 895.  27.   8.  12.  19.   7.   1.   9.]
 [  1.   0.  10. 971.   0.   5.   7.   2.   1.   3.]
 [ 21.  10.   2.   7. 940.   0.   7.   3.   8.   2.]
 [  5.  10.  28.   5.   4. 915.  20.   2.   3.   8.]
 [  4.   3.   5.   1.   4.   5. 977.   1.   0.   0.]
 [ 14.   1.   2.   0.   6.   0.   6. 957.   2.  12.]
 [  9.  11.   5.   0.   9.   2.   7.   2. 949.   6.]
 [  4.   4.   4.   0.   6.   1.   2.   3.   1. 975.]]

Test set: Average loss: 0.2143, Accuracy: 9483/10000 (95%)
```

4a) From the results shown above it is evident that the NetLin model is the least accurate with a final accuracy of 70%, the 2-layer fully connected NetFull model performs slightly better having a final accuracy of 85% whilst the NetConv model performs the best having a 95% final accuracy. This is because the convolutional network is better at capturing local information such as neighbouring pixels in an image as compared to normal linear networks and it also reduces the chance of overfitting due to having many-to-one mappings, which reduce the overall number of units required in the network (fewer parameters to learn).

b) To determine which characters will be mistaken for which, we will first need to analyse our final confusion matrix for each model above. To do so, we can create a simple table of our characters where each character is represented by a number i.e. “o” = 0, “ki” = 1, and so on (which is shown from left to right in the confusion matrices). This is the same order when going from top to bottom with 0 at the top left and 9 at bottom-left. Quite obviously, as shown in the confusion matrices - the 0^{th} character will have the highest accuracy for the 0^{th} character and so on i.e. 0 – 0, 1 – 1, 2 – 2, … , 9 – 9 characters will have the highest accuracy. This means that the **next highest accuracy** for a particular character would be **the character most likely to be mistaken**. We can put it into a table as shown below, where under each model we will write what character it will likely be mistaken for.

Character	NetLin Model	NetFull Model	NetConv Model
0 = o	7	7	4
1 = ki	2	6	6
2 = su	1	3	3
3 = tsu	**2**	**2**	**2**
4 = na	2	0	0
5 = ha	**2**	**2**	**2**
6 = ma	**2**	**2**	**2**/5 (equal)
7 = ya	8	9	0
8 = re	2	3	1
9 = wo	2	2	4

Dheeraj Viswanadham
(z5204820)

From the above, we can see that “su” (character 2) is most likely to be mistaken for characters 3, 5 and 6. This may be because the characters look similar to each other or the images of any untidy handwriting may cause the network to incorrectly identify the characters due to the visual structure of handwritten characters not being “clean” enough to be used as inputs, etc.

c) I experimented with different parameters for each of my models.

For example, for the NetLin model, I tried having different learning rates and momentum as below and got the following final accuracies after 10 epochs:

lr = 0.0001 = 58%	mom = 0.1 = 69%
lr = 0.001 = 67%	mom = 0.2 = 70%
lr = 0.01 = 70% (default)	mom = 0.5 = 70%
lr = 0.1 = 67%	mom = 1.0 = 61%
lr = 0.5 = 63%	mom = 2.0 = 10%
lr = 1.0 = 60%	

As seen for learning rate, if it is close to 0.01 then it will be fairly accurate however if it differs to far then the accuracy greatly decreases. For momentum, it decreases if momentum differs greatly from 0.2 – 0.5 default values.

For NetFull model, I tried different momentum values as below:

mom = 0.1 = 81%

mom = 0.2 = 82%

mom = 0.5 = 85%

mom = 0.7 = 87%

mom = 0.9 = 89%

mom = 0.95 = 88%

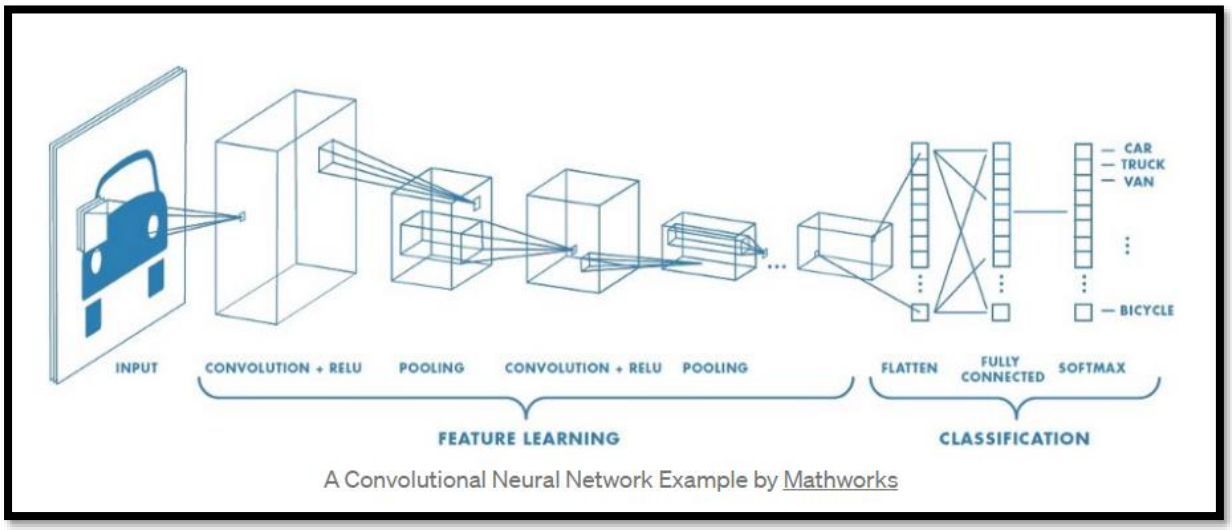
mom = 1.0 = 25%

mom = 2.0 = 10%

What I discovered was that as momentum went from 0.1 to 0.9, the final accuracy I was achieving was increasing with it peaking around 89% however when it went above 0.9 then the accuracy got significantly worse.

Dheeraj Viswanadham
(z5204820)

For the NetConv model, the structure of my network was similar to the below image where I slightly tweaked the max pooling layer:



https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d

Convolutional layer 1

- ➔ Relu

Convolutional layer 2

- ➔ Relu
- ➔ Max Pooling

Linear layer 1

- ➔ Relu

Linear layer 2

- ➔ And finally Log softmax to get my final output

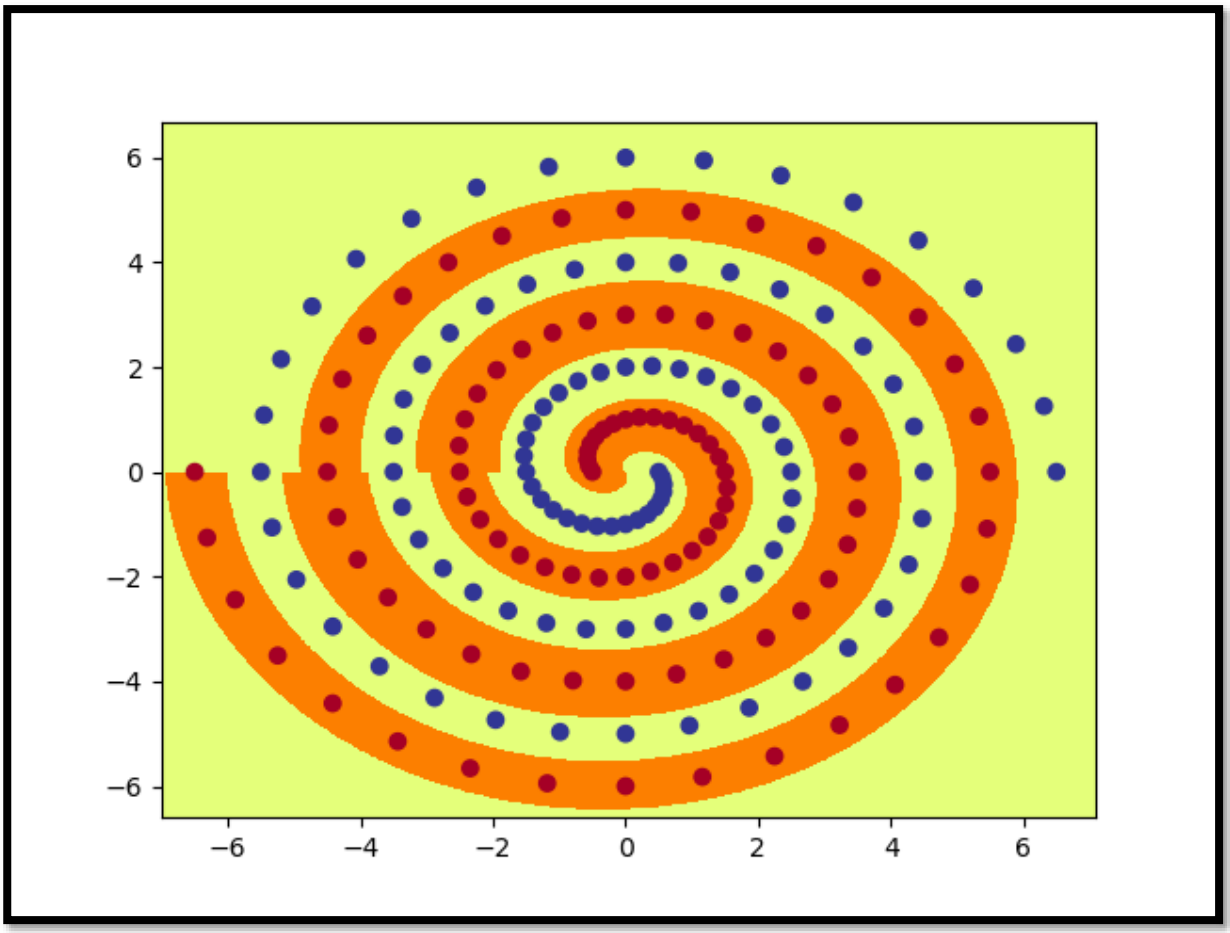
My values for the channels was 64 output channels in the first convolutional layer as I found that to be the optimal number during my experimentation as it mitigated any underfitting. By selecting my chosen kernel size and adding padding for my max pooling layer I found that my final accuracy had constantly increased to around the mid-90s.

Dheeraj Viswanadham
(z5204820)

Part 2:

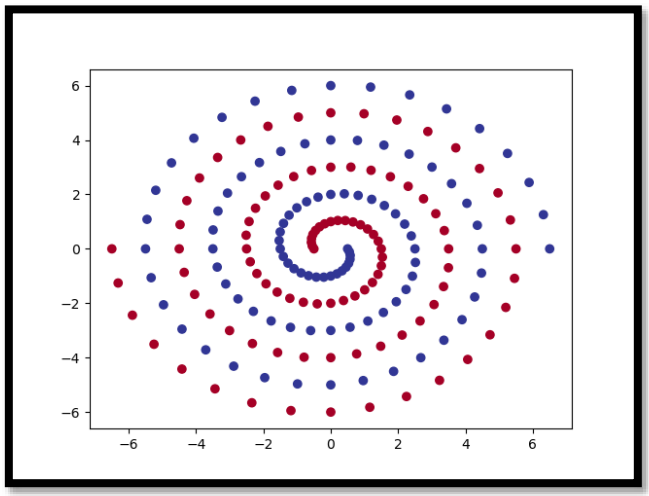
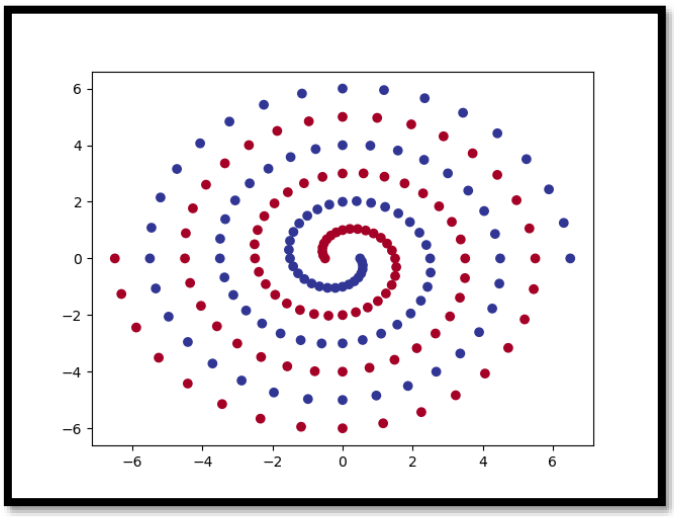
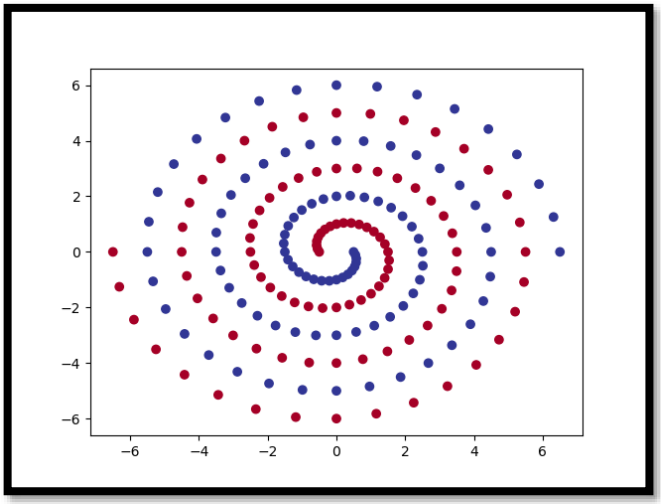
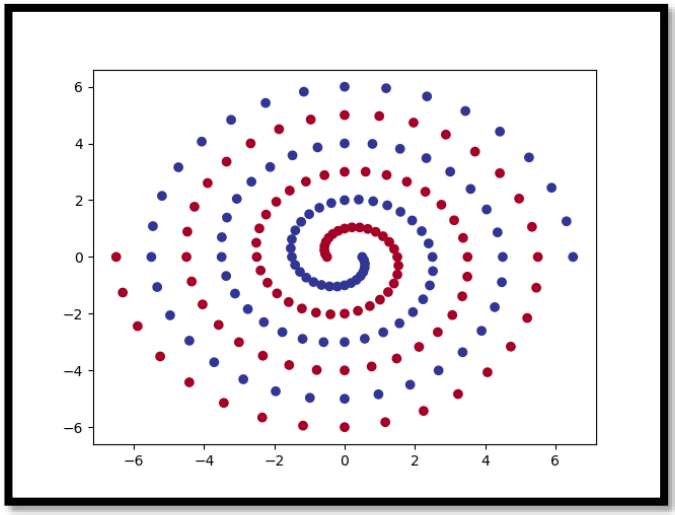
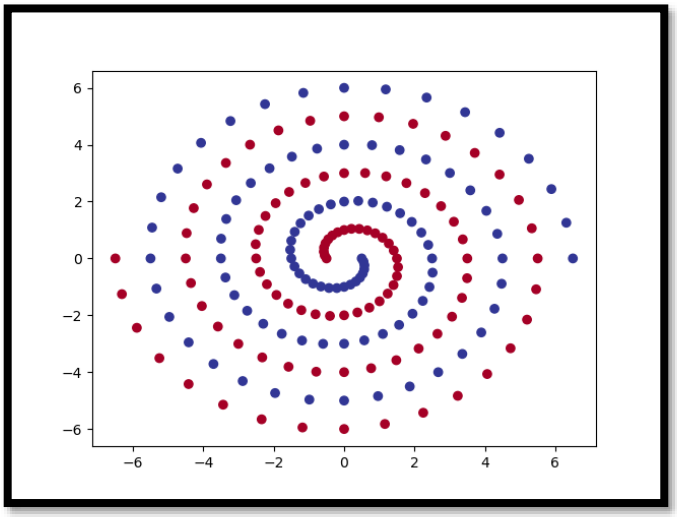
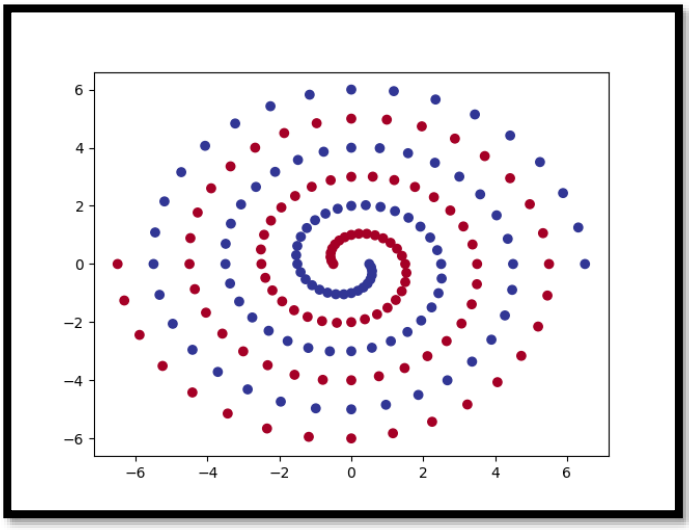
1) See spiral.py for PolarNet module code
2) By using the --hid parameter in the command line, I was able to tweak the number of hidden nodes required to correctly classify all the training data within 20,000 epochs on almost all my runs. I found that 7 nodes was the absolute minimum number of hidden nodes that would consistently and correctly classify the data. If I went lower and chose 6 hidden nodes as an example, I would sometimes get the data correctly classified within 20,000 epochs however most of the time it exceeded that amount, sometimes reaching up to 99,900 epochs at one point. Same thing happened if I chose 5 hidden nodes (accuracy < 100%) and reaching 99,900 epochs.

Graph output (polar_out.png):

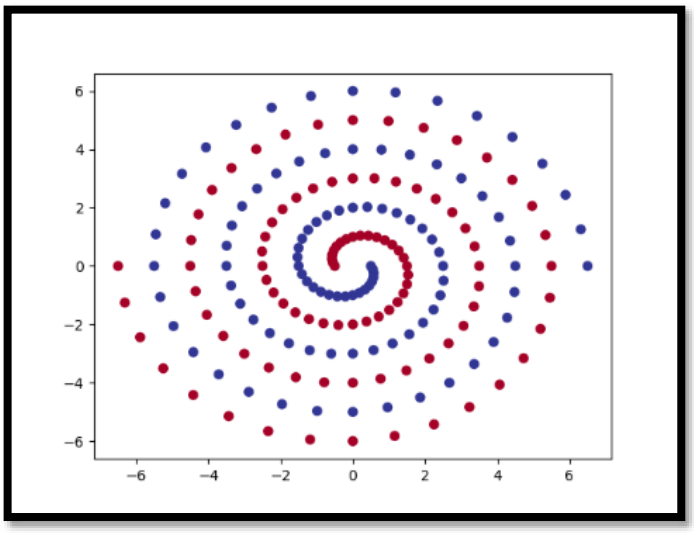
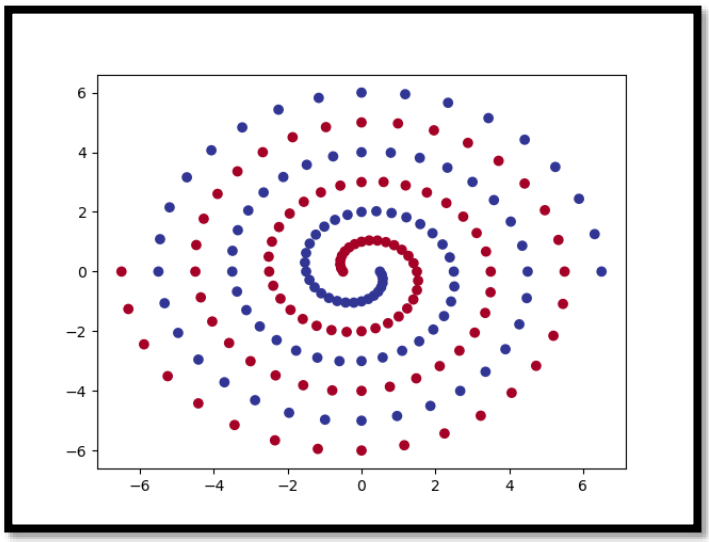
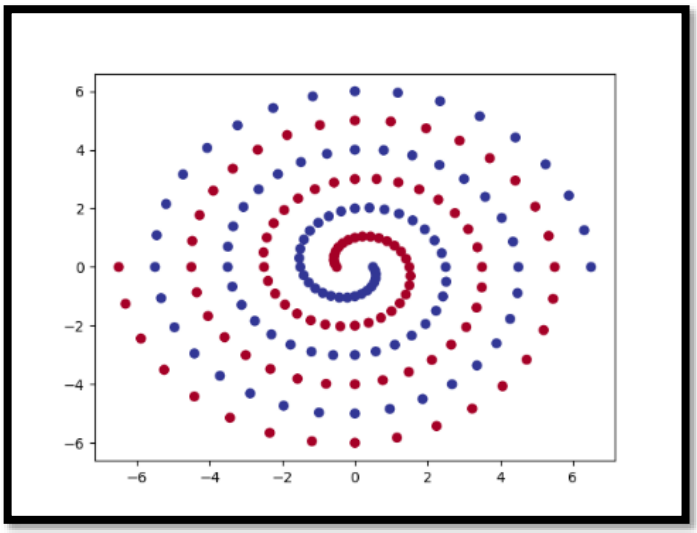
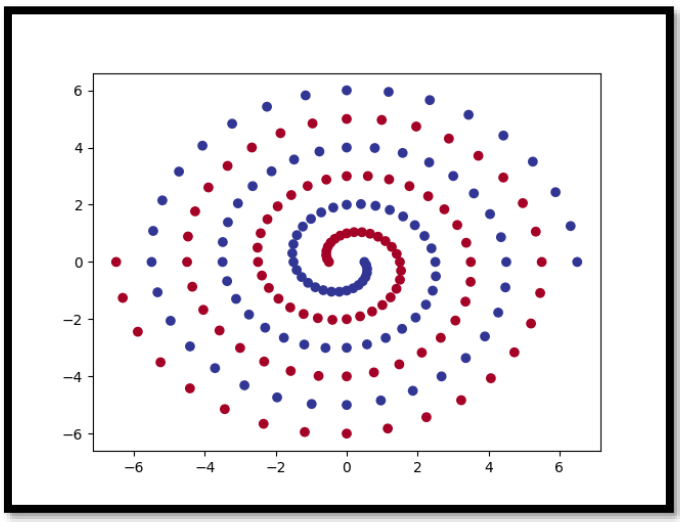


Dheeraj Viswanadham
(z5204820)

I have also made a table of the polar1_# png's that were generated, as shown below:

	
Polar1_0	**Polar1_1**
	
Polar1_2	**Polar1_3**
	

Dheeraj Viswanadham
(z5204820)

Polar1_4	Polar1_5
	
Polar1_6	**Polar1_7**
	
Polar1_8	**Polar1_9**

3) See spiral.py for RawNet module code
4) I experimented with the following parameters when num_hid and lr was kept at the default values.

Init = 0.001 = < 100%

Init = 0.01 = < 100% (more than 99,900 epochs)

Init = 0.1 = < 100% (more than 99,900 epochs)

Init = 0.11 = 100% (6800 epochs*)

Init = 0.12 = 100% (< 20,000 epochs some of the time)

Init = 0.13 = 100% (< 20,000 epochs most of the time)

Dheeraj Viswanadham
(z5204820)

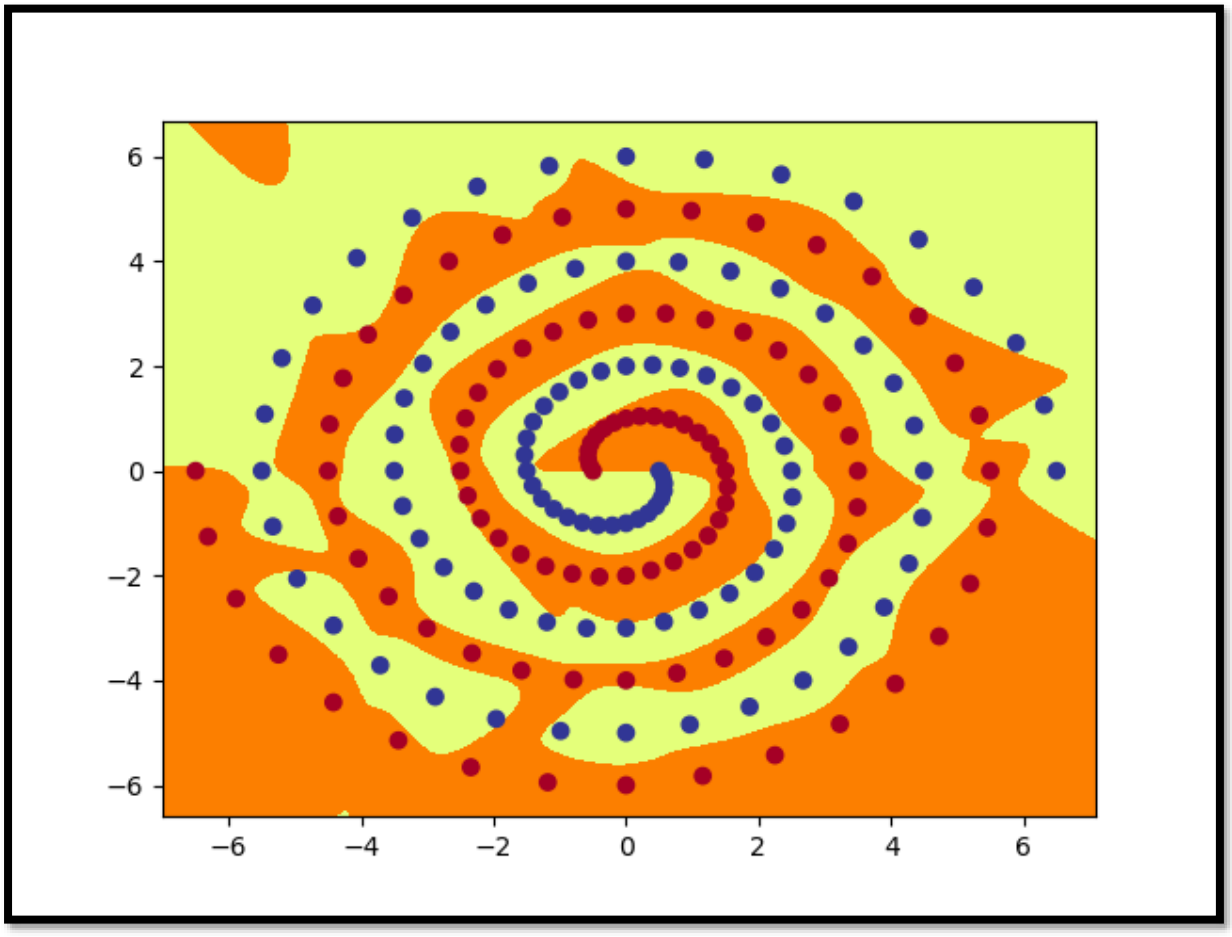
Init = 0.2 = 100% (9200 epochs)

Init = 0.3 = 100% (18,500 epochs)

If found that as I increased initial weight from 0.001 to 0.1 it was getting more accurate and at 0.11 - 0.2 I got 100% accuracy however sometimes it was greater than 20,000 epochs. At 0.3 I did get 100% accuracy, but it was again greater than 20,000 epochs most of the time.

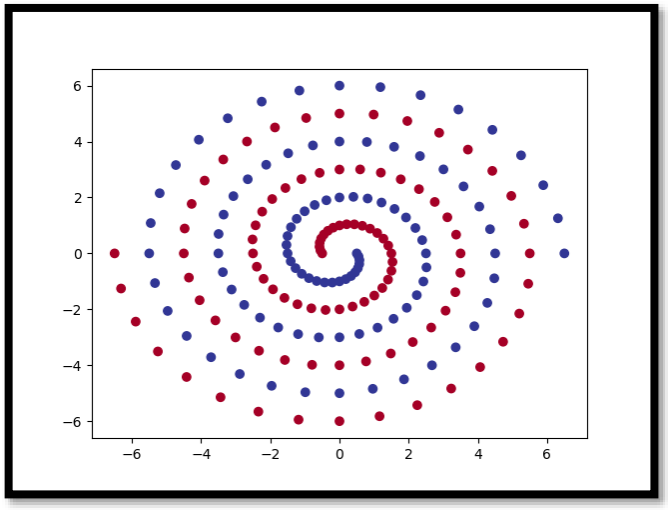
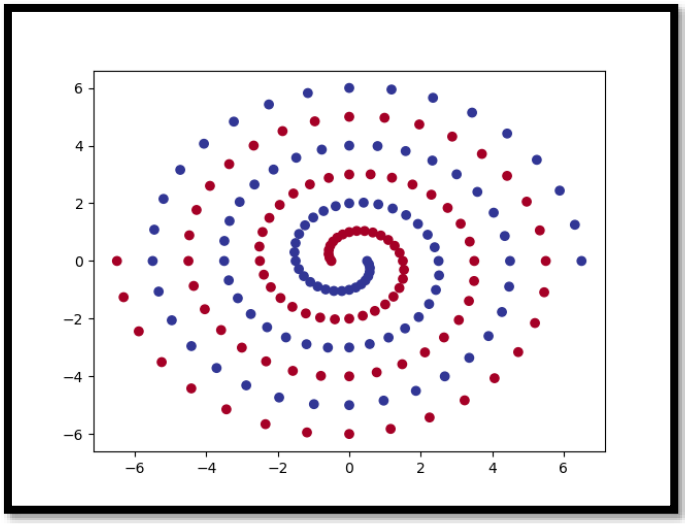
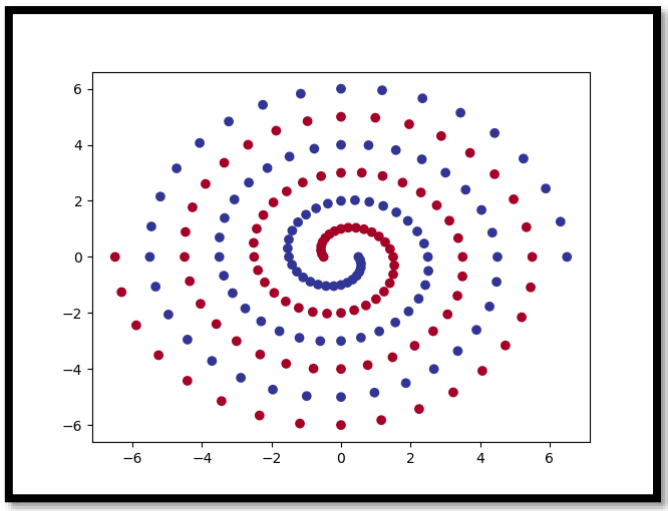
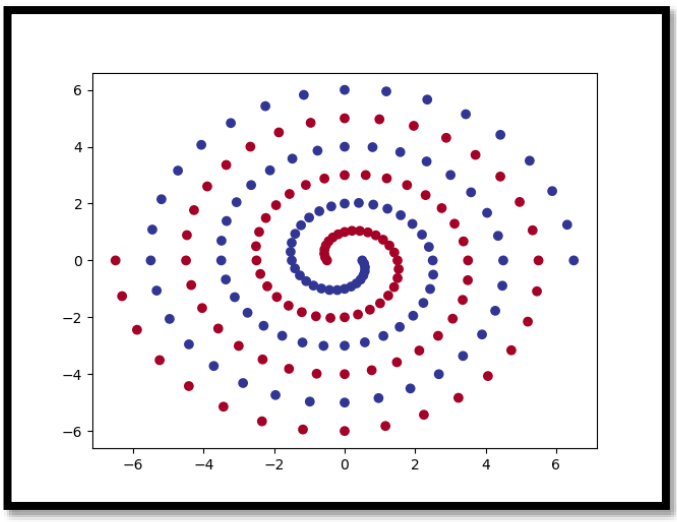
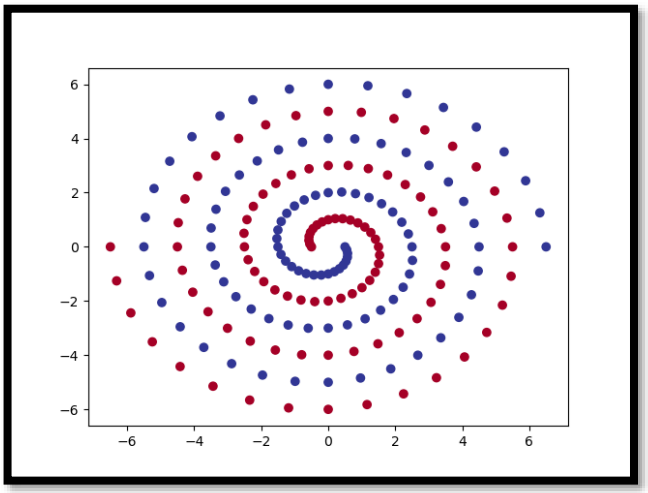
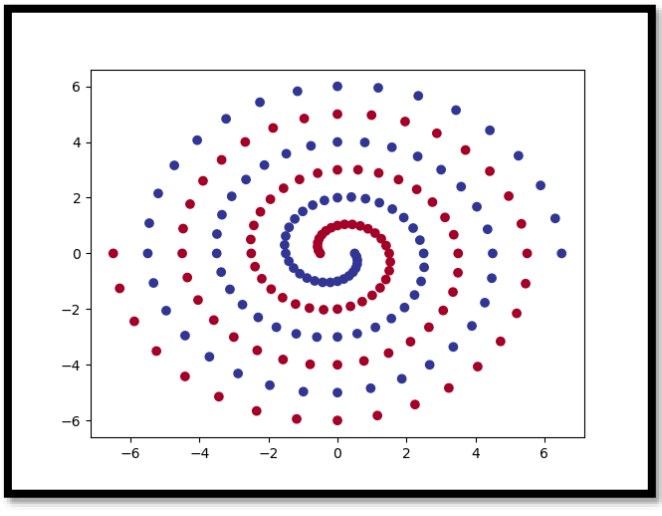
I also adjusted the number of hidden nodes and found that as I increased the hidden nodes from 5 to 10, it was getting completed within a smaller number of epochs where 10 hidden nodes was an optimal number to have. Hence, I found that by having initial weights at 0.13 and hidden nodes at 10, my network was able to correctly classify the training data within 20,000 epochs on almost all runs.

Graph output (raw_out.png):

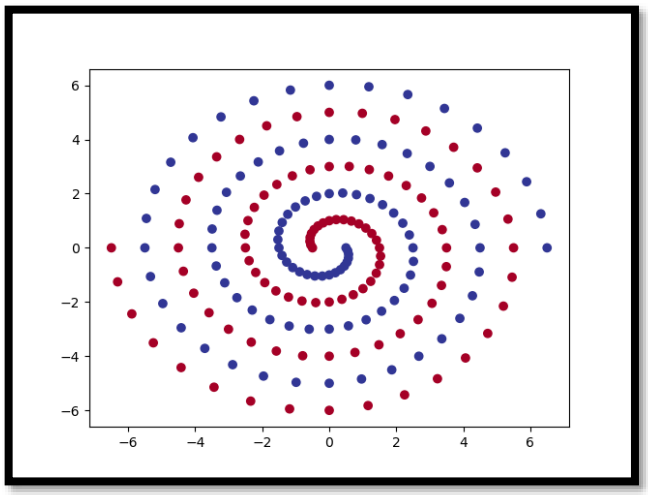
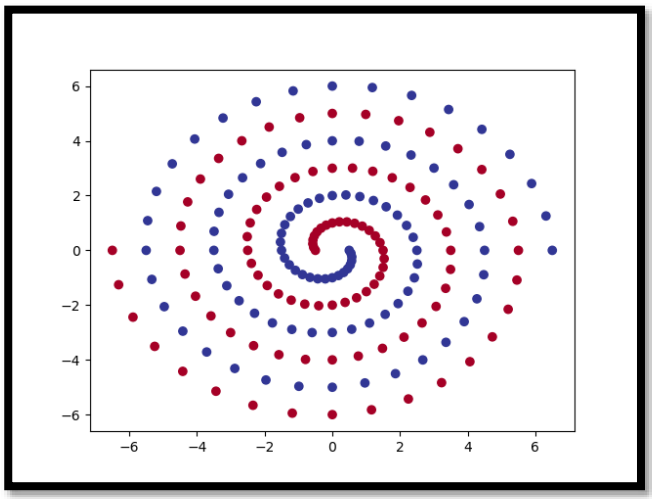
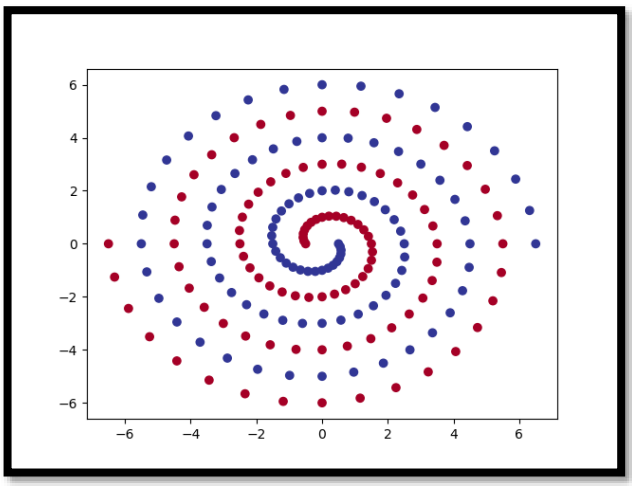
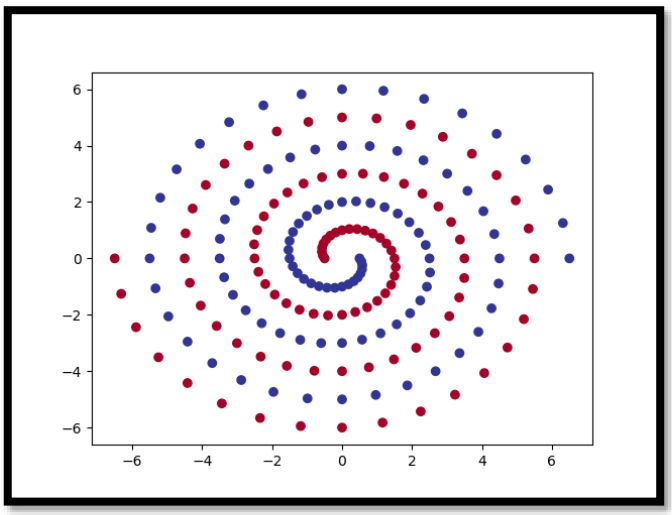
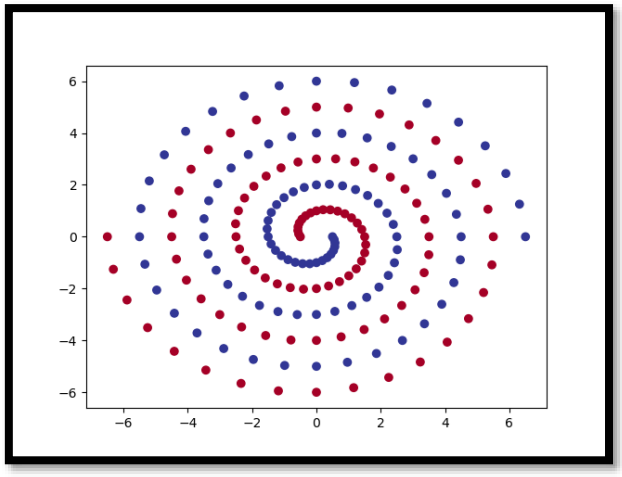
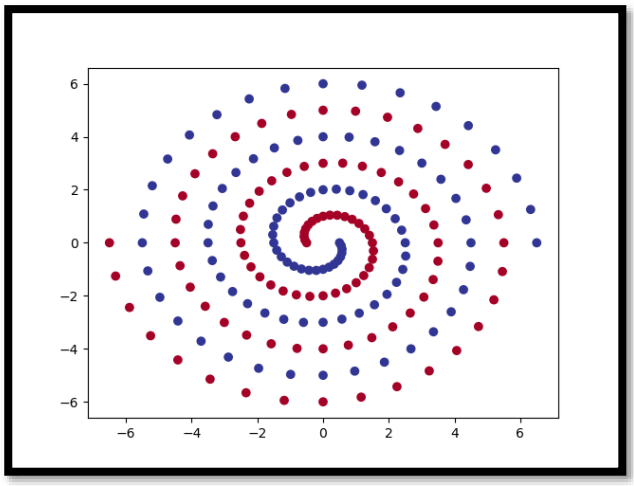


Dheeraj Viswanadham
(z5204820)

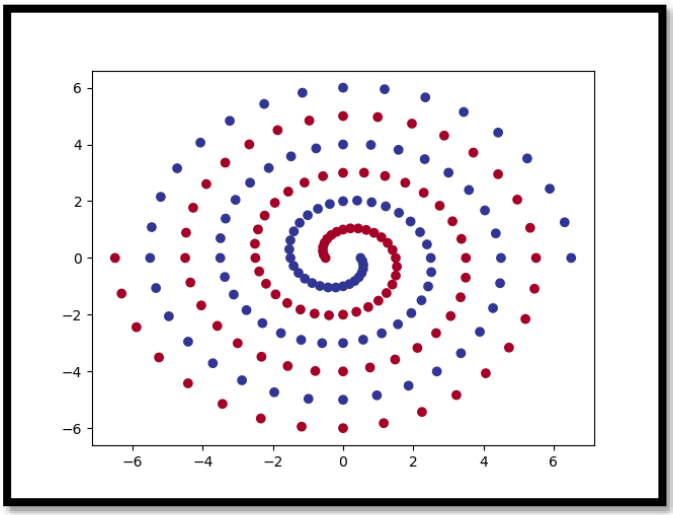
I have also made a table of the raw1_# and raw2_# png's that were generated, as shown below:

	
Raw1_0	**Raw1_1**
	
Raw1_2	**Raw1_3**
	

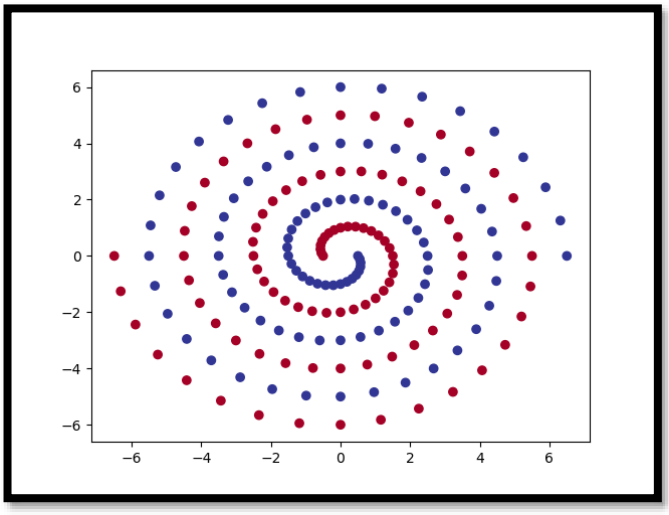
Dheeraj Viswanadham
(z5204820)

Raw1_4	Raw1_5
	
Raw1_6	**Raw1_7**
	
Raw1_8	**Raw1_9**
	
Raw2_0	**Raw2_1**

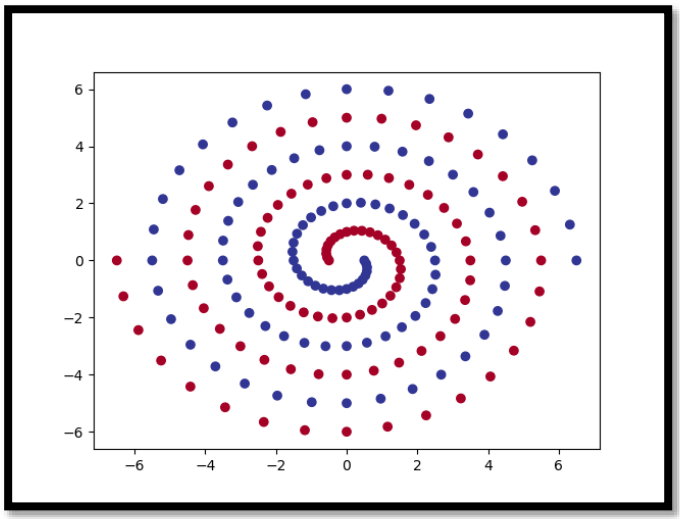
Dheeraj Viswanadham
(z5204820)



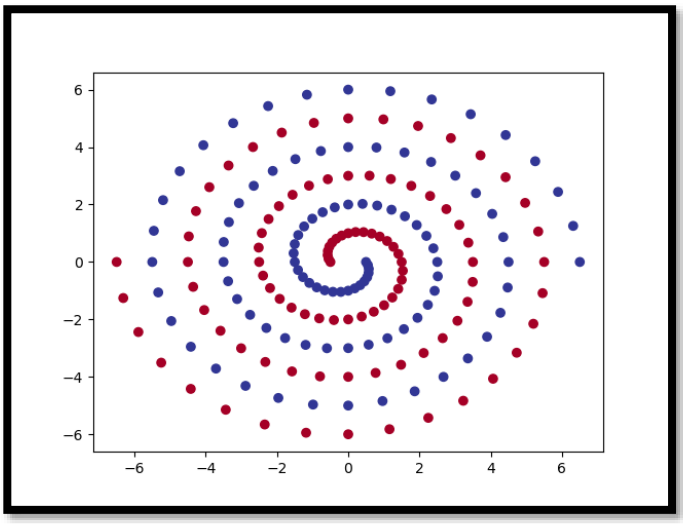
Raw2_2



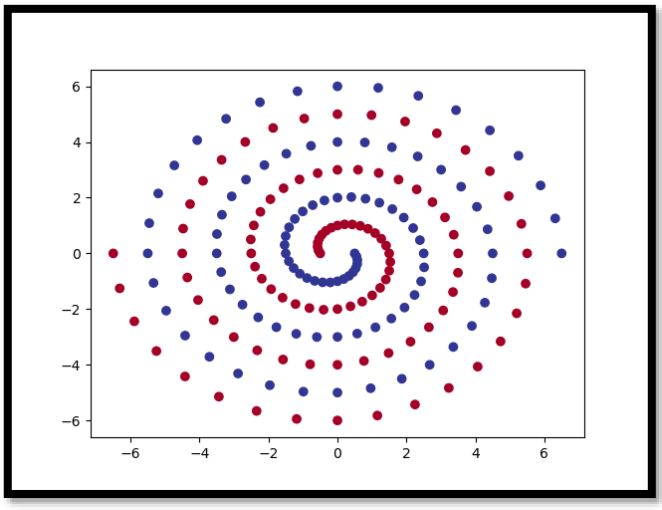
Raw2_3



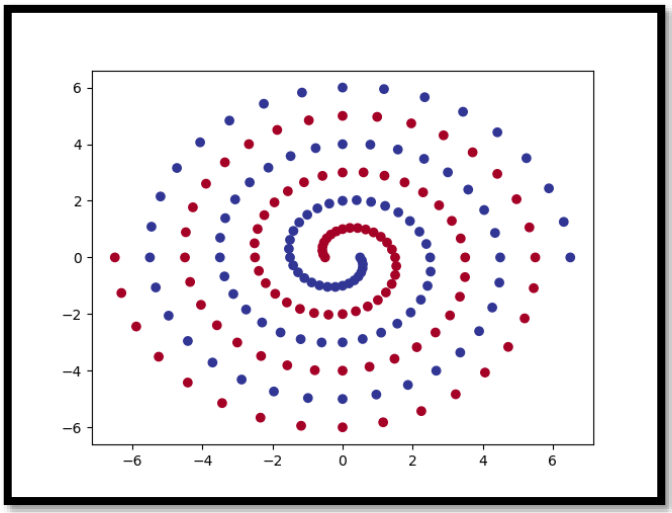
Raw2_4



Raw2_5

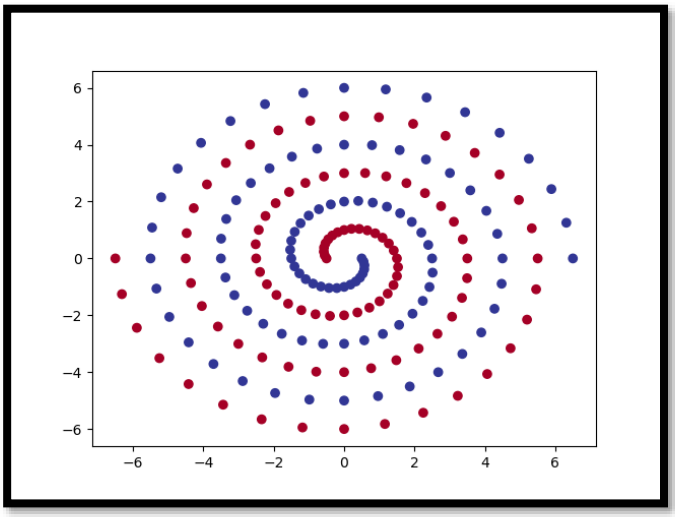
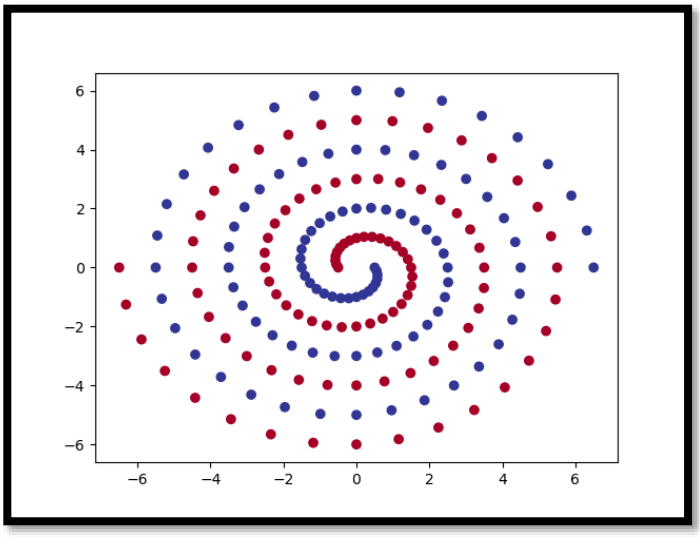


Raw2_6



Raw2_7

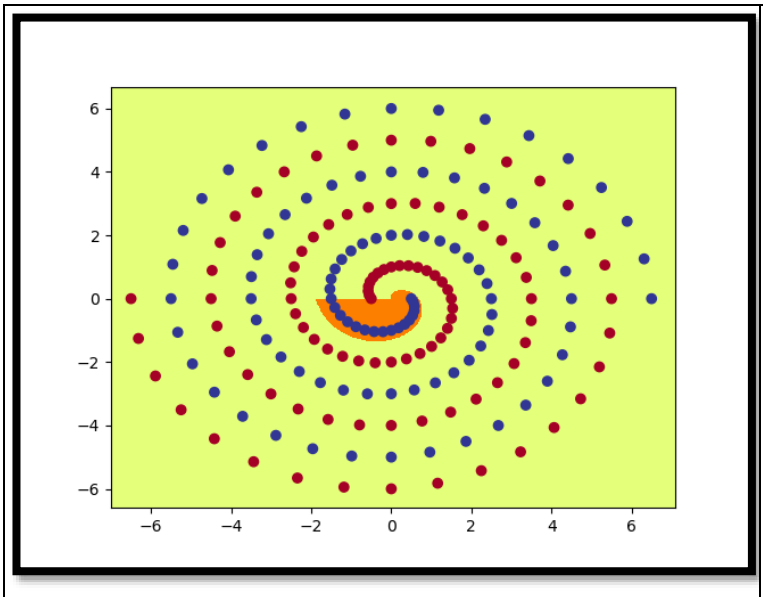
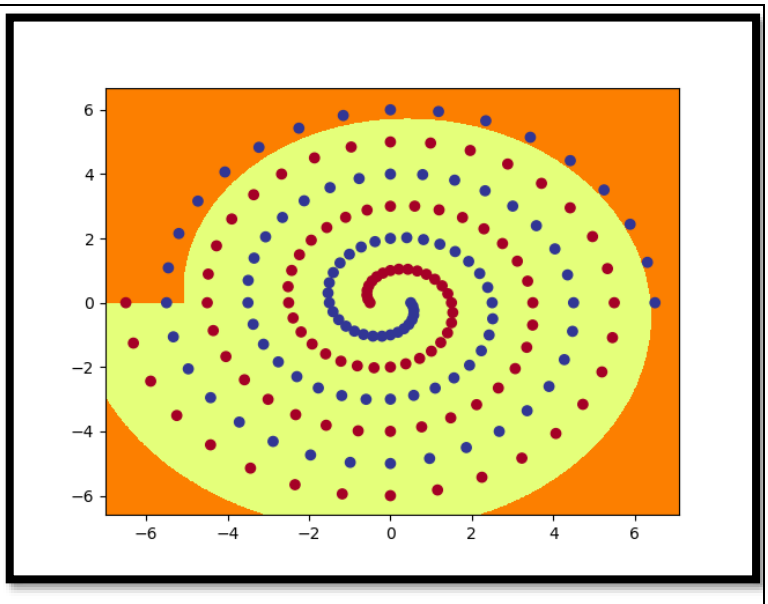
Dheeraj Viswanadham
(z5204820)

	
Raw2_8	**Raw2_9**

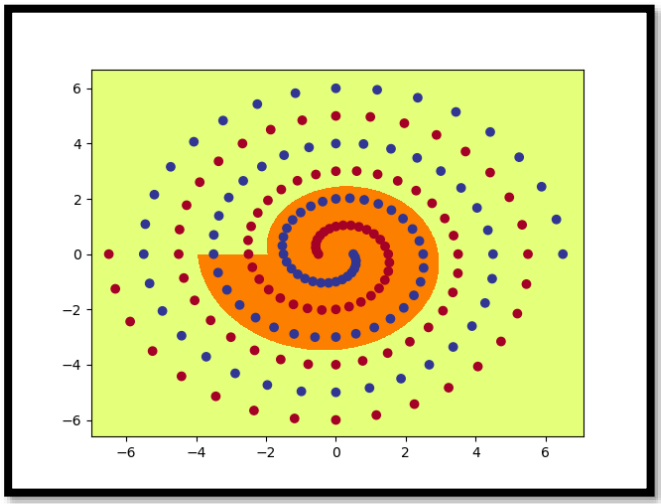
5) Note: code for graph_hidden in spiral.py file is **very similar** to graph_output in spiral_main.py file (as we were asked to base it off that).

Below are my results for both PolarNet (with hidden nodes at 7) and RawNet (with initial weight 0.13 and hidden nodes at 10). I only included the plots that were coloured and left the blank ones (that were white) out as I already included them above under each module for q2 and q4.

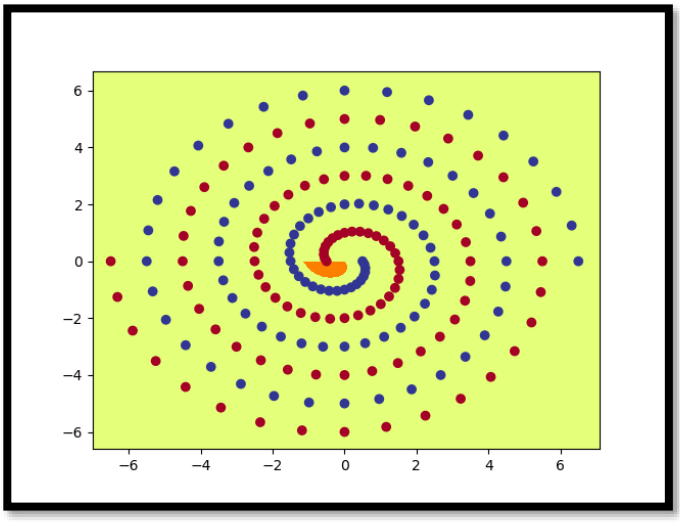
PolarNet:

	
Polar1_0	**Polar1_1**

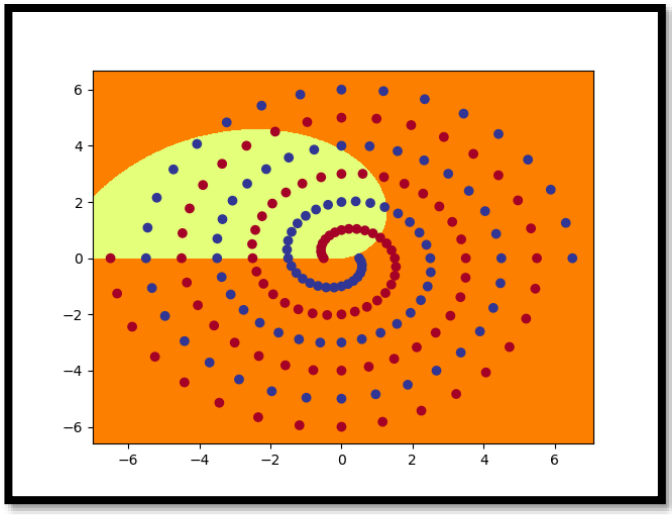
Dheeraj Viswanadham
(z5204820)



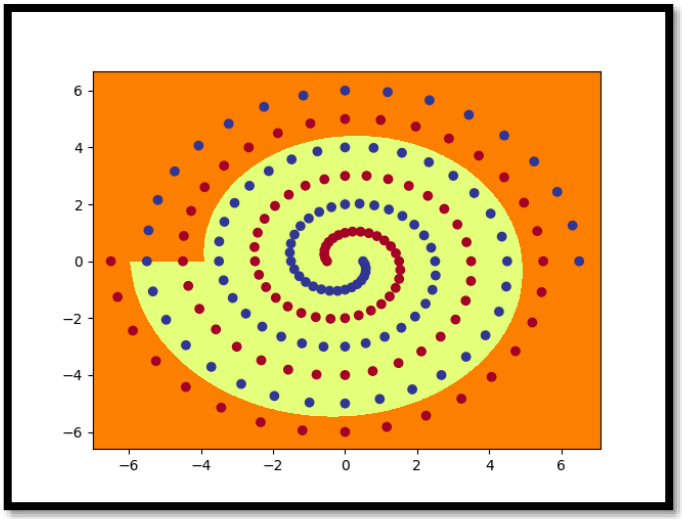
Polar1_2



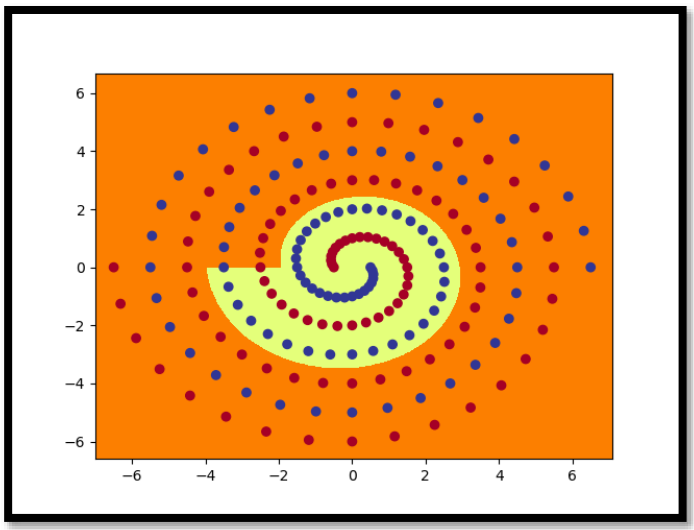
Polar1_3



Polar1_4



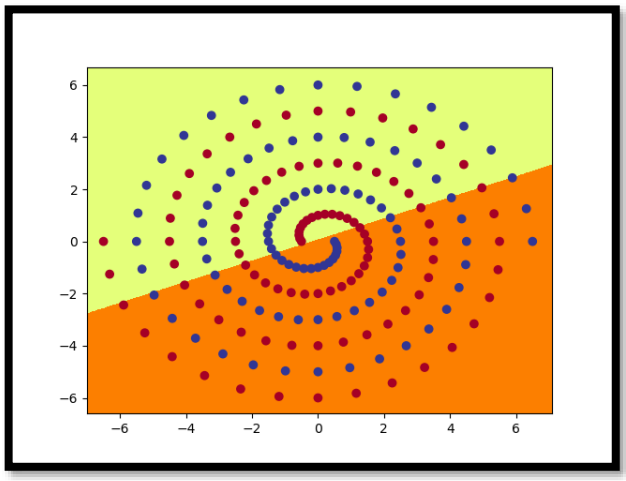
Polar1_5



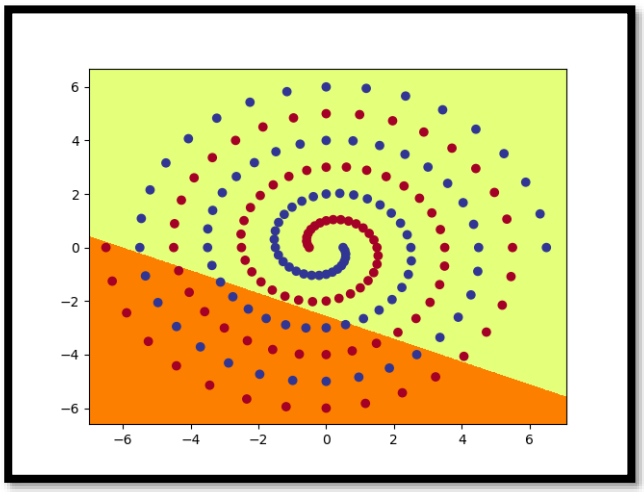
Polar1_6

Dheeraj Viswanadham
(z5204820)

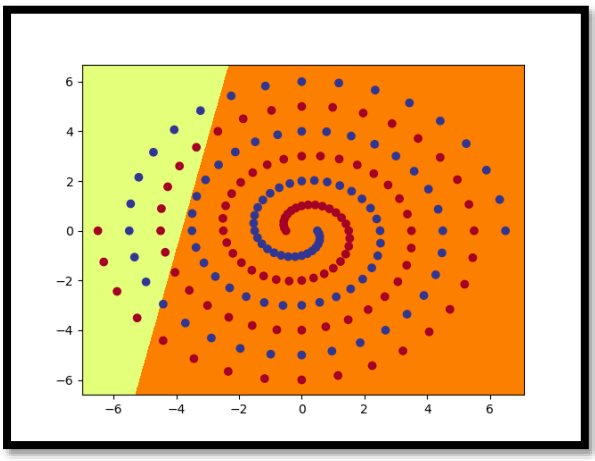
RawNet:



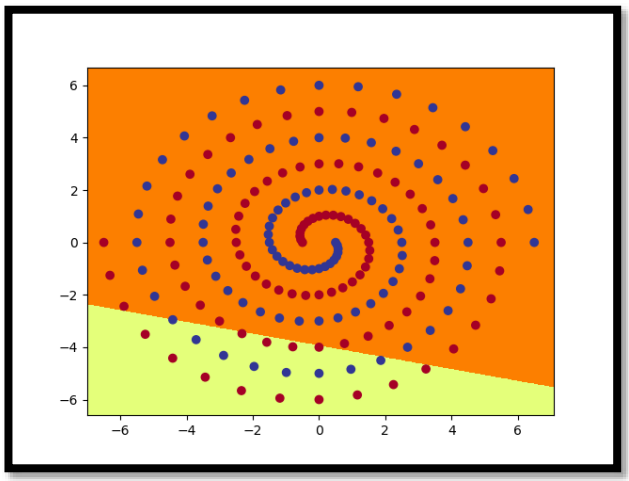
Raw1_0



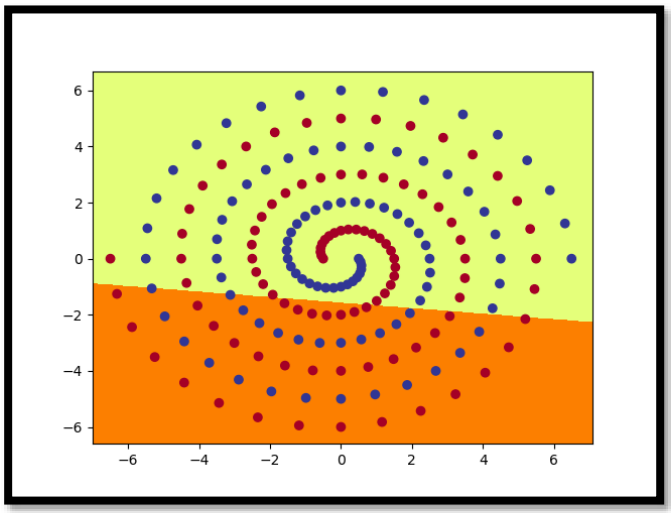
Raw1_1



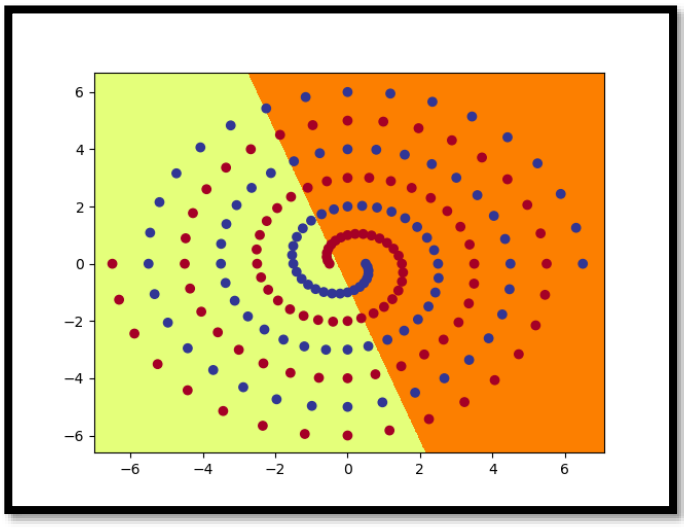
Raw1_2



Raw1_3

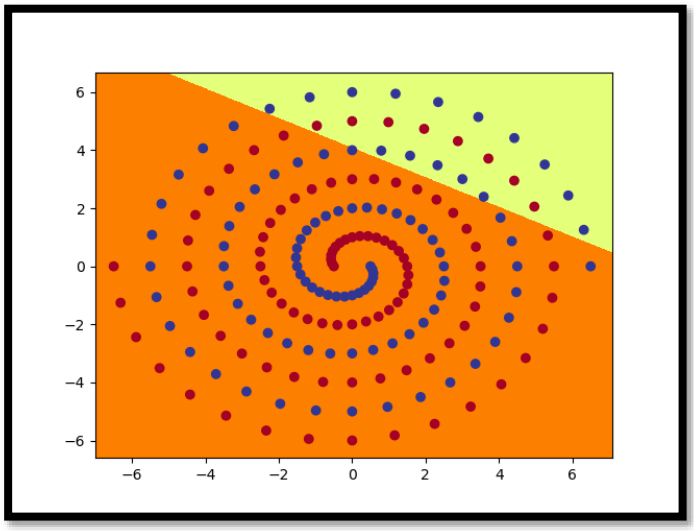


Raw1_4

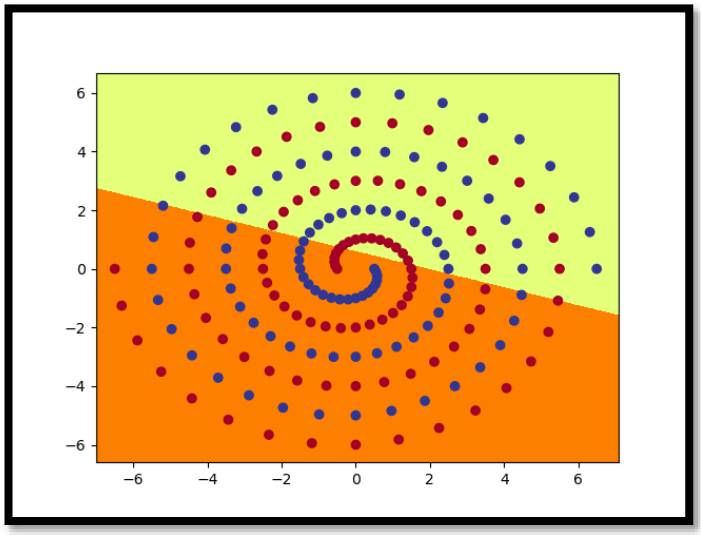


Raw1_5

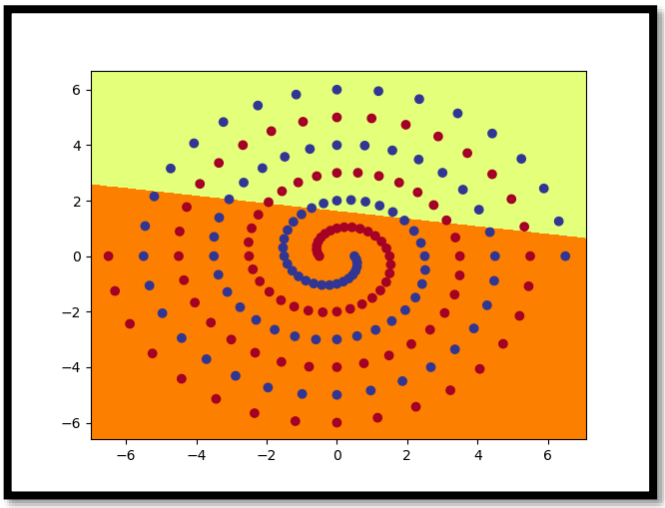
Dheeraj Viswanadham
(z5204820)



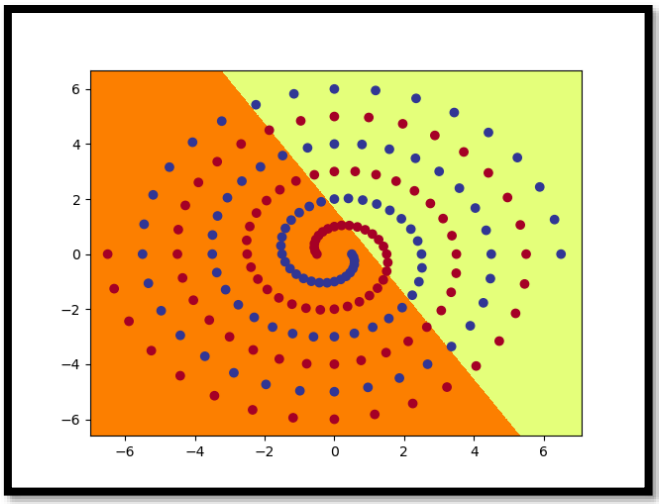
Raw1_6



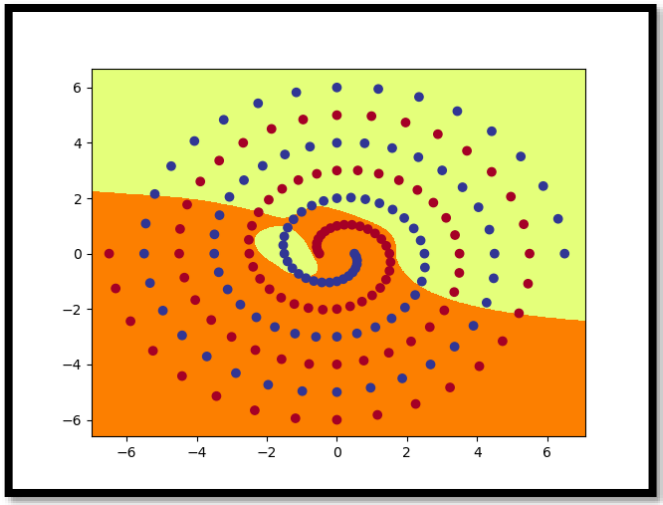
Raw1_7



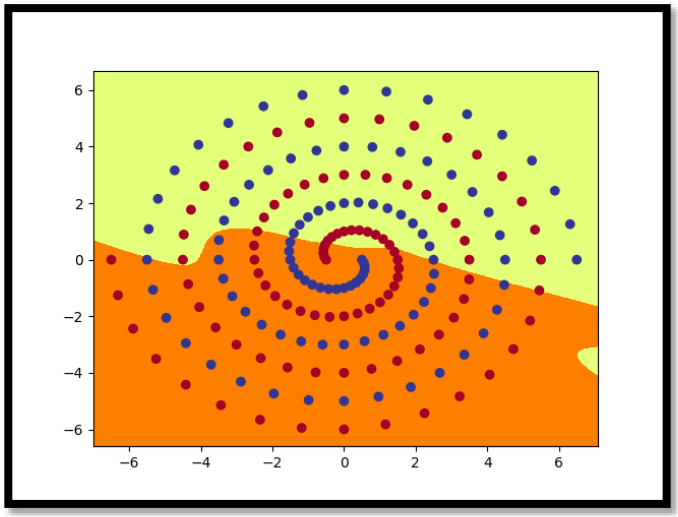
Raw1_8



Raw1_9

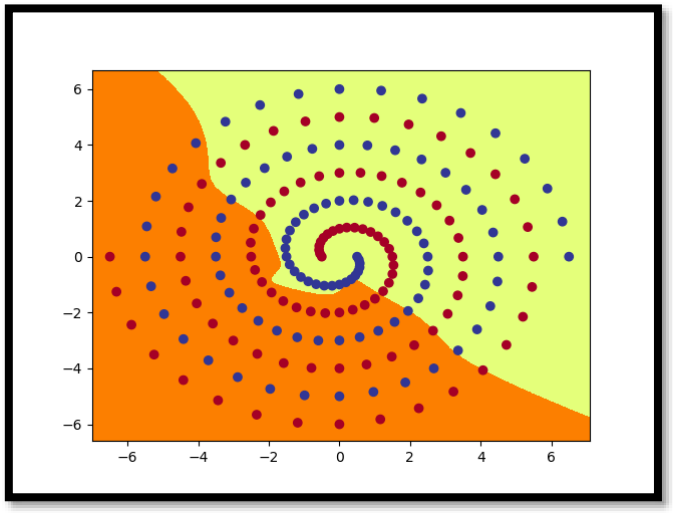


Raw2_0

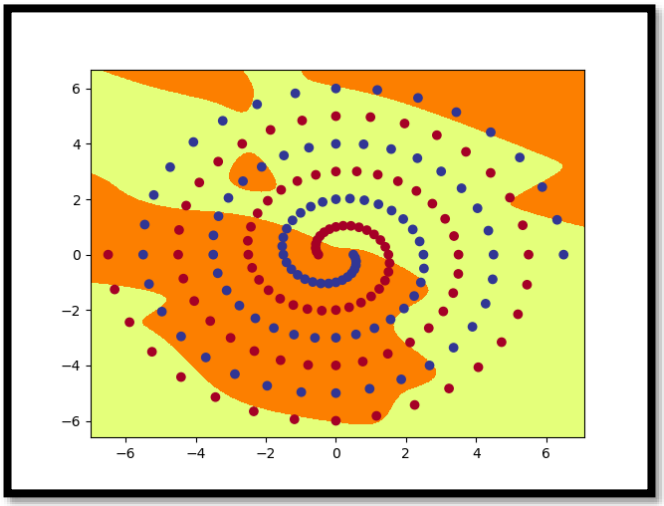


Raw2_1

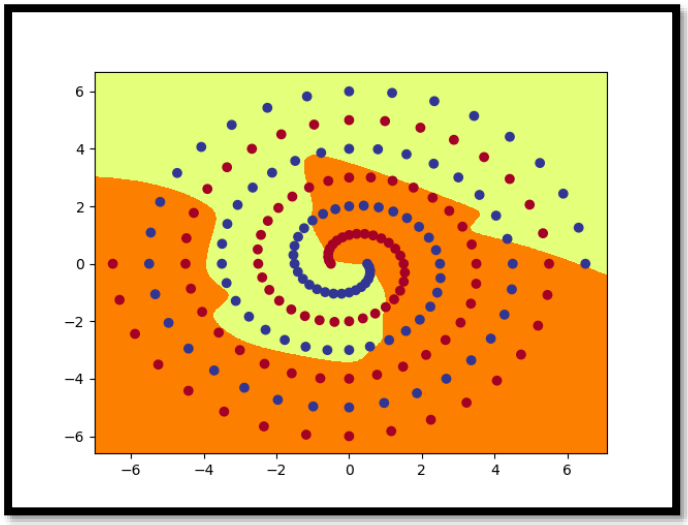
Dheeraj Viswanadham
(z5204820)



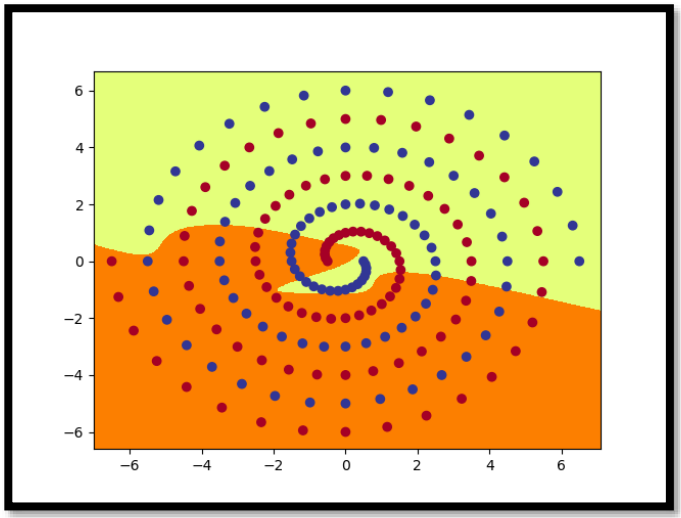
Raw2_2



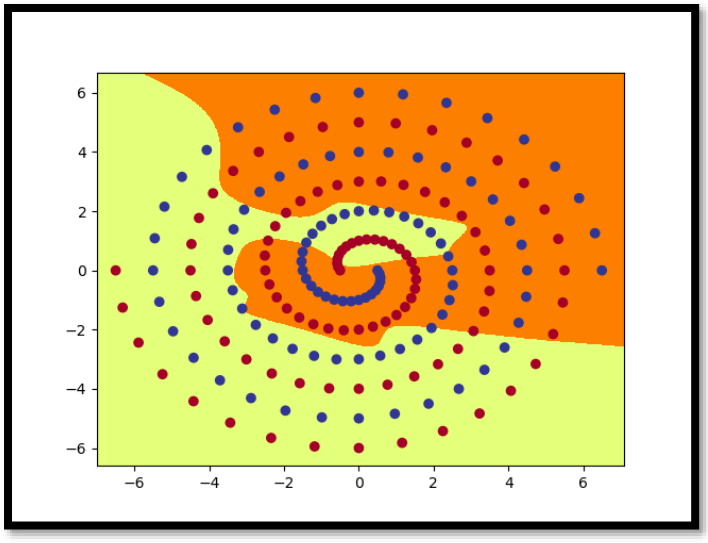
Raw2_3



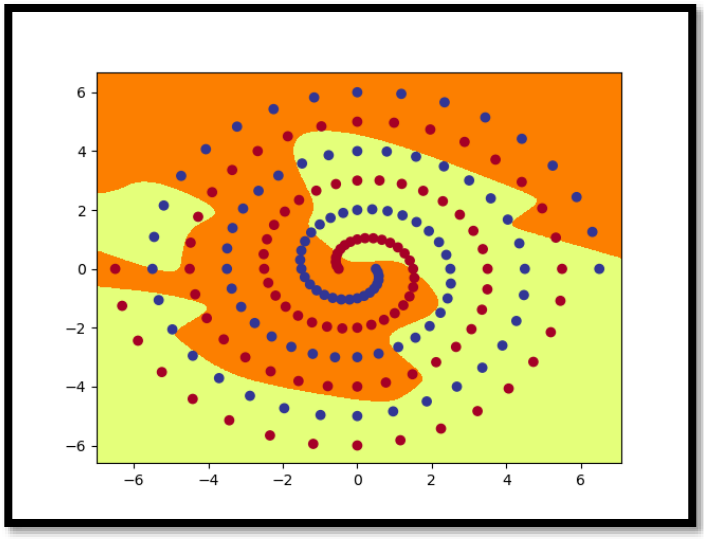
Raw2_4



Raw2_5

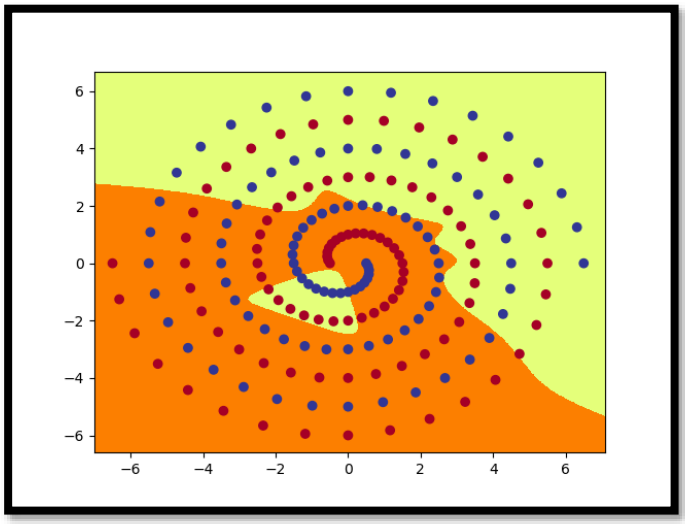
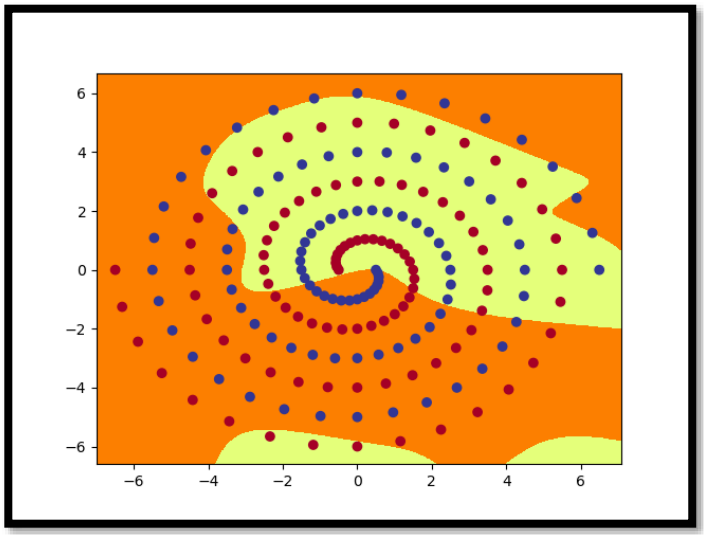


Raw2_6



Raw2_7

Dheeraj Viswanadham
(z5204820)

	
Raw2_8	**Raw2_9**

6a) From what can be observed above, we can note the main qualitative difference between the two modules, PolarNet and RawNet. The image that PolarNet outputs, polar_out.png, can be seen to have clean curves where the orange regions are clearly defined as compared to raw_out.png which is seen to be more linear but sporadic in nature. I.e. for PolarNet the function is non-linear, whilst for RawNet the function is linear for the first hidden layer and non-linear for the second hidden layer as shown by the above polar1_#, raw1_# and raw2_# png images.

b) For RawNet, I first tried experimenting with the initial weight parameter in the range of [0.001, 0.3] inclusive. I found through trial and error that an optimal initial weight parameter was around 0.13 and from here I experimented with the number of hidden nodes in the range of [5, 10] inclusive. Here I found that whilst the initial weight was 0.13, the network was able to correctly classify the data within 20,000 epochs most of the time when I had 10 hidden nodes.

I decided to go with a 0.13 initial weight as if it was too low (around 0.001) it would not finish within 20,000 epochs, often at times reaching 99,900 epochs and less than 100% but as I steadily increased it the success and speed of learning was steadily increasing.

c) For PolarNet, I first changed the batch size from 97 to 194. What I observed was that ceteris paribus, the number of epochs required to reach 100% accuracy was greatly reduced as previously it was around 99,900 epochs to now only 11,000 epochs. Observing this, I tried increasing the batch size further from 194 to 300. Here the number of epochs required to reach 100% slightly increased to 1400 epochs. I then tried increasing the batch size further to 500 and the number of epochs required reduced to 900 epochs. I then tried having a batch size of 1000 and the number of epochs increased again back to 1300 epochs. Seeing this, I determine that there existed a range within which the speed at which the networks learns is optimal. I also tried reducing the number of hidden nodes from 10 to 7 but this seemed to increase the number of epochs required for the network to reach 100% accuracy at all the batch sizes I trialled. I also tried changing the optimiser to SGD but the network seemed to perform worse than when I used Adam at all batch sizes.

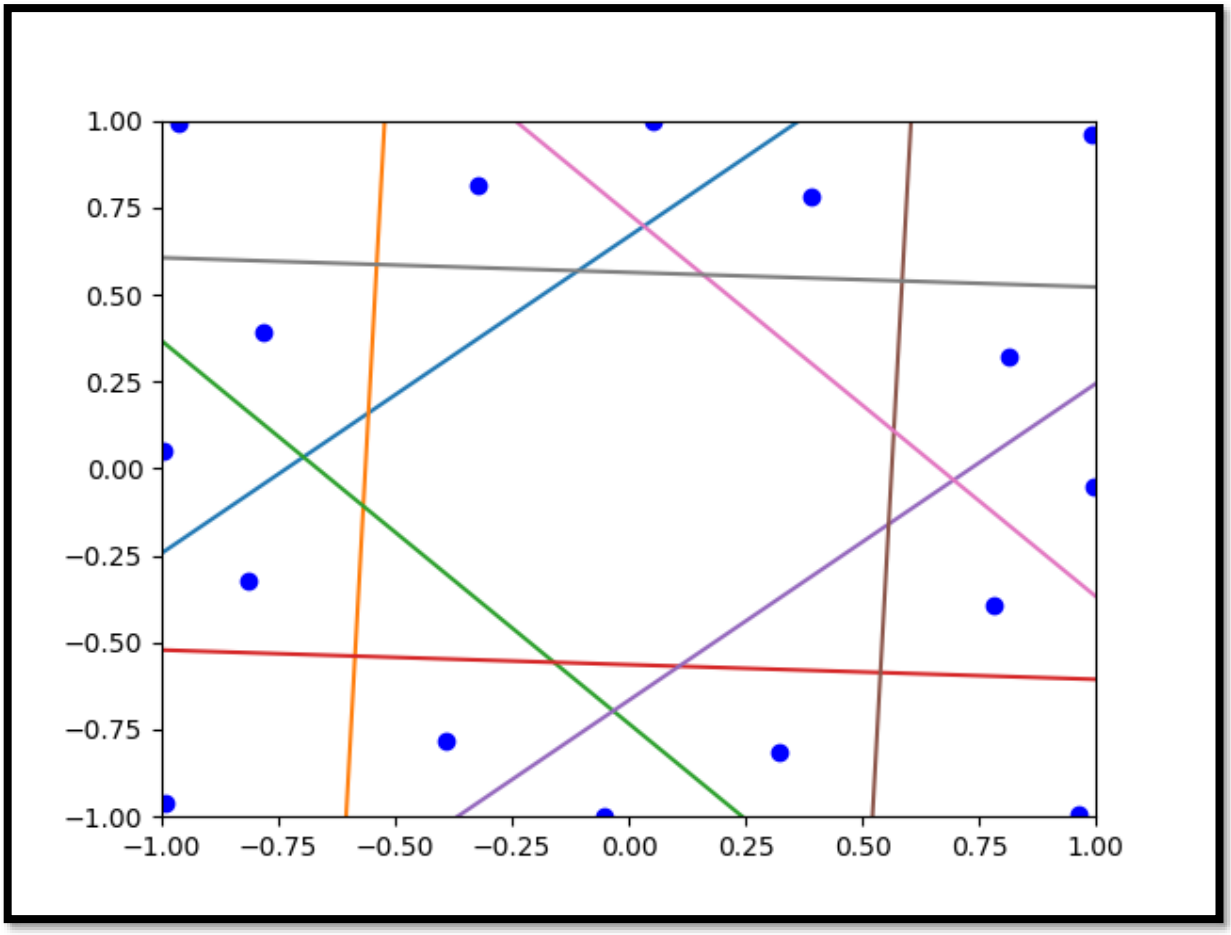
For RawNet, I again changed the batch size from 97 to 194. I tried using the SGD optimiser for a change and saw that the number of epochs required to reach 100% was much too long with everything apart from the batch size being at their default values. I then tried increasing the initial weight to 0.13 whilst keeping batch size at 97 but it reached 99,900 epochs having less than 100%. I then tried increasing batch size to 194 and it still did not perform well, reaching 99,900 epochs and less than 100% accuracy. Another major aspect I noticed that pertained to SGD was that even though the accuracy was increasing, it was increasing much slower than what Adam would i.e. increasing by smaller increments than what Adam would where Adam would sometimes increase by 3 – 5% at a time whilst SGD would only increase by 0.1 – 0.5% or so at a time.

Part 3 Next Page →

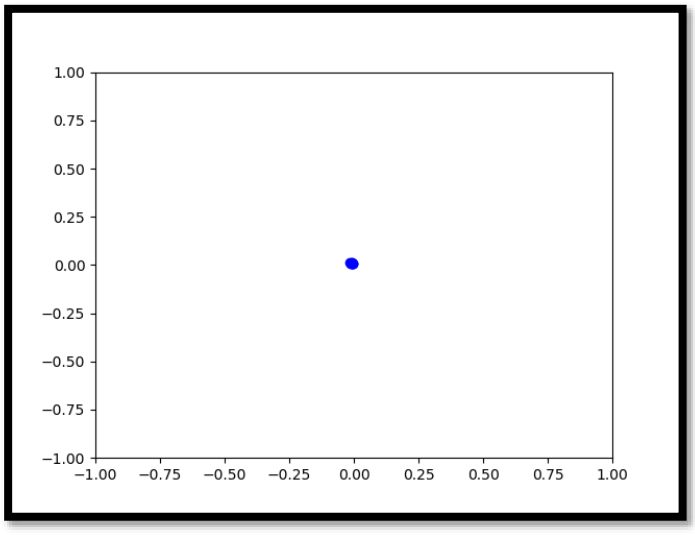
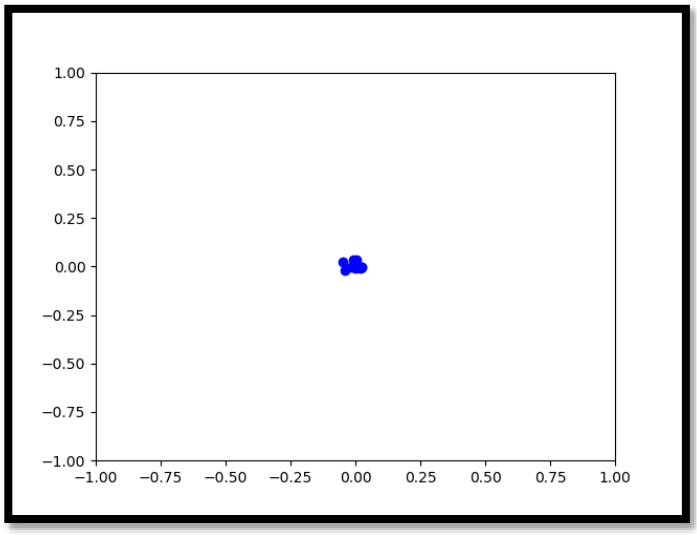
Dheeraj Viswanadham
(z5204820)

Part 3:

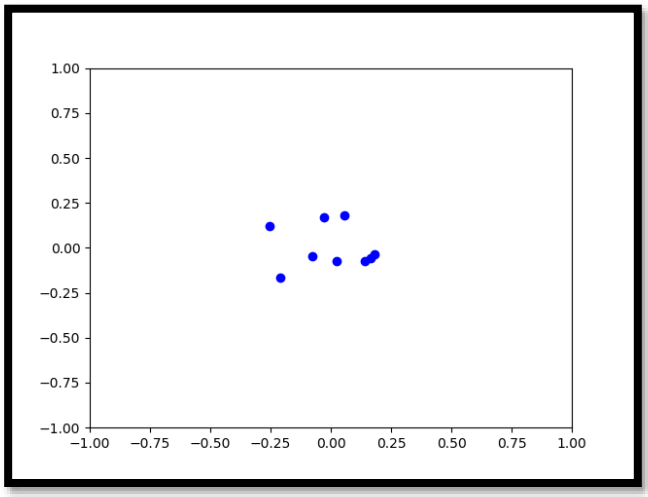
1) The final image that is generated when the program is run is shown below:



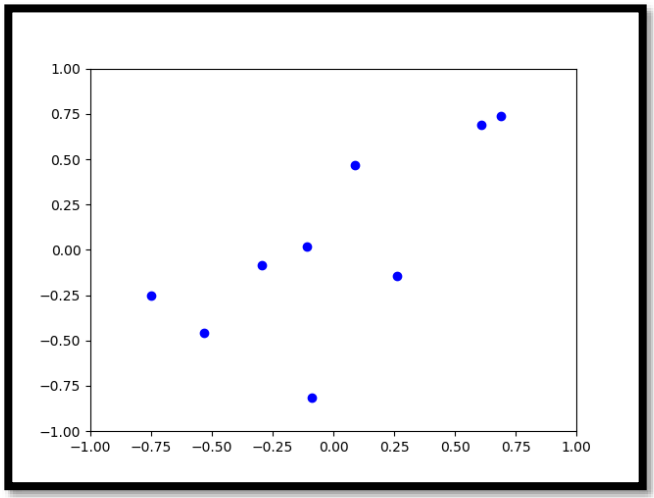
2) When the program is run the following images were generated:

	
Epoch 50	**Epoch 100**

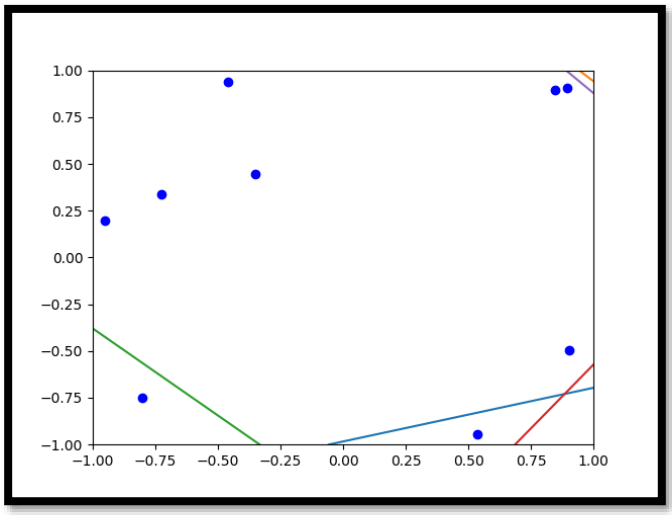
Dheeraj Viswanadham
(z5204820)



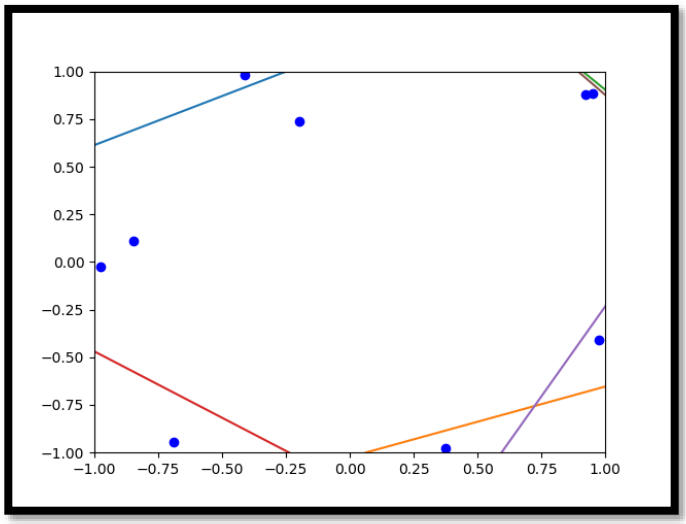
Epoch 150



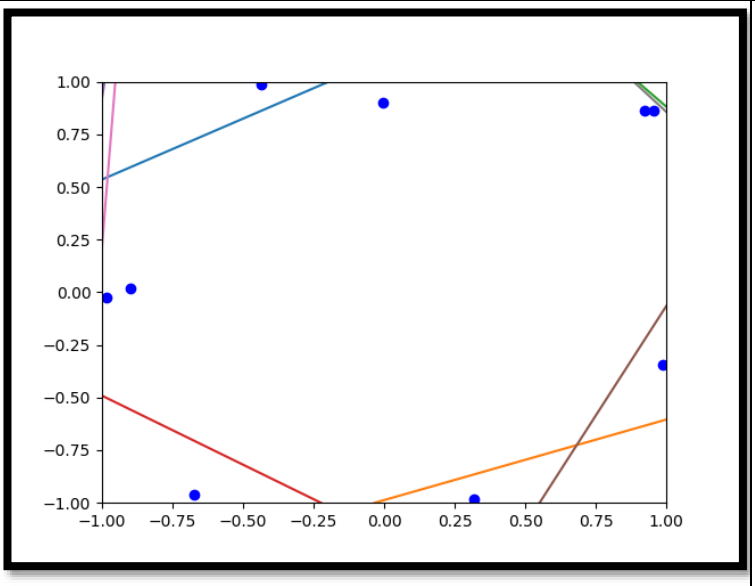
Epoch 200



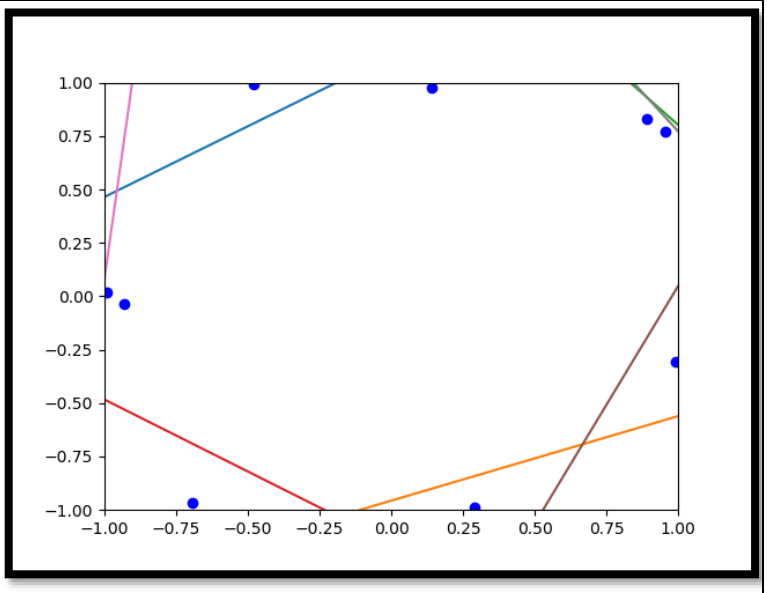
Epoch 300



Epoch 500

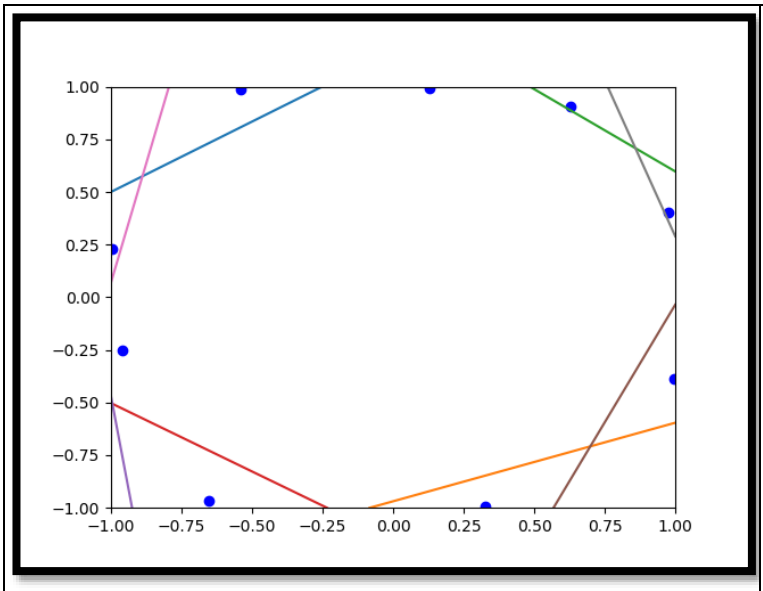


Epoch 700

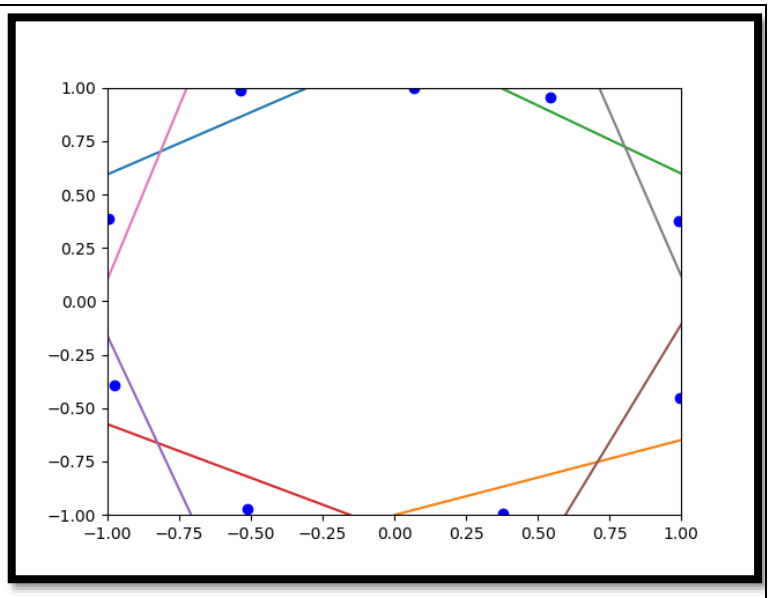


Epoch 1000

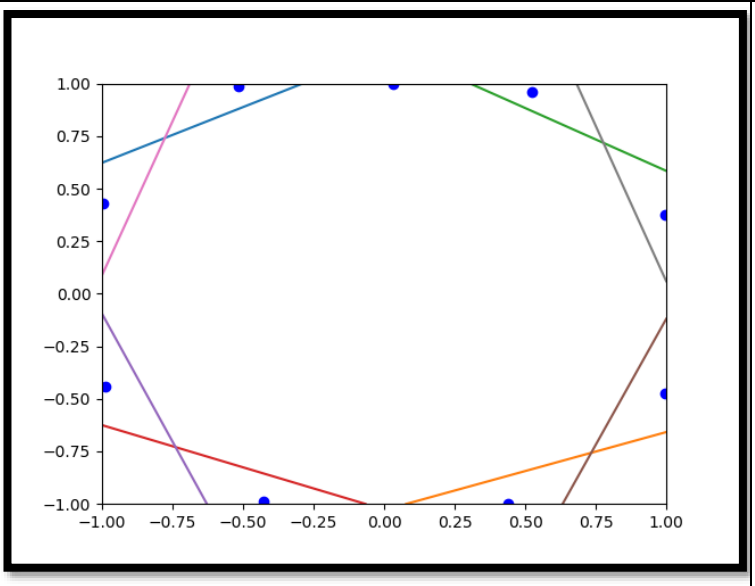
Dheeraj Viswanadham
(z5204820)



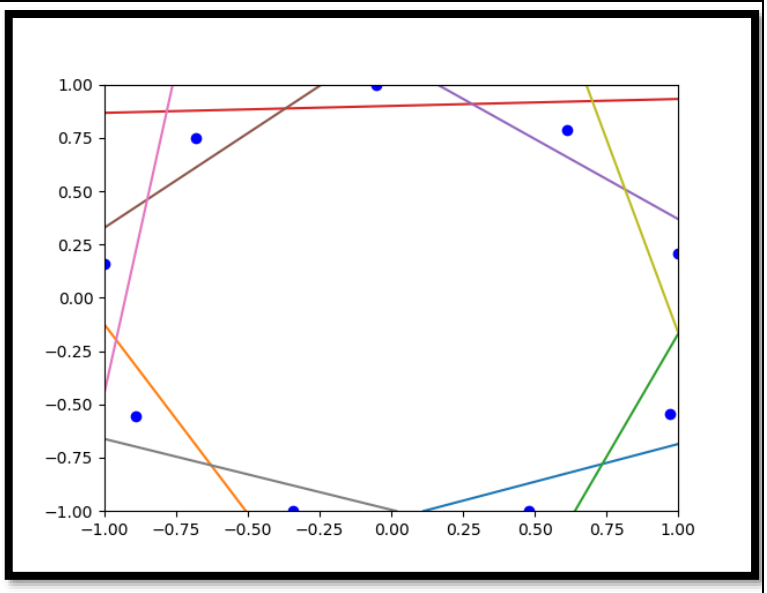
Epoch 1500



Epoch 2000



Epoch 3000

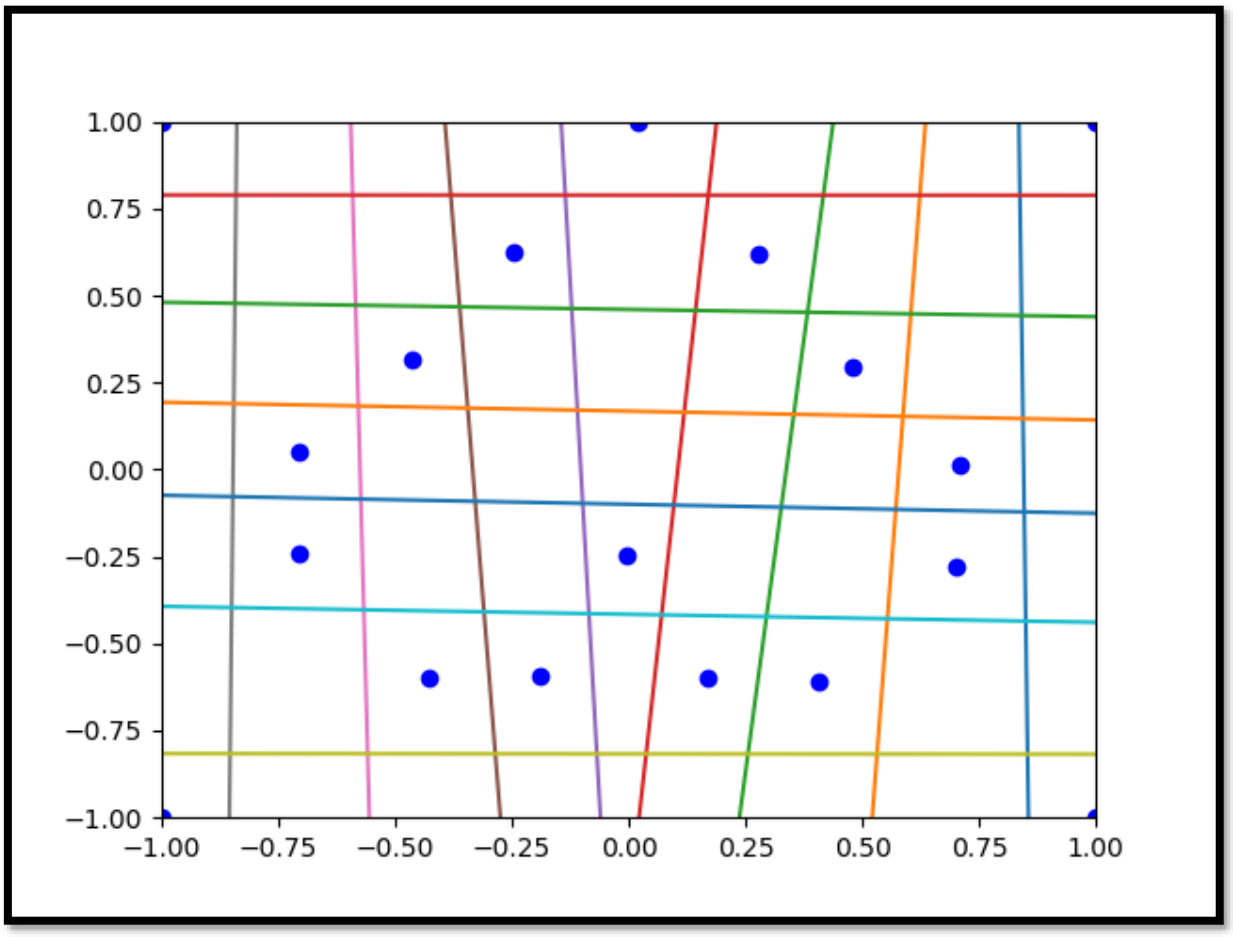


Final Image

As observed above, we can see that the lines are going in a clockwise direction first starting from the top-right and going clockwise. Initially, the dots are close together and as the network is trained, we can see it expanding across to the boundaries, creating a nonagonal shape.

Dheeraj Viswanadham
(z5204820)

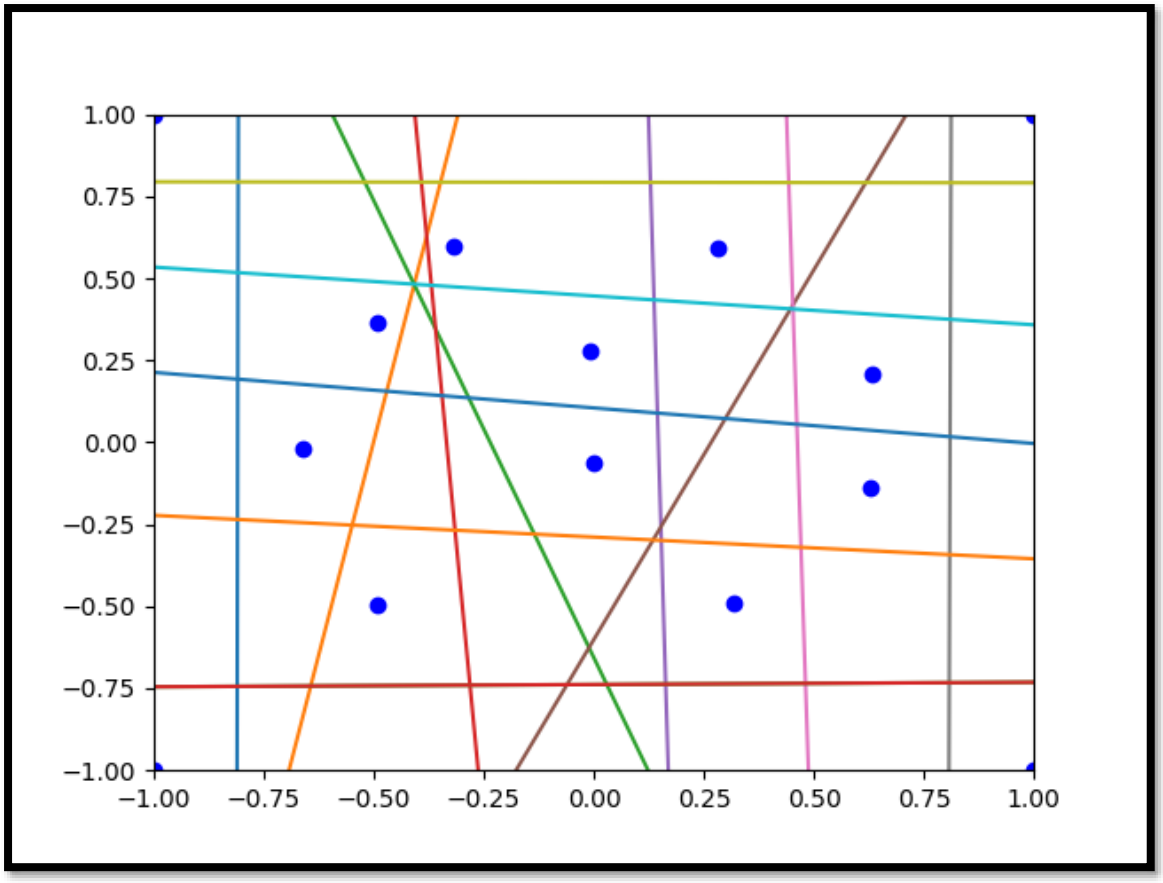
3) Please see encoder.py for the heart18 tensor code. In summary, I knew that the inputs (or dots) would be 18 – corresponding to the rows in my tensor, whilst the outputs (or lines) would be 14 – corresponding to the columns in my tensor. Knowing this, I first examined the example image of the heart shape tensor and identified where the dots were falling i.e. 4 dots on each of the corners whilst the remaining 14 dots formed the heart shape (similar to Exercise 5, Question 1 in the Tutorial Exercises). I then started small, identifying what way to input the code such that the dots were on the corners and the lines separated them approximately equally which is the same approach for Exercise 5, Question 1. I then tried my hand at forming the given heart shape once my building blocks were in place, and since I knew that the heart was symmetrical in shape, I could follow that pattern. By following the above, I was able to get approximately close to the example image as shown below (though for each run the heart shape was going in different directions e.g. sometimes it was flipped or rotated).



4) See code in encoder.py regarding my target1 and target2 tensors.

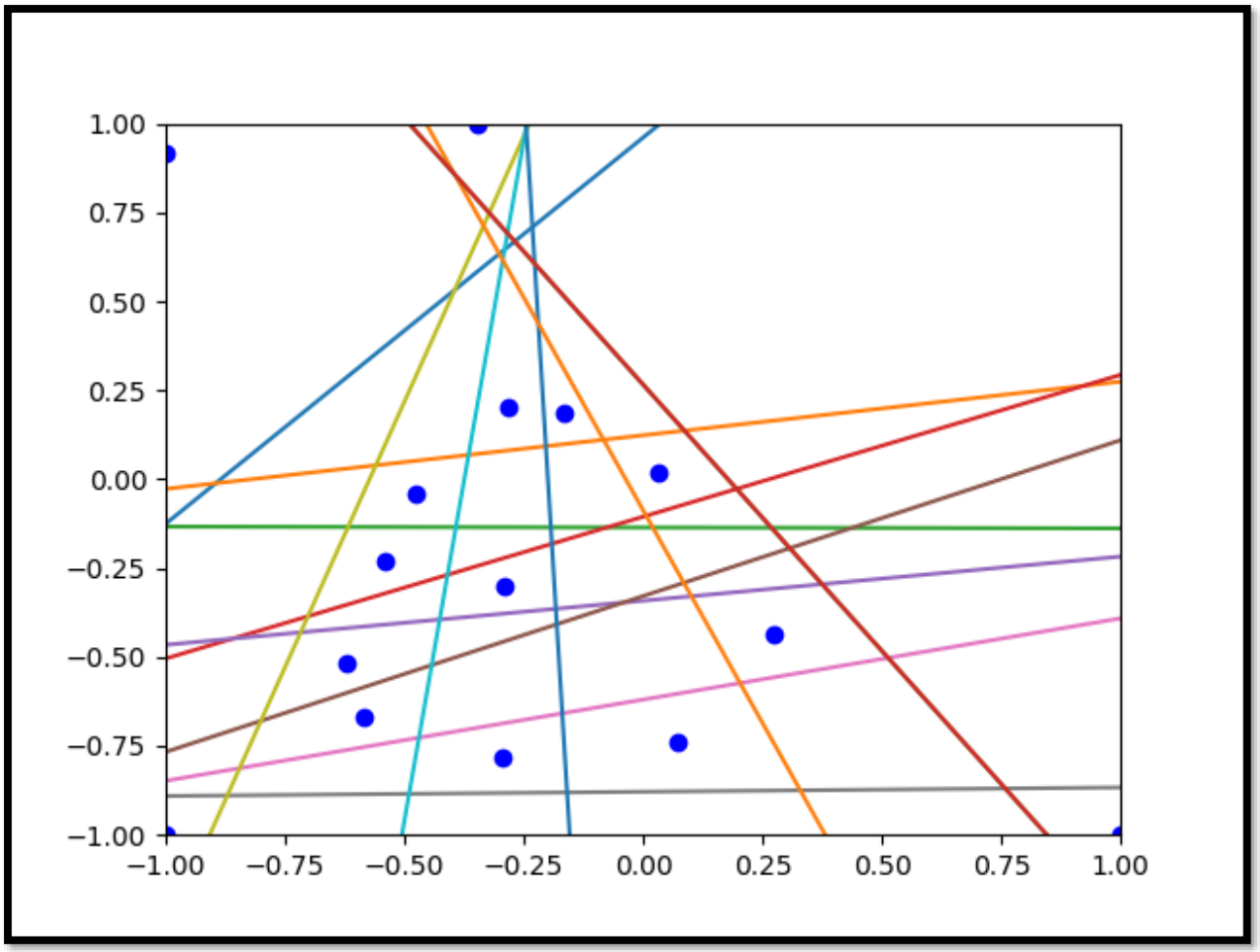
For target1, I tried experimenting with the shapes that I could create as well as an optimal learning rate for the network to train at. As I steadily increased the learning rate from the range [0.01, 1] I found that I was obtaining my desired shape faster within a lesser number of epochs and more consistently (though there were a few runs where the shape was distorted slightly). In the end I found that I could create an "∞" [infinity] symbol (or bowtie depending on how you look at it) by manipulating the tensor – via the inputs (dots) and outputs (lines). I

found that an optimal learning rate was at 1.5 where anything above that the runs would not return the infinity shape most of the time.



For target2, I experimented with some different styles of shapes that I could create with the tensors and I found that I could create unique shapes. At one point I was able to create a PacMan image but felt that to be too simple, so I decided to expand on that and add more intricacy. My final shape ended up being a tent (or YouTube Play button depending on the direction it was pointed at). I followed a similar process to target1 and ended up choosing the default learning rate at 0.01 as I found that to be resulting in the most consistent runs.

Dheeraj Viswanadham
(z5204820)



End of Assignment