## Part 1: Japanese Character Recognition

For Part 1 of the assignment you will be implementing networks to recognize handwritten Hiragana symbols. The dataset to be used is Kuzushiji-MNIST or KMNIST for short. The paper describing the dataset is available here. It is worth reading, but in short: significant changes occurred to the language when Japan reformed their education system in 1868, and the majority of Japanese today cannot read texts published over 150 years ago. This paper presents a dataset of handwritten, labeled examples of this old-style script (Kuzushiji). Along with this dataset, however, they also provide a much simpler one, containing 10 Hiragana characters with 7000 samples per class. This is the dataset we will be using.

Text from 1772 (left) compared to 1900 showing the standardization of written Japanese.

[1 mark] Implement a model NetLin which computes a linear function of the pixels in the image, followed by log softmax. Run the code by typing:

python3 kuzu_main.py --net lin

Copy the final accuracy and confusion matrix into your report. The final accuracy should be around 70%. Note that the rows of the confusion matrix indicate the target character, while the columns indicate the one chosen by the network. (0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo"). More examples of each character can be found here.

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.813031
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.635140
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.594049
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.598151
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.321043
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.514645
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.658641
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.617658
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.351468
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.669462
<class 'numpy.ndarray'>
[[768.   5.   8.  12.  30.  65.   2.  61.  30.  19.]
 [  6. 668. 106.  19.  32.  22.  59.  13.  24.  51.]
 [  8.  60. 692.  26.  25.  21.  46.  38.  46.  38.]
 [  4.  34.  57. 758.  16.  58.  14.  18.  30.  11.]
 [ 57.  54.  79.  21. 626.  19.  31.  36.  20.  57.]
 [  8.  27. 125.  17.  19. 724.  28.   9.  32.  11.]
 [  5.  22. 142.  10.  26.  24. 726.  20.  11.  14.]
 [ 17.  31.  29.  12.  83.  16.  53. 626.  85.  48.]
 [ 11.  40.  90.  41.   7.  31.  47.   7. 702.  24.]
 [  7.  52.  83.   3.  52.  34.  18.  31.  39. 681.]]

Test set: Average loss: 1.0091, Accuracy: 6971/10000 (70%)
```

Train Epoch: 10 [0/60000 (0%)] Loss: 0.813031

Train Epoch: 10 [6400/60000 (11%)] Loss: 0.635140

Train Epoch: 10 [12800/60000 (21%)] Loss: 0.594049

Train Epoch: 10 [19200/60000 (32%)] Loss: 0.598151

Train Epoch: 10 [25600/60000 (43%)] Loss: 0.321043

Train Epoch: 10 [32000/60000 (53%)] Loss: 0.514645

Train Epoch: 10 [38400/60000 (64%)] Loss: 0.658641

Train Epoch: 10 [44800/60000 (75%)] Loss: 0.617658

Train Epoch: 10 [51200/60000 (85%)] Loss: 0.351468

Train Epoch: 10 [57600/60000 (96%)] Loss: 0.669462

<class 'numpy.ndarray'>

[[768. 5. 8. 12. 30. 65. 2. 61. 30. 19.]

[ 6. 668. 106. 19. 32. 22. 59. 13. 24. 51.]

[ 8. 60. 692. 26. 25. 21. 46. 38. 46. 38.]

**[ 4. 34. 57. 758. 16. 58. 14. 18. 30. 11.]**

**[ 57. 54. 79. 21. 626. 19. 31. 36. 20. 57.]**

**[ 8. 27. 125. 17. 19. 724. 28. 9. 32. 11.]**

**[ 5. 22. 142. 10. 26. 24. 726. 20. 11. 14.]**

**[ 17. 31. 29. 12. 83. 16. 53. 626. 85. 48.]**

**[ 11. 40. 90. 41. 7. 31. 47. 7. 702. 24.]**

**[ 7. 52. 83. 3. 52. 34. 18. 31. 39. 681.]]**

**Test set: Average loss: 1.0091, Accuracy: 6971/10000 (70%)**

**[1 mark]** Implement a fully connected 2-layer network NetFull (i.e. one hidden layer, plus the output layer), using tanh at the hidden nodes and log softmax at the output node. Run the code by typing:

python3 kuzu_main.py --net full

Try different values (multiples of 10) for the number of hidden nodes and try to determine a value that achieves high accuracy (at least 84%) on the test set. Copy the final accuracy and confusion matrix into your report, and include a calculation of the total number of independent parameters in the network.

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.333290
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.265283
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.229493
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.232449
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.141605
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.271098
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.216205
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.370652
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.139398
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.274301
<class 'numpy.ndarray'>
[[856.   4.   1.   7.  29.  22.   2.  39.  34.   6.]
 [  5. 807.  33.   5.  23.  10.  60.   6.  20.  31.]
 [  7.  13. 831.  39.  13.  22.  25.  10.  26.  14.]
 [  2.  13.  41. 914.   4.   9.   2.   3.   5.   7.]
 [ 49.  30.  21.   7. 804.   6.  27.  16.  19.  21.]
 [  9.   8.  81.   7.  18. 822.  30.   2.  18.   5.]
 [  3.  14.  58.   9.  16.   5. 879.   7.   2.   7.]
 [ 17.  13.  24.   5.  35.   6.  38. 803.  25.  34.]
 [  8.  21.  26.  33.   3.  12.  32.   1. 853.  11.]
 [  3.  22.  48.   3.  35.   6.  23.  14.  15. 831.]]

Test set: Average loss: 0.5265, Accuracy: 8400/10000 (84%)
```

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.333290

Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.265283

Train Epoch: 10 [12800/60000 (21%)]  Loss: 0.229493

Train Epoch: 10 [19200/60000 (32%)]  Loss: 0.232449

Train Epoch: 10 [25600/60000 (43%)]  Loss: 0.141605

Train Epoch: 10 [32000/60000 (53%)]  Loss: 0.271098

Train Epoch: 10 [38400/60000 (64%)]  Loss: 0.216205

Train Epoch: 10 [44800/60000 (75%)]  Loss: 0.370652

Train Epoch: 10 [51200/60000 (85%)]  Loss: 0.139398

Train Epoch: 10 [57600/60000 (96%)]  Loss: 0.274301

<class 'numpy.ndarray'>

[[856.  4.  1.  7.  29.  22.  2.  39.  34.  6.]

 [  5. 807.  33.  5.  23.  10.  60.  6.  20.  31.]

 [  7.  13. 831.  39.  13.  22.  25.  10.  26.  14.]

**[ 2. 13. 41. 914. 4. 9. 2. 3. 5. 7.]**

**[ 49. 30. 21. 7. 804. 6. 27. 16. 19. 21.]**

**[ 9. 8. 81. 7. 18. 822. 30. 2. 18. 5.]**

**[ 3. 14. 58. 9. 16. 5. 879. 7. 2. 7.]**

**[ 17. 13. 24. 5. 35. 6. 38. 803. 25. 34.]**

**[ 8. 21. 26. 33. 3. 12. 32. 1. 853. 11.]**

**[ 3. 22. 48. 3. 35. 6. 23. 14. 15. 831.]]**

**Test set: Average loss: 0.5265, Accuracy: 8400/10000 (84%)**

**[2 marks]** Implement 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. You are free to choose for yourself the number and size of the filters, metaparameter values (learning rate and momentum), and whether to use max pooling or a fully convolutional architecture. Run the code by typing:

python3 kuzu_main.py --net conv

Your network should consistently achieve at least 93% accuracy on the test set after 10 training epochs. Copy the final accuracy and confusion matrix into your report, and include a calculation of the total number of independent parameters in the network.

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.096813
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.066735
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.076549
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.131862
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.072595
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.117876
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.058397
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.259117
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.015804
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.068783
<class 'numpy.ndarray'>
[[966.   2.   3.   0.  11.   5.   0.   8.   2.   3.]
 [  8. 914.   8.   1.  11.   1.  38.   5.   6.   8.]
 [ 10.   8. 842.  52.   8.  16.  35.  16.   3.  10.]
 [  1.   4.  20. 956.   1.   5.   6.   3.   2.   2.]
 [ 27.   3.   4.  12. 904.   0.  22.   8.  14.   6.]
 [  3.  14.  35.  10.   4. 901.  24.   3.   3.   3.]
 [  3.   2.  12.   4.   7.   2. 963.   3.   2.   2.]
 [  8.   5.   3.   1.   9.   2.   7. 934.  11.  20.]
 [  5.   7.   5.   2.   3.   3.   7.   2. 965.   1.]
 [  6.   4.   4.   4.   9.   4.   2.   5.   6. 956.]]

Test set: Average loss: 0.2496, Accuracy: 9301/10000 (93%)
```

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.096813

Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.066735

Train Epoch: 10 [12800/60000 (21%)]  Loss: 0.076549

Train Epoch: 10 [19200/60000 (32%)]  Loss: 0.131862

Train Epoch: 10 [25600/60000 (43%)]  Loss: 0.072595

Train Epoch: 10 [32000/60000 (53%)]  Loss: 0.117876

Train Epoch: 10 [38400/60000 (64%)]  Loss: 0.058397

Train Epoch: 10 [44800/60000 (75%)]  Loss: 0.259117

Train Epoch: 10 [51200/60000 (85%)]  Loss: 0.015804

Train Epoch: 10 [57600/60000 (96%)]  Loss: 0.068783

<class 'numpy.ndarray'>

[[966.  2.  3.  0.  11.  5.  0.  8.  2.  3.]

 [  8. 914.  8.  1.  11.  1.  38.  5.  6.  8.]

 [ 10.  8. 842.  52.  8.  16.  35.  16.  3.  10.]

[ 1. 4. 20. 956. 1. 5. 6. 3. 2. 2.]

[ 27. 3. 4. 12. 904. 0. 22. 8. 14. 6.]

[ 3. 14. 35. 10. 4. 901. 24. 3. 3. 3.]

[ 3. 2. 12. 4. 7. 2. 963. 3. 2. 2.]

[ 8. 5. 3. 1. 9. 2. 7. 934. 11. 20.]

[ 5. 7. 5. 2. 3. 3. 7. 2. 965. 1.]

[ 6. 4. 4. 4. 9. 4. 2. 5. 6. 956.]]

**Test set: Average loss: 0.2496, Accuracy: 9301/10000 (93%)**

**Nmber of independent parameters：**

**(1*64*5*5+64)+(64*128*3*3+128)+(128*7*7*10+10)=138250**

**[4 marks]** Briefly discuss the following points:

the relative accuracy of the three models,

the number of independent parameters in each of the three models,

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

| Mdel | Accuracy | Nmber of independent parameters |
|---|---|---|
| NetLin | 70% | 28*28*10+10=7850 |
| NetFull | 84% | (84*100+100)+(100*10+10) = 79510 |
| NetConv | 93% | (1*64*5*5+64)+ (64*128*3*3+128)+ (128*7*7*10+10)=138250 |

**Confusion Matrix Analysis:**

**NetLin:**

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.813031
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.635140
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.594049
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.598151
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.321043
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.514645
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.658641
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.617658
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.351468
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.669462
<class 'numpy.ndarray'>
[[768.   5.   8.  12.  30.  65.   2.  61.  30.  19.]
 [  6. 668. 106.  19.  32.  22.  59.  13.  24.  51.]
 [  8.  60. 692.  26.  25.  21.  46.  38.  46.  38.]
 [  4.  34.  57. 758.  16.  58.  14.  18.  30.  11.]
 [ 57.  54.  79.  21. 626.  19.  31.  36.  20.  57.]
 [  8.  27. 125.  17.  19. 724.  28.   9.  32.  11.]
 [  5.  22. 142.  10.  26.  24. 726.  20.  11.  14.]
 [ 17.  31.  29.  12.  83.  16.  53. 626.  85.  48.]
 [ 11.  40.  90.  41.   7.  31.  47.   7. 702.  24.]
 [  7.  52.  83.   3.  52.  34.  18.  31.  39. 681.]]

Test set: Average loss: 1.0091, Accuracy: 6971/10000 (70%)
```

ki->su 106 times

ma->su 142 times

ha->su 125 times

Reason: With only linear projection, the model relies heavily on global shape, leading to massive confusion between characters that look vaguely similar overall.

NetFull:

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.333290
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.265283
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.229493
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.232449
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.141605
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.271098
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.216205
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.370652
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.139398
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.274301
<class 'numpy.ndarray'>
[[856.   4.   1.   7.  29.  22.   2.  39.  34.   6.]
 [  5. 807.  33.   5.  23.  10.  60.   6.  20.  31.]
 [  7.  13. 831.  39.  13.  22.  25.  10.  26.  14.]
 [  2.  13.  41. 914.   4.   9.   2.   3.   5.   7.]
 [ 49.  30.  21.   7. 804.   6.  27.  16.  19.  21.]
 [  9.   8.  81.   7.  18. 822.  30.   2.  18.   5.]
 [  3.  14.  58.   9.  16.   5. 879.   7.   2.   7.]
 [ 17.  13.  24.   5.  35.   6.  38. 803.  25.  34.]
 [  8.  21.  26.  33.   3.  12.  32.   1. 853.  11.]
 [  3.  22.  48.   3.  35.   6.  23.  14.  15. 831.]]

Test set: Average loss: 0.5265, Accuracy: 8400/10000 (84%)
```

**ha->su 81 times**

**ki->ma 60 times**

**ma->su 58 times**

**Reason: the model is still sensitive to shape overlap.**

**NetConv:**

**su->tsu 52 times**

**ki->ma 38 times**

**ha->su 35 times**

**Reason: The remaining confusions mostly involve characters that are genuinely similar.**

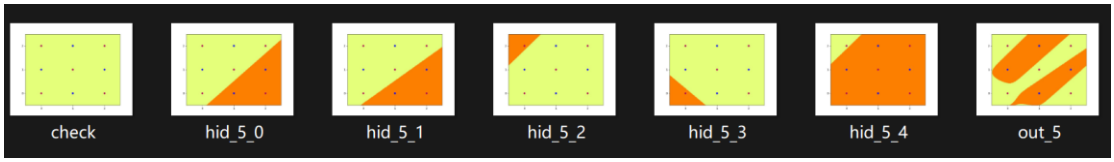
## Part 2: Multi-Layer Perceptron

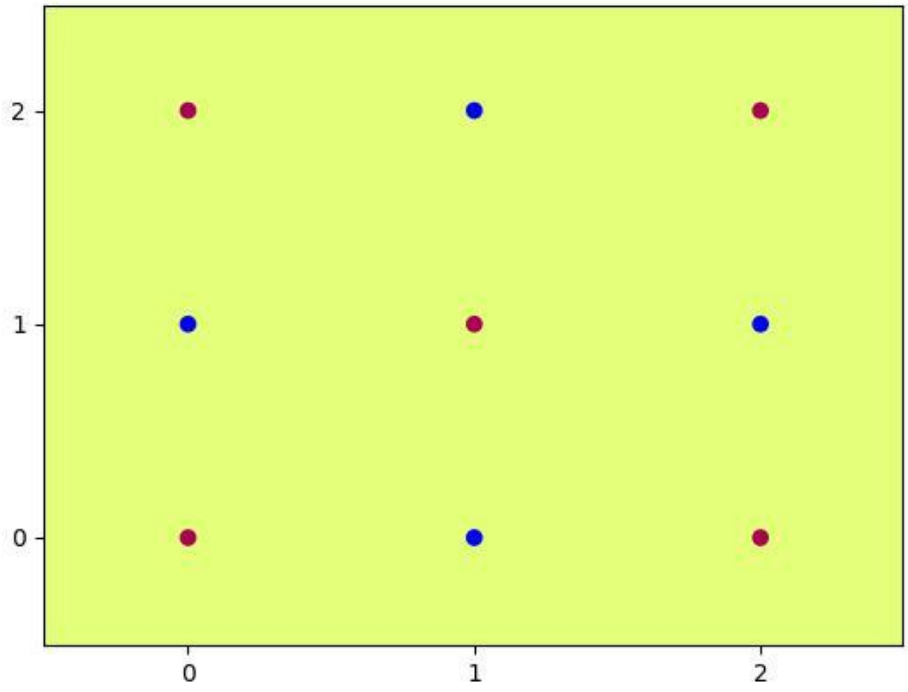
In Part 2 you will be exploring 2-layer neural networks (either trained, or designed by hand) to classify the following data:

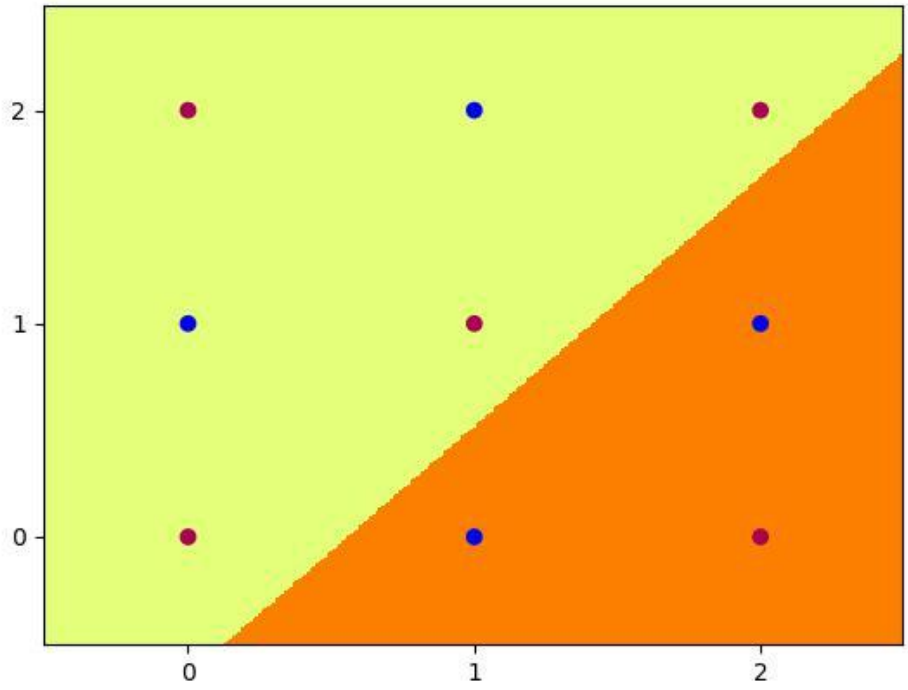
[1 mark] Train a 2-layer neural network with 5 hidden nodes, using sigmoid activation at both the hidden and output layer, on the above data, by typing:

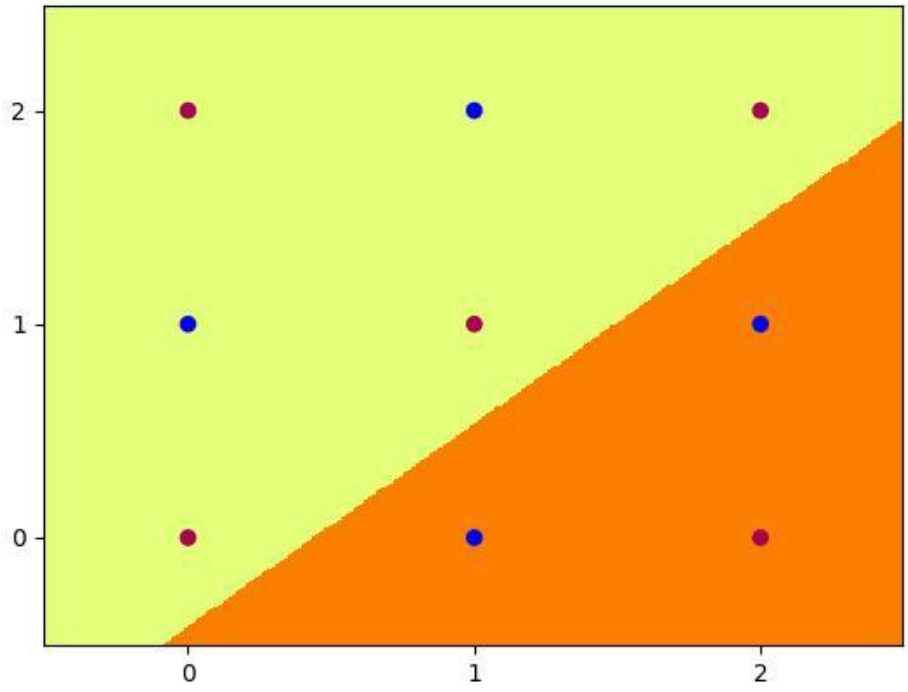
python3 check_main.py --act sig --hid 5

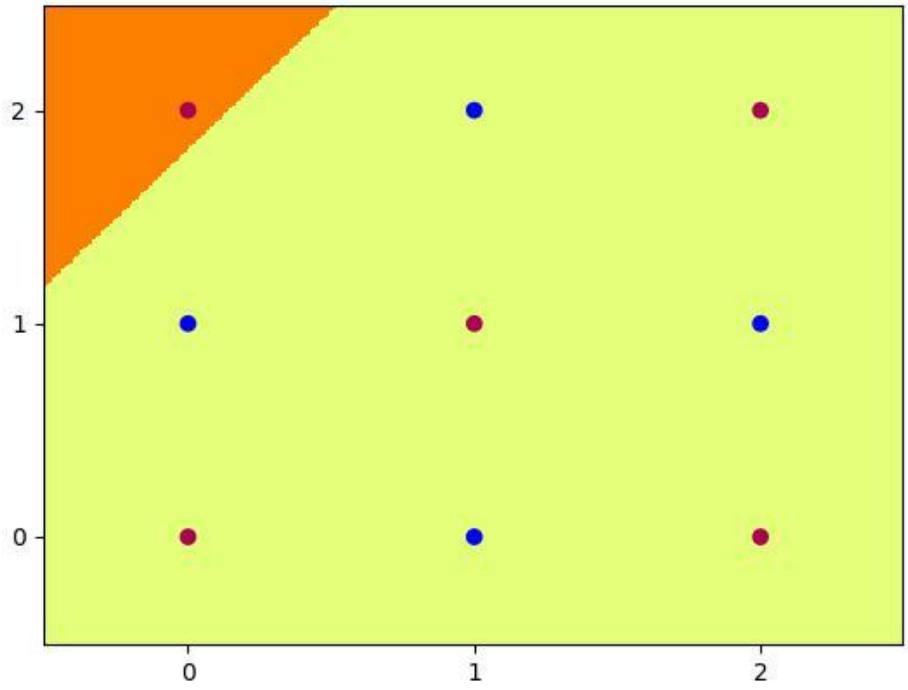
You may need to run the code a few times, until it achieves accuracy of 100%. If the network appears to be stuck in a local minimum, you can terminate the process with ⟨ctrl⟩-C and start again. You are free to adjust the learning rate and the number of hidden nodes, if you wish (see code for details). The code should produce images in the plot subdirectory graphing the function computed by each hidden node (hid_5_?.jpg) and the network as a whole (out_5.jpg). Copy these images into your report.

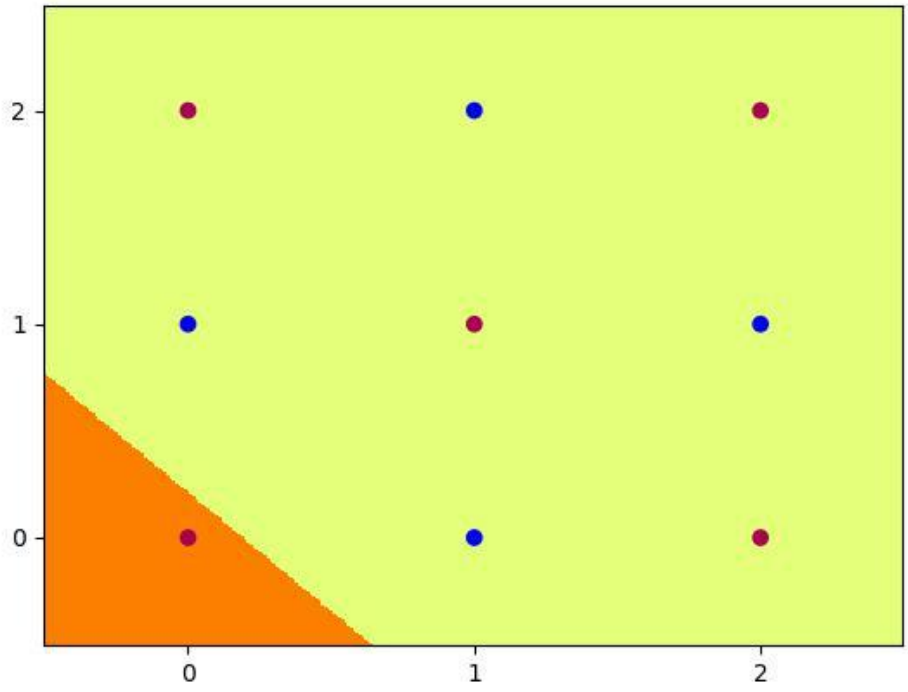


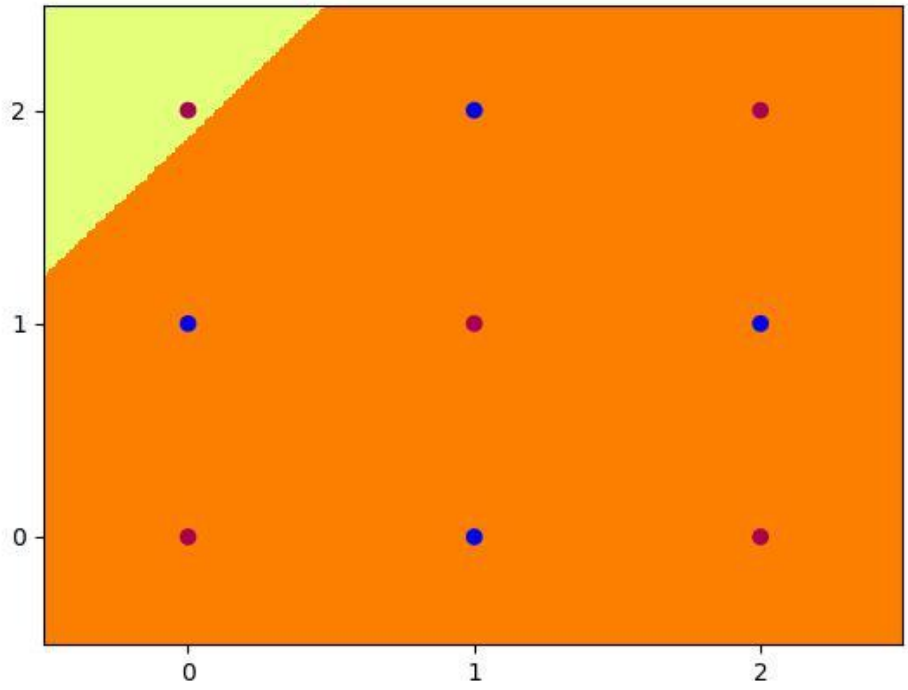

2
1
0
0
1
2

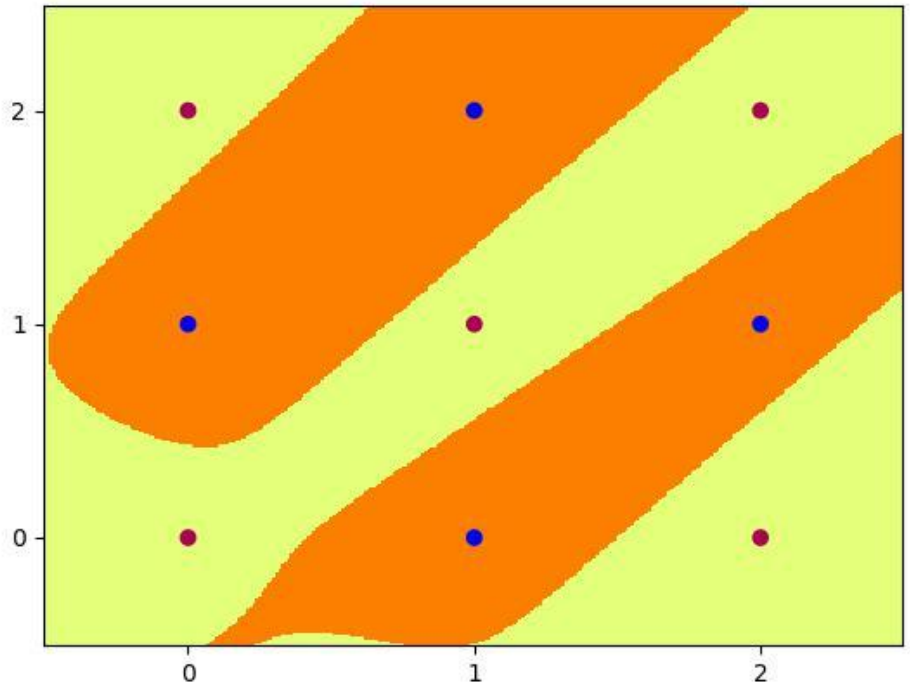

2
1
0
0
1
2


2
1
0
0
1
2


2
1
0
0
1
2

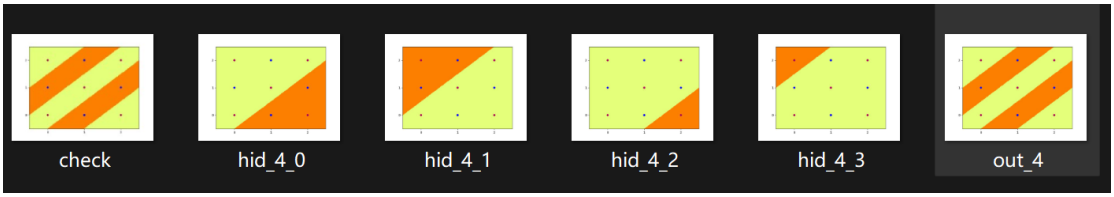

2
1
0
0
1
2


2
1
0
0
1
2

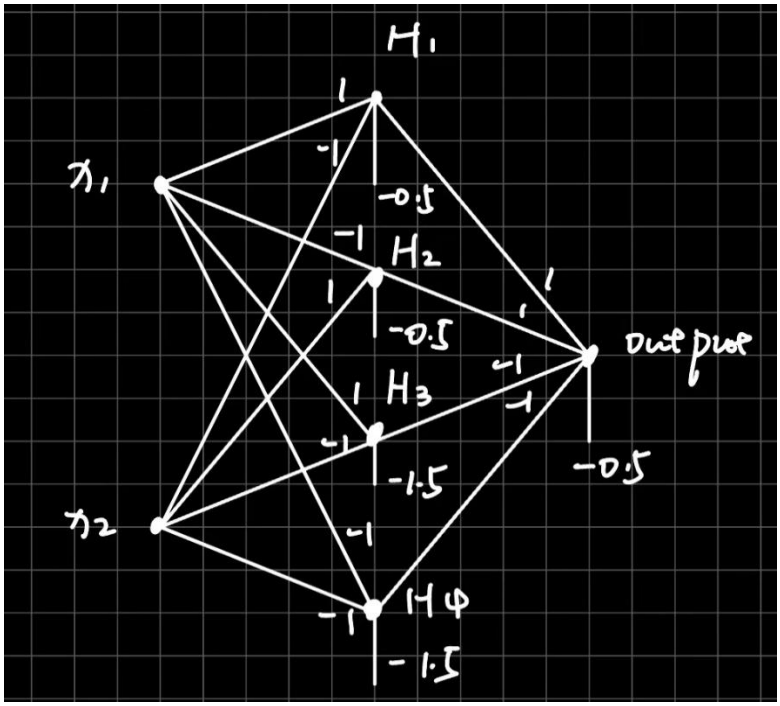



[2 marks] Design by hand a 2-layer neural network with 4 hidden nodes, using the Heaviside (step) activation function at both the hidden and output layer, which correctly classifies the above data. Include a diagram of the network in your report, clearly showing the value of all the weights and biases. Write the equations for the dividing line determined by each hidden node. Create a table showing the activations of all the hidden nodes and the output node, for each of the 9 training items, and include it in your report. You can check that your weights are correct by entering them in the section of check.py where it says "Enter Weights Here", and typing:

python3 check_main.py --act step --hid 4 --set_weights



```
PS C:\Users\admin\Desktop\document\9444\assignment\a1\a1> python check_main.py --act step --hid 4 --set_weights
Initial Weights:
tensor([[ 1., -1.],
        [-1.,  1.],
        [ 1., -1.],
        [-1.,  1.]])
tensor([-0.5000, -0.5000, -1.5000, -1.5000])
tensor([[ 1.,  1., -1., -1.]])
tensor([-0.5000])
Initial Accuracy:  100.0
PS C:\Users\admin\Desktop\document\9444\assignment\a1\a1>
```




| X1 | X2 | H1 | H2 | H3 | H4 | Output |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 0 | 2 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 2 | 0 | 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 1 | 1 | 0 | 0 | 0 | 1 |
| 2 | 2 | 0 | 0 | 0 | 0 | 0 |

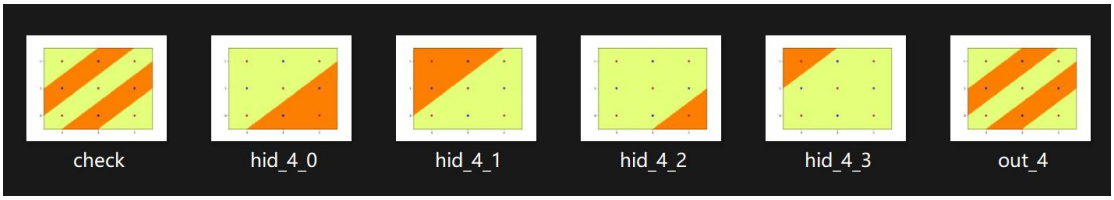
[1 mark] Now rescale your hand-crafted weights and biases from Part 2 by multiplying all of them by a large (fixed) number (for example, 10) so that the combination of

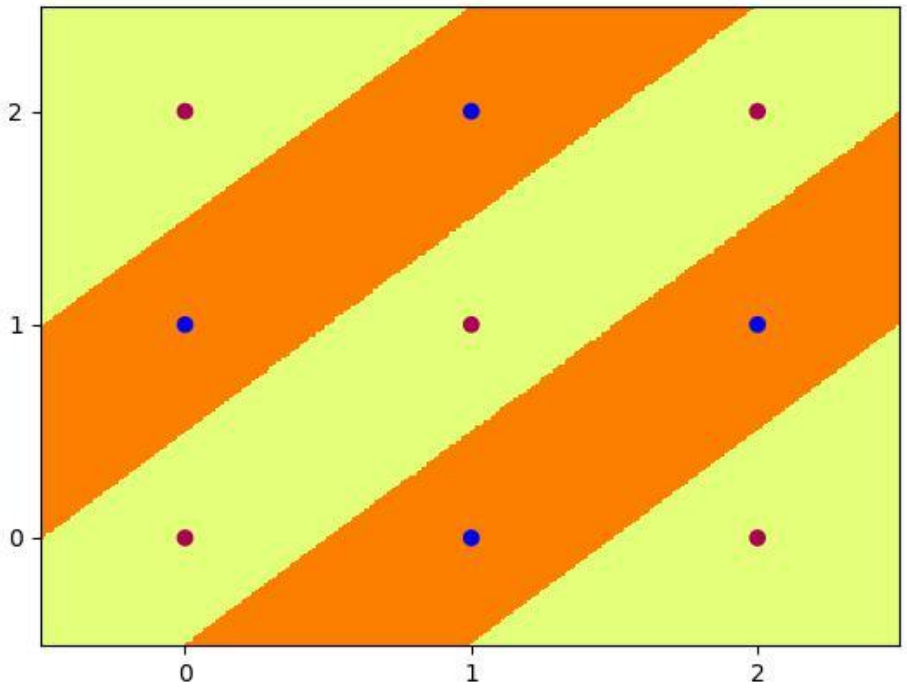
rescaling followed by sigmoid will mimic the effect of the step function. With these re-scaled weights and biases, the data should be correctly classified by the sigmoid network as well as the step function network. Verify that this is true by typing:

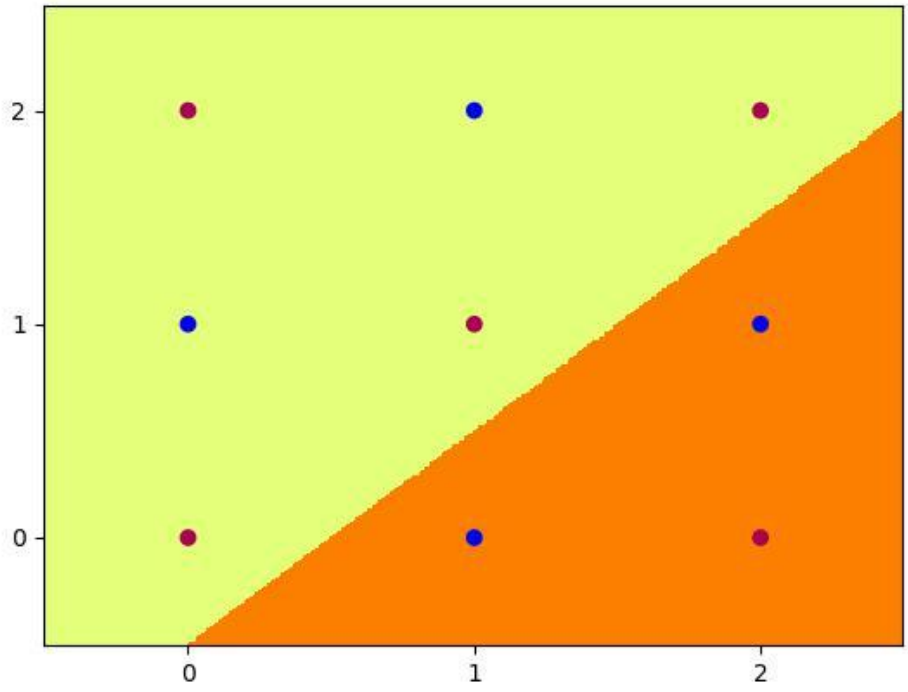
python3 check_main.py --act sig --hid 4 --set_weights

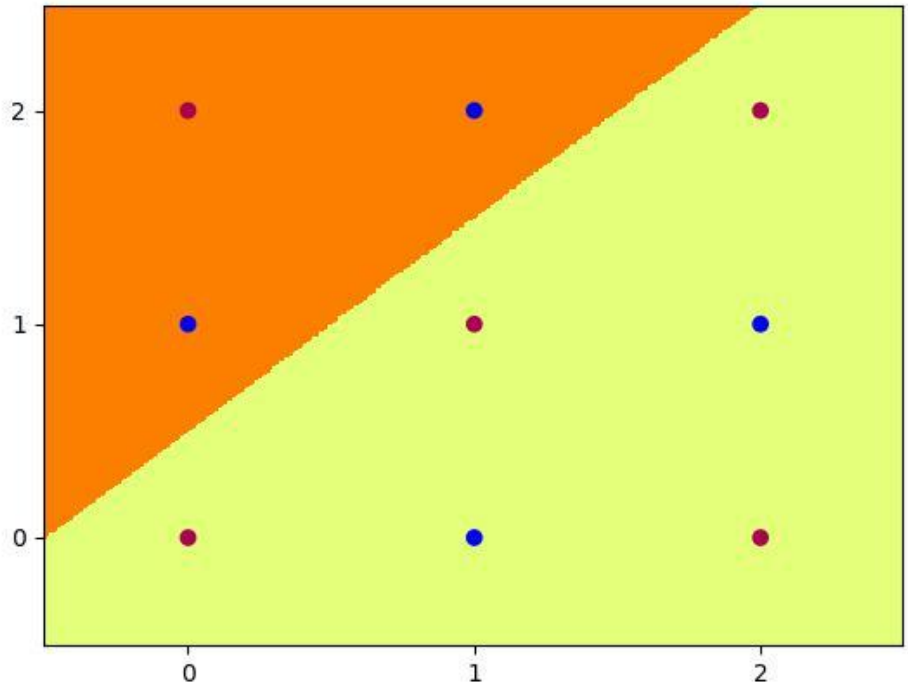
Once again, the code should produce images in the plot subdirectory showing the function computed by each hidden node (hid_4_?.jpg) and the network as a whole (out_4.jpg). Copy these images into your report, and be ready to submit check.py with the (rescaled) weights as part of your assignment submission.

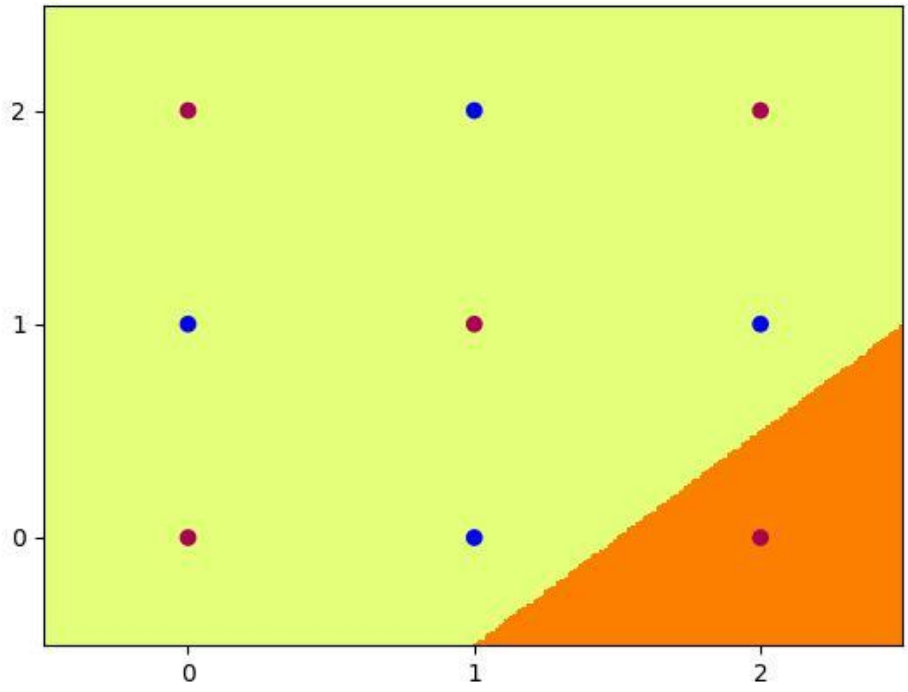
```
PS C:\Users\admin\Desktop\document\9444\assignment\a1\a1> python check_main.py --act sig --hid 4 --set_weights
Initial Weights:
tensor([[ 10., -10.],
        [-10.,  10.],
        [ 10., -10.],
        [-10.,  10.]])
tensor([ -5.,  -5., -15., -15.])
tensor([[ 10.,  10., -10., -10.]])
tensor([-5.])
Initial Accuracy:  100.0
```

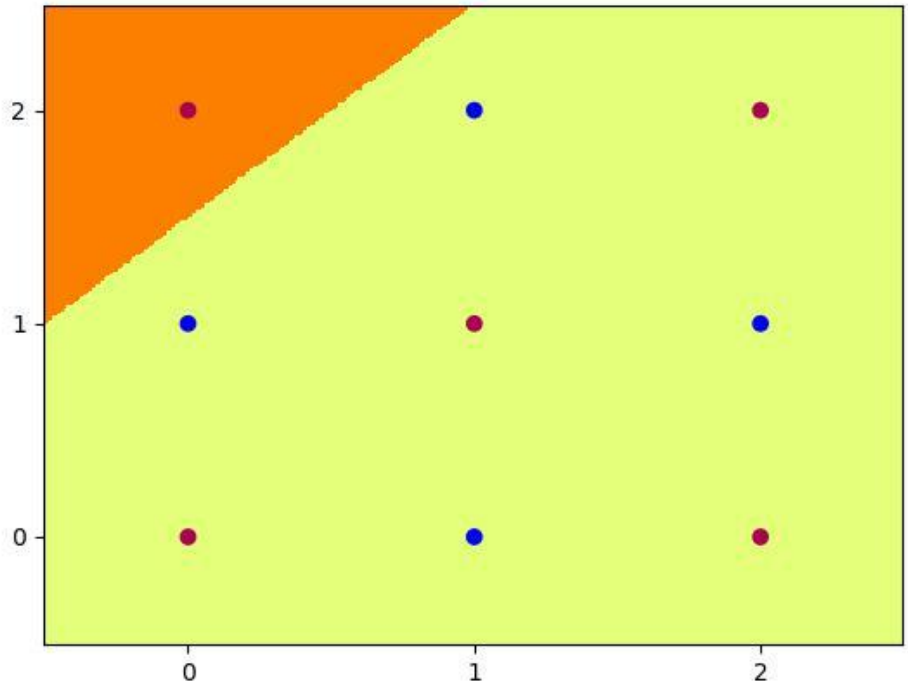


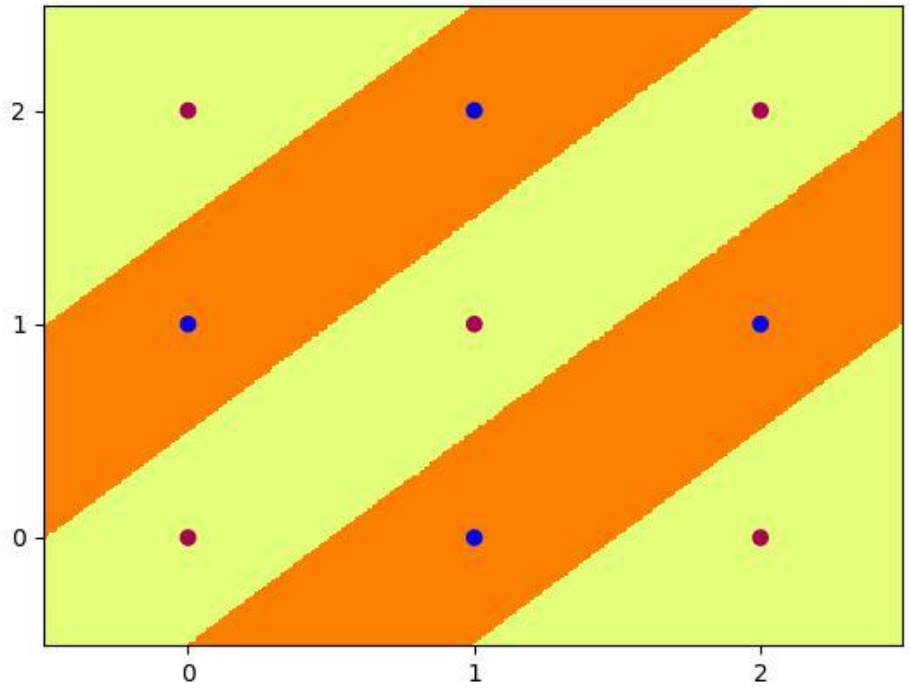




2
1
0
0
1
2


2
1
0
0
1
2


2
1
0
0
1
2


2
1
0
0
1
2




## Part 3: Hidden Unit Dynamics for Recurrent Networks

In Part 3 you will be training and analysing the hidden unit dynamics of recurrent networks trained on two language prediction tasks, anb2n and anb2nc3n, using the supplied code seq_train.py, seq_plot.py and anb2n.py.

[1 mark] Train a Simple Recurrent Network (SRN) with 2 hidden nodes on the anb2n language prediction task by typing:

python3 seq_train.py --lang anb2n

The anb2n language is a concatenation of a random number of A's followed by exactly twice that same number of B's. The generator produces concatenations of sequences anb2n for values of n between 1 and 4. This SRN has 2 inputs, 2 hidden nodes and 2 outputs.

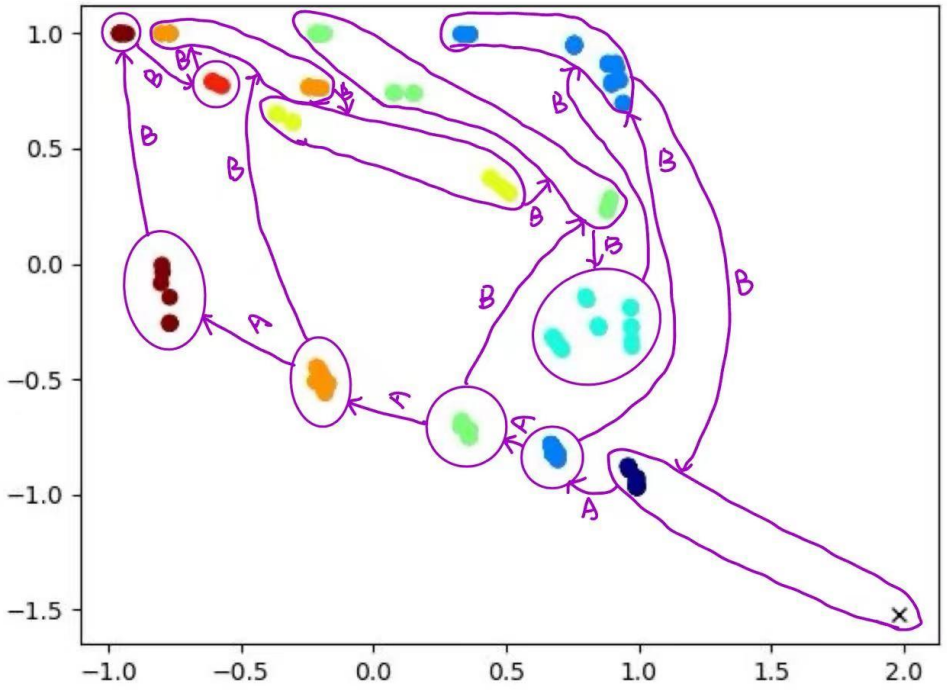
Look at the predicted probabilities of A and B as the training progresses. The first B in each sequence and all A's after the first A are non-deterministic, meaning that they can only be predicted in a probabilistic sense. But, if the training is successful, all other

symbols should be correctly predicted. In particular, the network should predict the last B in each sequence as well as the subsequent A (at the beginning of the next sequence). The loss should stay consistently below 0.05 (you might need to run the code a couple of times in order to achieve this). After the network has been trained successfully, plot the hidden unit activations at epoch 100000 by typing:

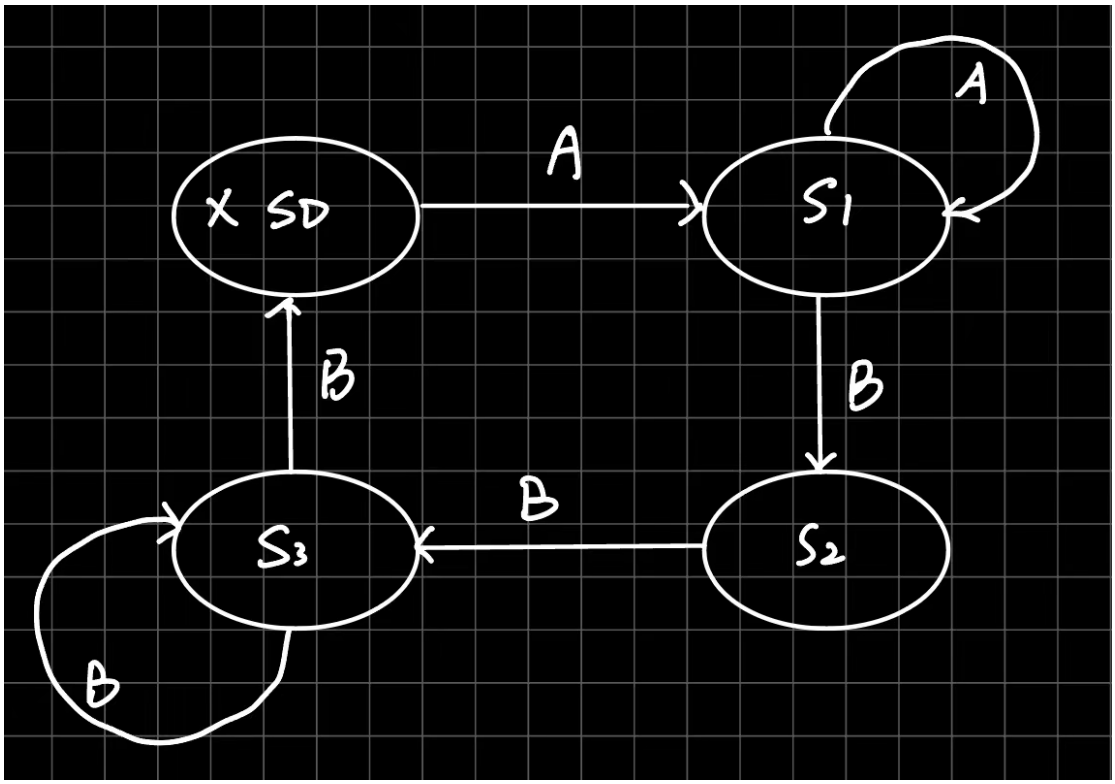
```
python3 seq_plot.py --lang anb2n --epoch 100
```

This should produce text output showing the hidden unit activations and associated probabilities, as each sequence is processed; it should also produce an image in the plot subdirectory called anb2n_srn2_01.jpg. After examining the hidden unit activations, annotate this image (either electronically, or with a pen on a printout) by grouping clusters of points together to form "states". Draw a boundary around each state, draw arrows between the states and label each arrow as 'A' or 'B'. The initial state is the one which includes the cross '×'. In order to assist with this task, we have tried to color the dots in a logical way using this colormap:

Include the annotated image in your report.




[1 mark] Draw a picture of a finite state machine, using circles and arrows, which is equivalent to the finite state machine in your annotated image from Step 1.




[1 mark] Briefly explain how the network accomplishes the anb2n task. Specifically, you should describe how the hidden unit activations change as the string is processed, and how it is able to correctly predict all B's after the first B, as well as the initial A following the last B in the sequence.

**The network learns to track the number of A's in the input sequence using its hidden unit activations. When A's are input, the hidden activations accumulate a count representation of how many A's have been seen. Upon encountering the first B, the hidden state transitions to a different region in activation space, indicating a shift from counting A's to counting B's.**

**For each B, the network decrements an internal representation of the expected number of B's which should be twice the number of A's. The activations follow a distinct trajectory through hidden space during this phase, moving closer to a boundary that signals the end of the B sequence.**

**The first B is not deterministic because the network can't be sure if the sequence of A's has ended yet, so its prediction is probabilistic. After the first B, the network's internal state confirms it's now in the B-processing phase, and subsequent B's become predictable through the hidden state changes in a structured, linearly progressing path, reflecting that it is counting down toward the expected 2n. When the last B is encountered, the network's hidden activations enter a known region associated with the start of a new sequence, enabling it to predict the next**

**A with high confidence.**

[1 mark] Train an LSTM with 3 hidden nodes on the anb2nc3n language prediction task by typing:

python3 seq_train.py --lang anb2nc3n --model lstm --hid 3

The anb2nc3n language is a concatenation of a random number of A's, followed by exactly twice that same number of B's, followed by three times that number of C's.

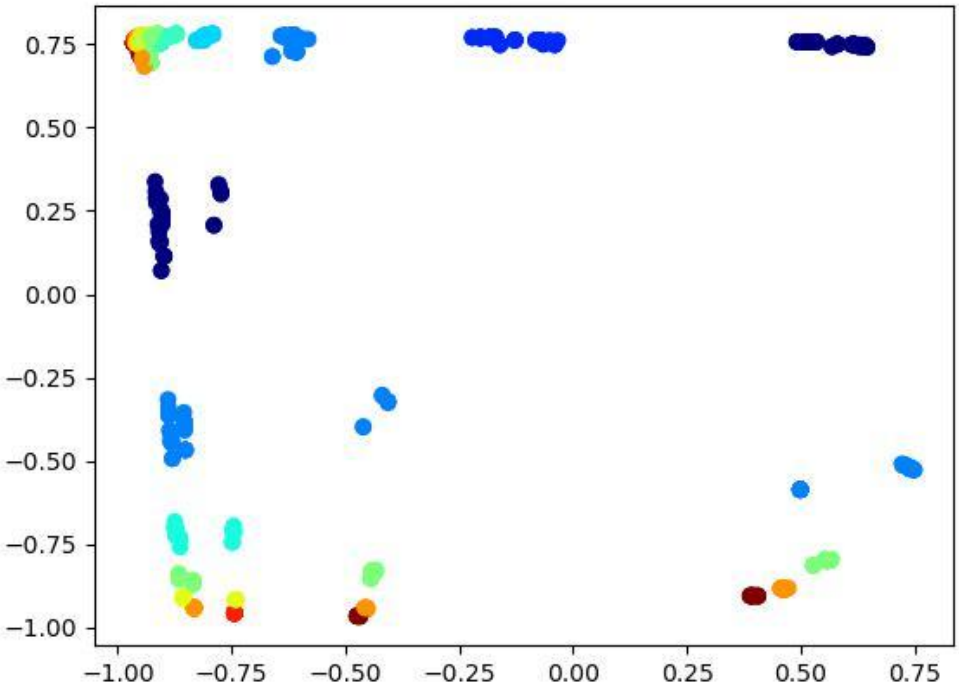
The loss should stay consistently below 0.02, and the network should correctly predict the non-initial B's, all the C's, and the subsequent A following the last C. You might need to run the code a couple of times in order to achieve this. If the network appears to have already learned the task successfully, or if it seems to be stuck in a local minimum, you can terminate the process with ⟨ctrl⟩-C (and start again, if necessary). After the network has been trained successfully, plot the hidden unit activations at epoch 50000 by typing:

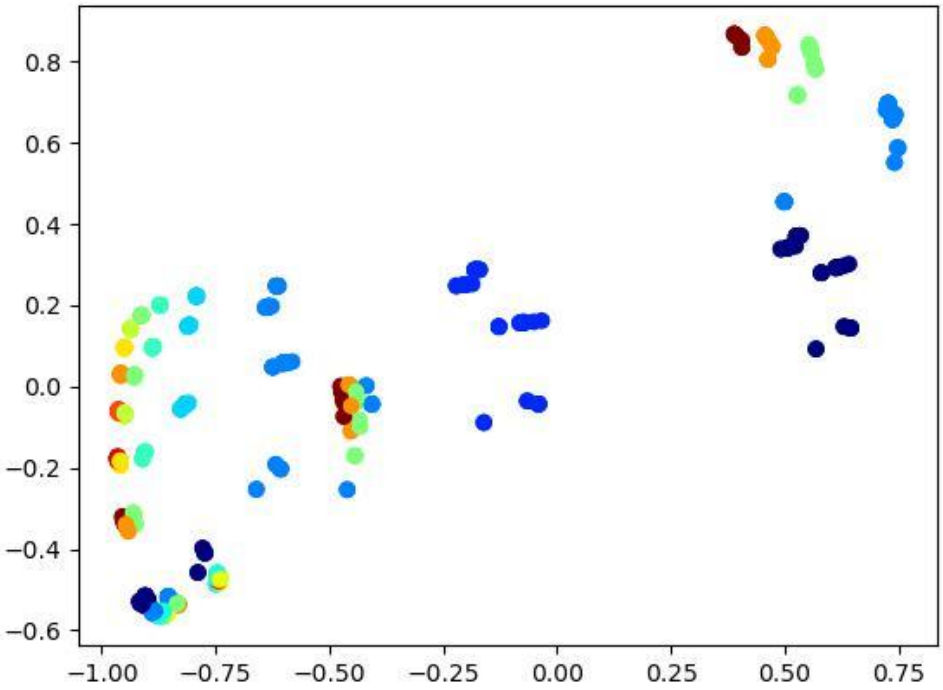
python3 seq_plot.py --lang anb2nc3n --model lstm --epoch 50

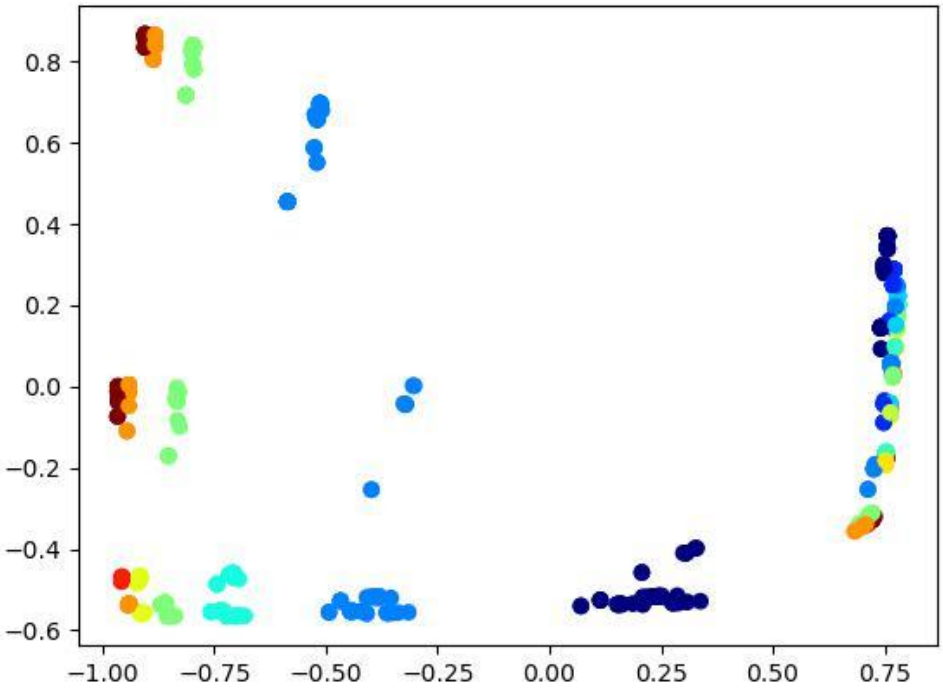
(you can choose a different epoch number, if you wish). This should produce three images from different angles, labeled anb2nc3n_lstm3_??.jpg, as well as an interactive 3D plot which you can rotate to a visually appealing angle and save as an image. Copy these images into your report.

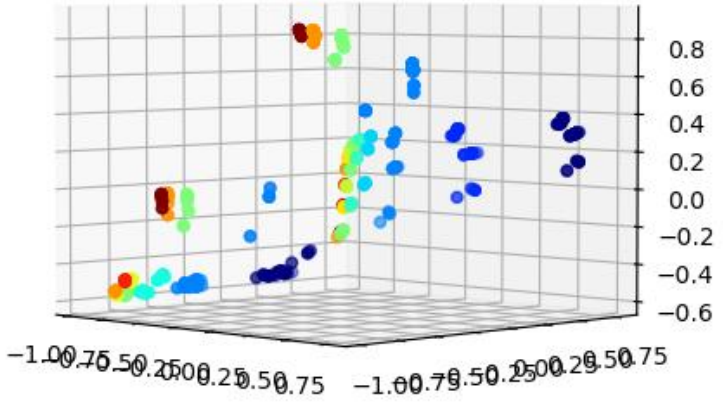
```
epoch: 100000
 loss: 0.0158
```

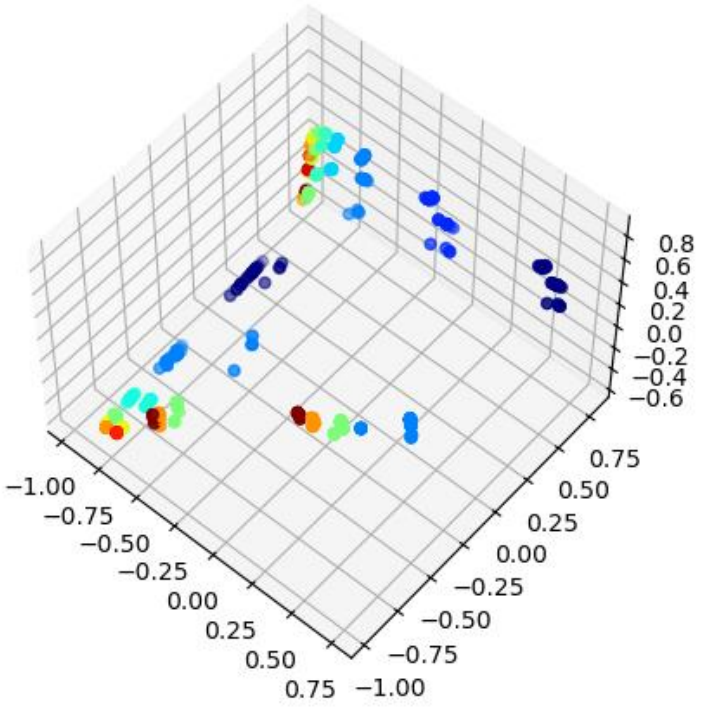
```
epoch: 9
 loss: 0.0082
```


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


0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

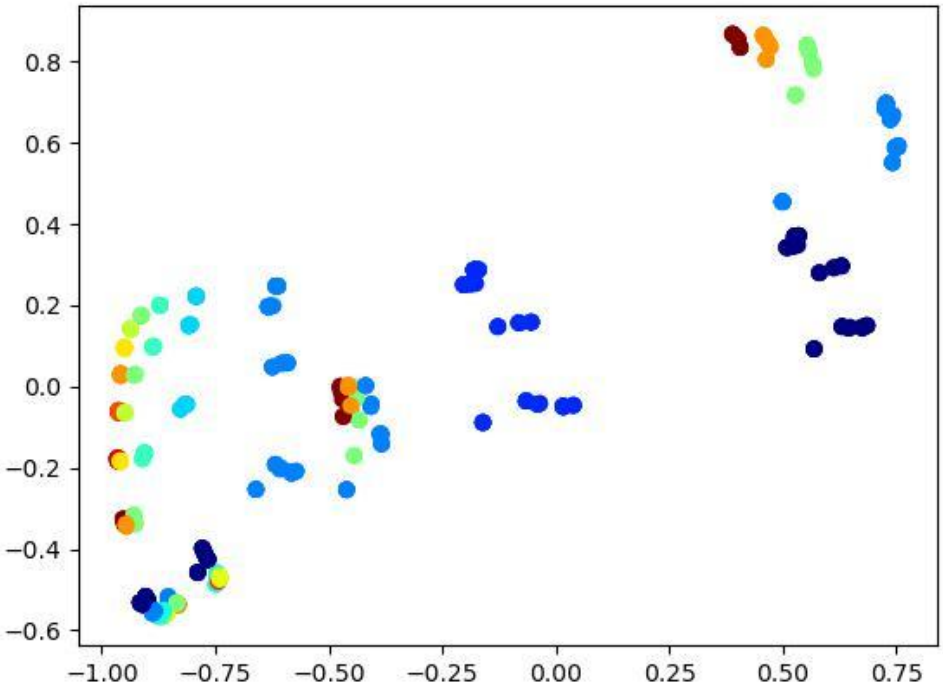

0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

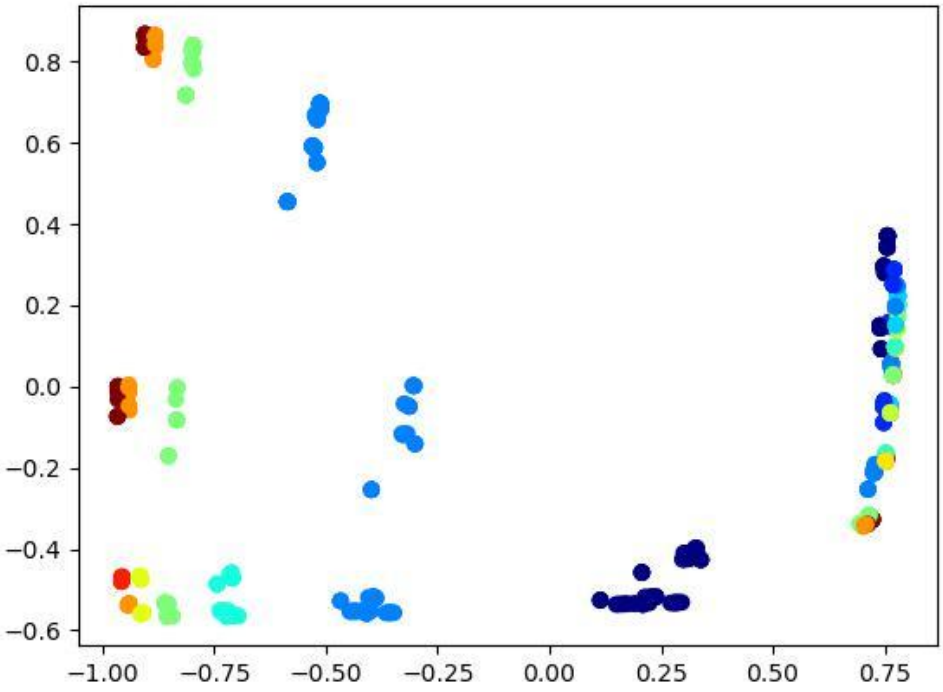

0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

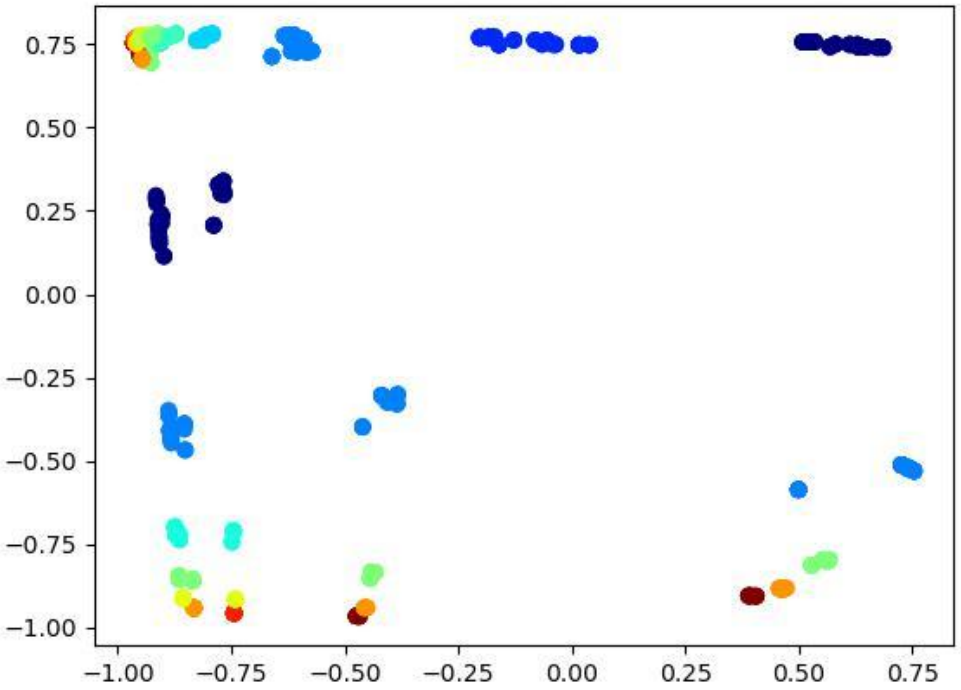



[4 marks] This question is intended to be a bit more challenging. By annotating the generated images from Step 4 (or others of your own choosing), try to analyse the dynamics of the hidden and/or context units, and explain how the LSTM successfully accomplishes the anb2nc3n prediction task (this might involve modifying the code so that it returns and prints out the context units as well as the hidden units).

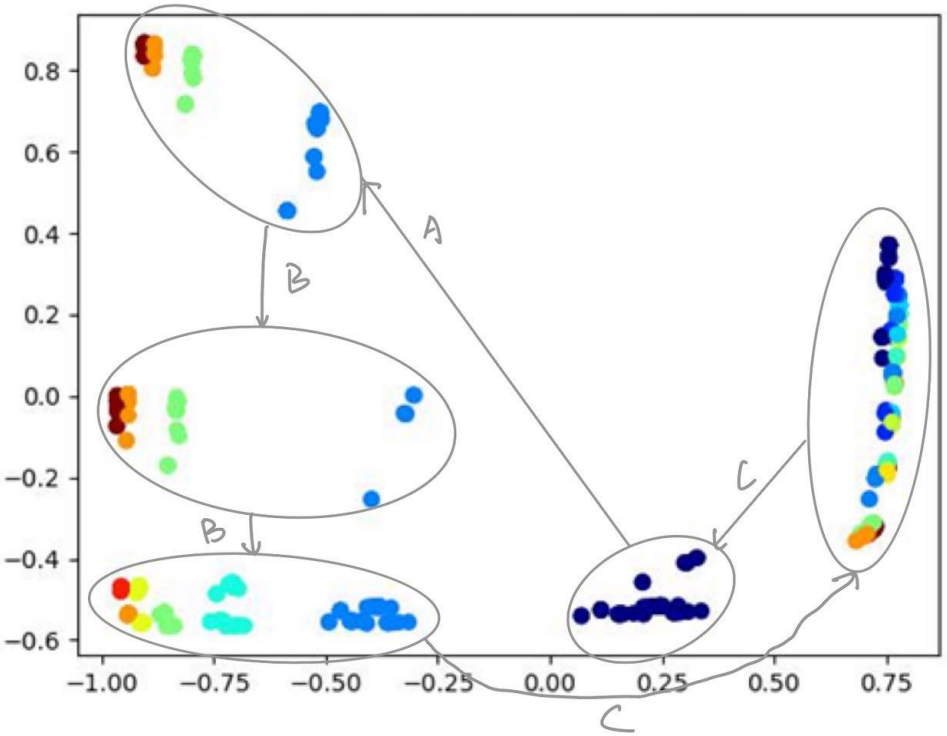
**Lstm:**


0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

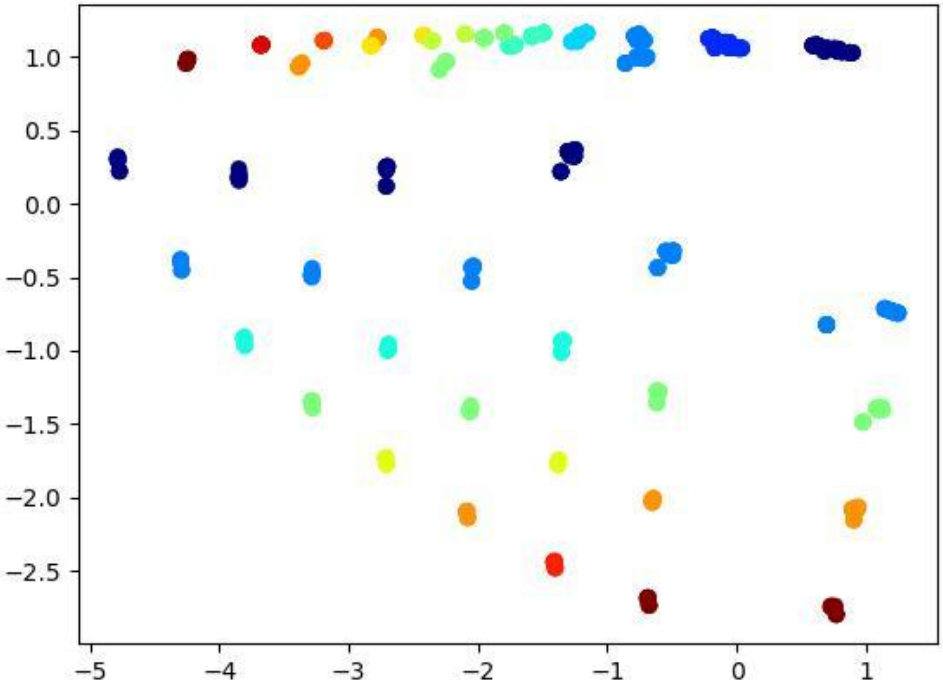

0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

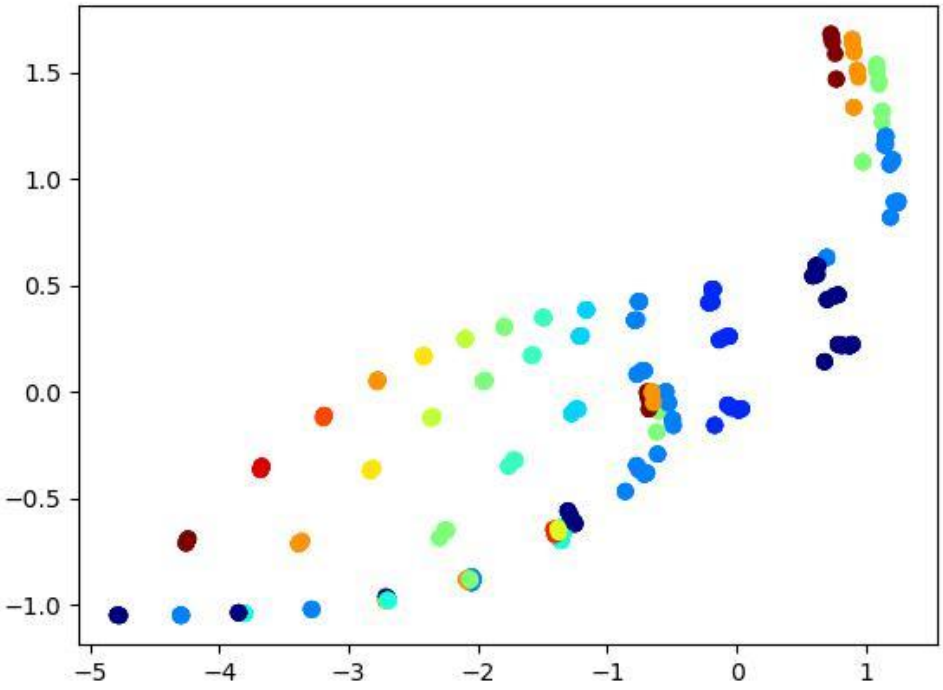

0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75

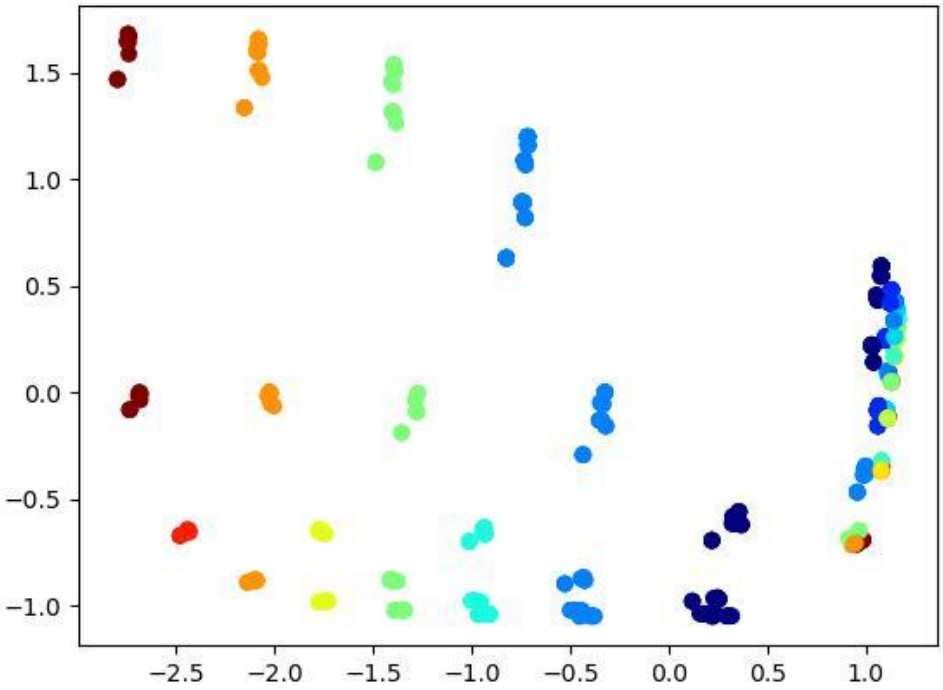
Annotating:


0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
A
B
B
C
C

the hidden and/or context units:






One context unit likely stores the original n value. During a phase, cell accumulates value; in b or c it stays steady or decays. The network uses context state to remember long-term structure.

The LSTM solves the anb2nc3n task by encoding the count of initial as in its context (cell) units, and using this persistent memory to correctly predict the subsequent structured pattern of bs and cs. Hidden units act as controllers for phase transitions, and the cell states store the critical counting information across long distances.