COMP9444

Assignment 1 Report

Kan-Lin Lu z3417618

---

**Part 1: Japanese Character Recognition**

1. We are requested to perform a linear function followed by log softmax, adjustment made to the code is:

```python
class NetLin(nn.Module):
    # linear function followed by log_softmax
    def __init__(self):
        super(NetLin, self).__init__()
        # INSERT CODE HERE
        # KMNIST dataset with 28x28 image = 784 pixels
        self.linear_function = nn.Linear(784, 10)
        """
        #Originally using sequential
        self.main = nn.Sequential(
            #First use a linear function to transform input size to output size
            nn.Linear(784, 10), # We have input 784 pixel and a output of 10 characters, defining input and output size
            nn.LogSoftmax(dim=1) # Followed by log_softmax
        )
        """

    def forward(self, x):
        x = x.view(x.shape[0], -1) #Reshape
        x = self.linear_function(x) #linear function
        x = F.log_softmax(x, dim = 1) #Log SoftMax
        return x # CHANGE CODE HERE
```

The output result is:

```
[[769.   5.   9.  12.  30.  64.   2.  62.  29.  18.]
 [  7. 670. 108.  18.  27.  22.  58.  14.  24.  52.]
 [  6.  57. 697.  26.  26.  20.  47.  36.  46.  39.]
 [  4.  35.  58. 759.  16.  59.  12.  17.  29.  11.]
 [ 60.  50.  82.  21. 621.  18.  33.  38.  20.  57.]
 [  8.  27. 124.  17.  19. 723.  29.   9.  33.  11.]
 [  5.  22. 146.  10.  27.  24. 723.  21.   8.  14.]
 [ 15.  28.  27.  12.  84.  16.  55. 624.  89.  50.]
 [ 12.  34.  95.  42.   5.  31.  45.   6. 707.  23.]
 [  8.  50.  86.   3.  49.  31.  19.  31.  39. 684.]]

Test set: Average loss: 1.0090, Accuracy: 6977/10000 (70%)
```

2. We are requested to implement a fully connected 2 layer network using tanh at hidden node and log softmax at output node, adjustment made to the code is:

```python
class NetFull(nn.Module):
    # two fully connected tanh layers followed by log softmax
    def __init__(self):
        super(NetFull, self).__init__()
        # INSERT CODE HERE
        self.linear_function_1 = nn.Linear(784, 140)
        self.linear_function_2 = nn.Linear(140, 10)
        """
        #Originally using sequential
        self.main = nn.Sequential(
            nn.Linear(784, 140), # Linear Transform to 140 nodes
            nn.Tanh(), # Tanh Layer
            nn.Linear(140,10), # Linear Transform to 10 nodes
            nn.LogSoftmax(dim=1) # followed by log_softmax
        )
        """
    def forward(self, x):
        x = x.view(x.shape[0], -1)
        x = F.tanh(self.linear_function_1(x)) #tanh as activation funtion
        x = F.log_softmax(self.linear_function_2(x), dim=1) # log softmax as activation function
        return x # CHANGE CODE HERE
```

Note: The number of hidden nodes are varied and the comparison is as follow, the experiment is first conducted by having large jump of 50 number of nodes, and locating the trend, then further finding the range to breakdown.

| Num | 10 | 50 | 100 | 110 | 120 | 130 | 140 | 150 | 200 | 250 |
|---|---|---|---|---|---|---|---|---|---|---|
| Acc | 68% | 82% | 84% | 84% | 84% | 84% | 85% | 85% | 85% | 85% |

From the above table we can observe 140 nodes provides an 85% accuracy, further increasing number of neurons do not have further improvement.

```
[[854.   4.   1.   5.  28.  32.   6.  35.  28.   7.]
 [  4. 814.  35.   5.  21.   8.  59.   5.  22.  27.]
 [  8.  14. 848.  36.  10.  18.  25.   9.  20.  12.]
 [  2.   9.  30. 919.   1.  11.   4.   7.   6.  11.]
 [ 40.  32.  21.   7. 818.  10.  24.  15.  20.  13.]
 [  8.  12.  67.  14.  13. 844.  24.   2.   9.   7.]
 [  3.  19.  41.   6.  20.   7. 886.  11.   2.   5.]
 [ 17.  20.  22.   8.  25.   7.  27. 818.  19.  37.]
 [ 13.  34.  24.  44.   5.   8.  31.   4. 831.   6.]
 [  4.  24.  38.   3.  36.   8.  21.  17.  16. 833.]]

Test set: Average loss: 0.5108, Accuracy: 8465/10000 (85%)
```

3. Implement 2 convolutional layers plus one fully connected layer, all using relu activation function, followed by the output layer. A random setting was set up and a 95% accuracy was reached. Adjustment are as follow:

```
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__()
        # INSERT CODE HERE
        self.convolution_1 = nn.Conv2d(in_channels=1, out_channels=64, kernel_size=5)
        self.max_pooling_1 = nn.MaxPool2d(5, stride=1)
        self.convolution_2 = nn.Conv2d(in_channels=64, out_channels=16, kernel_size=3)
        self.max_pooling_2 = nn.MaxPool2d(3, stride=1)
        self.convo_hidden = nn.Linear(4096, 126)
        self.hidden_out = nn.Linear(126, 10)
        """
        #Originally using sequential
        # 2 Convolution Layer
        self.main = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=5),
            nn.ReLU(),
            nn.MaxPool2d(5, stride=1), # Max Pooling to reduce size of previous, further reduce total operation
            nn.Conv2d(in_channels=64, out_channels=16, kernel_size=3),
            nn.ReLU(),
            nn.MaxPool2d(3, stride=1), # Max Pooling to reduce size, note this affects the upcoming linear layer size
            nn.Dropout(p=0.5),
        )

        #Follow by a lineary layer then output layer
        self.linear_out = nn.Sequential(
            nn.Linear(4096, 126),
            nn.ReLU(),
            nn.Linear(126, 10),
            nn.LogSoftmax(dim=1)
        )
        """
    def forward(self, x):
        #First layer
        x = self.max_pooling_1(F.relu(self.convolution_1(x)))
        #Second layer
        x = self.max_pooling_2(F.relu(self.convolution_2(x)))
        #Last layer
        #Now we need to flatten before inputting to fully connected layer
        x = F.relu(self.convo_hidden(x.view(x.shape[0], -1)))
        x = F.log_softmax(self.hidden_out(x), dim=1)
        return x
```

Confusion Matrix and Accuracy:

```
[[961.   4.   0.   1.  16.   0.   0.  13.   2.   3.]
 [  6. 936.   6.   0.   7.   2.  25.   4.   5.   9.]
 [ 10.  10. 899.  20.  11.   6.  26.   5.   7.   6.]
 [  2.   0.   5. 967.   2.   5.   9.   2.   4.   4.]
 [ 16.   6.   2.   4. 942.   2.   4.   5.   7.  12.]
 [  3.   5.  23.   2.   1. 937.  18.   1.   2.   8.]
 [  2.  12.   2.   3.   1.   2. 974.   2.   0.   2.]
 [  7.   5.   1.   3.   9.   0.  12. 939.   3.  21.]
 [  7.  10.   5.   3.  21.   5.   5.   2. 939.   3.]
 [  4.   5.   3.   2.   7.   1.   0.   0.   2. 976.]]

Test set: Average loss: 0.2307, Accuracy: 9470/10000 (95%)
```

Note: We try to keep kernel size for respective Conv2d and MaxPool2d consistent with one another. And after research the common Kernel size are 5 and 3, and consider it is a squared image, we use square kernel as well. It is also researched that first kernel size should be bigger, as extract first abstract information from raw image provides more useful abstraction. Note we also keep 1$^{st}$ Convolutional Layer Output smaller than 2$^{nd}$.

Note: Further test were conducted, and provided in part c.

4. Part a)

Overall, we observe an increase in accuracy with linear (70%), to 2 layer(85%) then further to convolutional network(95%). This aligns with the complexity of each network, in other words, a relationship between accuracy and complexity.

In more specific, considering the nature of the input, linear function and 2 layers network proposed in question 1 and 2, do not take into account the higher-level feature. Notice we flatten the input to 784 neurons at the very first layer, although we establish more layer in question 2, but that only takes into account the pairs of relationship.

Whereas, when using kernel method or convolutional network, a sliding window is used to grasp neighbourhood information that forms a higher-level understanding of the image. Then once the features are extracted, reducing to abstract vectors than flatten into a linear layer, further an output layer – providing better prediction.

Part b)

Without Looking at the part c), Let's first converting each questions' confusion matrix to a excel table and identify the highest

Question 1:

| | o | ki | su | tsu | na | ha | ma | ya | re | wo |
|---|---|---|---|---|---|---|---|---|---|---|
| o | 769 | 5 | 9 | 12 | 30 | 64 | 2 | 62 | 29 | 18 |
| ki | 7 | 670 | 108 | 18 | 27 | 22 | 58 | 14 | 24 | 52 |
| su | 6 | 57 | 697 | 26 | 26 | 20 | 47 | 36 | 46 | 39 |
| tsu | 4 | 35 | 58 | 759 | 16 | 59 | 12 | 17 | 29 | 11 |
| na | 60 | 50 | 82 | 21 | 621 | 18 | 33 | 38 | 20 | 57 |
| ha | 8 | 27 | 124 | 17 | 19 | 723 | 29 | 9 | 33 | 11 |
| ma | 5 | 22 | 146 | 10 | 27 | 24 | 723 | 21 | 8 | 14 |
| ya | 15 | 28 | 27 | 12 | 84 | 16 | 55 | 624 | 89 | 50 |
| re | 12 | 34 | 95 | 42 | 5 | 31 | 45 | 6 | 707 | 23 |
| wo | 8 | 50 | 86 | 3 | 49 | 31 | 19 | 31 | 39 | 684 |

False Identification: Red – Represents value above 100, Pink Represents Value above 80

Positive Identification: Yellow

Question 2:

| | o | ki | su | tsu | na | ha | ma | ya | re | wo |
|---|---|---|---|---|---|---|---|---|---|---|
| o | 854 | 4 | 1 | 5 | 28 | 32 | 6 | 35 | 28 | 7 |
| ki | 4 | 814 | 35 | 5 | 21 | 8 | 59 | 5 | 22 | 27 |
| su | 8 | 14 | 848 | 36 | 10 | 18 | 25 | 9 | 20 | 12 |
| tsu | 2 | 9 | 30 | 919 | 1 | 11 | 4 | 7 | 6 | 11 |
| na | 40 | 32 | 21 | 7 | 818 | 10 | 24 | 15 | 20 | 13 |
| ha | 8 | 12 | 67 | 14 | 13 | 844 | 24 | 2 | 9 | 7 |
| ma | 3 | 19 | 41 | 6 | 20 | 7 | 886 | 11 | 2 | 5 |
| ya | 17 | 20 | 22 | 8 | 25 | 7 | 27 | 818 | 19 | 37 |
| re | 13 | 34 | 24 | 44 | 5 | 8 | 31 | 4 | 831 | 6 |
| wo | 4 | 24 | 38 | 3 | 36 | 8 | 21 | 17 | 16 | 833 |

False Identification: Red – Represents value above 50, Pink Represents Value above 40

Positive Identification: Yellow

Question 3:

| | o | ki | su | tsu | na | ha | ma | ya | re | wo |
|---|---|---|---|---|---|---|---|---|---|---|
| o | 961 | 4 | 0 | 1 | 16 | 0 | 0 | 13 | 2 | 3 |
| ki | 6 | 936 | 6 | 0 | 7 | 2 | 25 | 4 | 5 | 9 |
| su | 10 | 10 | 899 | 20 | 11 | 6 | 26 | 5 | 7 | 6 |
| tsu | 2 | 0 | 5 | 967 | 2 | 5 | 9 | 2 | 4 | 4 |
| na | 16 | 6 | 2 | 4 | 942 | 2 | 4 | 5 | 7 | 12 |
| ha | 3 | 5 | 23 | 2 | 1 | 937 | 18 | 1 | 2 | 8 |
| ma | 2 | 12 | 2 | 3 | 1 | 2 | 974 | 2 | 0 | 2 |
| ya | 7 | 5 | 1 | 3 | 9 | 0 | 12 | 939 | 3 | 21 |
| re | 7 | 10 | 5 | 3 | 21 | 5 | 5 | 2 | 939 | 3 |
| wo | 4 | 5 | 3 | 2 | 7 | 1 | 0 | 0 | 2 | 976 |

False Identification: Red – Represents value above 20, Pink Represents Value above 10

Positive Identification: Yellow

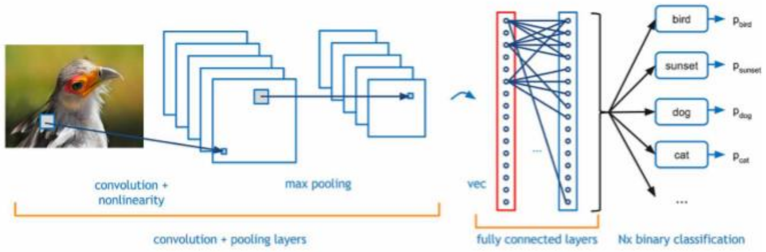
From these we can observe that overall, the misclassification has improved by the reduction in False Identification range.

Specifically, for which characters, one can observe the respective rows and columns, anything that is not in diagonal referring to either mislabelling. Therefore, there exist always mislabelling, unless a value of zero is observed. What we want to achieve is having low number of mislabelling. Specifically:

- For Question 1, we see “Su” is mislabelled with all other characters, mainly “ki”, “na”, “ha”, “ma”, “re” and “wo” have the highest mislabel rate. “na” is mislabelled with “ya” at high rate. And “re” is also mislabelled with “ya” at high rate. Note there is no, 0 mislabelling exists in this table. Considering the simplicity of this network, we can say that the reason for higher mislabelling is due to pixel level identification.
- For Question 2, we see slight improvement of overall mislabelling value, however still no existence of 0 mislabelling. The highest mislabelling still exists on “Su” column, referring back to previous statement, even though an extra layer is attached, the overall network still does not grasp a “neighbourhood” information. All the other characters still have slight chance of been mislabelled as the others.
- For Question 3, we see a huge improvement of overall mislabelling value. There is existence of 0 mislabel value, meaning there is none that is mislabelled as the other. In this table, the mislabelling is more spread out, this is due to a better network that grasp the higher-level abstract understanding of the image, hence better prediction. Taking “Ki” column as an example, it is not been mislabelled as “tsu”, but at low rate mislabelled as all other characters.

<u>Part c)</u>

Considering the following illustration from lecture note:




We set to examine different parameters in affecting the accuracy, and in order to do so, we keep certain parameters consistent and adjust the selected parameter. Note the observation here is specifically for given images, i.e. KMNIST, as other image set might have different behaviour. However this provides and understanding of the theoretical concept of convolutional network. Considering the nature of the given network, we first test adjusting stride of Max Pooling:

**Max Pooling Adjustment**

| Constant Parameters | | | |
|---|---|---|---|
| 1st Conv Output Ch | 64 | Max Pool Window | 5 |
| 2nd Conv Output Ch | 16 | Max Pool Window | 3 |
| Adjusting Parameter | | | |
| Stride | 1 | 2 | 3 |
| Acc | 96% | 91% | 80% |

From this we can tell that decrease value of stride for max pooling improves overall accuracy.

**Adjust 1$^{st}$ Conv Output Channels**

| Constant Parameters | | | |
|---|---|---|---|
| Stride | 1 | | |
| 1st Conv Max | 5 | | |
| 2nd Conv Max | 3 | | |
| Last Layer (From Research) | Mean of Desired Output plus 2nd Conv output Ch | | |
| Adjusting Parameters | | | |
| **2nd Conv** | **8** | | |
| 1st Conv | 32 | 64 | 128 |
| Acc | 94% | 94% | 94% |
| **2nd Conv** | **16** | | |
| 1st Conv | 32 | 64 | 128 |
| Acc | 95% | 95% | 96% |
| **2nd Conv** | **32** | | |
| 1st Conv | 32 | 64 | 128 |
| Acc | 96% | 96% | 96% |

From this we can conclude 2 things:

- 1$^{st}$ Convolution layer channel size has slight effect on overall accuracy.
- 2$^{nd}$ Convolution layer channel size has more major effect on overall accuracy, considering in theory this layer is the higher-level information abstraction.

Now we also look at the epoch level influence:

| Epoch | 2nd Conv = 8 | | | 2nd Conv = 16 | | | 2nd Conv = 32 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 32 | 64 | 128 | 32 | 64 | 128 | 32 | 64 | 128 |
| 1 | 86 | 87 | 88 | 86 | 86 | 89 | 87 | 88 | 89 |
| 2 | 90 | 91 | 92 | 91 | 91 | 92 | 92 | 93 | 93 |
| 3 | 92 | 93 | 93 | 93 | 93 | 94 | 94 | 94 | 94 |
| 4 | 93 | 93 | 93 | 93 | 94 | 94 | 94 | 94 | 95 |
| 5 | 93 | 94 | 94 | 94 | 94 | 95 | 95 | 94 | 95 |
| 6 | 94 | 94 | 94 | 94 | 94 | 95 | 95 | 95 | 95 |
| 7 | 94 | 94 | 94 | 94 | 94 | 95 | 95 | 95 | 95 |
| 8 | 94 | 94 | 94 | 94 | 95 | 95 | 95 | 95 | 96 |
| 9 | 94 | 94 | 94 | 95 | 95 | 96 | 95 | 95 | 96 |
| 10 | 94 | 94 | 94 | 95 | 95 | 96 | 96 | 96 | 96 |

From this we can conclude 2 things:

- Increase 1$^{st}$ Convolution layer output channel size allows the overall accuracy to converge faster
- Increase 2$^{nd}$ Convolution layer output channel size allows baseline accuracy to increase.

**Adjust Linear Layer nodes:**

From above we can tell that with 128 output channel in 1$^{st}$ layer and 32 output channel in 2$^{nd}$ layer provides a more stable network. However, it comes at high computational cost considering number of nodes involved. Therefore, as per the lecture node, one way in tackling is reducing last hidden layer number of nodes:

| Constant Parameters | | | | |
|---|---|---|---|---|
| Stride | 1 | | | |
| 1st Conv Max | 5 | | | |
| 2nd Conv Max | 3 | | | |
| 1st Conv | 128 | | | |
| 2nd Conv | 32 | | | |
| Adjusting Parameters | | | | |
| Last Hidden Layer | 4101 | 2050 | 1025 | 512 |
| Acc | 96 | 96 | 96 | 96 |

From above, it looks like it does not have an influence on overall performance, however take a look at the below per epoch output:

| Epoch | Number of nodes in last hidden layer | | | |
|---|---|---|---|---|
| | 4101 | 2050 | 1025 | 512 |
| 1 | 89 | 88 | 88 | 88 |
| 2 | 93 | 93 | 92 | 92 |
| 3 | 94 | 94 | 94 | 94 |
| 4 | 95 | 95 | 95 | 94 |
| 5 | 95 | 95 | 95 | 95 |
| 6 | 95 | 95 | 95 | 95 |
| 7 | 95 | 95 | 95 | 95 |
| 8 | 96 | 95 | 95 | 95 |
| 9 | 96 | 95 | 95 | 95 |
| 10 | 96 | 96 | 96 | 96 |

From this we can tell that reducing number of nodes in last hidden layer reduces convergence speed.

**Apply Drop out:**

As another experiment, drop out is applied after pooling at the second layer, which avoids feature been omitted.

| Constant Parameters | | | |
|---|---|---|---|
| Stride | 1 | | |
| 1st Conv Max | 5 | | |
| 2nd Conv Max | 3 | | |
| 1st Conv | 128 | | |
| 2nd Conv | 32 | | |
| Adjusting Parameters | | | |
| **Last Layer** | **4101** | | |
| Rate | 1 | 0.7 | 0.5 |
| Acc | 96 | 96 | 96 |
| **Last Layer** | **2050** | | |
| Rate | 1 | 0.7 | 0.5 |
| Acc | 96 | 96 | 96 |

| Last Layer | 1025 | | |
|---|---|---|---|
| Rate | 1 | 0.7 | 0.5 |
| Acc | 96 | 96 | 96 |
| **Last Layer** | **512** | | |
| Rate | 1 | 0.7 | 0.5 |
| Acc | 96 | 96 | 96 |

Initially, from this we can conclude that applying drop out assists in reducing size of last layer. As with 0 to 0.5 drop rate, we see no influence over the overall accuracy, however, a deeper look is taken into per epoch influence:

| Epoch | Last Layer = 4101 | | | Last Layer = 2050 | | | Last Layer = 1025 | | | Last Layer = 512 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 0.7 | 0.5 | 1 | 0.7 | 0.5 | 1 | 0.7 | 0.5 | 1 | 0.7 | 0.5 |
| 1 | 89 | 87 | 88 | 88 | 87 | 88 | 88 | 87 | 87 | 88 | 86 | 88 |
| 2 | 93 | 91 | 92 | 93 | 91 | 92 | 92 | 91 | 92 | 92 | 92 | 92 |
| 3 | 94 | 93 | 94 | 94 | 93 | 93 | 94 | 93 | 94 | 94 | 93 | 94 |
| 4 | 95 | 94 | 94 | 95 | 94 | 95 | 95 | 94 | 95 | 94 | 94 | 94 |
| 5 | 95 | 94 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 |
| 6 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 96 | 95 |
| 7 | 95 | 95 | 96 | 95 | 95 | 96 | 95 | 95 | 95 | 95 | 95 | 96 |
| 8 | 96 | 96 | 96 | 95 | 96 | 96 | 95 | 96 | 95 | 95 | 96 | 96 |
| 9 | 96 | 96 | 96 | 95 | 96 | 96 | 95 | 96 | 96 | 95 | 96 | 96 |
| 10 | 96 | 96 | 96 | 96 | 96 | 96 | 96 | 96 | 96 | 96 | 96 | 96 |

From this we can tell that having 0.5 drop out rate assists in avoiding overfitting onto the training set, therefore, having faster convergence to 96% accuracy. This aligns with the theoretical concept given in lecture. Another thing to test out is Apply drop out at first layer

**Sequential VS Module**

We also examined the difference between using sequential and modules for each questions, and it is found that no difference were observed, this is as per research, where Sequential stacks up the module and make the code “cleaner”.

Another function call ModuleList can also be used, but that is mainly for iterating purpose.

**Part 2: Twin Spirals Task**

1. In this question we are required to implement a Polar Net that first convert x and y coordinates to r and a, than apply Tanh with Sigmoid. Adjustment is as follow:

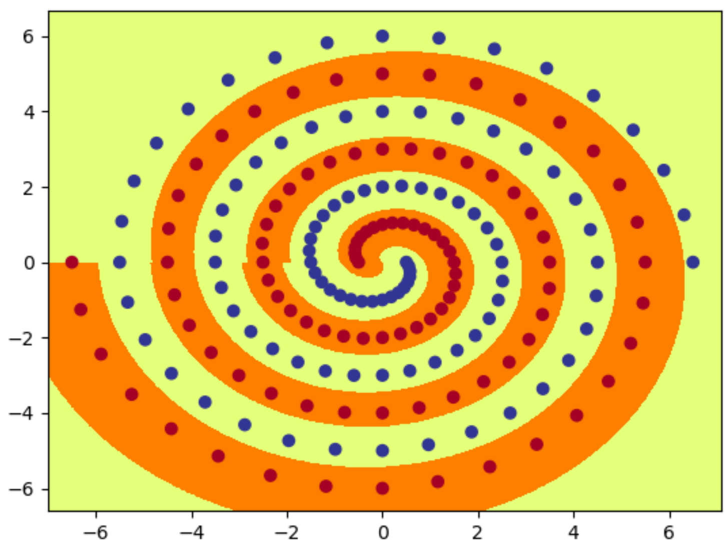
```
class PolarNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(PolarNet, self).__init__()
        # INSERT CODE HERE
        self.in_hidden = nn.Linear(2, num_hid)
        self.hidden_out = nn.Linear(num_hid, 1)
        """
        #Originally using Sequential
        self.main = nn.Sequential(
            nn.Linear(2, num_hid),
            nn.Tanh(),
            nn.Linear(num_hid, 1),
            nn.Sigmoid()
        )
        """

    def forward(self, input):
        # First we convert the input to polar co-ordinates
        x,y = input[:,0], input[:, 1]
        r,a = torch.sqrt((x**2) + (y**2)), torch.atan2(y,x) # Might need to reshape
        r,a = r.view(r.shape[0], -1), a.view(a.shape[0], -1)
        #Now we have r and a, we need to concatenate into 1
        output = torch.cat((r,a), dim=1)
        #Network:
        output = torch.tanh(self.in_hidden(output))
        output = torch.sigmoid(self.hidden_out(output))
        return output
```

2. Find Minimum Number of hidden nodes using provided Polar Net. Following provides the initial run of number of nodes from 10 to 2, as we can observe number of nodes 7 seems to be the optimal choice:

| Number of Nodes | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 |
|---|---|---|---|---|---|---|---|---|---|
| Number of Epochs | 2800 | 2700 | 11400 | 17800 | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ |

The following provides the attaching polar_out.png for number of nodes = 7:




Now considering what the question is hinting, there may be variation in different runs, therefore, we further conduct the experiments for 10 times:

| Node | 6 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Iteration Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 14400 | 12600 | 10600 | 4500 | 20800 | 6600 | 99900+ | 99900+ | 14300 | 11100 |
| Loss | 0.0306 | 0.1028 | 0.0128 | 0.0572 | 0.0463 | 0.0178 | 0.0548 | 0.0234 | 0.0347 | 0.0216 |
| Node | 7 | | | | | | | | | |
| Iteration Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 3200 | 12400 | 5800 | 5200 | 9500 | 5200 | 6400 | 10500 | 5700 | 17800 |
| Loss | 0.0239 | 0.0295 | 0.0171 | 0.0177 | 0.0187 | 0.0773 | 0.0959 | 0.0261 | 0.102 | 0.0483 |
| Node | 8 | | | | | | | | | |
| Iteration Number | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 13100 | 13400 | 13300 | 99900+ | 14000 | 5300 | 12800 | 99900+ | 14700 | 99900+ |
| Loss | 0.0205 | 0.0215 | 0.0184 | 0.011 | 0.0286 | 0.0802 | 0.0124 | 0.011 | 0.0297 | 0.0234 |

From that we can conclude that 7 hidden nodes seem to be stable, therefore we choose 7, as out of 10 iterations they all under 20000 Epoch to converge.

3. In this question we are requested to use x and y coordinates in directly learning the input, with 2 tanh layer and 1 sigmoid activation at output layer. Number of hidden nodes between two hidden layer remains consistent. The adjustment is as follow:

```python
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        # INSERT CODE HERE
        self.in_hidden1 = nn.Linear(2, num_hid)
        self.hidden1_hidden2 = nn.Linear(num_hid, num_hid)
        self.hidden2_out = nn.Linear(num_hid, 1)
        """
        #Originally using Sequential
        self.main = nn.Sequential(
            #Two fully connect Tanh
            nn.Linear(2, num_hid),
            nn.Tanh(),
            nn.Linear(num_hid, num_hid),
            nn.Tanh(),
            #Linear out with Sigmoid
            nn.Linear(num_hid, 1),
            nn.Sigmoid()
        )
        """
    def forward(self, input):
        output = torch.tanh(self.in_hidden1(input))
        output = torch.tanh(self.hidden1_hidden2(output))
        output = torch.sigmoid(self.hidden2_out(output))
        return output
```

4. Find minimum number of hidden nodes and size of initial weights such that this RawNet learns correctly classify all of the training data within 20000 epochs. In order to perform the experiment, the best approach is for every number of hidden nodes we run different initial weight and observe number of epochs required. Discussion is provided in question 6 Part b, as the question did not ask for a discussion.

| Node | 30 | | | | | |
|---|---|---|---|---|---|---|
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 800 | 1000 | 800 | 1100 | 2800 | 99900+ |
| Node | 25 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 1000 | 1600 | 1500 | 1500 | 4100 | 99900+ |
| Node | 20 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 3200 | 1700 | 2600 | 2000 | 5000 | 99900+ |
| Node | 15 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 4200 | 4700 | 3000 | 2200 | 4500 | 99900+ |
| Node | 10 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 99900+ | 9200 | 26000 | 6000 | 10700 | 99900+ |
| Node | 9 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ |
| Node | 8 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 99900+ | 99900+ | 99900+ | 9500 | 99900+ | 99900+ |
| Node | 7 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 38900 | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ |
| Node | 6 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ |
| Node | 5 | | | | | |
| Initial Weights | 0.5 | 0.4 | 0.3 | 0.2 | 0.1 | 0.01 |
| Number of Epochs | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ | 99900+ |

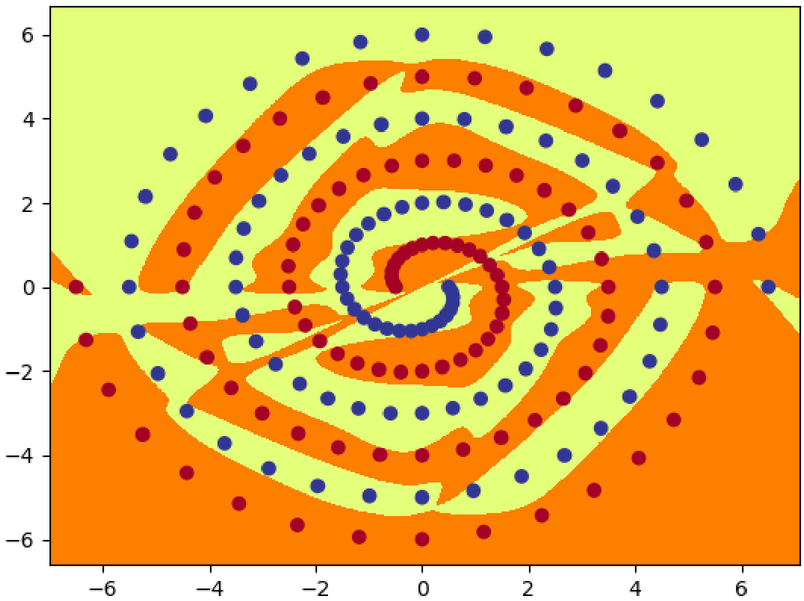
Therefore, any cell highlighted in red is response to question 4, an example is:

Python spiral_main.py --hid 8 --init 0.2

Note, the question did not ask for “on almost all run”, experiment is conducted and also shown in question 6 part b, where another value is selected for a stable network.

Note, 15 nodes seem to be a better and consistent choice then 8 nodes, as observed that 8 nodes do not show consistency of convergence under 20000 under various attempts.

The image for hid_num = 8 with initial weight = 0.2:

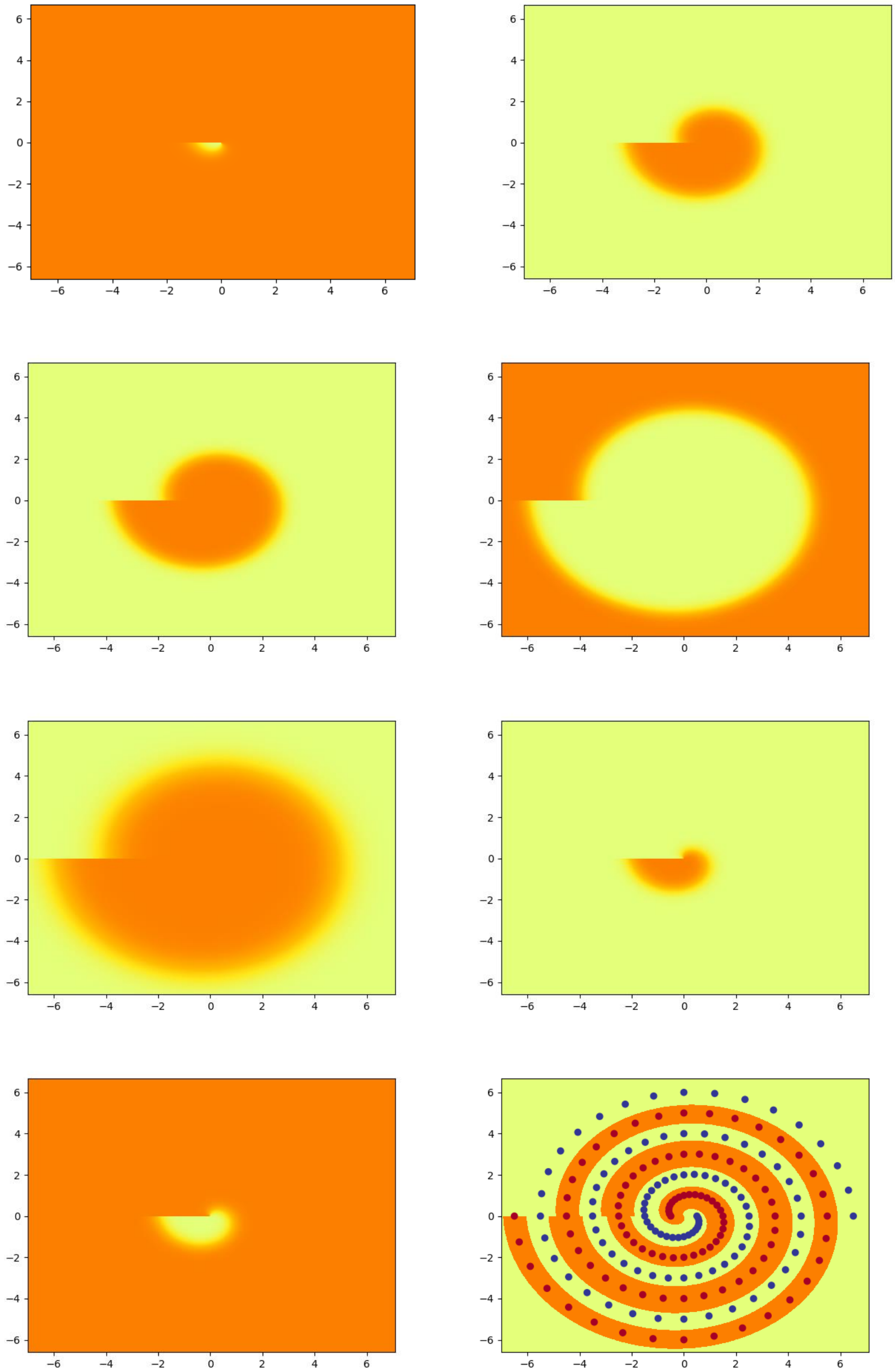


5. Develop graph_hidden(net, layer,node) function and include plots of all hidden nodes in PolarNet and Raw Net and include in report. Note no discussion is requested

**Adjusted Code:**

```
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) #Set up a grid
    with torch.no_grad(): # Temporarily set all the requires_grad flag to false
        net.eval() #Sets the model at evaluation mode
        net(grid) #Using the pre-defined grid
        if layer == 1: #Mainly for Raw - as there is 2 layers
            pred = (net.output_1[:, node]).float()
        elif layer == 2:
            pred = (net.output_2[:, node]).float()
        # plot function computed by model
        plt.pcolormesh(xrange, yrange, pred.cpu().view(yrange.size()[0], xrange.size()[0]), cmap='Wistia')
        #Tmp
        file_name = "hidden_layer_" + str(layer) + "_node_" + str(node) + ".png"
        plt.savefig(file_name)
        #Showing Graphs
        #plt.show()
```

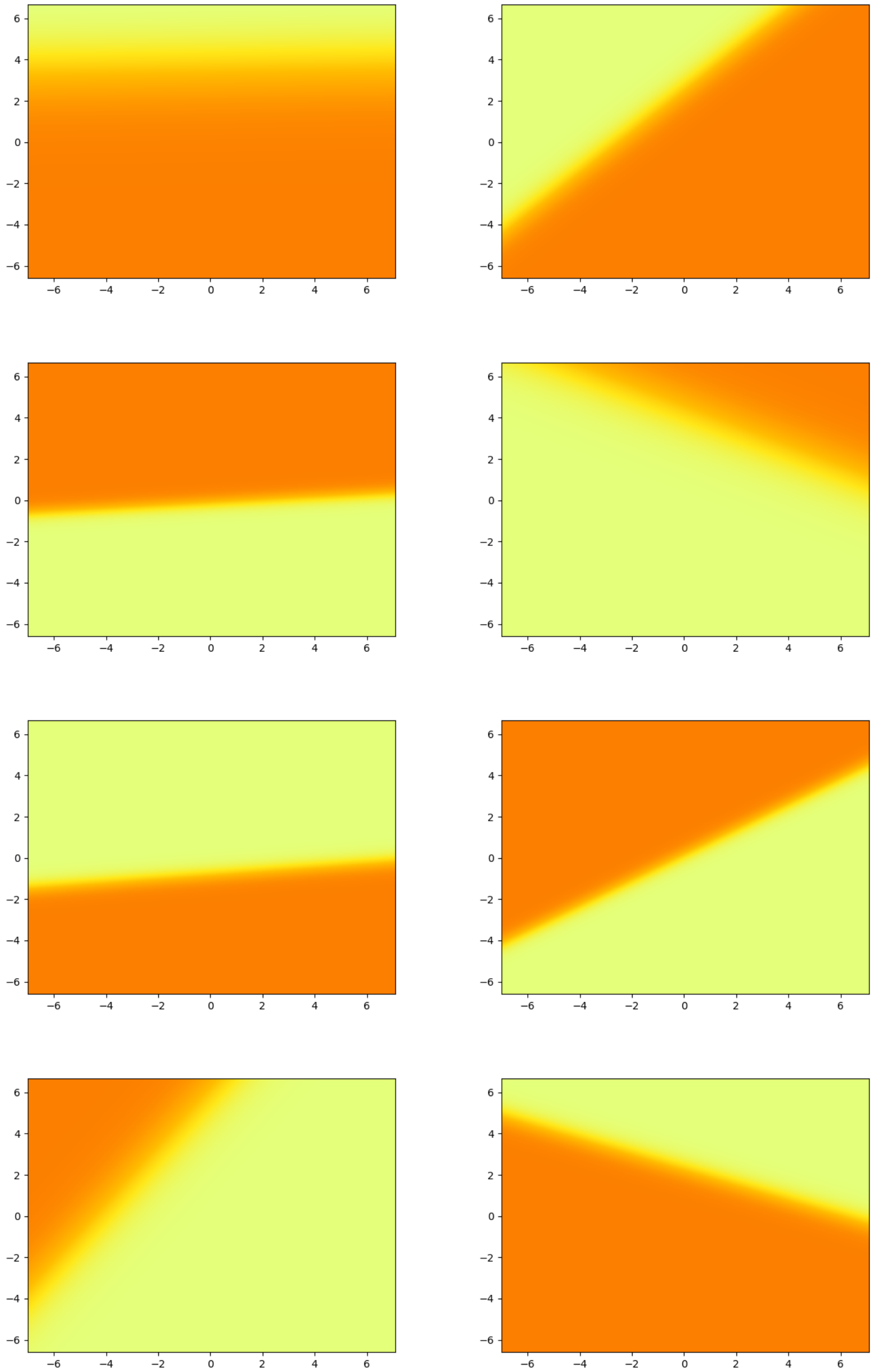
**Polar Net**

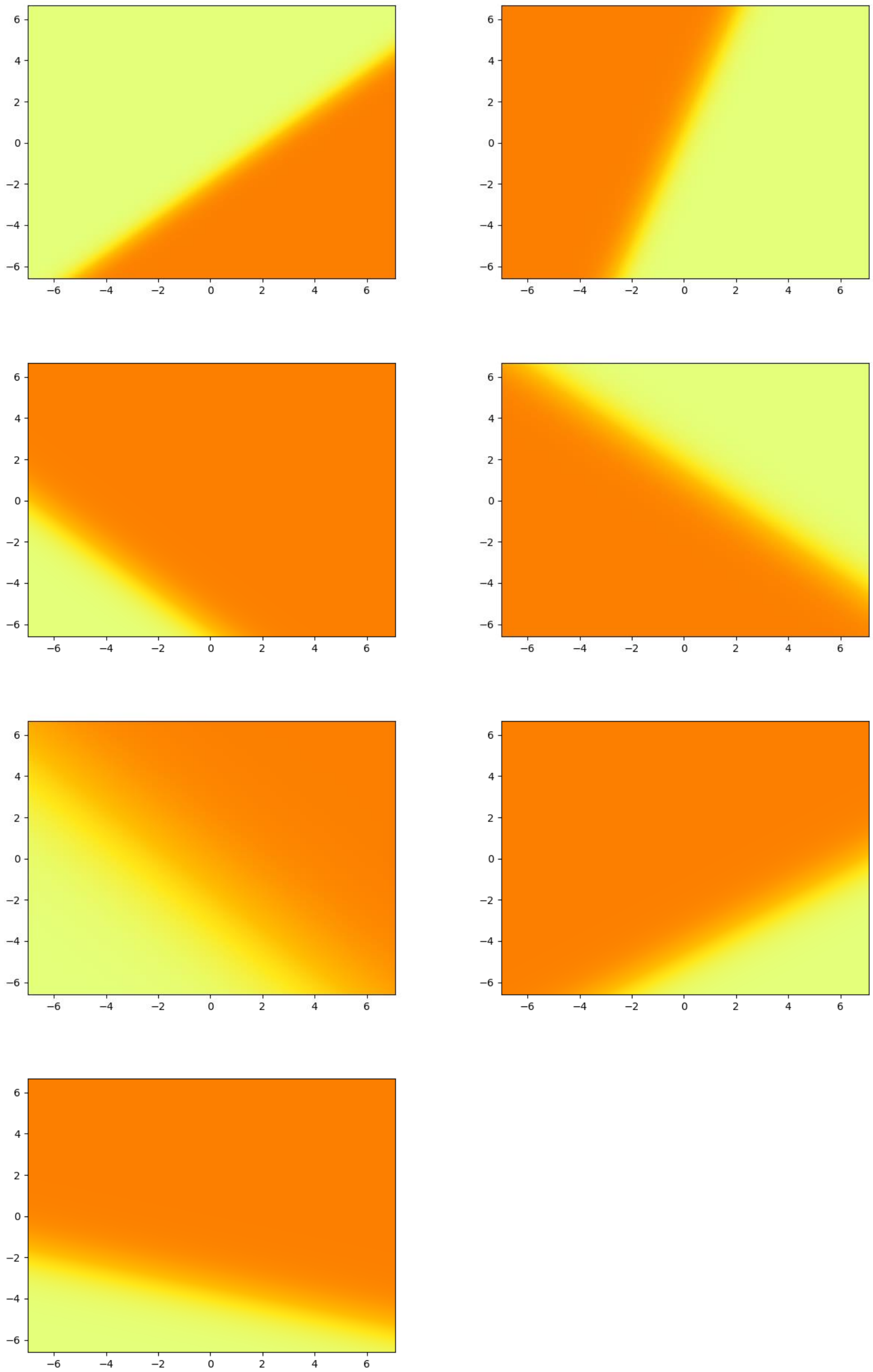


The order is Right to Left, top to bottom, and last picture is output. We can see that each hidden node represents different parts of the graph.

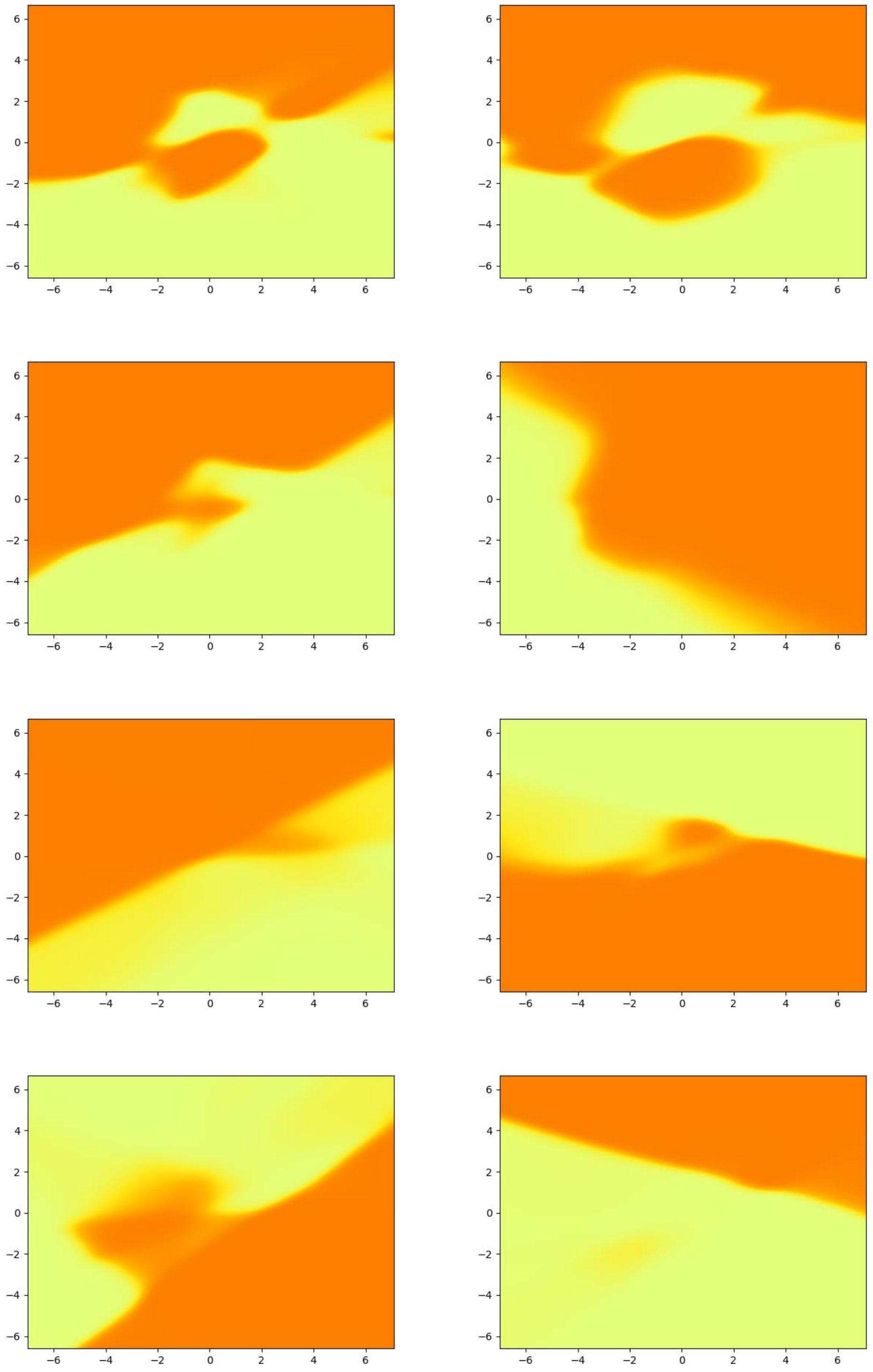
**Raw Net (We choose num hidden = 15 with 0.2 initial weight – refer to Q 6)**

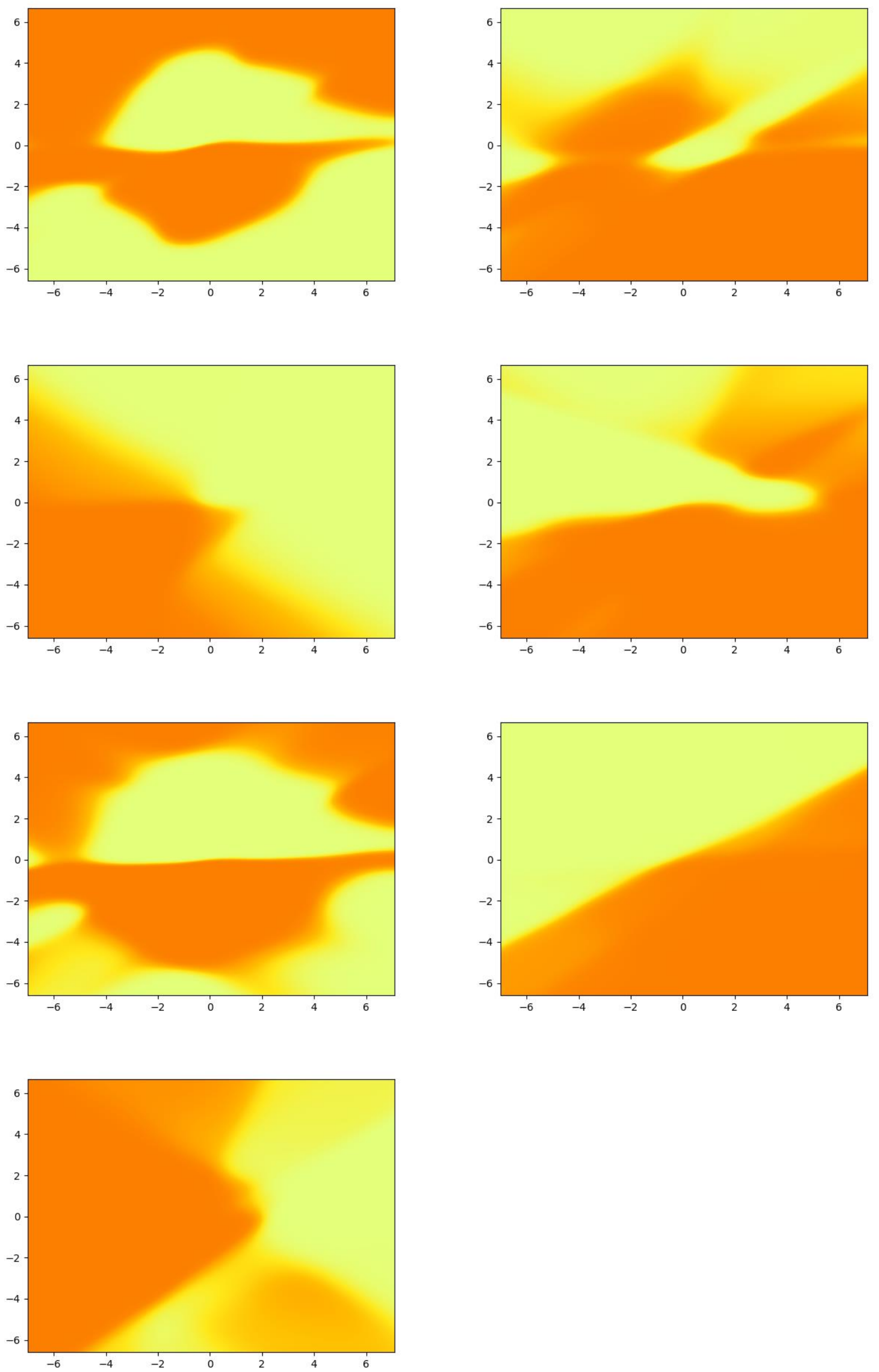
<u>Layer 1</u>



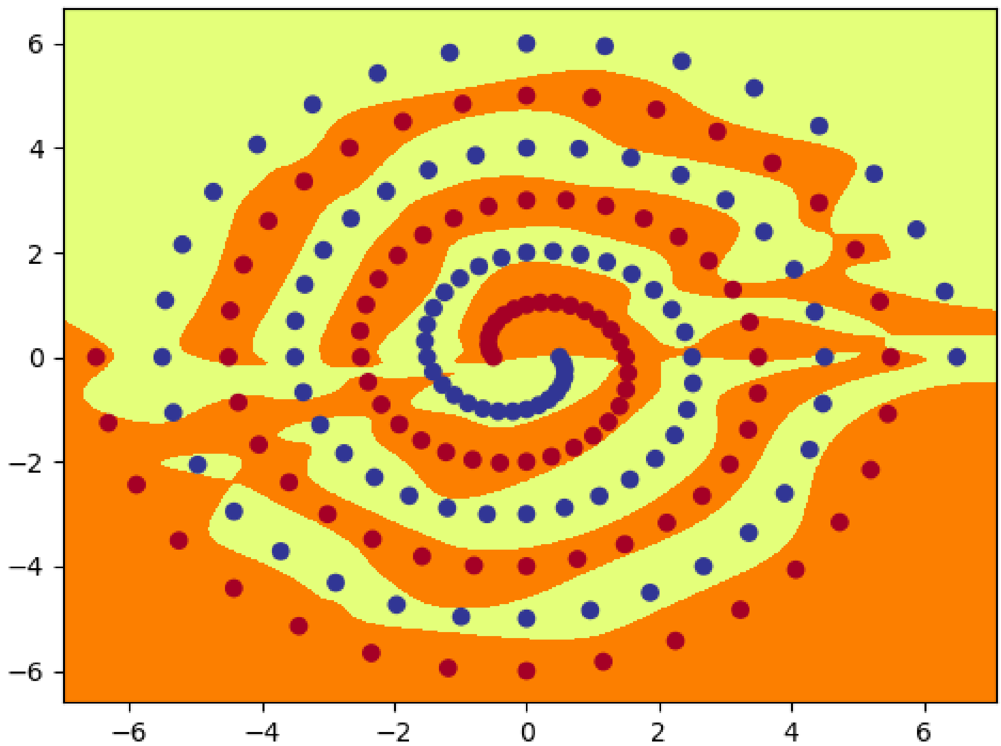

−6
−4
−2
0
2
4
6

Layer 2





RawNet Out:



6. Part a)

On both, the networks learn via drawing boundaries around the input, however the main difference between PolarNet and RawNet are the input and layer of network. In terms of input, the first is inputted r and a, where as the latter is using raw x and y coordinates. This results in harder interpretation for the latter, as coming up with a perceptron based on x and y coordinates in dividing the nodes – as observed more hidden nodes are required in latter. Whereas for the former (r and a), the given input algins more to the desired outcome, and the neural network can pick up that easier. In terms of layers of network, both uses tanh and sigmoid. PolarNet only contains a single layer, where as RawNet contains 1 extra layer, this because of the input, where for RawNet in order to capture higher level information based on x and y coordinates, it requires that extra level.

We can also observe the difference via the graphs shown in the previous sections.

Part b)

From the table in question 4, we observed that increase number of nodes assists the overall network to converge under 99900+, where for nodes less than 10, we see most of them does not converge to 100% accuracy. In terms of weight, whilst bare in mind that the conducted experiment varies each run, we observe that increasing weight assist in making the network converge faster (The average for 0.5 Weights across different number of nodes>10 is 2300, where as the average for 0.1 Weights across different number of nodes >10 is 4100).

An experiment is conducted to find a more stable choice, note we pick 15 number of nodes, as it is the minimum with most number of epochs < 20000:

| Node 15 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Weight | 0.1 | | | | | | | | | |
| Iteration | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 3400 | 99900+ | 6500 | 6200 | 4500 | 3900 | 6400 | 99900+ | 5300 | 5200 |
| Loss | 0.0162 | 0.692 | 0.0256 | 0.0543 | 0.0448 | 0.03 | 0.0491 | 0.692 | 0.0685 | 0.0688 |
| Weight | 0.2 | | | | | | | | | |
| Iteration | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 10100 | 3200 | 3100 | 2000 | 3500 | 2400 | 5300 | 2400 | 2800 | 1700 |
| Loss | 0.0087 | 0.0329 | 0.022 | 0.0352 | 0.015 | 0.0417 | 0.0121 | 0.0095 | 0.0413 | 0.0725 |
| Weight | 0.3 | | | | | | | | | |
| Iteration | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 5100 | 2300 | 2200 | 7500 | 4000 | 4200 | 2800 | 1700 | 2700 | 2100 |
| Loss | 0.0072 | 0.0205 | 0.0276 | 0.0052 | 0.0061 | 0.007 | 0.0129 | 0.0218 | 0.0193 | 0.0304 |
| Weight | 0.4 | | | | | | | | | |
| Iteration | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 3200 | 7100 | 3200 | 4100 | 15500 | 3500 | 7400 | 79700 | 7300 | 4500 |
| Loss | 0.0055 | 0.0067 | 0.0077 | 0.0062 | 0.0043 | 0.0063 | 0.0055 | 0.0051 | 0.0045 | 0.0105 |
| Weight | 0.5 | | | | | | | | | |
| Iteration | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Number of Epochs | 5800 | 6500 | 12600 | 22700 | 8400 | 9500 | 4700 | 20900 | 30200 | 37200 |
| Loss | 0.004 | 0.0054 | 0.0065 | 0.0051 | 0.0052 | 0.0056 | 0.0069 | 0.0045 | 0.0054 | 0.0039 |

Therefore, we see that Number of nodes = 15 with initial weights between 0.2 and 0.3 to be a more stable choice for convergence below 20000 epochs.

Part c)

**Batch Size from 97 to 194**

```
train_dataset = torch.utils.data.TensorDataset(full_input,full_target)
#train_loader  = torch.utils.data.DataLoader(train_dataset,batch_size=97)
train_loader  = torch.utils.data.DataLoader(train_dataset,batch_size=194) #Twice the original
```

| Batch: | 97 | 194 | Setting<br>Adam, Tanh |
|---|---|---|---|
| Polar Net | 17800 | 5600 | hid = 7 |
| Raw Net | 2200 | 1000 | hid = 15 layer =2<br>weight=0.2 |

We can see that increasing batch size reduces number of Epoch, however overall, if batch size it too large it will also lead to poor generalization.

**Using SGD (Batch Size = 97)**

```
"""
# use Adam optimizer
optimizer = torch.optim.Adam(net.parameters(),eps=0.000001,lr=args.lr,
                             betas=(0.9,0.999),weight_decay=0.0001)

"""
optimizer = torch.optim.SGD(net.parameters(),lr=args.lr, weight_decay=0.0001)
```

| Optim: | Adam | SGD | Setting<br>Batch 97, Tanh |
|---|---|---|---|
| Polar Net | 17800 | 99900+ | hid = 7 |
| Raw Net | 2200 | 99900+ | hid = 15 layer =2<br>weight=0.2 |

According to research Adam generalize poorly compared to SGD, however Adam has better training performance, which aligns without results here, that SGD does not converge within 100000 epochs in both scenarios.

**Changing to ReLu (Batch Size = 97, Adam)**

```
#Relu
class PolarNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(PolarNet, self).__init__()
        # INSERT CODE HERE
        self.in_hidden = nn.Linear(2, num_hid)
        self.hidden_out = nn.Linear(num_hid, 1)

    def forward(self, input):
        # First we convert the input to polar co-ordinates
        x,y = input[:,0], input[:, 1]
        r,a = torch.sqrt((x**2) + (y**2)), torch.atan2(y,x) # Might need to reshape
        r,a = r.view(r.shape[0], -1), a.view(a.shape[0], -1)
        #Now we have r and a, we need to concatenate into 1
        output = torch.cat((r,a), dim=1)
        #Network:
        self.output_1 = torch.relu(self.in_hidden(output))
        output = torch.sigmoid(self.hidden_out(self.output_1))
        return output
```

```
#For question 6 part c
#relu
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        # INSERT CODE HERE
        self.in_hidden1 = nn.Linear(2, num_hid)
        self.hidden1_hidden2 = nn.Linear(num_hid, num_hid)
        self.hidden2_out = nn.Linear(num_hid, 1)
    def forward(self, input):
        self.output_1 = torch.relu(self.in_hidden1(input))
        self.output_2 = torch.relu(self.hidden1_hidden2(self.output_1))
        output = torch.sigmoid(self.hidden2_out(self.output_2))
        return output
```

| Activation: | Tanh | Relu | Setting<br>Batch 97, Adam |
|---|---|---|---|
| Polar Net | 17800 | 99900+ | hid = 7 |
| Raw Net | 2200 | 99900+ | hid = 15 layer =2<br>weight=0.2 |

By looking at the graphical interpretation of each activation function, with Tanh, negative value can be fired off, where as Relu 0 value is given if a certain neuron does not contribute to the process. We can't simply justify which one is better than the other, rather it highly depends on the given task at hand, which in terms affect the model and training process.

In this case, considering the nature of the given image, it takes longer for ReLu to finish training, as the weights do not have negative value.

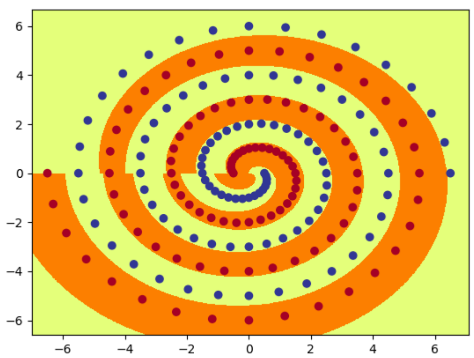
**Extra Layer added to each Network**

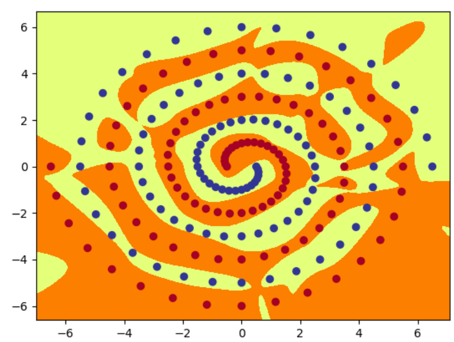
```python
# 1 extra layer
class PolarNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(PolarNet, self).__init__()
        # INSERT CODE HERE
        self.in_hidden = nn.Linear(2, num_hid)
        self.hidden_intermediate = nn.Linear(num_hid, num_hid)
        self.hidden_out = nn.Linear(num_hid, 1)

    def forward(self, input):
        # First we convert the input to polar co-ordinates
        x,y = input[:,0], input[:, 1]
        r,a = torch.sqrt((x**2) + (y**2)), torch.atan2(y,x) # Might need to reshape
        r,a = r.view(r.shape[0], -1), a.view(a.shape[0], -1)
        #Now we have r and a, we need to concatenate into 1
        output = torch.cat((r,a), dim=1)
        #Network:
        self.output_1 = torch.tanh(self.in_hidden(output))
        self.output_2 = torch.tanh(self.hidden_intermediate(self.output_1))
        output = torch.sigmoid(self.hidden_out(self.output_2))
        return output
```

```python
# 1 extra layer
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        # INSERT CODE HERE
        self.in_hidden1 = nn.Linear(2, num_hid)
        self.hidden1_hidden2 = nn.Linear(num_hid, num_hid)
        self.hidden2_hidden3 = nn.Linear(num_hid, num_hid)
        self.hidden3_out = nn.Linear(num_hid, 1)
    def forward(self, input):
        self.output_1 = torch.tanh(self.in_hidden1(input))
        self.output_2 = torch.tanh(self.hidden1_hidden2(self.output_1))
        self.output_3 = torch.tanh(self.hidden2_hidden3(self.output_2))
        output = torch.sigmoid(self.hidden3_out(self.output_3))
        return output
```

| Extra Layers: | 0 | 1 | Setting<br>Batch 97, Adam, Tanh |
|---|---|---|---|
| Polar Net | 17800 | 4000 | hid = 7 |
| Raw Net | 2200 | | hid = 15 layer =2<br>weight=0.2 |

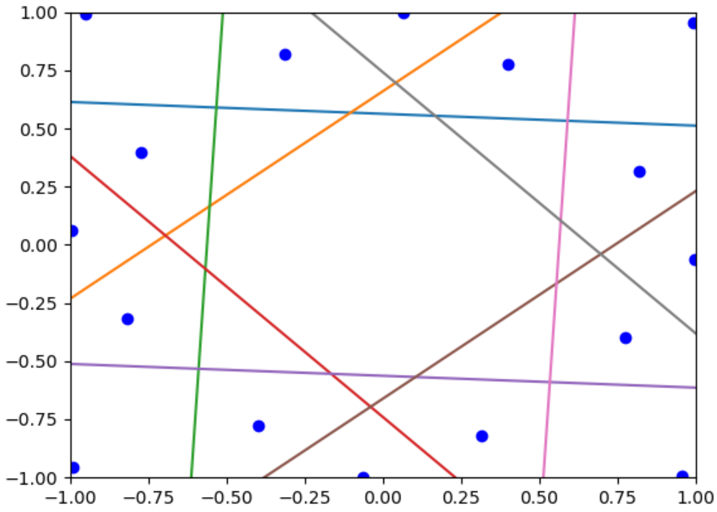



We can see that for rawNet, there is slight more refined boundary area, if we observe the hidden nodes, we will see a better graphical output.

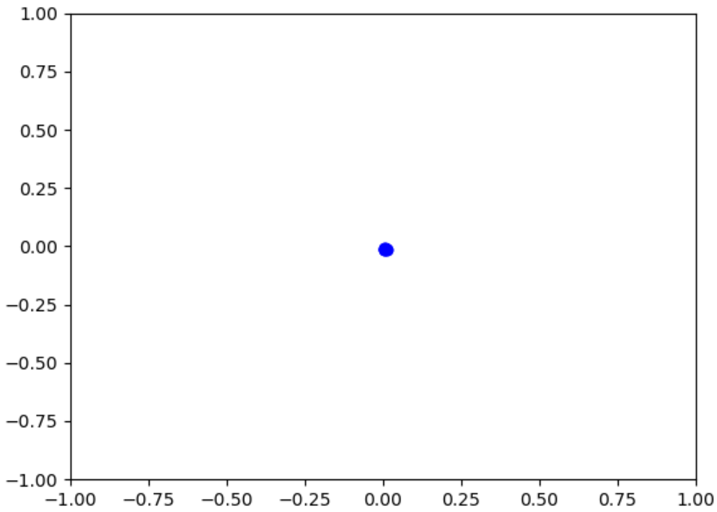
**Part 3: Hidden Unit Dynamics**

1. We are only requested to execute the code and attach the image:

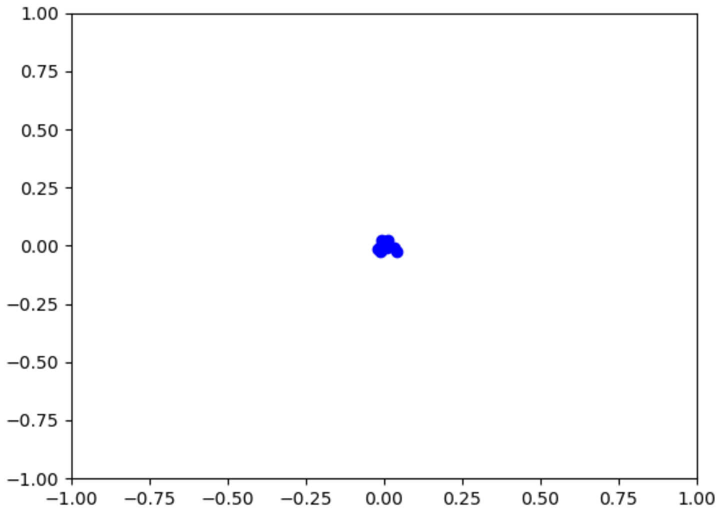


2. We are only requested to execute the code, attach images of 50 to 3000 epochs, then discuss:

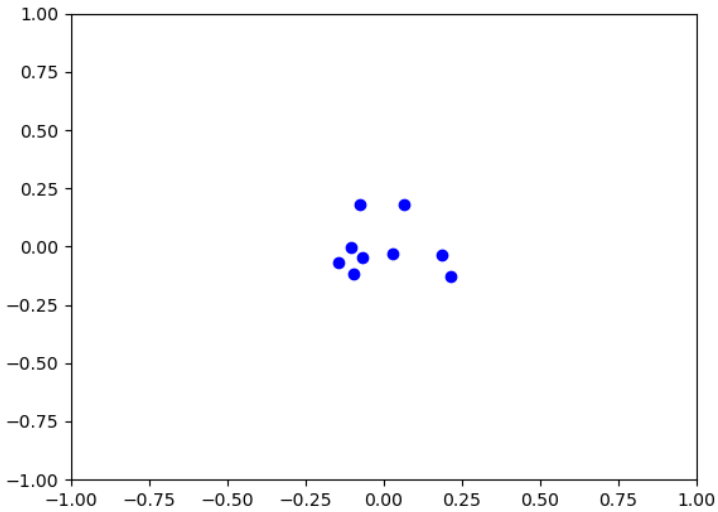
Epoch = 50:



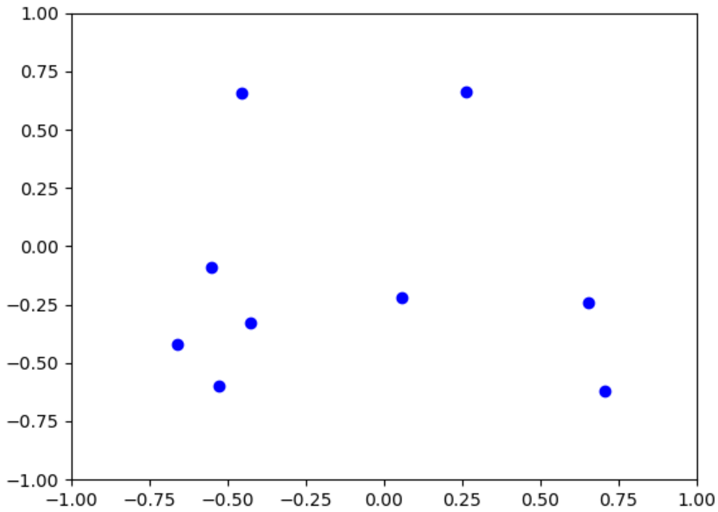
Epoch = 100



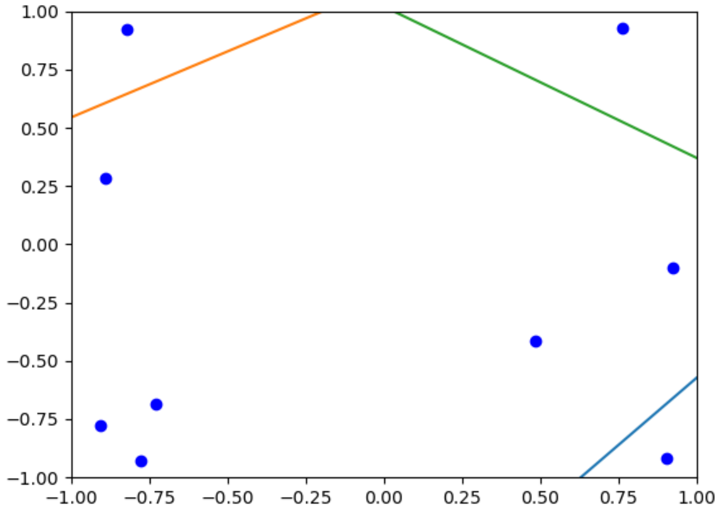

Epoch = 150



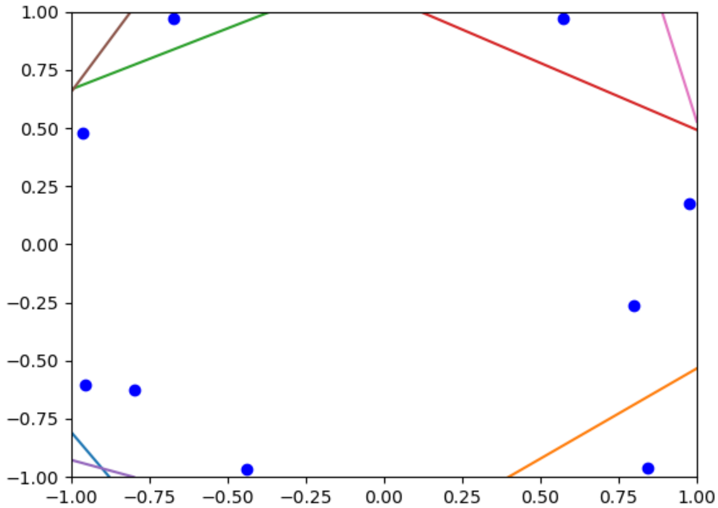

Epoch = 200



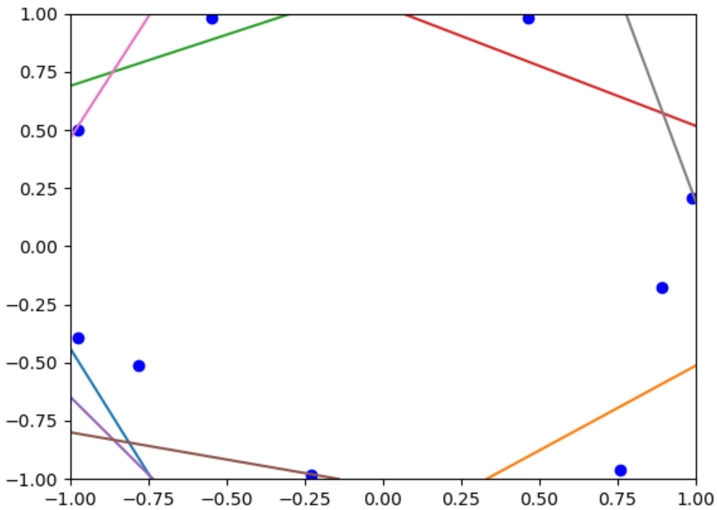
Epoch = 300



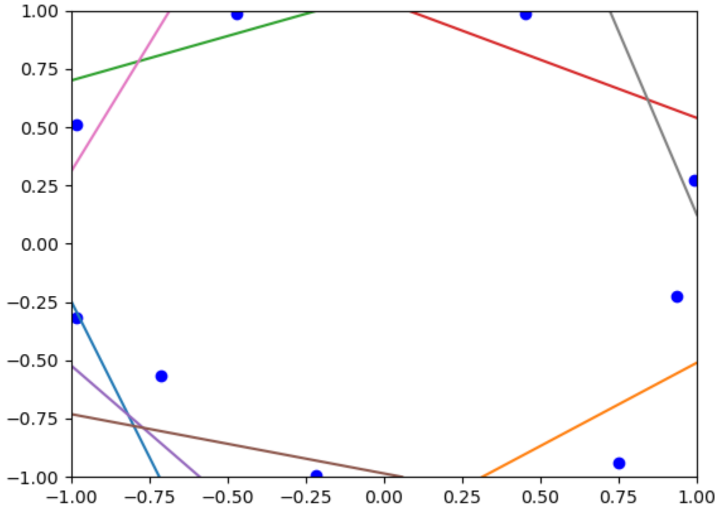

Epoch = 500



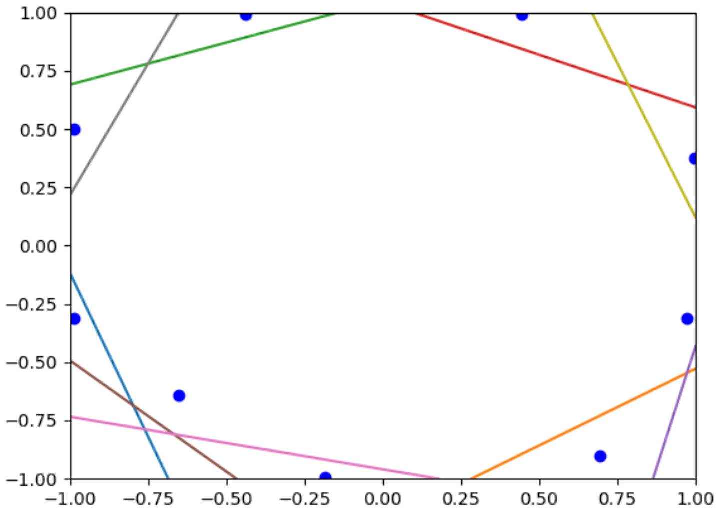

Epoch = 700



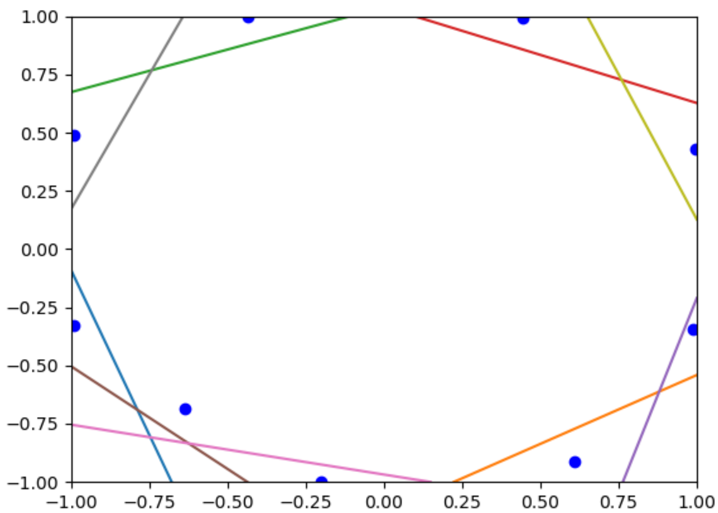
Epoch = 1000

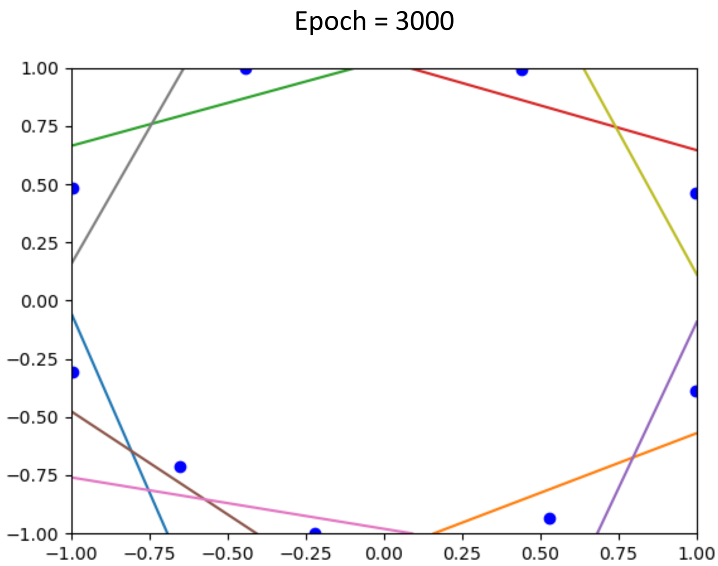


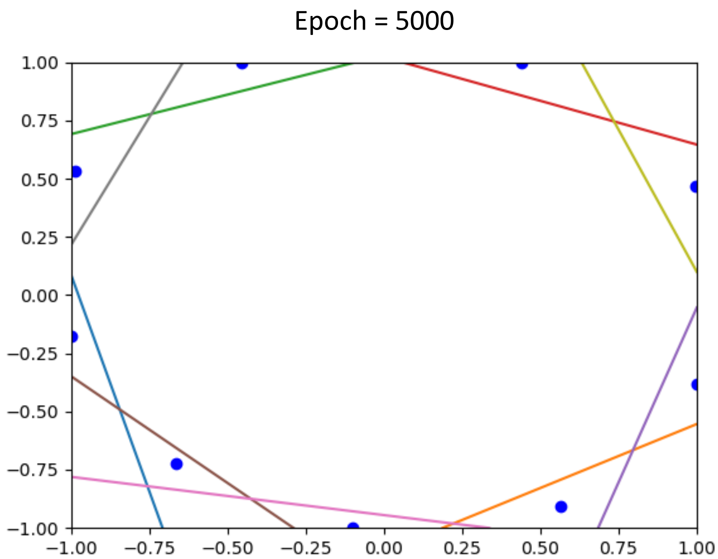

Epoch = 1500

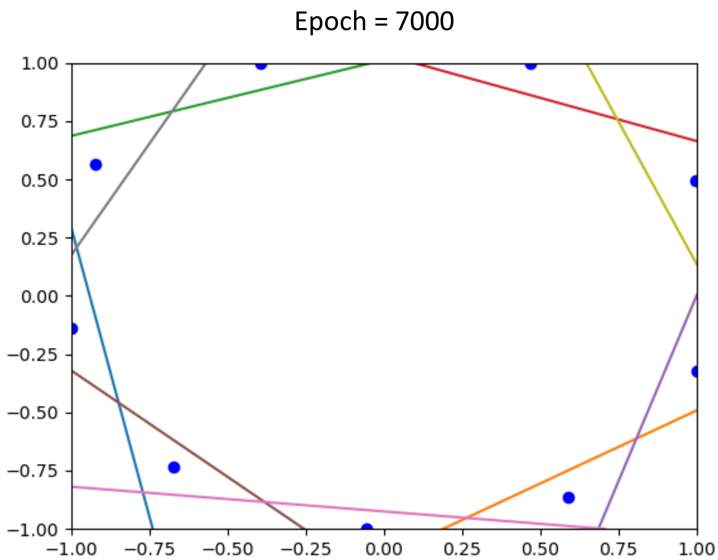



Epoch = 2000

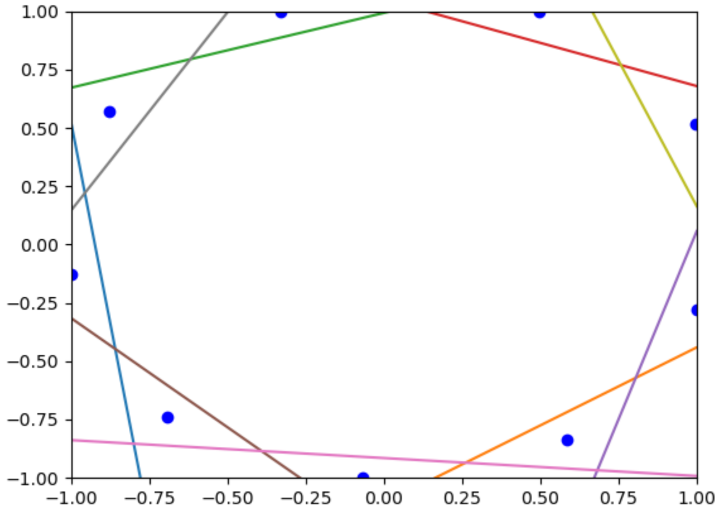



Epoch = 3000
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00

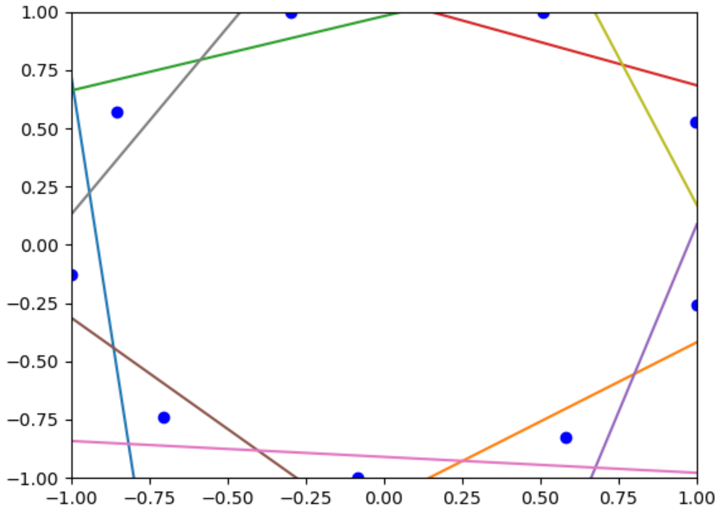

Epoch = 5000
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00


Epoch = 7000
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

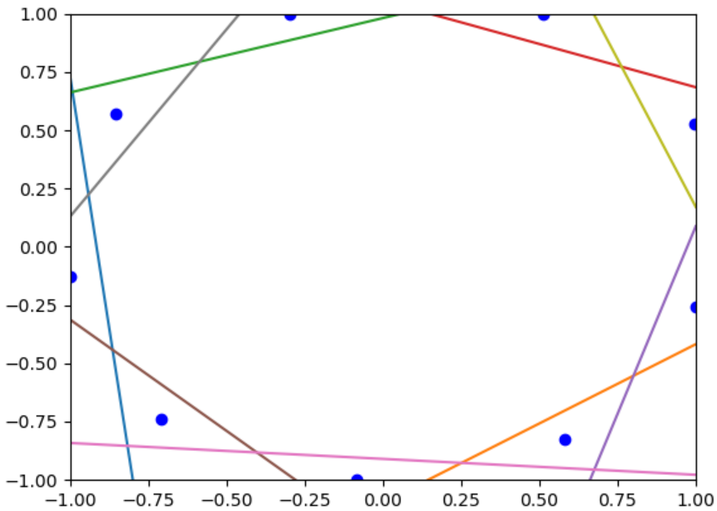
Epoch = 10000



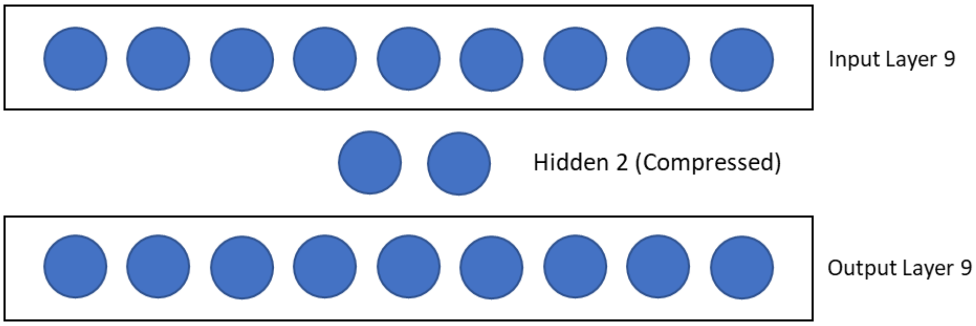

Epoch = 15000




Epoch = 15080

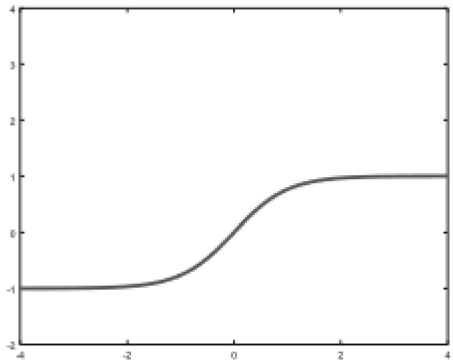


Given by the lecture, the dots represent hidden unit activations, the lines represent the output boundaries.




Providing a description for the output graph:

- First Hidden Unit is along X axis, Second hidden unit is y axis
- Activation Function is Tanh, as it is bounded between 1 to -1:



Hyperbolic Tangent

- No Bias observed
- Dot Coordinates are obtained by:
    - X Axis – Input times weight and apply Tanh for one of the hidden nodes
    - Y Axis – Input times weight and apply Tanh for the other hidden node
    - This gives us the input to hidden unit representation on HU Space
- Line Coordinates are obtained by:
    - Perceptron Approach
    - Draws the boundary where the linear combination where output is positive or negative

To describe the steps:

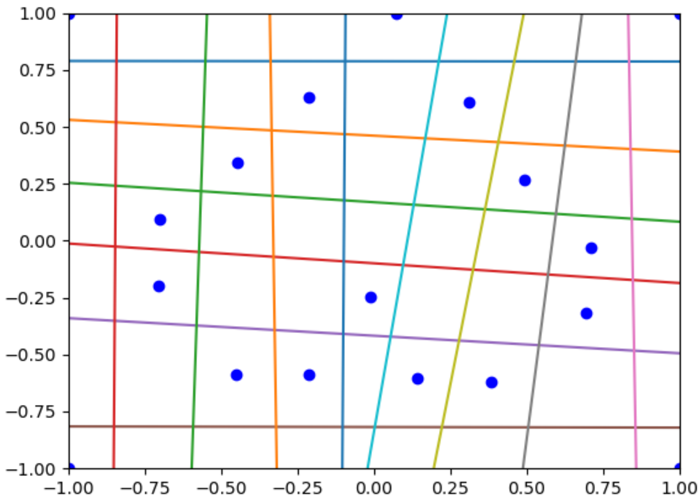
1) At 50 Epoch the input to hidden nodes weights are still initializing hence clouded at center
2) At 100 Epoch the input to hidden nodes weights start to spread out, same with 150 and 200.
3) At 300 Epoch we start to see hidden to output perceptron line been learnt and drawn, however this is still at early stage hence only 3 lines. The dots start to move in to respective "area"
4) At 500 Epoch movement continues in properly dividing the HU space with hidden to output boundaries introduced (7 lines)
5) At 700 Epoch more boundaries are drawn and dots move to respective region, extra line introduced

6) At 1000 Epoch the boundaries start to shift and bounding the dots into centre of respect region
7) At 1500 Epoch we start to see most dots been bounded into a unique region
8) At 2000 Epoch All dots are within unique region; however, a single dot is within the large bounded area within centre
9) At 5000 Epoch, the single dot is moved to a bounded triangular area
10) At 7000 Epoch, the boundaries and dots shift together to divide each sub region to more proportion – perceptron learning
11) At 10000 and 15000 Epoch, little movement is observed, where now the system is stabilizing

Overall the training process, as visualized by the graph, is showing how input-hidden and hidden-output weights update its value to learn the problem at task and shifting to a more stabilized network.

3. In this task we are requested to create heart shape, regardless of rotation, with 18 rows and 14 columns, various variation is examined and the selected on is called heart18 Following provides detailed breakdown, note discussion is not required:

   **1st_attempt:**



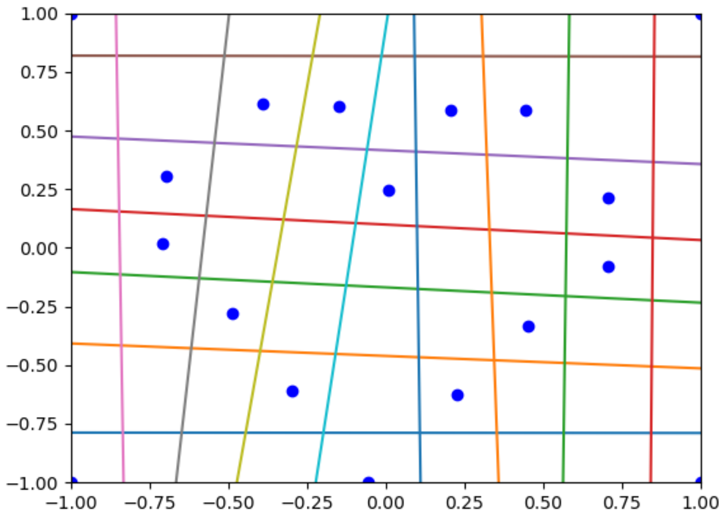

This provides an upside-down heart, the rule is as follow:

| | |
|---|---|
| Dot to the Right | 0 |
| Dot to the Left | 1 |
| Dot to the Top | 0 |
| Dot to the Bottom | 1 |

Tensor is as follow:

```
#1st
heart18 = torch.Tensor([
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,1,1,1,1,1,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,1,1,1,1,1,1,1,1],
    [1,1,1,1,1,1,1,1,1,1,0,0,0,0],
    [0,1,1,1,1,1,1,1,1,0,0,0,0,0],
    [0,0,1,1,1,1,1,1,0,0,0,0,0,0],
    [0,0,0,1,1,1,1,0,0,0,0,0,0,0],
    [0,0,0,0,1,1,1,0,0,0,0,0,0,0],
    [0,0,0,0,0,1,1,1,0,0,0,0,0,0],
    [0,0,0,0,0,1,1,1,1,0,0,0,0,0],
    [0,0,0,0,1,1,1,1,1,1,0,0,0,0],
    [0,0,0,0,0,1,1,1,1,1,1,0,0,0],
    [0,0,0,0,0,1,1,1,1,1,1,1,0,0],
    [0,0,0,0,1,1,1,1,1,1,1,1,1,0],
    [0,0,0,1,1,1,1,1,1,1,1,1,1,0],
    [0,0,1,1,1,1,1,1,1,1,1,1,0,0],
    [0,1,1,1,1,1,1,1,1,1,1,0,0,0]
    ])
```

**2nd_attempt:**



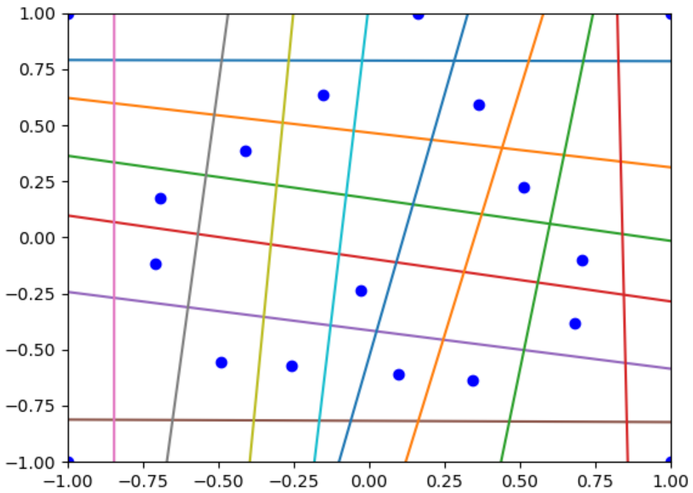

This provides the exact same heart, the rule is as follow

| | |
|---|---|
| Dot to the Right | 0 |
| Dot to the Left | 1 |
| Dot to the Top | 1 |
| Dot to the Bottom | 0 |

Tensor is as follow:

```
#2nd
heart18 = torch.Tensor([
    [0,0,0,0,0,0,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,1,1,1,1,0,0,0,0],
    [1,0,0,0,0,0,1,1,1,0,0,0,0,0],
    [1,1,0,0,0,0,1,1,0,0,0,0,0,0],
    [1,1,1,0,0,0,1,0,0,0,0,0,0,0],
    [1,1,1,1,0,0,1,0,0,0,0,0,0,0],
    [1,1,1,1,1,0,1,1,0,0,0,0,0,0],
    [1,1,1,1,1,0,1,1,1,0,0,0,0,0],
    [1,1,1,1,0,0,1,1,1,1,0,0,0,0],
    [1,1,1,1,1,0,1,1,1,1,1,0,0,0],
    [1,1,1,1,1,0,1,1,1,1,1,1,0,0],
    [1,1,1,1,0,0,1,1,1,1,1,1,1,0],
    [1,1,1,0,0,0,1,1,1,1,1,1,1,0],
    [1,1,0,0,0,0,1,1,1,1,1,1,0,0],
    [1,0,0,0,0,0,1,1,1,1,1,0,0,0]
    ])
```

**3rd_attempt:**



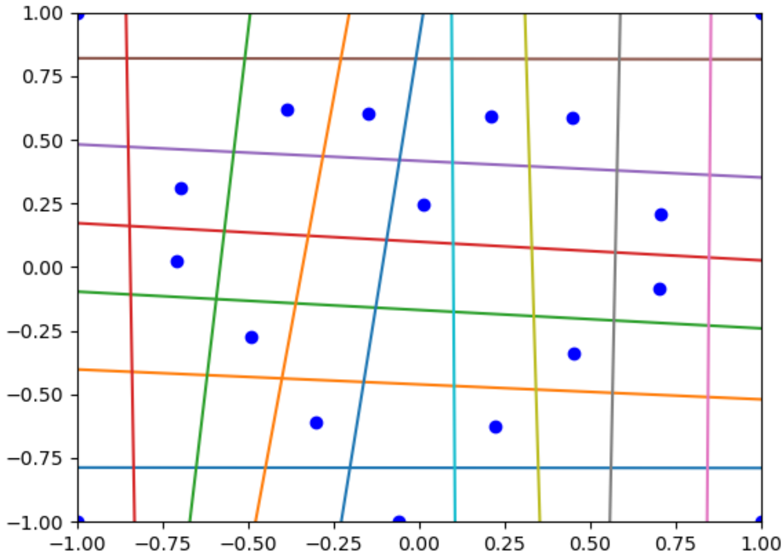

This provides again an upside-down heart, the rule is as follow

| | |
|---|---|
| Dot to the Right | 1 |
| Dot to the Left | 0 |
| Dot to the Top | 1 |
| Dot to the Bottom | 0 |

Tensor is as follow:

```
#3rd
heart18 = torch.Tensor([
    [1,1,1,1,1,1,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,0,0,0,0,1,1,1,1],
    [0,1,1,1,1,1,0,0,0,1,1,1,1,1],
    [0,0,1,1,1,1,0,0,1,1,1,1,1,1],
    [0,0,0,1,1,1,0,1,1,1,1,1,1,1],
    [0,0,0,0,1,1,0,1,1,1,1,1,1,1],
    [0,0,0,0,0,1,0,0,1,1,1,1,1,1],
    [0,0,0,0,0,1,0,0,0,1,1,1,1,1],
    [0,0,0,0,1,1,0,0,0,0,1,1,1,1],
    [0,0,0,0,0,1,0,0,0,0,0,1,1,1],
    [0,0,0,0,0,1,0,0,0,0,0,0,1,1],
    [0,0,0,0,1,1,0,0,0,0,0,0,0,1],
    [0,0,0,1,1,1,0,0,0,0,0,0,0,1],
    [0,0,1,1,1,1,0,0,0,0,0,0,1,1],
    [0,1,1,1,1,1,0,0,0,0,0,1,1,1]
    ])
"""
```

**$4^{th}$_attempt:**




This provides the exact same heart, the rule is as follow

| | |
|---|---|
| Dot to the Right | 1 |
| Dot to the Left | 0 |
| Dot to the Top | 0 |
| Dot to the Bottom | 1 |

Tensor is as follow:

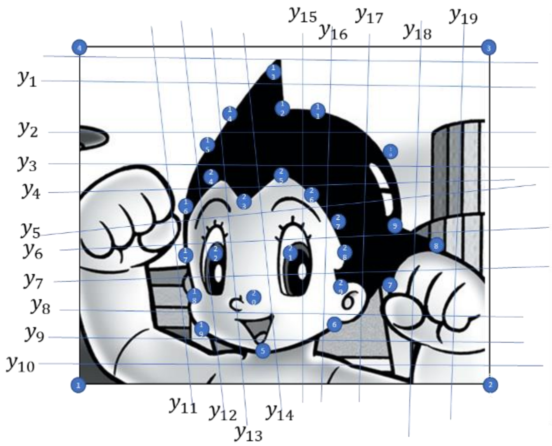
```
#4th
heart18 = torch.Tensor([
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,1,1,1,1,1,1,1,1],
        [1,1,1,1,1,1,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,1,1,1,1],
        [1,0,0,0,0,0,0,0,0,1,1,1,1,1],
        [1,1,0,0,0,0,0,0,1,1,1,1,1,1],
        [1,1,1,0,0,0,0,1,1,1,1,1,1,1],
        [1,1,1,1,0,0,0,1,1,1,1,1,1,1],
        [1,1,1,1,1,0,0,0,1,1,1,1,1,1],
        [1,1,1,1,1,0,0,0,0,1,1,1,1,1],
        [1,1,1,1,0,0,0,0,0,0,1,1,1,1],
        [1,1,1,1,1,0,0,0,0,0,0,1,1,1],
        [1,1,1,1,1,0,0,0,0,0,0,0,1,1],
        [1,1,1,1,0,0,0,0,0,0,0,0,0,1],
        [1,1,1,0,0,0,0,0,0,0,0,0,0,1],
        [1,1,0,0,0,0,0,0,0,0,0,0,1,1],
        [1,0,0,0,0,0,0,0,0,0,0,1,1,1]
        ])
```

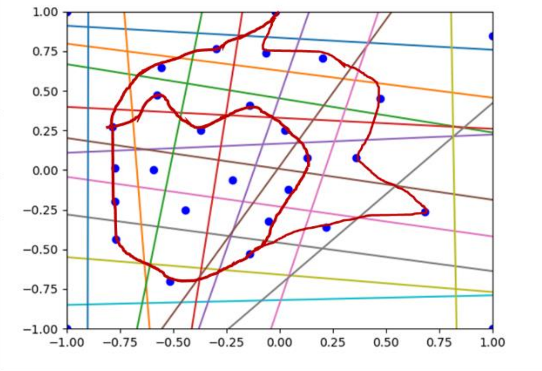
Both attempt 2 and 4 provides the outcome desired. It is understood that depending on various factor, the picture might differ slightly across same tensor.

4. We are asked to provide 2 images of our own design:

**Traget1 – Attempt1**

This is the first attempt in drawing:




Tensor is as follow (29 Rowsx19 Columns):

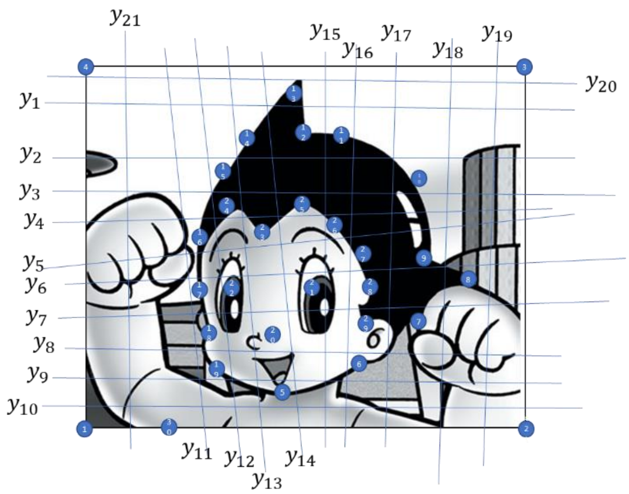
```
target1 = torch.Tensor([
    [0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,1],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0],
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,1],
    [0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,1,1],
    [0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,1],
    [0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,1,1],
    [0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,1],
    [0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1],
    [0,1,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1],
    [1,1,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1],
    [0,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1],
    [0,0,1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1],
    [0,0,0,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,1,1,1,1,0,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,1,1,1,0,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,0,1,1,0,1,1,1,1,1,1,1,1],
    [0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1,1,1,1],
    [0,0,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1],
    [0,0,0,0,0,0,1,1,1,1,0,0,1,1,1,1,1,1,1],
    [0,0,0,0,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1],
    [0,0,0,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1],
    [0,0,0,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1],
    [0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,1,1,1,1],
    [0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,1,1,1],
    [0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,1,1,1],
    [0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,1,1,1]
    ])
```

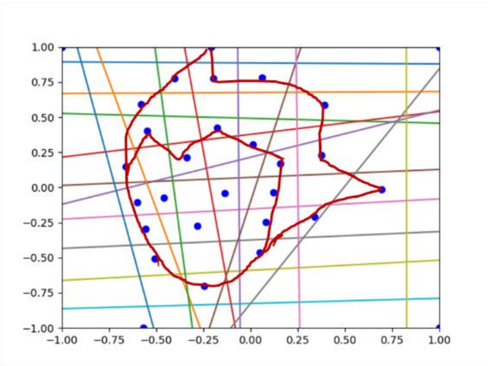
**Target1 – Attempt 2:**

Adjustment has been made slightly to make it better (2 extra columns and 1 extra row = 30 Rows X 21 Columns), ran using the following command, note the learning rate is increased to 0.7 to converge faster:

python encoder_main.py --target=target1 --lr=0.7

This will be our final picture for target 1



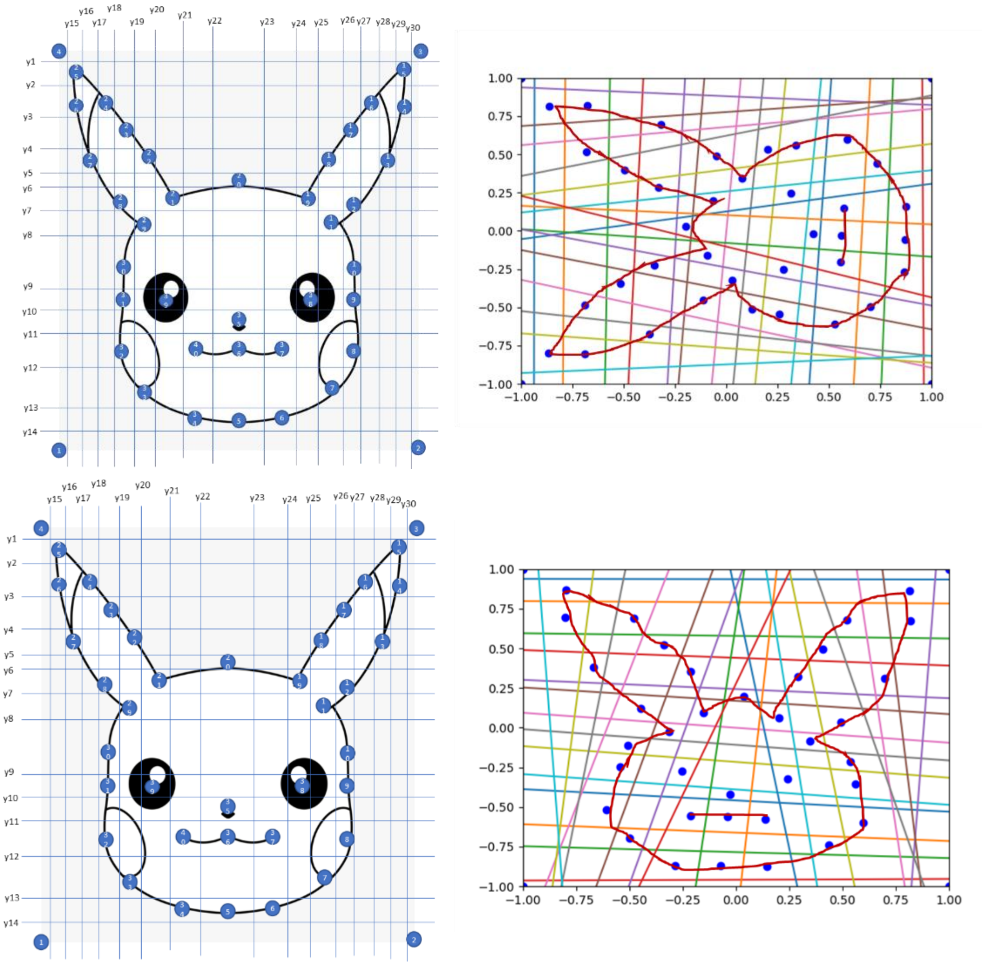




Tensor is as follow:

```
target1 = torch.Tensor([
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    ])
```

**Target2 - Attemp1:**

As for the second target, we want to try something more symmetric and interesting, a 40 rows X 30 Columns image is established:




Tensor is as follow:

```
#Attempt 1
target2 = torch.Tensor([
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
    ])
```