# PROJECT 1 – CHARACTERS, SPIRALS AND HIDDEN UNIT DYNAMICS

By Lyndon Chang (z5122528)

## PART 1: JAPANESE CHARACTER RECOGNITION

### 1.1 NetLin model

```
class NetLin(nn.Module):
    # linear function followed by log_softmax
    def __init__(self):
        super(NetLin, self).__init__()
        self.lin = nn.Linear(784,10)

    def forward(self, x):
        x = x.view(-1,784) # reshape to have 784 columns
        x = F.log_softmax(self.lin(x), dim=1)
        return x
```

```
[[766.   5.   7.  13.  30.  64.   2.  64.  31.  18.]
 [  7. 669. 107.  17.  29.  24.  59.  14.  23.  51.]
 [  8.  60. 691.  27.  27.  20.  45.  36.  47.  39.]
 [  3.  38.  62. 754.  14.  59.  12.  18.  29.  11.]
 [ 61.  53.  82.  20. 620.  20.  33.  35.  19.  57.]
 [  8.  27. 122.  17.  19. 727.  27.   9.  34.  10.]
 [  5.  24. 145.  10.  26.  26. 719.  20.  10.  15.]
 [ 18.  28.  29.  10.  83.  16.  55. 624.  88.  49.]
 [ 10.  37.  93.  41.   7.  31.  44.   7. 706.  24.]
 [  8.  50.  86.   3.  52.  30.  19.  30.  39. 683.]]

Test set: Average loss: 1.0097, Accuracy: 6959/10000 (70%)
```

### 1.2 NetFull model

```
class NetFull(nn.Module):
    # two fully connected tanh layers followed by log softmax
    def __init__(self):
        super(NetFull, self).__init__()
        self.hiddennodes = 100
        self.main = nn.Sequential(
            nn.Linear(784, self.hiddennodes),
            nn.Tanh(),
            nn.Linear(self.hiddennodes, 10),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        x = x.view(-1,784)
        x = self.main(x)
        return x
```

```
[[847.   5.   3.   7.  30.  31.   2.  38.  30.   7.]
 [  8. 805.  40.   6.  21.   9.  60.   2.  20.  29.]
 [  8.  11. 834.  41.  15.  22.  23.  13.  21.  12.]
 [  4.   8.  37. 901.   2.  19.   9.   3.   8.   9.]
 [ 53.  33.  19.   9. 793.   8.  30.  16.  22.  17.]
 [  7.  10.  75.  10.  14. 832.  27.   2.  17.   6.]
 [  5.  15.  49.  12.  15.   5. 883.   9.   3.   4.]
 [ 20.  14.  22.   6.  32.   8.  31. 808.  27.  32.]
 [ 13.  29.  28.  52.   3.   8.  30.   3. 826.   8.]
 [  2.  28.  50.   5.  35.   9.  16.  14.  13. 828.]]

Test set: Average loss: 0.5356, Accuracy: 8357/10000 (84%)
```

### 1.3 NetConv model

```python
class NetConv(nn.Module):
    # two convolutional layers and one fully connected layer,
    # all using relu, followed by log_softmax
    def __init__(self):
        super(NetConv, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=4, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2,2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=4, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2,2)
        )
        self.layer1 = nn.Linear(2304,10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.reshape(x.size(0),-1)
        x = self.layer1(x)
        x = F.log_softmax(x, dim=1)
        return x
```

```
[[951.   3.   5.   0.  24.   4.   0.   7.   3.   3.]
 [  2. 928.   9.   1.  14.   0.  36.   4.   4.   2.]
 [  9.  11. 855.  42.  14.  12.  29.  11.   7.  10.]
 [  1.   6.  24. 944.   5.   8.   3.   3.   2.   4.]
 [ 17.   8.   7.   6. 916.   2.  21.  10.   9.   4.]
 [  3.  18.  40.   5.   3. 898.  21.   3.   4.   5.]
 [  3.  11.  13.   3.  11.   2. 952.   2.   2.   1.]
 [  4.   7.   5.   2.   6.   1.  11. 932.  10.  22.]
 [  5.  16.   6.   5.   4.   4.   5.   1. 954.   0.]
 [  7.  18.   6.   9.   9.   2.   0.   2.   1. 946.]]

Test set: Average loss: 0.2688, Accuracy: 9276/10000 (93%)
```

### 1.4 Discussion

#### 1.4.a Relative accuracy of the three models

The accuracy of the three models increased with layers added and through the use of activation functions and max pooling.

The first model used, ‘NetLin’, was a single linear function followed by log softmax which produced 70% accuracy. This was surprising as even with such a simple model it was able to classify the characters significantly better than a random guess.

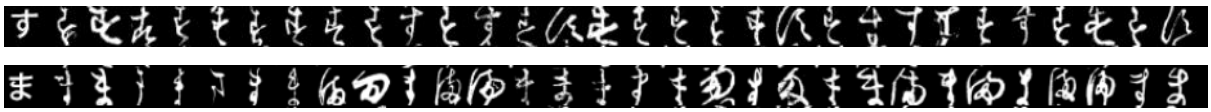
Adding another layer to this model for ‘NetFull’ and using tanh at hidden nodes increased the accuracy to 84%. More layers in a network increases the number of weights and helps to extract more features. Using tanh (hyperbolic tangent) which activation is in [-1,1] allows the model to utilise negative values in classification. This could be advantageous as it removes the “vanishing gradient problem” for datasets with activity near zero.

The final model architecture, ‘NetConv’, which included two convolutional layers and one fully connected layer, all using ReLU activation function and max pooling had the highest accuracy of 93%. The convolutional layers perform convolution operations to output a new matrix of the sum of the products, ReLU converts any number below zero to zero and max pooling passes the highest value amongst the incoming matrix to the output. The improvement of accuracy to 93% can be seen as a result of using the two convolutional layers with ReLU activation and max pooling.

#### 1.4.b Confusion matrix analysis

```
[[766.   5.   7.  13.  30.  64.   2.  64.  31.  18.]
 [  7. 669. 107.  17.  29.  24.  59.  14.  23.  51.]
 [  8.  60. 691.  27.  27.  20.  45.  36.  47.  39.]
 [  3.  38.  62. 754.  14.  59.  12.  18.  29.  11.]
 [ 61.  53.  82.  20. 620.  20.  33.  35.  19.  57.]
 [  8.  27. 122.  17.  19. 727.  27.   9.  34.  10.]
 [  5.  24. 145.  10.  26.  26. 719.  20.  10.  15.]
 [ 18.  28.  29.  10.  83.  16.  55. 624.  88.  49.]
 [ 10.  37.  93.  41.   7.  31.  44.   7. 706.  24.]
 [  8.  50.  86.   3.  52.  30.  19.  30.  39. 683.]]

Test set: Average loss: 1.0097, Accuracy: 6959/10000 (70%)
```

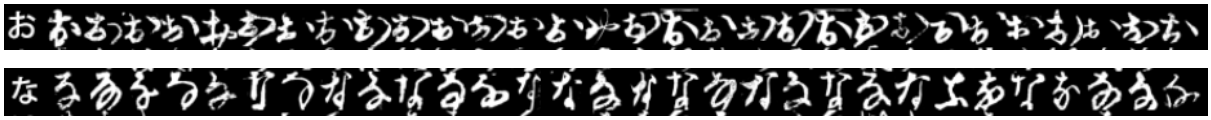
The ‘NetLin’ confusion matrix above demonstrates that the worst classified character was 4=”na” at ~62% (red box). This confusion matrix also shows that 2=”su” was most likely mistaken for 1=”ki”, 5=”ha” and 6=”ma”.



“su” was mistaken for “ma” 145 times. In the figures above you can see that both characters have similar features such as vertical lines in the centre. This could be a reason for the wrong classifications.

```
[[847.   5.   3.   7.  30.  31.   2.  38.  30.   7.]
 [  8. 805.  40.   6.  21.   9.  60.   2.  20.  29.]
 [  8.  11. 834.  41.  15.  22.  23.  13.  21.  12.]
 [  4.   8.  37. 901.   2.  19.   9.   3.   8.   9.]
 [ 53.  33.  19.   9. 793.   8.  30.  16.  22.  17.]
 [  7.  10.  75.  10.  14. 832.  27.   2.  17.   6.]
 [  5.  15.  49.  12.  15.   5. 883.   9.   3.   4.]
 [ 20.  14.  22.   6.  32.   8.  31. 808.  27.  32.]
 [ 13.  29.  28.  52.   3.   8.  30.   3. 826.   8.]
 [  2.  28.  50.   5.  35.   9.  16.  14.  13. 828.]]

Test set: Average loss: 0.5356, Accuracy: 8357/10000 (84%)
```

The ‘NetFull’ confusion matrix above demonstrates that the worst classified character was also 4=”na” (red box). This confusion matrix also shows that 2=”su” was most likely mistaken for 6=”ma” (blue box) similar to the results from ‘NetLin’. Another character which was more likely to be mistaken was 4=”na” for 0=”o” (green box).



The figures above compare 4=”na” with 0=”o”. It can be seen that both characters have curved strokes on the right which could be the main reason for misclassifications.

```
[[951.   3.   5.   0.  24.   4.   0.   7.   3.   3.]
 [  2. 928.   9.   1.  14.   0.  36.   4.   4.   2.]
 [  9.  11. 855.  42.  14.  12.  29.  11.   7.  10.]
 [  1.   6.  24. 944.   5.   8.   3.   3.   2.   4.]
 [ 17.   8.   7.   6. 916.   2.  21.  10.   9.   4.]
 [  3.  18.  40.   5.   3. 898.  21.   3.   4.   5.]
 [  3.  11.  13.   3.  11.   2. 952.   2.   2.   1.]
 [  4.   7.   5.   2.   6.   1.  11. 932.  10.  22.]
 [  5.  16.   6.   5.   4.   4.   5.   1. 954.   0.]
 [  7.  18.   6.   9.   9.   2.   0.   2.   1. 946.]]

Test set: Average loss: 0.2688, Accuracy: 9276/10000 (93%)
```

The ‘NetConv’ confusion matrix above demonstrates that the worst classified character was 2=”su” (red box). This confusion matrix also shows that 2=”su” was most likely mistaken for 3=”tsu” (blue box).

The figure to the left compares 2=”su” with 3=”tsu”. Shown in the blue boxes, some variations of “tsu” are very similar to “su” and dissimilar to the single stroke character it should be. This could be the reason for the highest number of wrong classifications from this model.

**1.4.c Experimenting other architectures/metaparameters**

Using the 'NetConv' model, the sigmoid, tanh and reLU activation functions were compared.

```
Test set: Average loss: 0.6258, Accuracy: 8054/10000 (81%)
```
Using sigmoid

```
Test set: Average loss: 0.3029, Accuracy: 9089/10000 (91%)
```
Using Tanh

```
Test set: Average loss: 0.2688, Accuracy: 9276/10000 (93%)
```
Using reLU

Using the tanh activation function gave the model at 10% increase in accuracy compared to the sigmoid function. This significant increase could be due to this dataset having activity around zero which when using the sigmoid activation function creates the "vanishing gradient problem" making the network harder to train. Tanh removes this problem as its activation is in the set [-1,1], allowing negative values to contribute to training the network. There was also an increase of 2% accuracy from using ReLU over tanh. ReLU is a nonlinear function which activation is in [0, inf] which could have contributed to the increase of accuracy due to activations being set higher and not as dense compared to the set between [-1,1].

```
Test set: Average loss: 0.2305, Accuracy: 9474/10000 (95%)
```
Using reLU with BatchNorm2d

The final improvement of accuracy was made by using batch normalisation in the two convolutional layers. This technique standardises the inputs to a layer for each mini-batch and allows the network to be trained much faster.

Using the original 'NetConv' model an increase of 1% accuracy was obtained by increasing the learning rate.

```
Test set: Average loss: 0.2651, Accuracy: 9377/10000 (94%)
```
Learning rate set to 0.02

```
Test set: Average loss: 0.3961, Accuracy: 9384/10000 (94%)
```
Learning rate set to 0.1

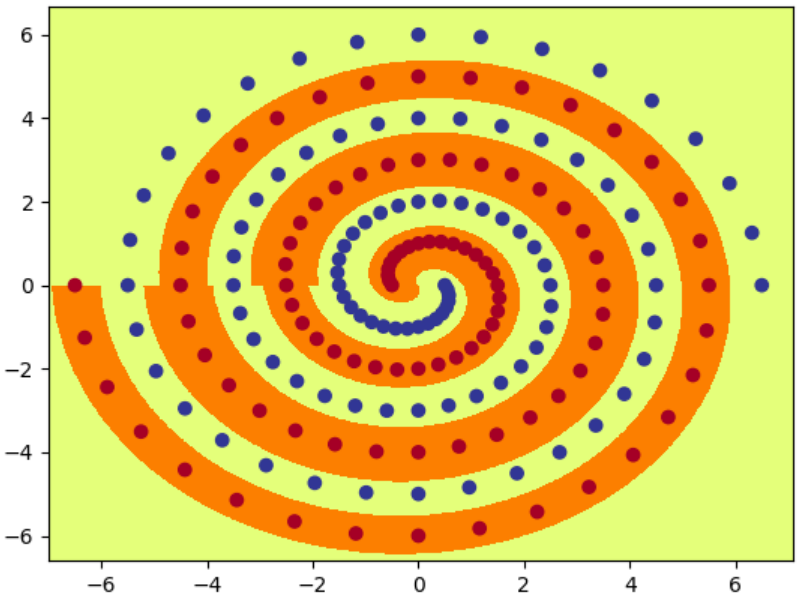
Learning rate controls how significant changes are based on the weights, where a higher learning rate results in more changes over fewer epochs. Changing this metaparameter was only able to influence the accuracy by a little in this model, as when set much higher to 0.1, accuracy was not improved.

## PART 2: TWIN SPIRALS TASK

### 2.1 PolarNet

```
class PolarNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(PolarNet, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(2, num_hid),
            nn.Tanh()
        )
        self.layer2 = nn.Linear(num_hid,1)

    def forward(self, input):
        # convert to polar co-ordinates
        x = input[:,0]
        y = input[:,1]
        r = torch.sqrt((x*x) + (y*y))
        a = torch.atan2(y,x)
        output = torch.stack([r,a], dim=1)

        output = self.main(output)
        output = torch.sigmoid(self.layer2(output))
        return output
```

### 2.2 PolarNet minimum hidden nodes




**Trained with 6 hidden nodes**

### 2.3 RawNet

```python
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(2, num_hid),
            nn.Tanh(),
            nn.Linear(num_hid, num_hid),
            nn.Tanh(),
        )
        self.layer2 = nn.Linear(num_hid,1)

    def forward(self, input):
        output = self.main(input)
        output = torch.sigmoid(self.layer2(output))
        return output
```

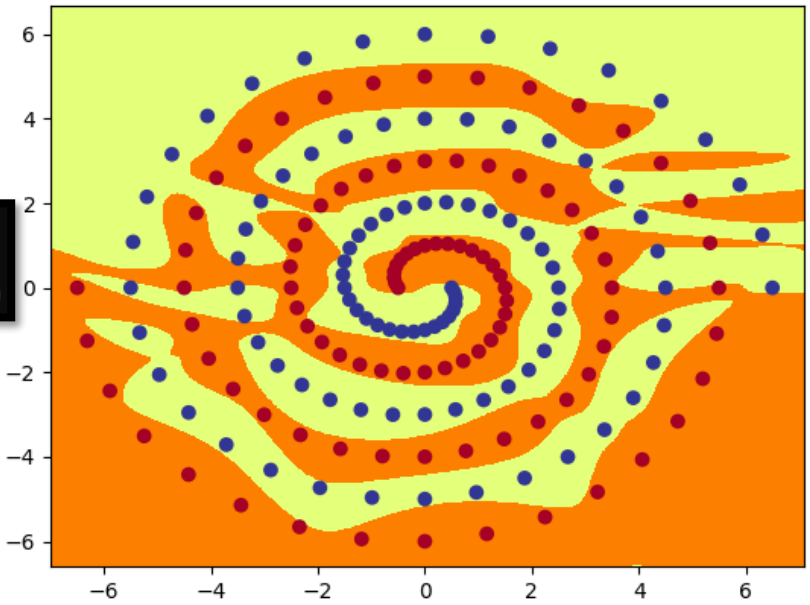
### 2.4 RawNet correct classification within 20,000 epochs

```
ep: 3500 loss: 0.0265 acc: 98.97
ep: 3600 loss: 0.0253 acc: 99.48
ep: 3700 loss: 0.0235 acc: 100.00
```

Trained with 3700 epochs.

Hidden nodes = 12.

Learning rate = 0.02.

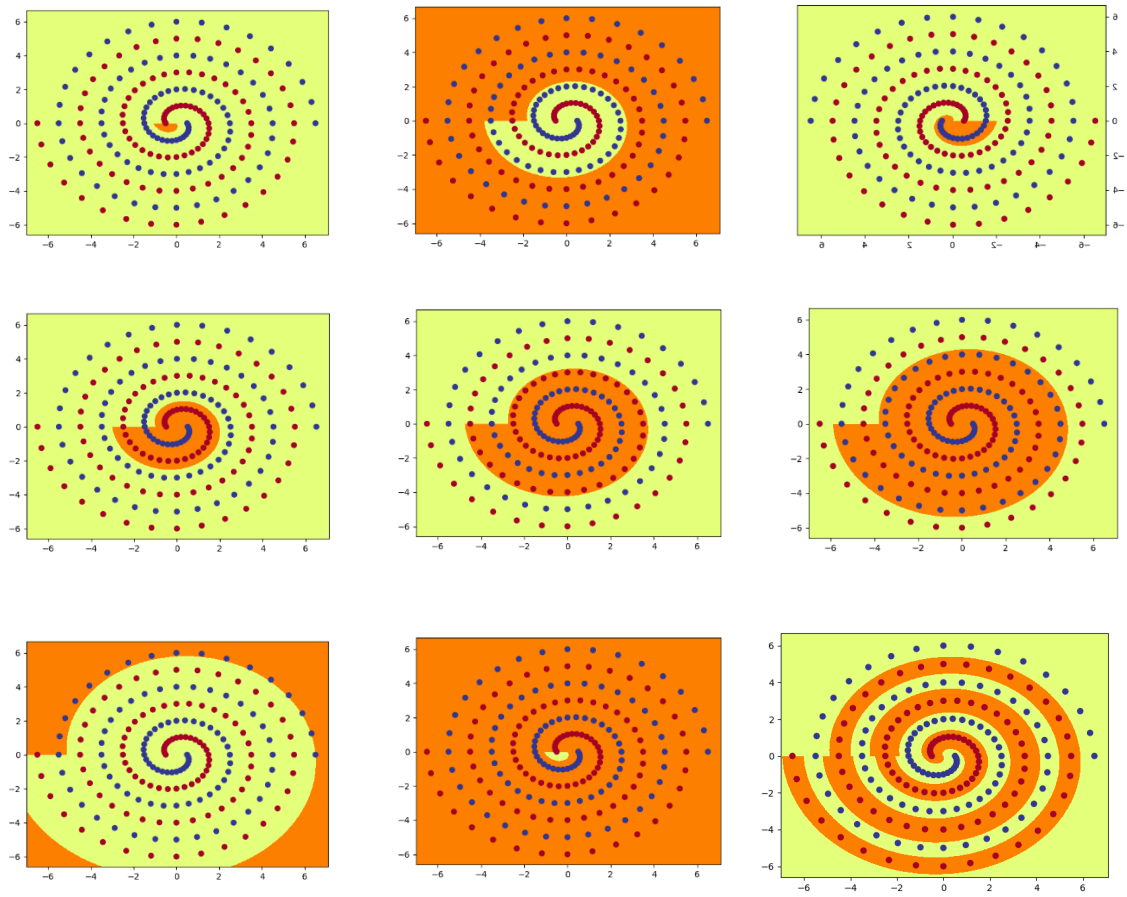



```
python3 spiral_main.py --net raw --hid 12 --lr 0.02
```

**2.5 Graph_hidden**

```
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

        # PolarNet
        netType = isinstance(net, PolarNet)
        if (netType):
            pgrid = []
            for [x,y] in pgrid:
                pgrid.append([math.sqrt(x*x + y*y), math.atan2(y,x)])
            pgrid = torch.tensor(grid)
            output = net.main(pgrid)

        # RawNet
        else:
            if layer >= 1:
                output = net.main(grid)
                output = torch.tanh(output)
            if layer >= 2:
                output = net.layer2(output)
                output = torch.tanh(output)

        net.train() # toggle batch norm, dropout back again
        pred = (output[:, node] >= 0).float()

        # plot function computed by model
        plt.clf()
        activation = pred.cpu().view(yrange.size()[0], xrange.size()[0])
        plt.pcolormesh(xrange,yrange,activation, cmap='Wistia')
```

```
python3 spiral_main.py --net polar --hid 8
```



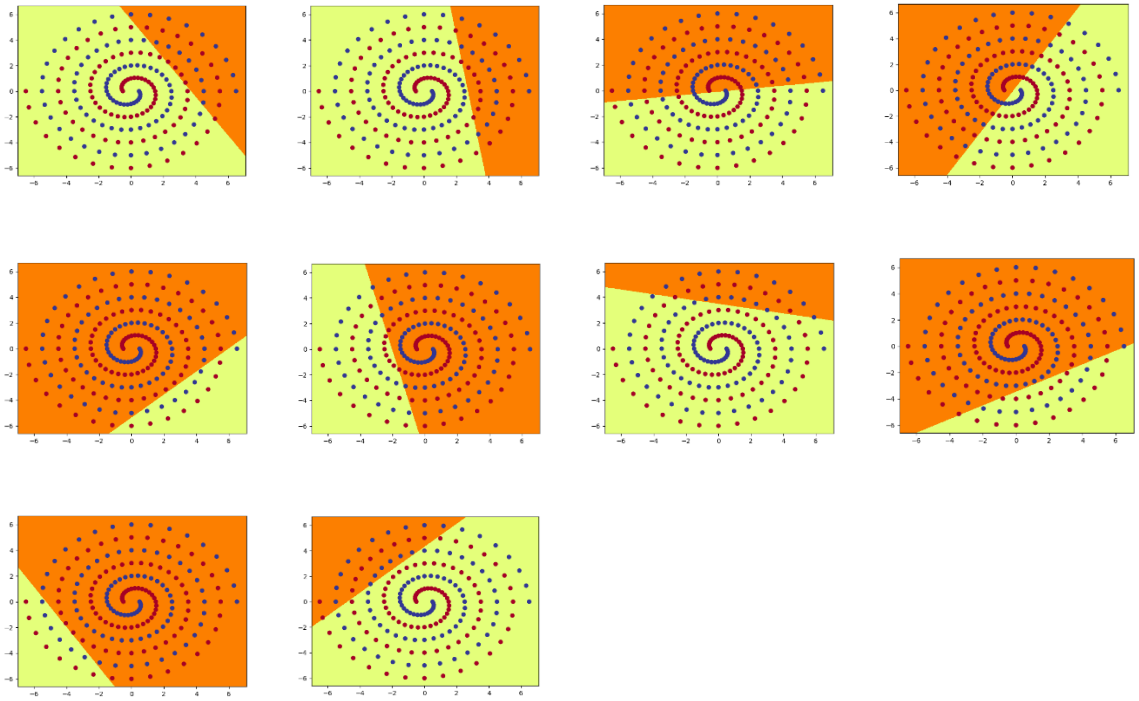
**Output**

**Modified RawNet to allow printing of hidden nodes of both layers**

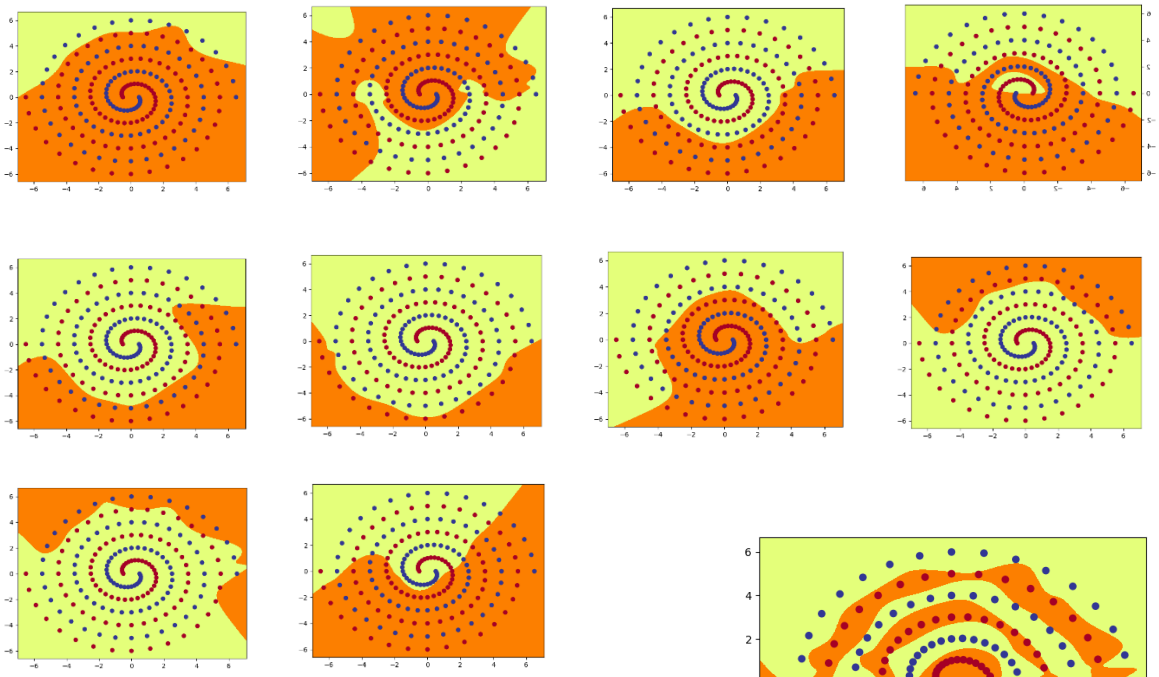
```
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        # Changed to allow printing of first two layers
        self.main = nn.Linear(2, num_hid)
        self.layer2 = nn.Linear(num_hid,num_hid)
        self.layer3 = nn.Linear(num_hid,1)

    def forward(self, input):
        output = torch.tanh(self.main(input))
        output = torch.tanh(self.layer2(output))
        output = torch.sigmoid(self.layer3(output))
        return output
```

```
16:02:44 $> python3 spiral_main.py --net raw --hid 10 --lr 0.02
```

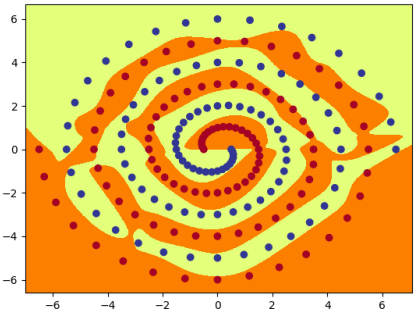
**Layer 1**



**Layer 2**



**Output**



### 2.6 Discussion

#### 2.6.a Qualitative differences

Comparing the plots of the hidden nodes in PolarNet and RawNet, it is clear that the hidden layers in PolarNet are more organised and sequential while the RawNet hidden layers appear more random and chaotic. In PolarNet, the raw data is converted into polar coordinates and the activations of consecutive hidden layers seem to build on the previous one, approximating the different layers of spirals in 'sea-shell' shapes until all of them are mapped out. RawNet however uses the raw data in cartesian coordinates. In the first layer the activations show linear functions cutting across the spirals while the second layers show morphed cut-outs of each consecutive spiral. In this network each consecutive layer it is trying to include points from the previous spiral and exclude all others. This results in slow training initially due to the activations cutting across the spirals, however in the following layer morphed plots map out all the twin spirals.

#### 2.6.b Effect of initial weight size

Weight initialisation is important for a network as it aims to prevent the loss from either being too large or small which affects the rate at which nodes converge. From the data on the right, we can see that an initial weight between 0.15-0.2 produced quickest training times. Using initial weights between this range also proved to be most reliable, as the average epochs of each run was relatively close to each other. In contrast using an initial weight of 0.05 for the model, was unstable. It would train the network very quick (2300 epochs) on some runs while on other runs take over 15,000 epochs.

**Hidden nodes = 10 for all runs**

| INITIAL WEIGHT | EPOCHS TRAINED WITHIN |
|---|---|
| 0.05 | 17300 |
| 0.1 | 7700 |
| 0.15 | 5000 |
| 0.2 | 8200 |
| 0.25 | 22400 |
| 0.3 | 32,300 |
| 0.35 | 50,000 (~98% acc) |
| 0.4 | 50,000 (~97% acc) |

As the weight increased, training times increased. Using initial weights over 0.3, the model was unable to train to 100% accuracy in under 50,000 epochs. In the runs the accuracy would plateau around 95-98% accuracy, unable to find the solution.

From experimenting with initial weight sizes for RawNet it is clear that weights between 0.15-2 are most reliable in terms of speed and success. Weights lighter than this proved to be more unstable and weights heavier were unable to achieve 100% accuracy due to plateaus.

#### 2.6.c Experimenting with other changes

**Increasing batch size**

Increasing the batch size to 194 gives the model more data (examples) per epoch which increases training times. The figures below show more than a double increase in training speeds compared to the original RawNet settings used. The initial weight was also doubled to compensate for the doubling of batch size.

```
train_loader  = torch.utils.data.DataLoader(train_dataset,batch_size=194)
```

```
> python3 spiral_main.py --net raw --hid 12 --lr 0.02 --init 0.3
ep:  100 loss: 0.5928 acc: 63.40
ep:  200 loss: 0.4356 acc: 72.16
ep:  300 loss: 0.3644 acc: 79.90
ep:  400 loss: 0.3317 acc: 82.47
ep:  500 loss: 0.3111 acc: 84.02
ep:  600 loss: 0.2893 acc: 86.08
ep:  700 loss: 0.2615 acc: 87.63
ep:  800 loss: 0.1713 acc: 92.27
ep:  900 loss: 0.0448 acc: 99.48
ep: 1000 loss: 0.0248 acc: 100.00
```

```
> python3 spiral_main.py --net raw --hid 12 --lr 0.02 --init 0.3
ep:  100 loss: 0.6407 acc: 58.76
ep:  200 loss: 0.4475 acc: 72.68
ep:  300 loss: 0.3057 acc: 77.84
ep:  400 loss: 0.1717 acc: 94.33
ep:  500 loss: 0.1160 acc: 94.85
ep:  600 loss: 0.0987 acc: 95.88
ep:  700 loss: 0.0919 acc: 95.88
ep:  800 loss: 0.0725 acc: 95.88
ep:  900 loss: 0.0595 acc: 97.42
ep: 1000 loss: 0.0467 acc: 99.48
ep: 1100 loss: 0.0324 acc: 100.00
```

**ReLU vs Tanh**

Using all the same settings as above, we change the activation function from tanh to reLU to get these results.

```
ep:19700 loss: 0.2215 acc: 88.66
ep:19800 loss: 0.1952 acc: 90.21
ep:19900 loss: 0.3975 acc: 82.47
```

```
ep:19700 loss: 0.1393 acc: 91.75
ep:19800 loss: 0.1067 acc: 95.88
ep:19900 loss: 0.0667 acc: 97.94
```

```
ep:19700 loss: 0.0294 acc: 98.45
ep:19800 loss: 0.0294 acc: 98.45
ep:19900 loss: 0.0294 acc: 98.45
```

From these runs we can see that using tanh over reLU is significantly better as reLU is unable to attain 100% accuracy in under 20,000 epochs. ReLU activation is between [0, inf] while tanh [-1, 1] is able to use negative values to contribute to training the network. This may be a reason for its superiority in this model.

**Adding another hidden layer**

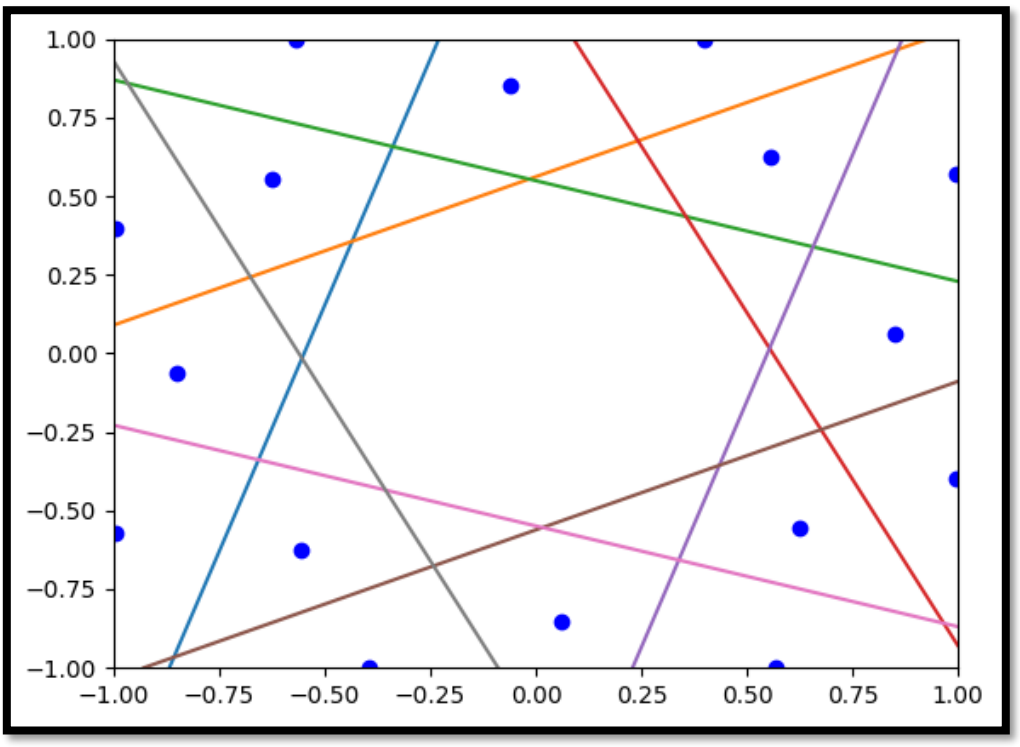
```python
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(2, num_hid),
            nn.Tanh(),
            nn.Linear(num_hid,num_hid),
            nn.Tanh(),
            nn.Linear(num_hid,num_hid),
            nn.Tanh()
        )

        self.layer2 = nn.Linear(num_hid,1)

    def forward(self, input):
        output = self.main(input)
        output = torch.sigmoid(self.layer2(output))
        return output
```

Adding another hidden layer to the model made it more reliable. It was able to train to 100% accuracy in around 1000 epochs on every run.

```
> python3 spiral_main.py --net raw
ep:  100 loss: 0.6005 acc: 59.28
ep:  200 loss: 0.3973 acc: 78.87
ep:  300 loss: 0.2337 acc: 87.63
ep:  400 loss: 0.1311 acc: 95.36
ep:  500 loss: 0.0711 acc: 98.45
ep:  600 loss: 0.0389 acc: 98.45
ep:  700 loss: 0.0296 acc: 98.97
ep:  800 loss: 0.0200 acc: 99.48
ep:  900 loss: 0.0102 acc: 100.00
```

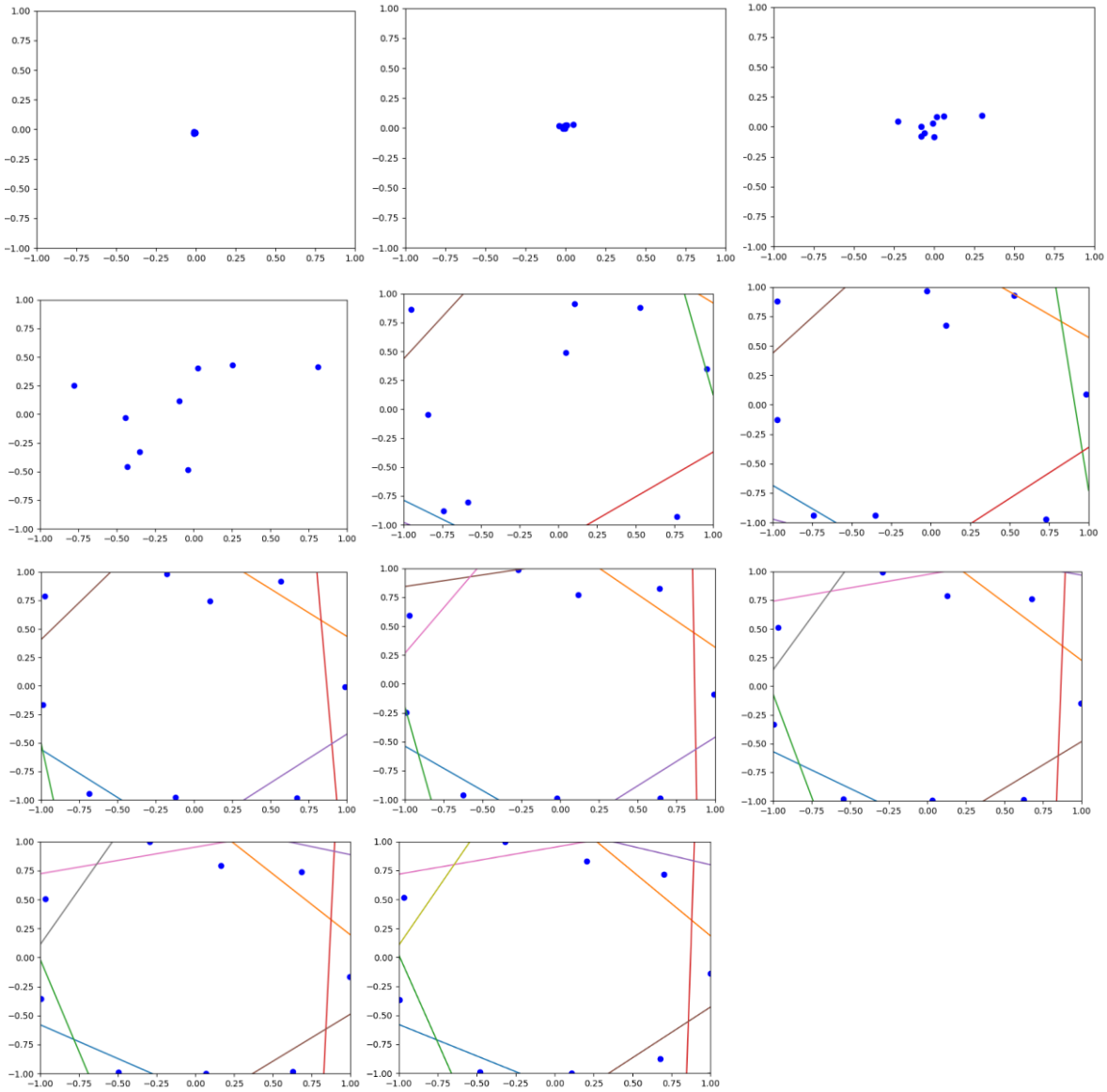
## PART 3: HIDDEN UNIT DYNAMICS

**3.1 Star16**

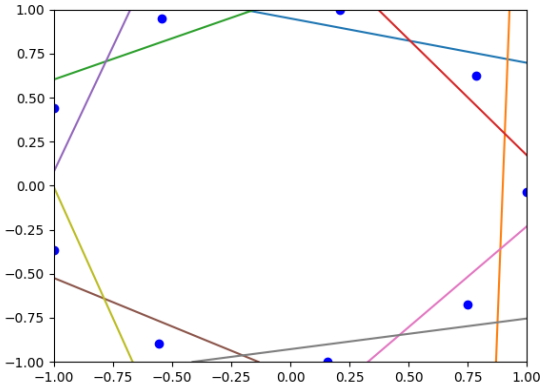


### 3.2 9-2-9 encoder

First 11 images generated (epochs 50 to 3000)



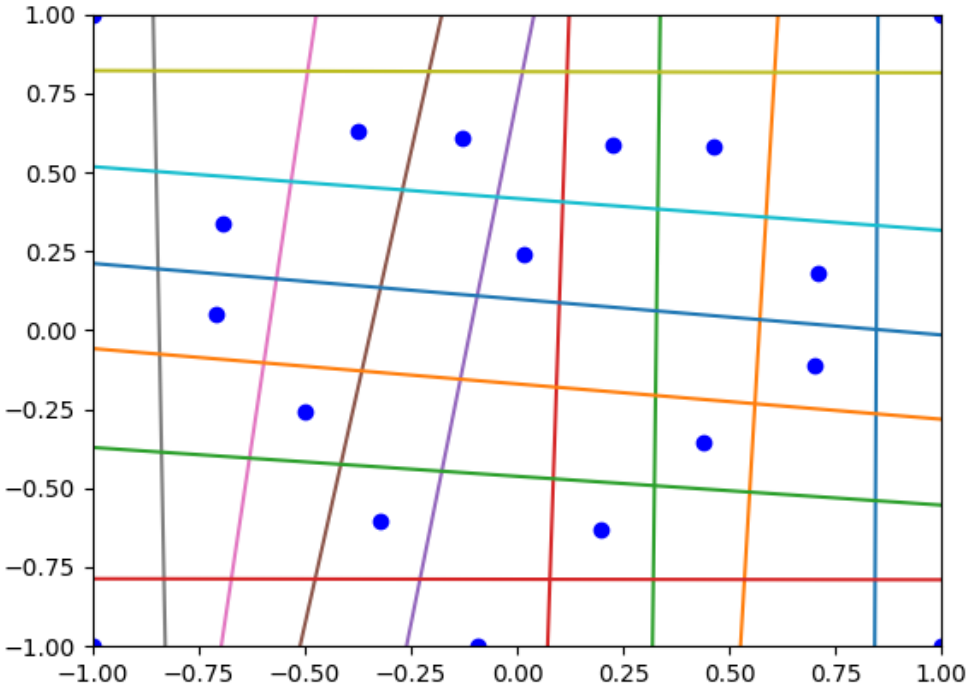
### Final Image



The 9-2-9 encoder is a three layer network that must learn to compress 9 inputs into 2 hidden units and output the original 9 inputs. From these 11 images we can begin to interpret how the network learns this task. Training begins with the hidden unit activations (dots) all centred as the network cannot differentiate between the different inputs. As training progresses, they begin to spread out towards the edges with the lines (output boundaries). This occurs as the network is better able to differentiate the types of inputs as when the lines become further apart there is a larger space for inputs that can be classified as 0 or 1 with respect to a certain node. In the final image, the network has completed the encoding task,

which is represented as the dots and lines evenly spread out as far as possible. This is visualised because since the network is able to distinguish between all inputs, all the dots are evenly separated by the lines, representing the points where inputs are classified as 0 or 1.

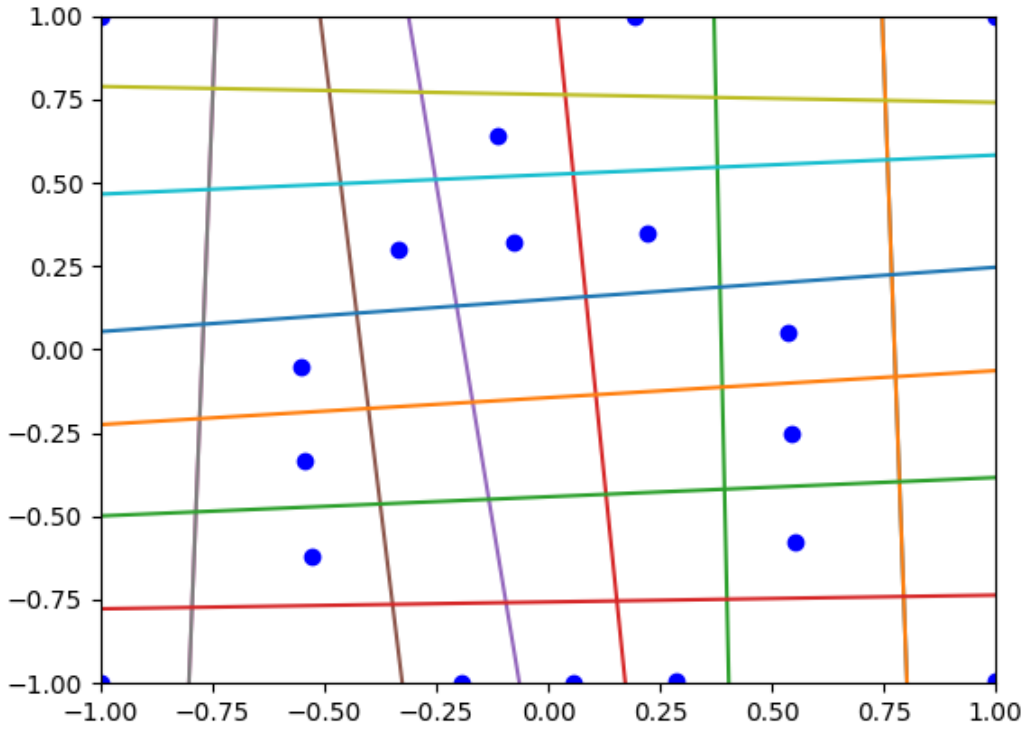
**3.3 Heart18**

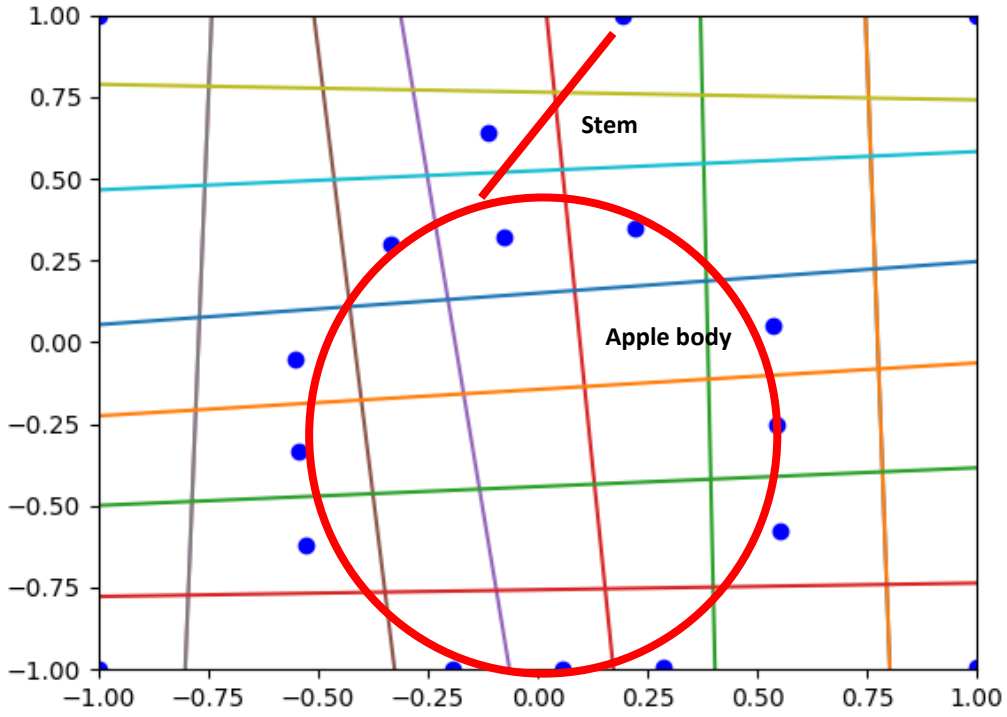



```
heart18 = torch.Tensor(
    [[1,1,1,1,1,1,1,1,1,1,1,1,1,1], # top left corner
     [0,0,1,1,1,1,1,1,0,1,1,1,1,1], # top heart slice
     [0,0,0,1,1,1,1,1,0,1,1,1,1,1], # '' ''
     [0,0,0,0,0,1,1,1,0,1,1,1,1,1], # '' ''
     [0,0,0,0,0,0,1,1,0,1,1,1,1,1], # '' ''
     [0,1,1,1,1,1,1,1,0,0,1,1,1,1], # 2nd slice
     [0,0,0,0,0,0,0,1,0,0,1,1,1,1], # '' ''
     [0,1,1,1,1,1,1,1,0,0,0,1,1,1], # 3rd slice
     [0,0,0,0,0,0,0,1,0,0,0,1,1,1], # '' ''
     [0,0,1,1,1,1,1,1,0,0,0,0,1,1], # 4th slice
     [0,0,0,0,0,0,1,1,0,0,0,0,1,1], # '' ''
     [0,0,0,1,1,1,1,1,0,0,0,0,0,1], # 5th slice left
     [0,0,0,0,0,0,0,0,1,1,1,1,1,1], # top right corner
     [1,1,1,1,1,1,1,1,0,0,0,0,0,0], # bottom left corner
     [0,0,0,0,0,0,0,0,0,0,0,0,0,0], # bottom right corner
     [0,0,0,0,1,1,1,1,0,0,1,1,1,1], # top middle heart
     [0,0,0,0,1,1,1,1,0,0,0,0,0,0], # bottom middle heart
     [0,0,0,0,0,1,1,1,0,0,0,0,0,1]])# 5th slice right
```

**3.4 target1 and target2**

**Target 1: Apple design**



**Target 2:**

**@ symbol**