Part 1: Japanese Character Recognition

1. Final accuracy and confusion matrix

```
 # 0   #1   #2   #3   #4 #5   #6   #7   #8   #9
[[764.  5.  9.  14.  31.  63.  2.  63.  31.  18.]
 [ 7. 671. 109.  18.  28.  22.  55.  15.  26.  49.]
 [ 9.  61. 692.  26.  24.  21.  46.  38.  46.  37.]
 [ 5.  37.  57. 759.  15.  55.  14.  19.  28.  11.]
 [ 61.  52.  79.  22. 623.  17.  33.  36.  21.  56.]
 [ 8.  29. 122.  17.  19. 727.  28.   7.  33.  10.]
 [ 5.  22. 149.  10.  26.  24. 721.  21.   9.  13.]
 [ 17.  31.  27.  11.  81.  17.  55. 621.  90.  50.]
 [ 12.  37.  96.  43.   6.  30.  46.   7. 703.  20.]
 [ 9.  54.  83.   3.  55.  33.  18.  31.  41. 673.]]
Test set: Average loss: 1.0081, Accuracy: 6973/10000 (70%)
```

2. Fully connected 2-layer network `NetFull`, using tanh at the hidden nodes and log SoftMax at the output node. Final accuracy and confusion matrix.

```
# 0   #1   #2   #3   #4 #5   #6   #7   #8   #9
[[852.  5.  1.  6.  27.  33.  3.  39.  29.  5.]
 [  4. 792.  39.  5.  23.  13.  70.  8.  21.  25.]
 [  5.  15. 845.  30.  15.  14.  24.  13.  26.  13.]
 [  4.  10.  36. 912.  4.  11.  6.  3.  7.  7.]
 [ 49.  26.  14.  8. 806.  12.  30.  16.  17.  22.]
 [ 10.  8.  66.  16.  15. 830.  28.  2.  18.  7.]
 [  3.  11.  66.  13.  18.  4. 869.  10.  1.  5.]
 [ 18.  14.  20.  5.  27.  8.  41. 810.  21.  36.]
 [ 11.  34.  26.  50.  3.  10.  29.  4. 822.  11.]
 [  3.  21.  49.  3.  31.  10.  27.  12.  16. 828.]]

Test set: Average loss: 0.5345, Accuracy: 8366/10000 (84%)
```

Achieved 84% at 80 hidden nodes.

3. A convolutional network called `NetConv`, with two convolutional layers plus one fully connected layer, all using relu activation function, followed by the output layer, using log SoftMax.

Final accuracy, 93%. Momentum was used at .9. Architecture of NetConv was 1,30,30,30,480,10. Using relu and log_softmax.

```
 # 0   #1   #2   #3   #4   #5   #6  #7   #8   #9
[[939.   5.   4.   2.  26.   3.   3.  15.   1.   2.]
 [  1. 942.   5.   1.   5.   0.  23.   7.   2.  14.]
 [ 10.  11. 867.  36.   5.   3.  25.  24.   6.  13.]
 [  1.   1.  18. 961.   2.   1.   7.   4.   2.   3.]
 [ 12.   9.   4.   6. 920.   5.  18.   5.   9.  12.]
 [  4.  22.  24.   4.   7. 896.  19.  14.   2.   8.]
 [  2.  14.  10.   2.   5.   1. 962.   3.   0.   1.]
 [  6.   8.   1.   1.   8.   0.   7. 945.   4.  20.]
 [  5.  19.  14.   6.  14.   3.   8.   7. 916.   8.]
 [  3.   7.   4.   3.   6.   1.   0.   6.   3. 967.]]
```

4. Test set: Average loss: 0.3383, Accuracy: 9315/10000 (93%)

   Discuss

   a. the relative accuracy of the three models

      Simpler architectures such as the Linear function tend to have lower accuracy levels than more complex architectures like convolutional neural networks and fully connected networks.

      This is due the non-linearity component captured by these types of networks.

      The fully connected has lower accuracy than the convolutional network as the fully connected neural network may fail to capture some of the spatial features the convolutional neural network can capture. It also experiences a well-known problem called exploding vanishing gradients. The problem affects the backpropagation algorithm. In this case, as the number of hidden neurons increases past 80 the gradient explodes.

      For the convolutional network different Kernels, learning rates, momentum and channel outputs were used to increase the accuracy.
      The most effective was the use of momentum and the number of channel outputs.

   b. the confusion matrix for each model: which characters are most likely to be mistaken for which other characters, and why?

In the three Models, character 2="su" is misclassified with character 5=”ha”.

For lin, the character “su” is also mistaken with 1=”ki” and 6="ma".

For full, the character “su” is mistaken with 6=”ma”. Also, the character 6="ma" is mistaken with character “su”.

Finally for conv, the character “su” is mistaken with 5="ha", but also the character 3=“tsu” is mistaken with “su”.

The reason for the misclassification is one of the challenges of the KMNIST dataset and the challenges representing the mapping to the one-character situation.

Hentaigana or variant kana, are Hiragana characters that have more than one form of writing, as they were derived from different Kanji. Therefore, one Hiragana class of Kuzushiji-MNIST may have many characters.

c. Other experiments

C1. Trying Different values for momentum and Kernel Size

Some of the factors that may affect the accuracy of the model is when the network starts oscillating towards a point causing the network to get stuck. By adding momentum, we are adding a weight on the update of past gradients to determine the new weight (new direction to go). In the table below, Momentum at a higher weight .9 was found to have higher impact on the accuracy of the model.

Another hyperparameter than can affect the accuracy is the kernel size. In the convolution the filter slides over all pixels of the image taking their dot product.

The algorithm does this in the hope of extracting some features of the image. In this case as shown below a larger Kernel size increased the performance of the network significantly.

The architecture of NetConv was 1,10,10,20,320,10 initially.

Kernel 3x3	Kernel 5x5	Momentum
Accuracy=87%	Accuracy=90%	.5
Accuracy=89%	Accuracy=91%	.6
Accuracy=88%	Accuracy=91%	.7
Accuracy=90%	Accuracy=91%	.8

Accuracy=91%	Accuracy=92%	.9

C2. Changing the learning rate

The learning rate is a very important hyperparameter that updates the weights at each step of the backpropagation algorithm. Different values where used for the network some lower than the default value .01 and some greater. A little value was found to be more accurate at each step but slower, while a higher learning step would cause the network to diverge oscillating around a point until the epochs were finished. Therefore, the same learning rate .01 was applied for all the other networks.

Learning Rate	Kernel 5x5, 10 output channels each conv, mom=.9
.001	87%
.0001	71%
.01	92%
.1	81%
.2	46%
.3	10%
.4	10%
.5	10%
.6	10%
.7	10%
.8	10%
.9	10%

C3. Changing the number of Output Channels

Finally, using different convolution output channels with the same kernel size 5x5 and momentum .9 was used. The networks increase significantly in accuracy, but the training time took twice as much due to the size of the convolutions.

Size of Convolution	Accuracy
1,10,10,20,320,10	92%
1,20,20,20,320,10	92%
1,30,30,30,480,10	93%
1,40,40,40,640,10	93%
1,50,50,50,800,10	93%

The calculation of the dimensions was calculated as follows, take for example the first row above 1,10,10,20,320,10:

- (28x28x1) -> conv1 -> (28-5+1) -> (24x24x10)
- (24x24x10) ->max1 -> (24/2) -> (12x12x10)
- (12x12x10) ->conv2 -> (12-5+1) -> (8x8x20)
- (8x8x20) -> max2 -> (8/2) -> (4 x 4 x 20)
- (4x4x20) = 320 -> Linear -> 10

Part 2: Twin Spirals Task

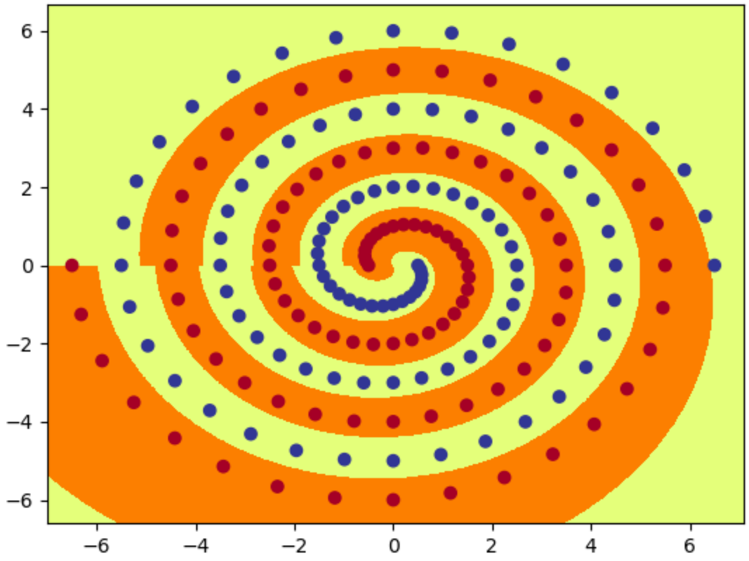
1. Code

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import math
import torch.nn.functional as F

class PolarNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(PolarNet, self).__init__()
        self.feed = nn.Linear(2,num_hid)
        self.hid = nn.Linear(num_hid,1)

    def forward(self, input):
        x = input[:,0]
        y = input[:,1]
        out = torch.zeros(input.shape,dtype=torch.float32)
        out[:,0] = torch.sqrt(x*x + y*y) #tranform data
        out[:,1] = torch.atan2(y,x)
        #here we start the forward pass
        self.active1 = torch.tanh(self.feed(out))
        self.active2 =  self.hid(self.active1)
        self.hidlayer = [self.active1]
        return F.sigmoid(self.active2)
```

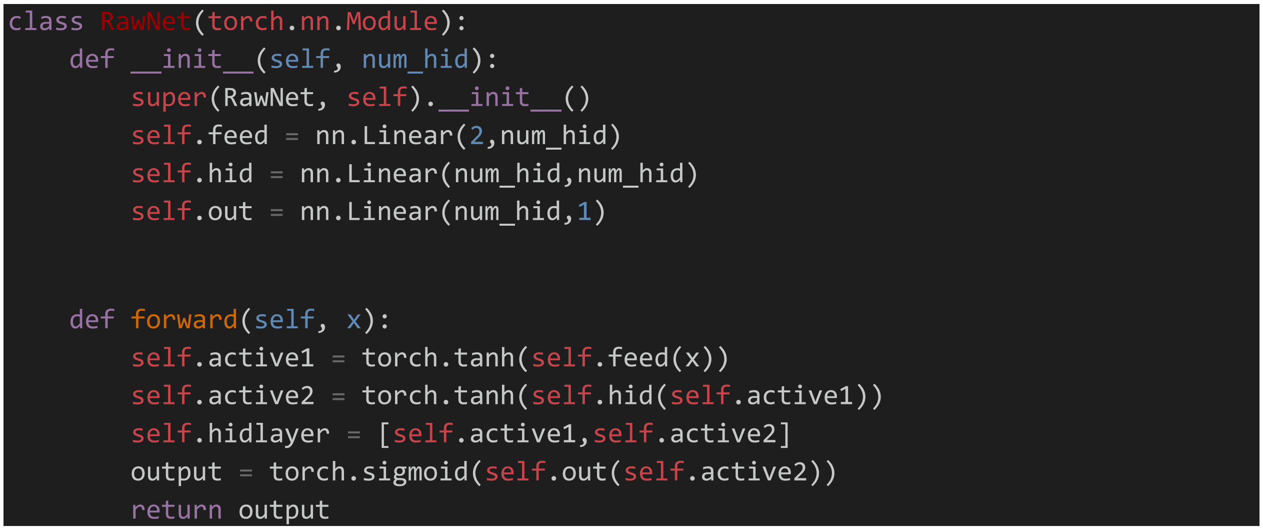
2. Minimum number of hidden nodes required so that this PolarNet learns to correctly classify all the training data within 20000 epochs, on almost all runs.

Minimum number of hidden nodes was found to be 6 after running all models 3 times until convergence in less than 20,000 epochs.

Graph:



3. Two fully connected hidden layers with tanh activation, plus the output layer, with sigmoid activation. Code:



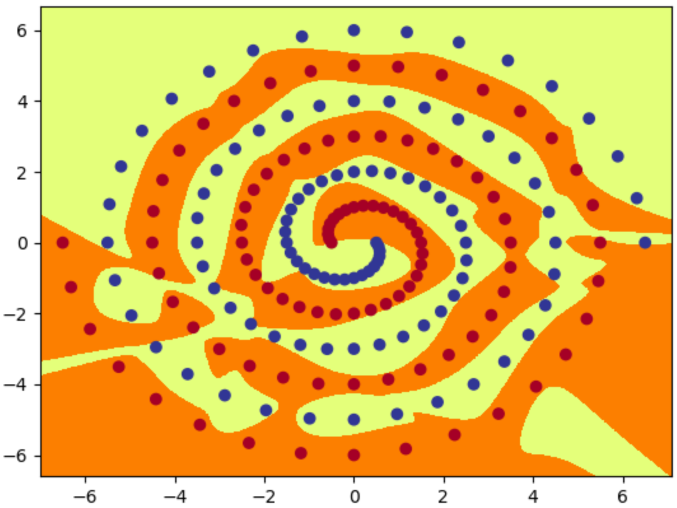
```
class RawNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(RawNet, self).__init__()
        self.feed = nn.Linear(2,num_hid)
        self.hid = nn.Linear(num_hid,num_hid)
        self.out = nn.Linear(num_hid,1)


    def forward(self, x):
        self.active1 = torch.tanh(self.feed(x))
        self.active2 = torch.tanh(self.hid(self.active1))
        self.hidlayer = [self.active1,self.active2]
        output = torch.sigmoid(self.out(self.active2))
        return output
```

4. Value for the number of hidden nodes (--hid) and the size of the initial weights (--init) such that this RawNet learns to correctly classify all the training data within 20000 epochs, on almost all runs.

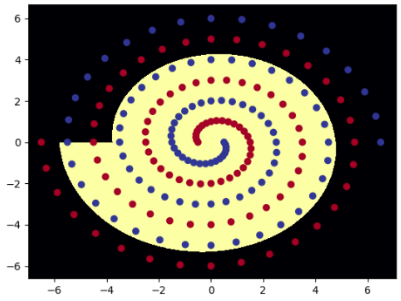
   Value of hidden nodes and other metaparameters:

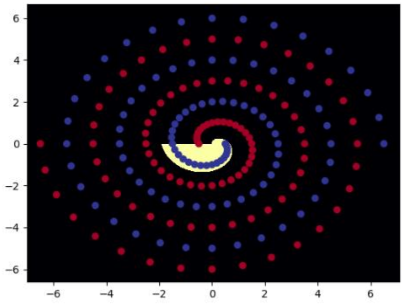
Hidden: from 10+ hidden nodes 100% accuracy is achieved in less than 20,000 epochs. To limit the number of activation images number of hidden nodes = 10 was chosen with init=.9 to reduce the number of epochs.

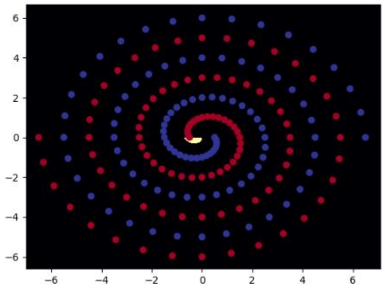
Graph `raw_out.png`:

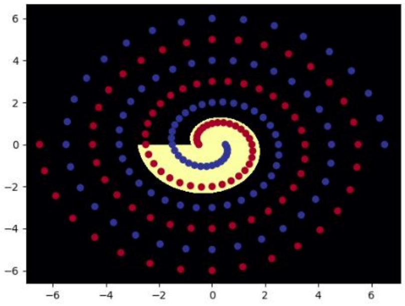


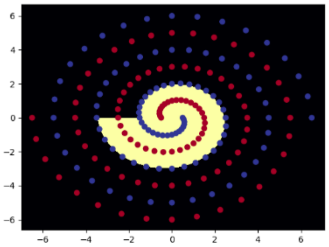
5. Plots of all the hidden nodes in PolarNet, and all the hidden nodes in both layers of RawNet:
   PolarNet:

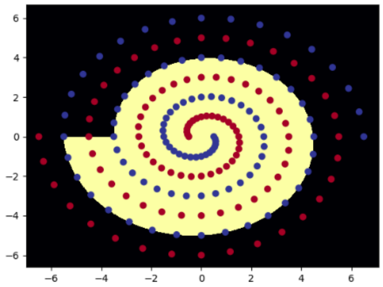




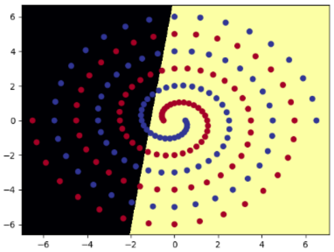

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

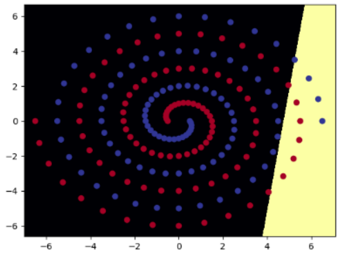

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

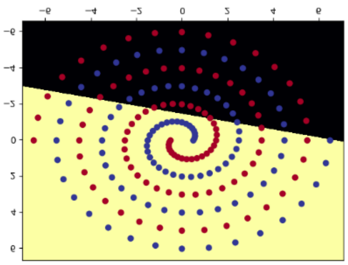

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

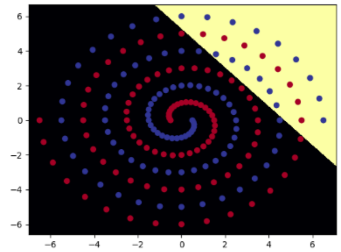

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

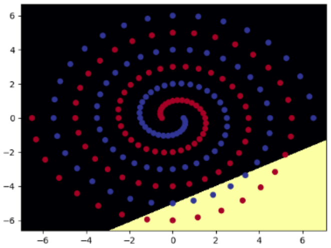
RawNet:

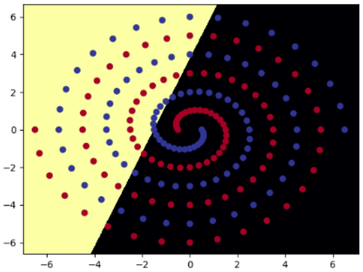

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

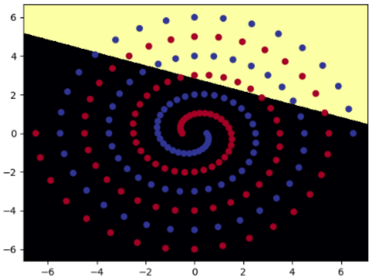

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

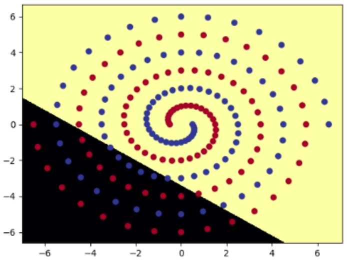


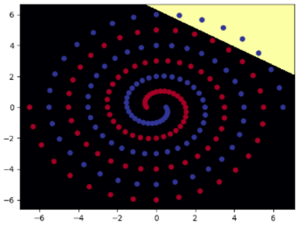

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

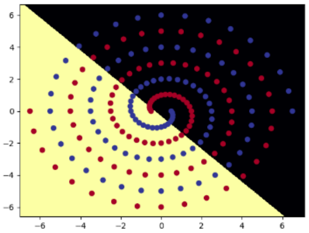

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

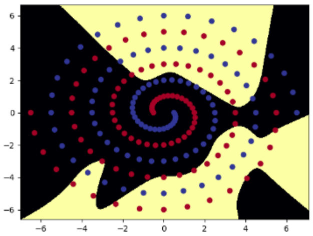

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

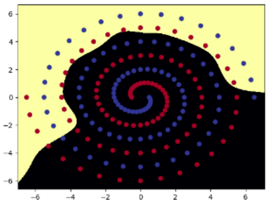

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

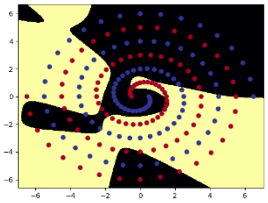

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

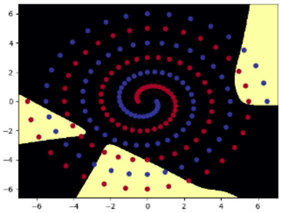

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

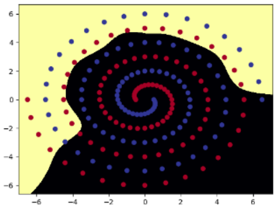

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

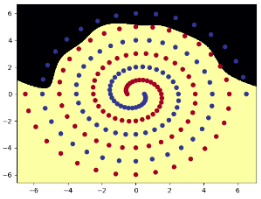

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

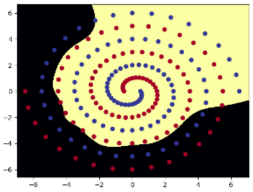

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

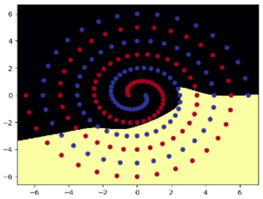

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

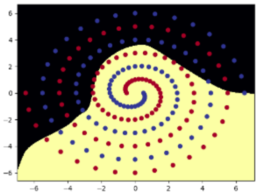

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

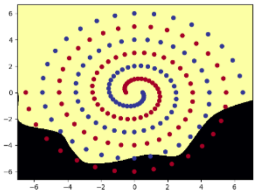

6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6


6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6


6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6


6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6


6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6


6
4
2
0
−2
−4
−6
−6 −4 −2 0 2 4 6

6. Discuss

a. the qualitative difference between the functions computed by the hidden layer nodes PolarNet and RawNet, and a brief description of how the network uses these functions to achieve the classification

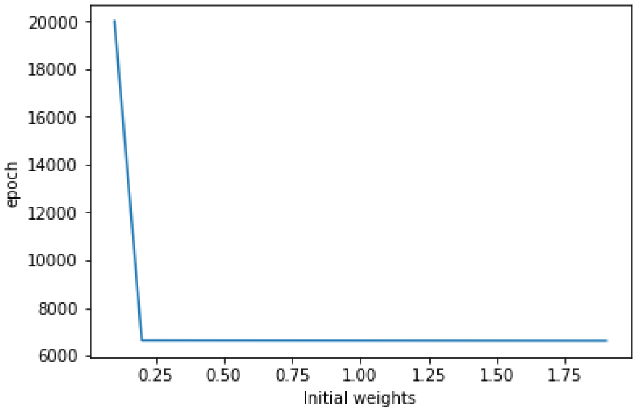
The difference between PolarNet and RawNet is that RawNet has an extra forward pass and backpropagates the network twice while PolarNet only backpropagates once.

In this case the transformation done to PolarNet helps enhance the classification of the x, y coordinates.

How it works is the activation function considers how different parameters interact with one another and does a transformation in this case tanh which gets to decide the neurons who go forward into the next layer.

The tanh activation function has the nice property that gets values of different signs the range is between 1 and -1 which is what is needed as the x, y values have negative values. While the sigmoid function is between one and zero which is what we need to classify the values as higher than .5 or lower than the threshold.

b. the effect of different values for initial weight size on the speed and success of learning for RawNet



As the initialization of the weights increases the number of epochs required to achieve 100% accuracy decreases. This is not surprising as the weights can determine how quickly the network converges or not at all. The distribution of the weights affects the gradients of the network and therefore how effective the training is.

c. Experiment with other changes and comment on the result

c.1 Changing batch size from 97 to 194

The number of epochs for PolarNet with 10 hidden nodes reduced significantly from 2300 epochs to only 1500. However, for RawNet the batch made the model unstable and could not converge. The accuracy of the network got stuck at 53.61%. This is not surprising as there is a tradeoff between stability and speed of the learning process.

c.2. Using SGD instead of Adam

Unlike stochastic Gradient Descent, Adam computes the learning rate dynamically which can help the optimization converge faster, rather than a static increase or decrease in the learning rate.

SGD was implemented but never reached convergence in RawNet or Polar Net with 10 hidden nodes under 100,000 epochs.

```
optimizer = torch.optim.SGD(net.parameters(), lr=args.lr, momentum=.1)
```

An increase in the accuracy was noticed but the algorithm took too long to train the model.

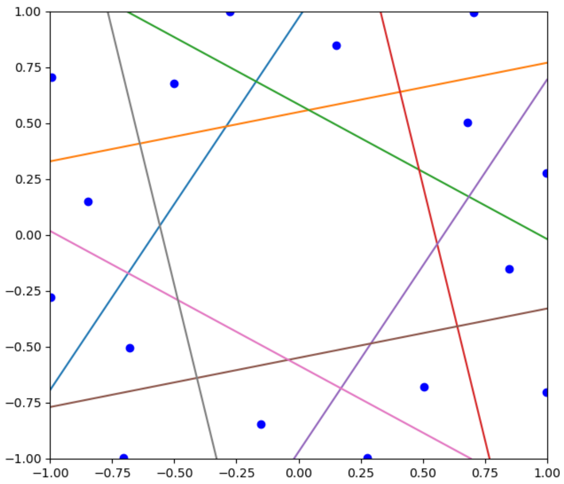
c3. Changing tanh to relu

One of the main advantages of relu is that it can approximate any function by stacking multiple relus. The relu activation function was implemented in the RawNet with 15 hidden nodes. However, the function took too long to train compared to using tanh. This could be due to the nature of Relu as it takes the positive values. The gradient could go towards zero and get stuck.

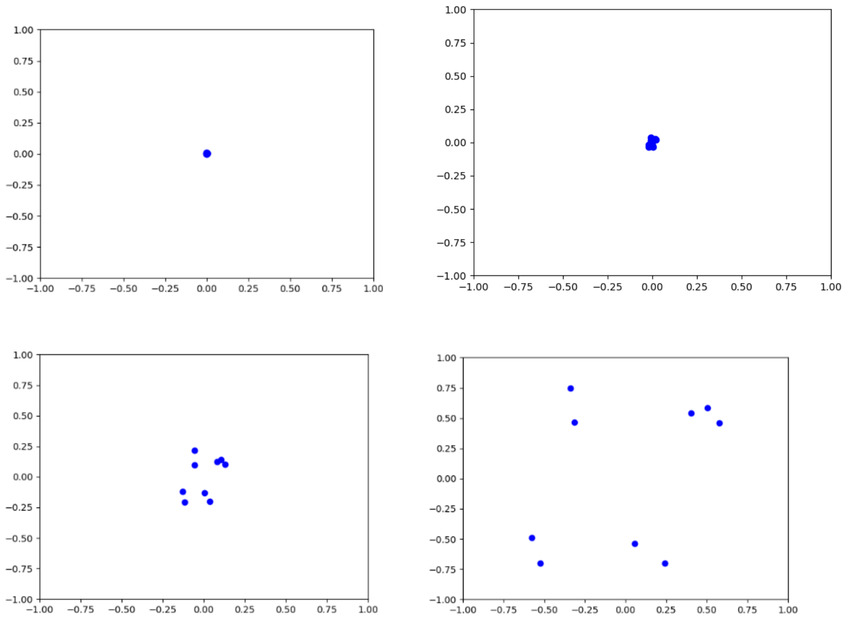
The RawNet with 15 hidden nodes and 2 relu activation functions took around 91,200 epochs to converge.

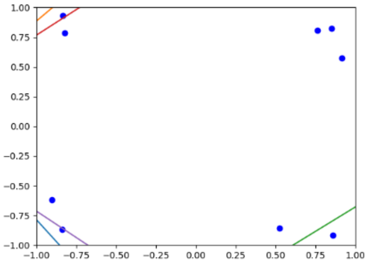
Part 3: Hidden Unit Dynamics

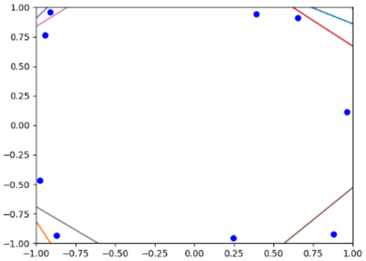
1. Image:

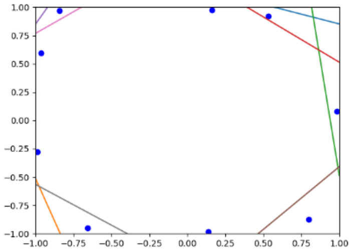


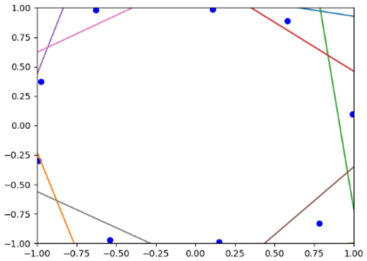
2. 9-2-9 encoder with both input and target determined by a one-hot encoding. First eleven images generated (epochs 50 to 3000), plus the final image.

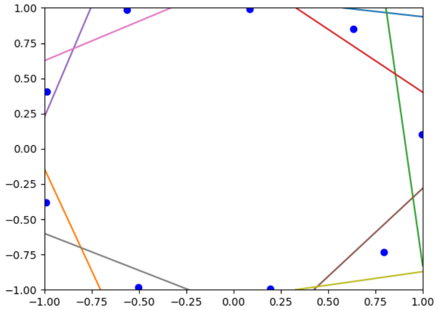


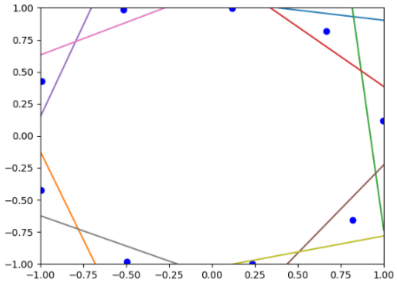

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

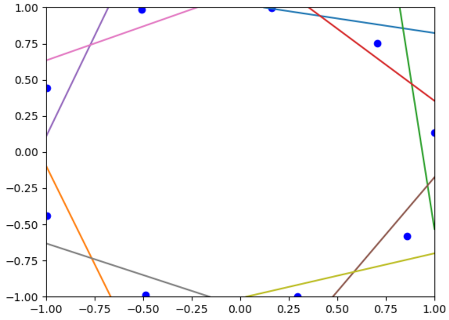

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


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

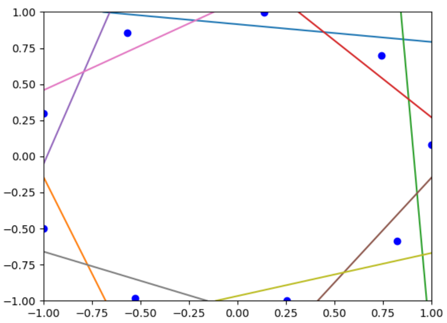

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


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


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


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

Final Image:



At every iteration, the input matrix is just an identity matrix with all diagonals equal to one and all other entries equal to zero. The target value is always the same desired output we are trying to match.

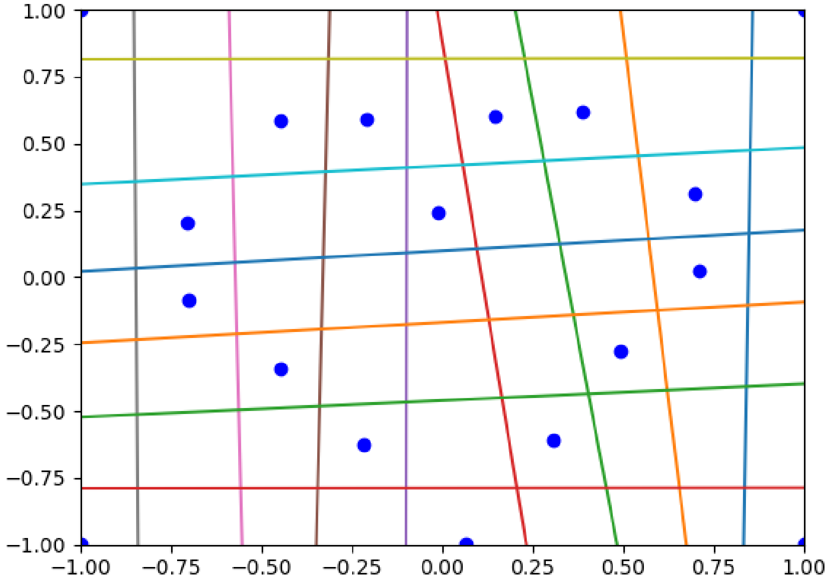
The network applies the forward step, which involves updating a single neuron at the time. For example, the first row will have a "1" at each row. Therefore, only the weight of that neuron will affect the network. It is then applied to the transformation function tanh which activates that neuron and applies the sigmoid function. Then computes the loss and applies the backpropagation algorithm to the network.

For the first 11 images the epochs are 50,100,150,200,300,500,700,1000,1500,2000 and 3000.
In any of those epochs the algorithm takes the input to hidden weights and biases and plots those weights in the range -1 to 1 (after computing tanh). Then, the algorithm takes the output hidden weights and draws a line interval to show the decision boundary of each output.

At the end, the network will solve (if there is a solution) the decision boundaries of each output of the target. The columns of the target can be seen as decision boundaries while the rows as the blue dots.

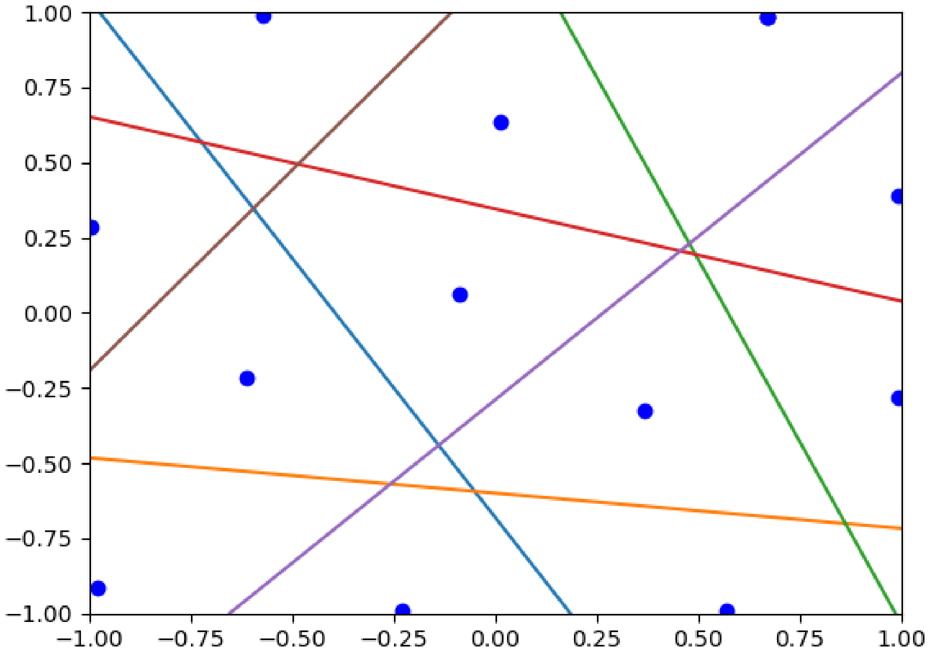
Sometimes there are multiple solutions, the image plotted is dependent on the initialization of the weights.

3. Final image in your report, and include the tensor heart18 in your file encoder.py



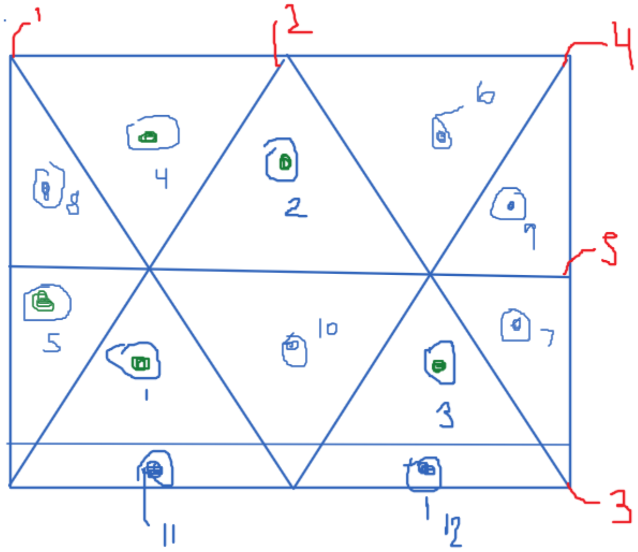
4. Create training data in tensors target1 and target2, which will generate two images of your own design.

Target 1, Zelda's Triforce:

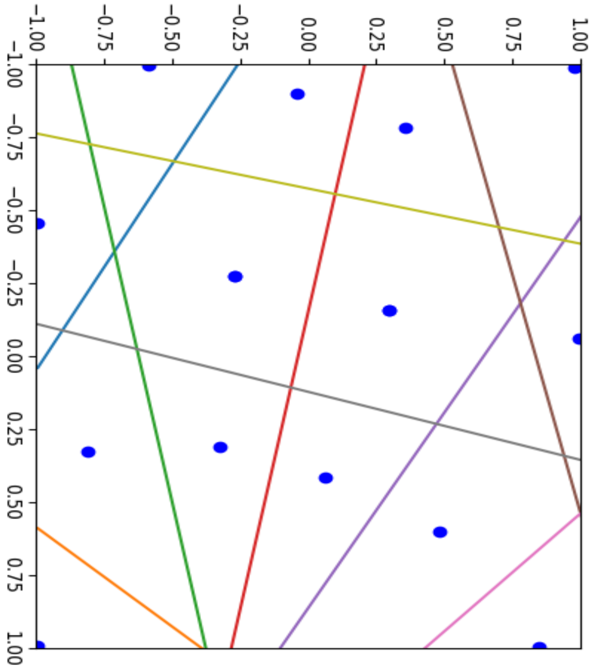


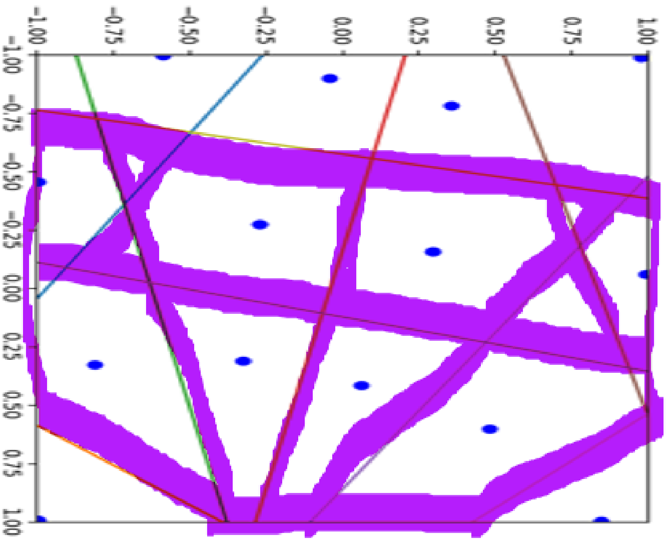
Learning rate is the default value for both targets, increasing the learning rate to .9 decreases the time taken to find the solution but are not necessary for these targets.

Design of Zelda’s Triforce:

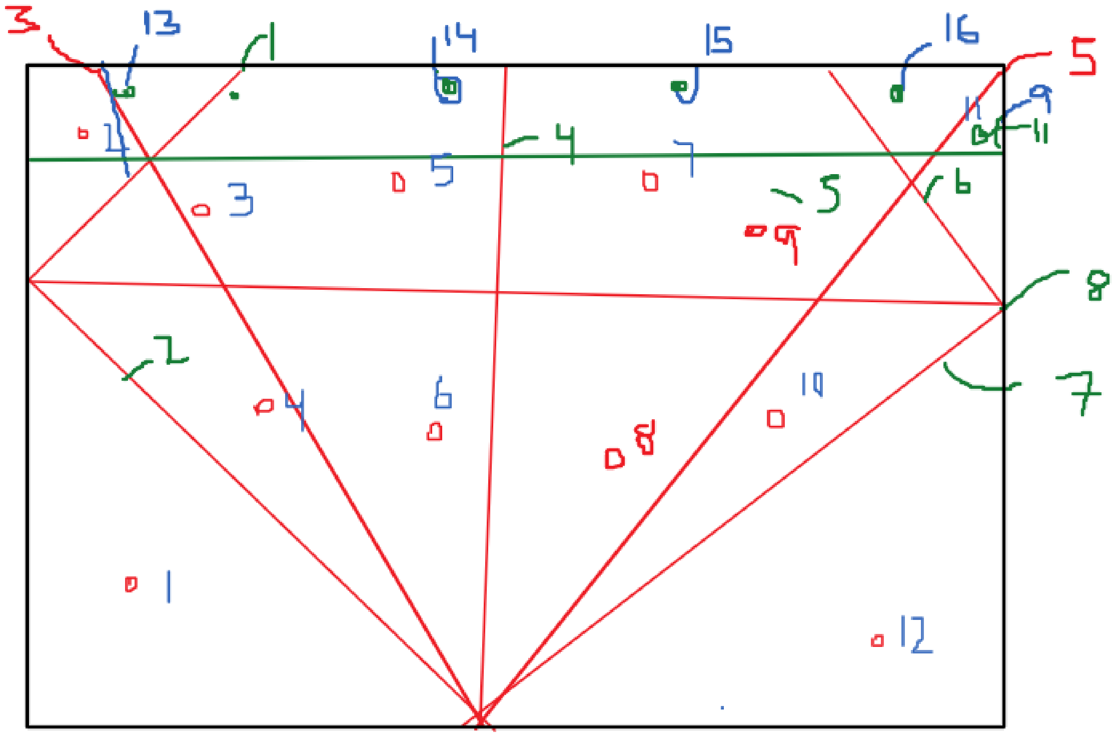


Target 2, Diamond:





Design of the diamond:



Reference:

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/#:~:text=A%20very%20popular%20technique%20that%20is%20used%20along%20with%20SGD,determine%20the%20direction%20to%20go.

https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363

https://medium.com/comet-ml/selecting-the-right-weight-initialization-for-your-deep-neural-network-780e20671b22#:~:text=The%20weight%20initialization%20technique%20you,tune%2C%20they%20are%20incredibly%20important.