Implementing Neural networks – Image processing

Waikei Lau

Part 1 – Japanese character recognition

Step 1 [1 mark]

Confusion matrix and accuracy rate of NetLin computing a linear function of the 28x28 pixels to 10 target characters:

```
<class 'numpy.ndarray'>
[[763.   5.   8.  13.  30.  67.   2.  63.  31.  18.]
 [  7. 671. 106.  18.  26.  22.  57.  15.  25.  53.]
 [  9.  60. 694.  25.  24.  21.  47.  38.  45.  37.]
 [  4.  38.  59. 756.  15.  56.  13.  18.  28.  13.]
 [ 59.  54.  78.  22. 625.  17.  33.  35.  19.  58.]
 [  8.  28. 122.  17.  20. 727.  27.   8.  33.  10.]
 [  5.  21. 148.   9.  25.  25. 723.  20.  10.  14.]
 [ 18.  31.  26.  11.  86.  17.  53. 622.  87.  49.]
 [ 10.  34.  91.  43.   8.  32.  45.   6. 709.  22.]
 [  9.  52.  89.   2.  51.  31.  20.  33.  42. 671.]]

Test set: Average loss: 1.0097, Accuracy: 6961/10000 (70%)
```

Step 2 [1 mark]

Confusion matrix and accuracy rate of NetFull using tanh at hidden layer with 150 hidden nodes and log softmax at the output layer:

```
<class 'numpy.ndarray'>
[[847.   5.   4.   4.  33.  33.   2.  34.  32.   6.]
 [  7. 810.  32.   3.  16.  13.  63.   5.  22.  29.]
 [  9.  16. 841.  35.   9.  18.  26.  11.  19.  16.]
 [  3.   8.  33. 912.   1.  14.   5.   5.   9.  10.]
 [ 40.  31.  19.   7. 805.   9.  32.  16.  22.  19.]
 [  7.  10.  72.  15.  10. 835.  23.   2.  20.   6.]
 [  3.   8.  53.   8.  13.   4. 892.   9.   1.   9.]
 [ 22.   7.  25.   4.  23.  11.  27. 819.  27.  35.]
 [ 10.  26.  28.  46.   3.   8.  30.   3. 841.   5.]
 [  2.  17.  55.   6.  28.   4.  21.  17.  14. 836.]]

Test set: Average loss: 0.5067, Accuracy: 8438/10000 (84%)
```

Step 3 [2 marks]

Confusion matrix and accuracy rate of NetConv. This network utilised two convolutional layers; with max pooling after each convolution, plus one fully connected layer, using relu activation function, followed by the output layer, using log softmax. This model utilised learning rate of 0.05, momentum of 0.6, kernel sizes of 3 for convolutional layers and the first max pooling layer.

```
<class 'numpy.ndarray'>
[[971.   1.   1.   1.  11.   2.   0.   9.   2.   2.]
 [  0. 947.  10.   0.   5.   1.  24.   1.   7.   5.]
 [  9.   6. 920.  33.   5.   9.   6.   6.   2.   4.]
 [  1.   0.  10. 975.   0.   4.   4.   1.   3.   2.]
 [ 17.   7.   6.   8. 923.   5.   9.   2.  20.   3.]
 [  1.   3.  26.   4.   1. 951.   8.   0.   4.   2.]
 [  3.   4.  16.   1.   3.   4. 965.   1.   2.   1.]
 [  7.   2.  11.   2.   3.   3.   6. 944.   7.  15.]
```

```
 [  3.   6.   8.   5.   3.   3.   3.   0. 969.   0.]
 [ 10.   4.   9.   0.   6.   1.   3.   2.   7. 958.]]

Test set: Average loss: 0.2720, Accuracy: 9523/10000 (95%)
```

Step 4 [6 marks]

Observations:

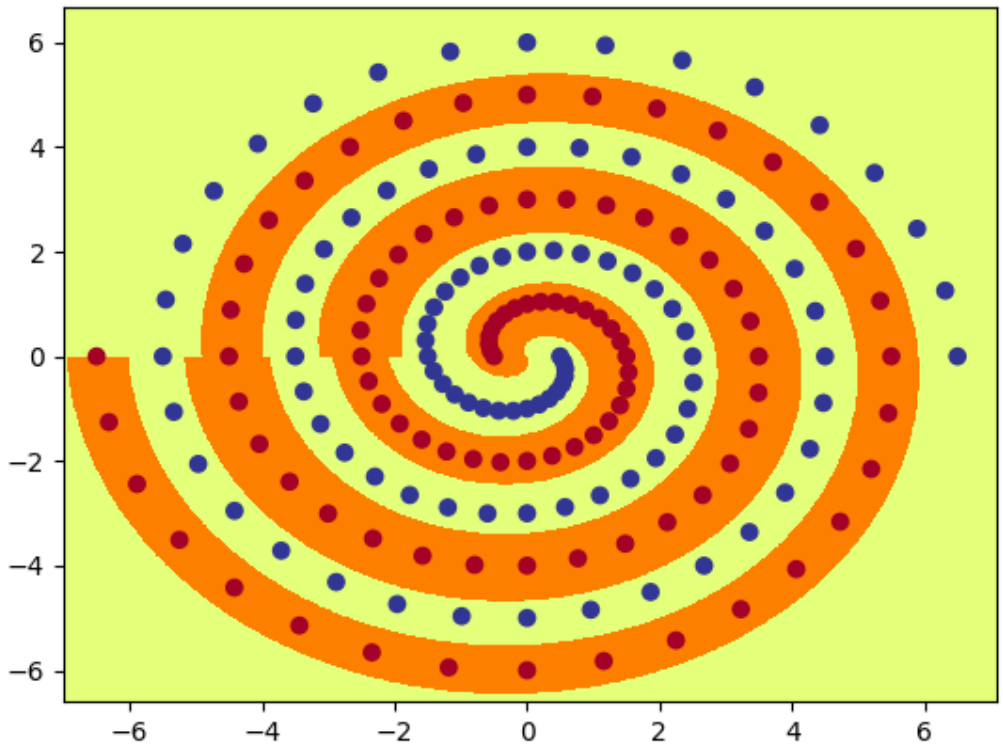
- Our testing indicates that deeper neural networks with optimisation have significant accuracy advantage on the Kuzushiji-MNIST dataset. NetLin was the least accurate of the models (~70%): using single linear layer. Netfull (~84% accuracy) utilised 1x hidden and output layer; while the most accurate model was NetConv (~95%) which utilised 2x convolution and max pooling layers, 1x linear and output layers. It suggests that deeper networks are better equipped to identify curvature characteristics such as convexity and concavity in addition to linearly separable features such as straight vertical/horizontal strokes. Since Japanese characters utilise both components in their typography, deeper neural network is better able to decern characters with such features.
- The confusion matrix indicates that the networks are most prone to err on 2='Su' す and tended to confuse this with 5='ha' は and 6='ma' ま. After visual inspection, the reason appears to be the close typographic style of these characters which utilise similar rounded strokes under crossed horizontal and vertical strokes. These feature similarities make the characters harder to differentiate.
- Our initial testing used 2x traditional convolution layers, with up to 10,000 neurons. The large number of neurons meant training speed was slow and produced an accuracy rate of ~92%. To improve accuracy, the kernel size was reduced from 5 to 3 and zero padding was added. Additionally max pooling was introduced with kernel size of 3 in first layer and 2 in the second layer to simplify and reduce the large number of neurons to ~625. This significantly sped up training and improved accuracy (Ockham's razer) by having a simpler network in terms of the number of neurons, we increase the relative impact of each weight, avoiding exploding or vanishing weights, while avoiding overfitting. In combination with an increase in learning rate from 0.01 to 0.05, the network achieved an accuracy of 95% within 10 epochs.
- Further increasing the learning rate to 0.1 displayed no additional benefit to accuracy and at 0.2 the accuracy decreased to 94%

Part 2 – Twin spirals

Step 1 [1 mark]

Step 2 [1 mark]

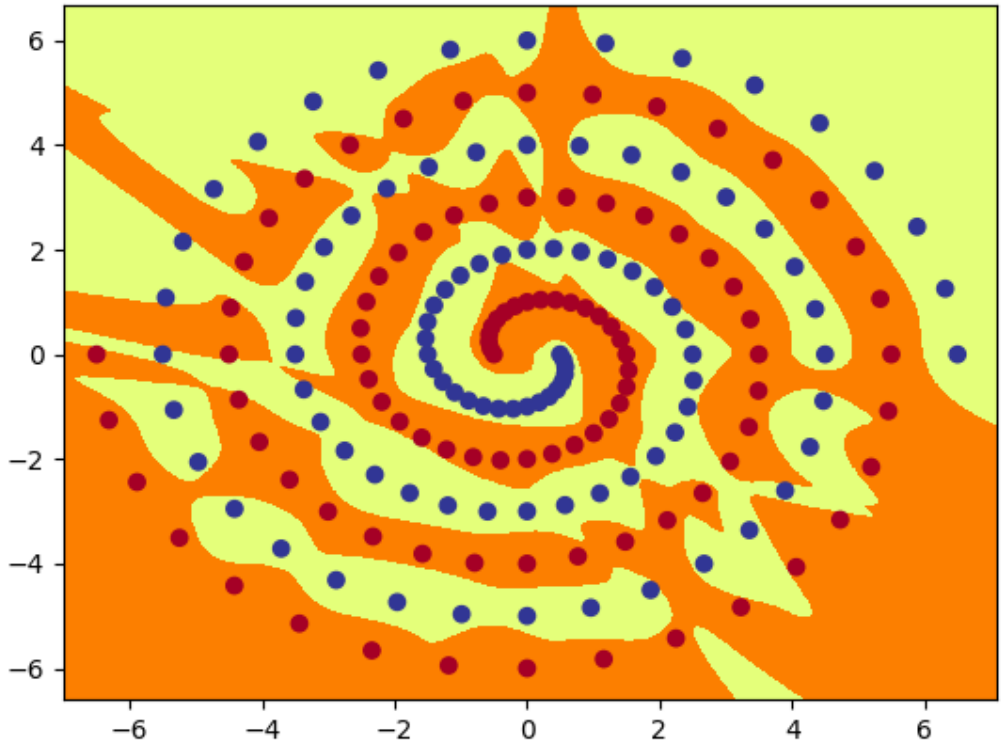
7 hidden nodes are required so that PolarNet learns to correctly classify all the training data within 20000 epochs, on almost all runs. polar_out.png:



Step 3 [1 mark]

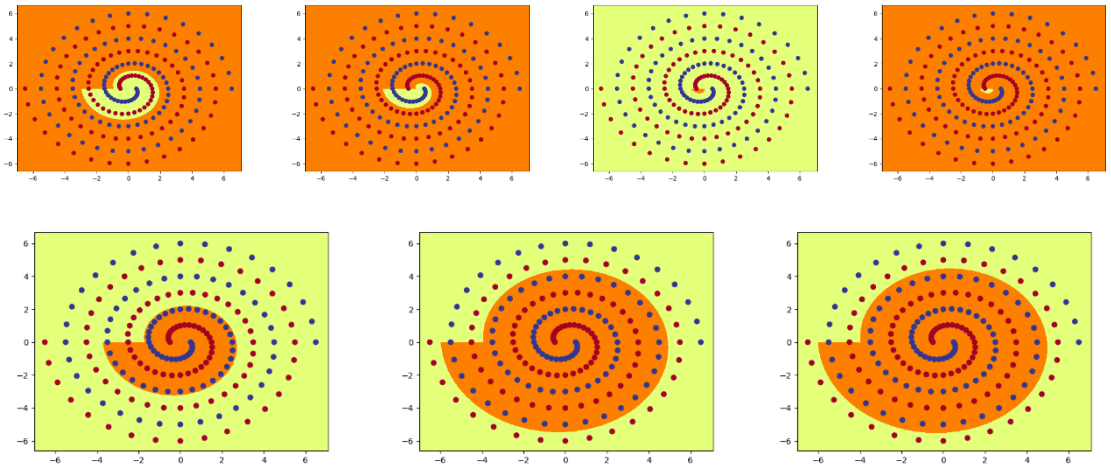
Step 4 [1 mark]

The optimal number of nodes such that this RawNet learns to correctly classify all of the training data within 20000 epochs is 15. Using Xavier initialisation, we can calculate the optimal weight initialisation to be around 0.8

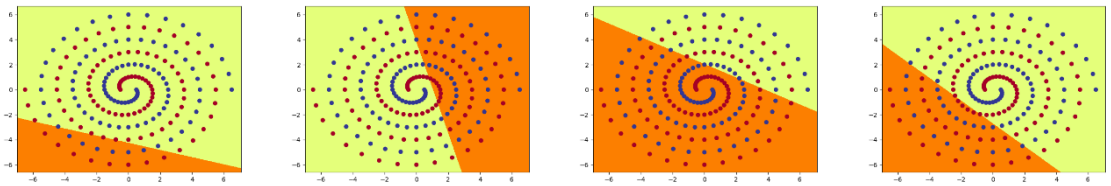


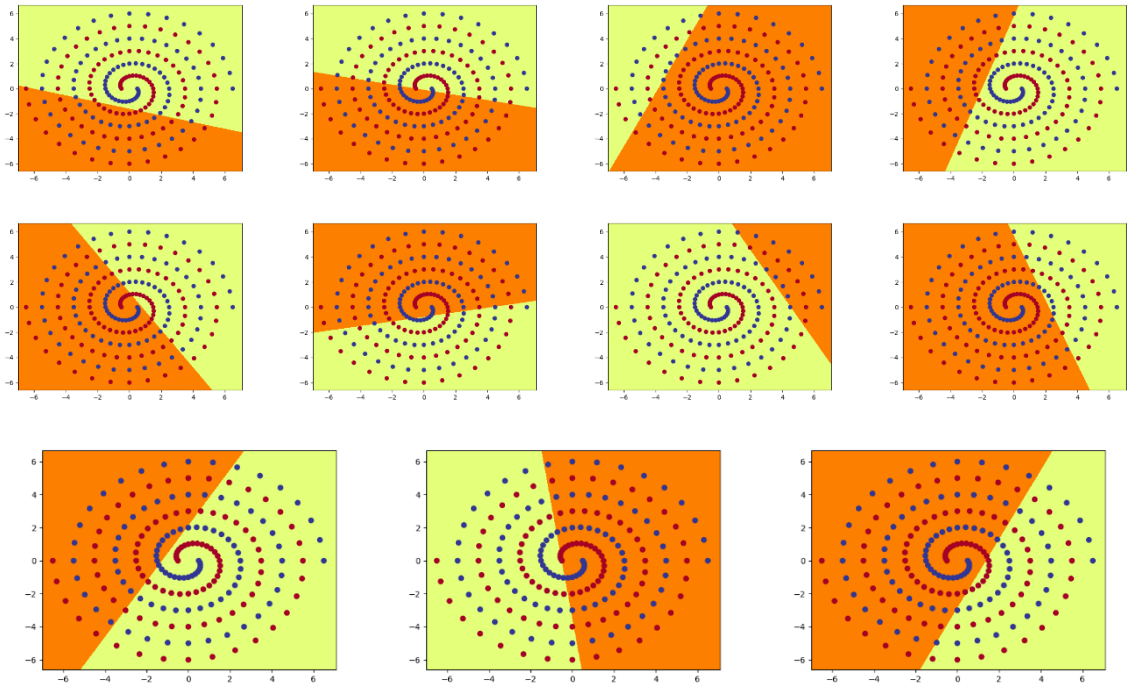
Step 5 [2 marks]

PolarNet layer 1 hidden nodes consecutively from 0 to 6:

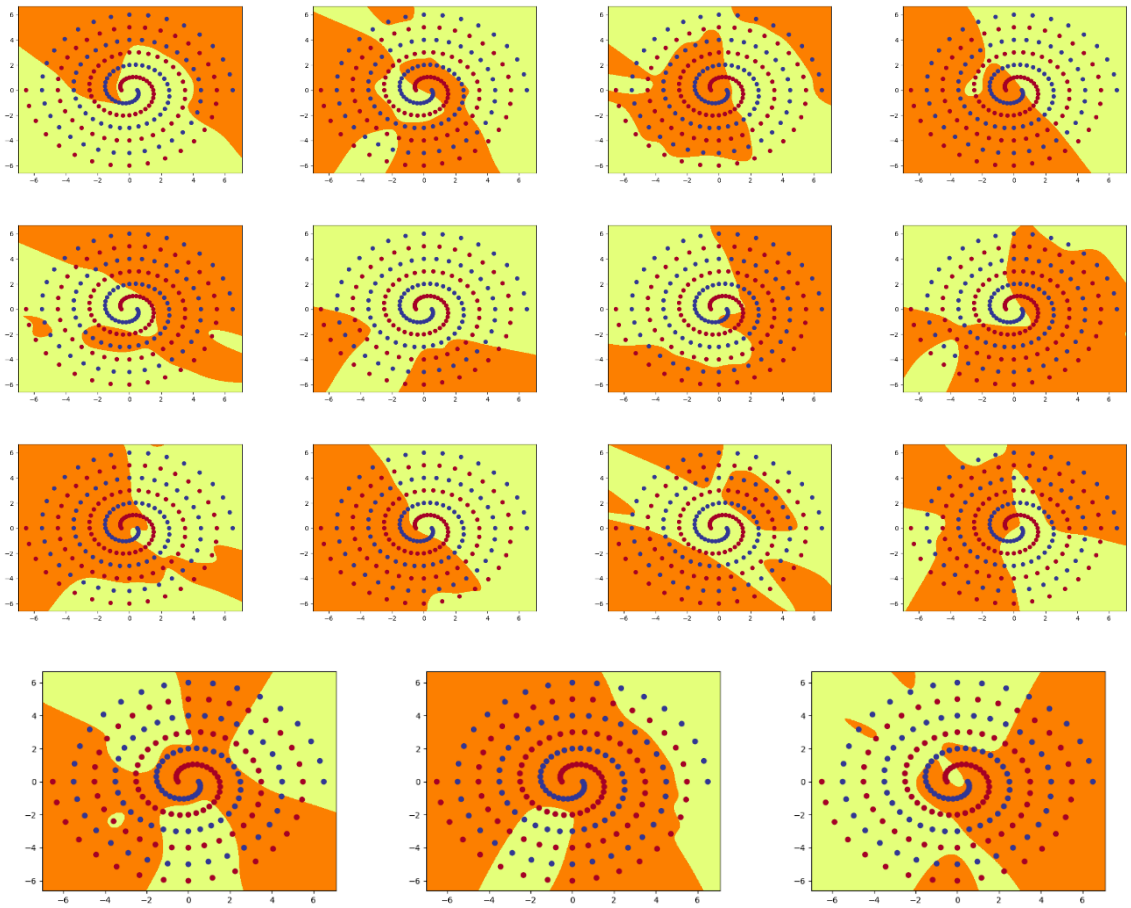


RawNet layer 1 hidden nodes consecutively from 0 to 14:





RawNet layer 2 hidden nodes consecutively from 0 to 14:



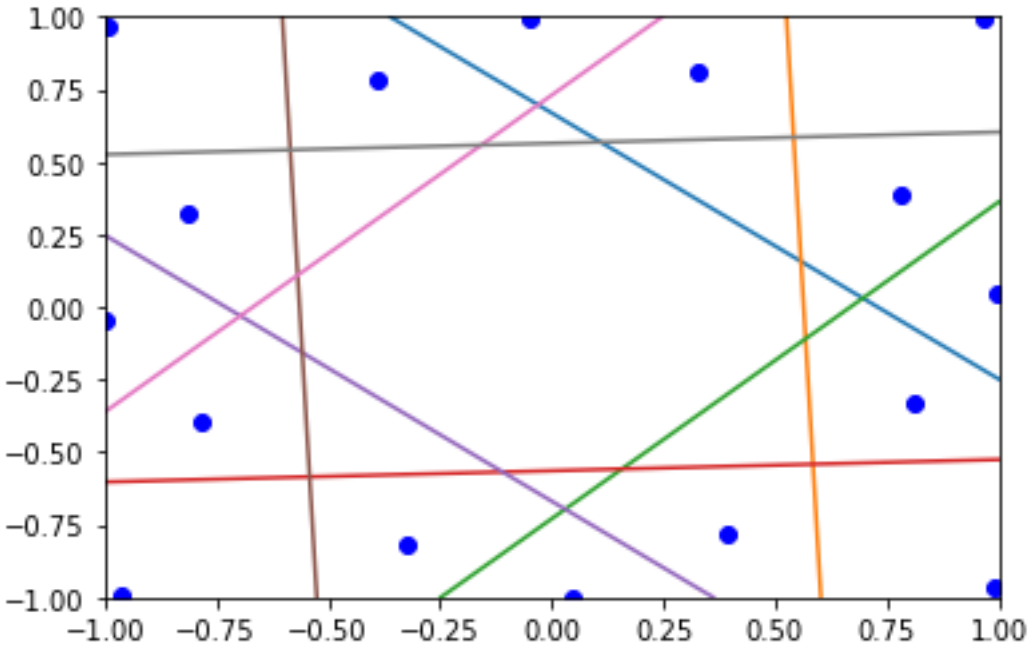
Step 6 [6 marks]

Observations:

- In RawNet, the first hidden layer learns linearly separable features, whilst the second hidden layer learns convex features. The output layer combines these to produce concave features.
- RawNet learns & expresses the twin spirals through continually minimising a cartesian loss function through backpropagation and approximating the linearity, convexity & concavity of the data. On the other hand, PolarNet computes the polar coordinates which expresses data in relation to angles and distances. Thus, the nature of the weights that get adjusted and backpropagated in RawNet are different to the ones being used in RawNet, as can be seen in the node diagrams for both networks. PolarNet is especially suited to application in the twin spirals problem as it can achieve 100% accuracy with only 1 hidden layer and 7 hidden nodes.
- It is to be noted however that weight initialisation is required for RawNet to achieve 100% accuracy. Values from 0.1 to 0.8 were tested (batch size 194) and all enabled the net to achieve 100% accuracy
- Increasing batch size from 97 to 194 significantly improved training speed for both PolarNet (lr 0.05) and RawNet (lr 0.01) to within 5000 epochs
- For both PolarNet and RawNet, utilising SGD optimiser instead of Adam, significantly decreased performance such that the network was not able to learn within 20000 epochs.
- Increasing the learning rate to 0.05 significantly improved PolarNet performance achieving 100% accuracy within 10000 epochs almost always. However the same was not true for RawNet at lr = 0.05 which had a tendency to jump back and forth in terms of accuracy, and failed to achieve 100% accuracy within 20000 epochs.
- It was found that PolarNet (lr 0.01) would occasionally experience the problem of vanishing weights, despite using weight initialisation. In this scenario it is suggested that further testing with batch normalisation may mitigate the issue. On the other hand, weight initialisation at 0.8 was able to mitigate the vanishing weights problem for RawNet.
- Relative to RawNet (lr 0.01), PolarNet at lr 0.05 displayed similar performance, consistently achieving 100% accuracy within 10000 epochs
- In considering the usage of ReLu vs Tanh, it was found that ReLu significantly decreased the performance of both networks and did not achieve 100% accuracy within 20000 epochs. Since ReLu is mainly used for deep networks (including convolutional networks) its application here is less effective than Tanh.

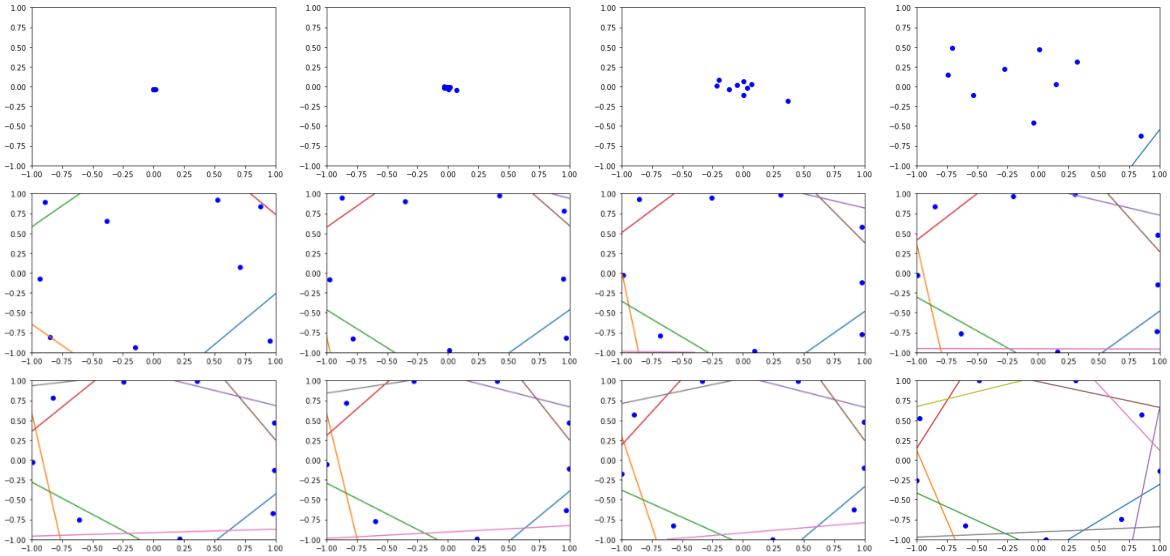
Part 3 – Hidden unit dynamics

Step 1 [1 mark]

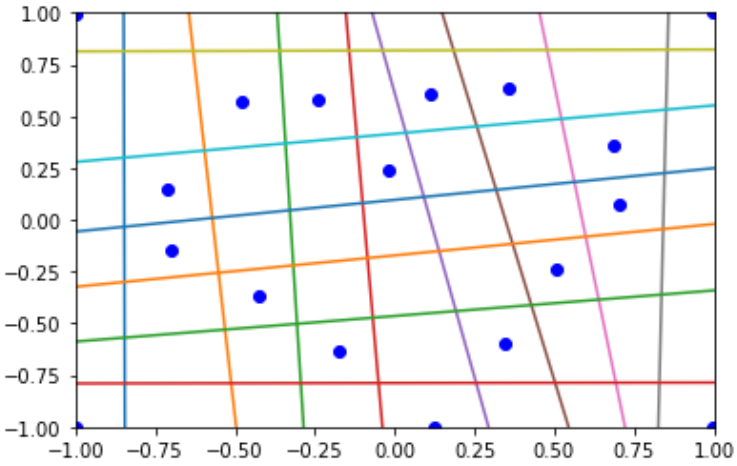


Step 2 [2 marks]

9-2-9 Encoder outputs displayed sequentially from epochs 50 – 3000. We see the net learn the weights over the epochs as the dots intersperse to follow the coordinates of the weights applied to the 9 inputs from each node (on the axis). As the dots reach their optimal weight coordinates, each line represents an output node boundary condition that defines activation (>50%) for the corresponding dot (input). It is evident that the dots are well dispersed around the edges of the map and indicate that the weights being applied to each input are significantly differentiable (separable) from each other, thus the effects of each input are also distinguishable.



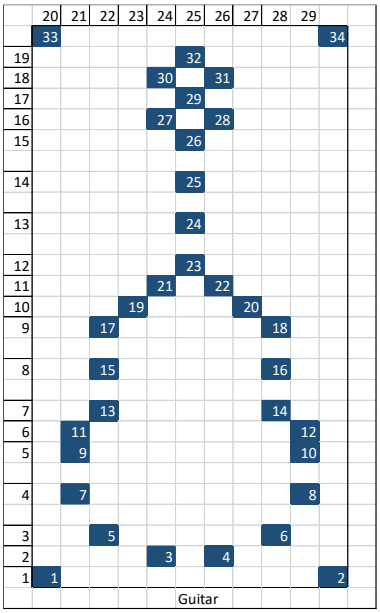
Step 3 [2 marks]



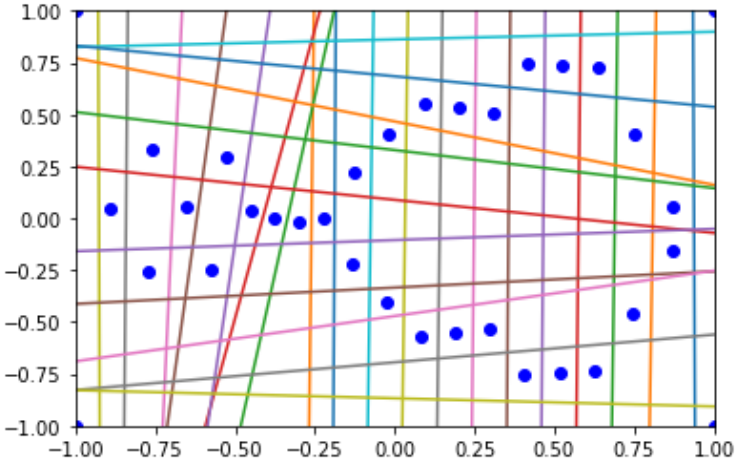
Step 4 [3 marks]

Using --lr 0.6 --epochs 50000

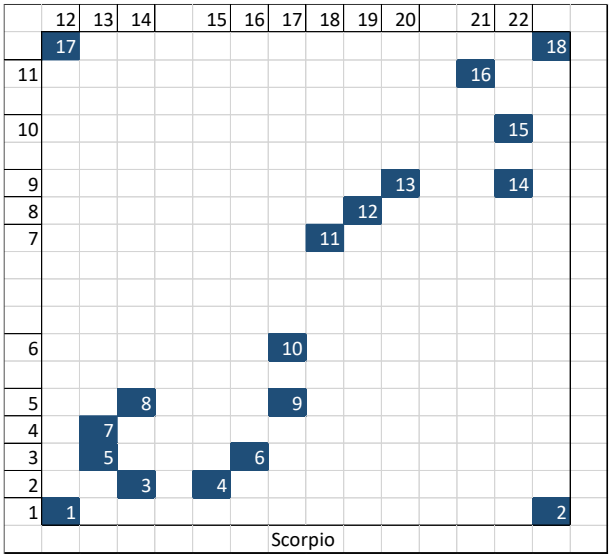
Original mapping for Guitar:



Target 1 output: Guitar



Original mapping for star sign Scorpio:



Target 2 output: Scorpio