Part 1

The input images are grey, hence the number of channels in input images would be 1.

1. python kuzu_main.py --net lin

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.808456
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.629005
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.590719
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.598157
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.320940
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.522108
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.661597
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.608241
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.351361
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.674648
<class 'numpy.ndarray'>
[[767.   5.   7.  15.  30.  63.   2.  63.  31.  17.]
 [  7. 670. 107.  19.  28.  22.  57.  13.  25.  52.]
 [  7.  61. 692.  26.  29.  20.  45.  36.  45.  39.]
 [  2.  36.  62. 760.  16.  56.  14.  18.  25.  11.]
 [ 62.  51.  83.  20. 623.  20.  32.  35.  20.  54.]
 [  8.  28. 126.  17.  19. 724.  28.   8.  33.   9.]
 [  5.  23. 147.  10.  24.  23. 723.  20.  10.  15.]
 [ 16.  30.  26.  11.  83.  17.  56. 624.  89.  48.]
 [ 11.  37.  96.  43.   6.  29.  46.   6. 704.  22.]
 [  8.  51.  88.   4.  54.  31.  19.  30.  40. 675.]]

Test set: Average loss: 1.0099, Accuracy: 6962/10000 (70%)
```

   All other parameters remain as default(lr=0.01, mom=0.5, epochs=10). The final result is shown above.

   - total number of parameters in the network: 7850 = (1 + 28 * 28) * 10, which is the size of an input image 28 * 28 plus 1 bias and multiplied by the number of output classes (10 Hiragana charaters).

Does not count batch size (64)

2. python kuzu_main.py --net full

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.353146
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.247114
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.232035
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.227153
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.120101
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.254057
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.267277
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.367975
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.147859
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.286056
<class 'numpy.ndarray'>
[[854.   3.   2.   4.  28.  24.   3.  47.  31.   4.]
 [  6. 819.  33.   4.  16.   9.  53.   5.  22.  33.]
 [ 10.  12. 838.  36.   9.  20.  24.  12.  20.  19.]
 [  4.  10.  27. 912.   3.  16.   7.   6.   6.   9.]
 [ 42.  28.  19.   5. 800.   9.  34.  25.  21.  17.]
 [  9.  11.  76.  10.  10. 835.  22.   3.  17.   7.]
 [  3.  13.  53.   7.  15.   8. 879.  12.   2.   8.]
 [ 18.   9.  17.   4.  22.  14.  34. 820.  26.  36.]
 [ 10.  26.  27.  51.   3.   9.  34.   5. 829.   6.]
 [  7.  21.  58.   4.  26.   7.  19.  16.  11. 831.]]

Test set: Average loss: 0.5189, Accuracy: 8417/10000 (84%)
```

The number of hidden nodes is determined to be 120 since it helps the network to reach 84% accuracy consistently. All other parameters remain as default(lr=0.01, mom=0.5, epochs=10). The final result is shown above.

total number of parameters in the network 95410 = 94200 + 1210,

- hidden layer: 94200 = (1 + 28 * 28) * 120, which is the size of an input image 28 * 28 plus 1 bias and multiplied by the number of hidden nodes (120).
- output layer: 1210 = (1 + 120) * 10, which is the 120 hidden nodes from last layer plus 1 bias and multiplied by the number of output classes (10 Hiragana charaters).

3. python kuzu_main.py --net conv

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 2.103817
Train Epoch: 10 [6400/60000 (11%)]      Loss: 2.027671
Train Epoch: 10 [12800/60000 (21%)]     Loss: 2.050928
Train Epoch: 10 [19200/60000 (32%)]     Loss: 2.154761
Train Epoch: 10 [25600/60000 (43%)]     Loss: 1.997241
Train Epoch: 10 [32000/60000 (53%)]     Loss: 2.013685
Train Epoch: 10 [38400/60000 (64%)]     Loss: 2.039845
Train Epoch: 10 [44800/60000 (75%)]     Loss: 2.228362
Train Epoch: 10 [51200/60000 (85%)]     Loss: 1.968650
Train Epoch: 10 [57600/60000 (96%)]     Loss: 2.113540
<class 'numpy.ndarray'>
[[945.   5.   1.   1.  19.   3.   3.  16.   4.   3.]
 [  2. 922.   8.   0.  13.   3.  36.   4.   3.   9.]
 [ 10.   6. 863.  53.   4.   6.  22.  15.   8.  13.]
 [  1.   1.  11. 973.   0.   3.   5.   0.   4.   2.]
 [ 26.  10.   6.   6. 894.  11.  19.  10.  12.   6.]
 [  5.  13.  42.  13.   4. 900.  10.   2.   4.   7.]
 [  2.   9.  20.   3.   5.   4. 953.   2.   1.   1.]
 [  4.   4.   9.   2.   9.   2.  18. 933.   6.  13.]
 [  5.  30.   6.   5.   5.  10.   4.   1. 932.   2.]
 [  5.   7.  12.   4.  14.   1.   5.   2.   2. 948.]]

Test set: Average loss: 2.2439, Accuracy: 9263/10000 (93%)
```

Besides the two convolutional layers, one fully connected layer and one output layer, I have added two additional max pooling layers after the two convolution layers to help the network to 93% accuracy consistently. All other parameters remain as default(lr=0.01, mom=0.5, epochs=10). The final result is shown above.

total number of parameters in the network 216614 = 364 + 14040 + 200200 + 2010,

- First convolution layer: 364 = (1 + 5 * 5) * 14, because of the weight sharing, the kernel size (5 * 5) plus 1 bias would be the number of independent parameters in one convolutional, and it is multiplied by the number of its filters (14).
- Second convolution layer: 14040 = (1 + 5 * 5 * 14) * 40, the kernel size (5 * 5) multiplied by the number of filters in last layer plus 1 bias, and multiplied by the number of filters in current layer (40).
- Fully connected layer: 200200 = (1 + 5 * 5 * 40) * 200, the size of last layer (5 * 5) multiplied by the number of filters of last layer (40) plus 1 bias (also the number of neurons in last layer), and multiplied by the number of hidden nodes in this layer (200).
- output layer: 2010 = (1 + 200) * 10, which is the 200 hidden nodes from last layer plus 1 bias and multiplied by the number of output classes (10 Hiragana charaters).
- Max pooling layer: no learnable independent parameters to backpropagate since it only get the biggest number from a 2 * 2 matrices.

4. Discussion
    a. The accuracy goes from low to high respect to model Netlin, fully connected 2-layer network Netfull and convolutional network NetConv
    b. The number of independent parameters also goes from low to high (7850, 95410 and 216614), it shows that more complicated models with more independent parameters indicate more learnable things, hence have a higher accuracy result. However, the number of independent parameters is not proportional to the accuracy, and need to be careful that more layers and independent parameters might result in problem such as overfitting, vanishing and exploding gradients.
    c. Most of the accuracy of recognitions for each character from last model are improved as the model includes more independent parameters and becomes more complicated. Netlin only contains a linear function to compute the output. Fully connected 2-layer network has one hidden layer with 120 filters more than the Netlin model, which makes it able to classify features that are not linearly seperable. Convolutional network uses 2 convolutional layers, the first convolutional layer gives a brief sketch, while the second convolutional layer focus on the different features of the input image. Convolutional network is able to recognize the character with its feature instead of simply black, white location data. However, some characters are still not able to be correctly classified, or even becomes worse as the network becomes more complicated.
        - Fully connected 2-layer network have the same mistaken as Netlin in characters such as,
            - 0 = “o” mistaken as 8 = “re”
            - 1 = “ki” mistaken as 6 = “ma”
            - 1 = “ki” mistaken as 8 = “re”
            - 4 = “na” mistaken as 6 = “ma”
            - 4 = “na” mistaken as 8 = “re”
            - 5 = “ha” mistaken as 0 = “o”
            - 9 = “wo” mistaken as 6 = “ma”
            - 7 = “ya” mistaken as 0 = “o”

          Netlin sense the location of white and blacke, and produces a model with different weights for different x and y. Fully connected 2-layer network can achieve nonlinear classification. However, if the problem was that the characters is not in the same location in the image or not the same size. Adding one more layer cannot solve the problem. These mistaken “o”, “ki” and “na” might looks differently to others, and “re”, “ma” and “o” have a more general characteristic compares to other characters, which is why those mistaken characters are recognized as them.

        - Fully connected 2-layer network have mistaken even more than Netlin in characters such as,
            - 8 = “re” mistaken as 3 = “tsu”

“tsu” is like a fall down U, while “re”contains a fall down U and a slash. “re” is more complicated. If the characters is not in the same size and location in the image, the network might get confused. Also, from the previous discussioin it is showed that “re” is more general in characteristics compares to other characters. “tsu” somehow fits the mistaken input “re” better.

- Convolutional network have similar mistaken as Fully connected 2-layer network in characters such as,
  - 8 = “re” mistaken as 1 = ”ki”

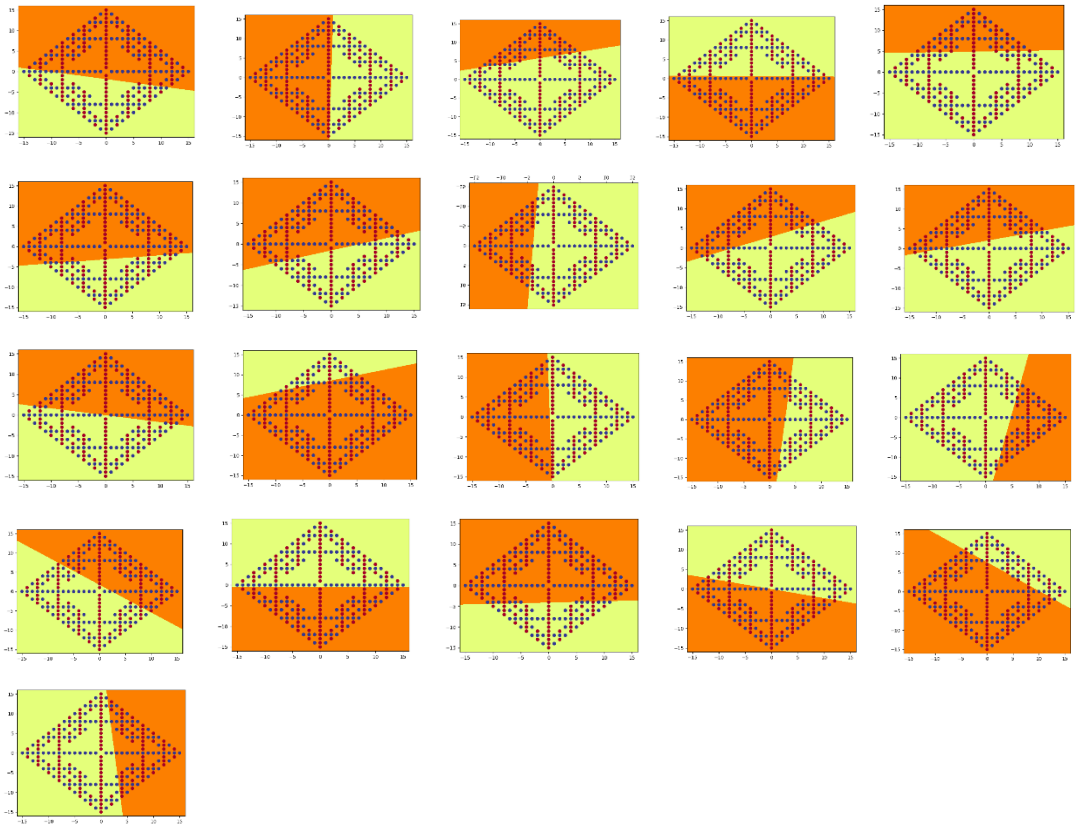
  “ki” is a ++ sign on top with U opening facing right at the bottom, while “re” is a ++ sign on left with U opening facing right at the right. Possible reason could be convolutional neural network recognized these features while care less about their relative location.

Part 2

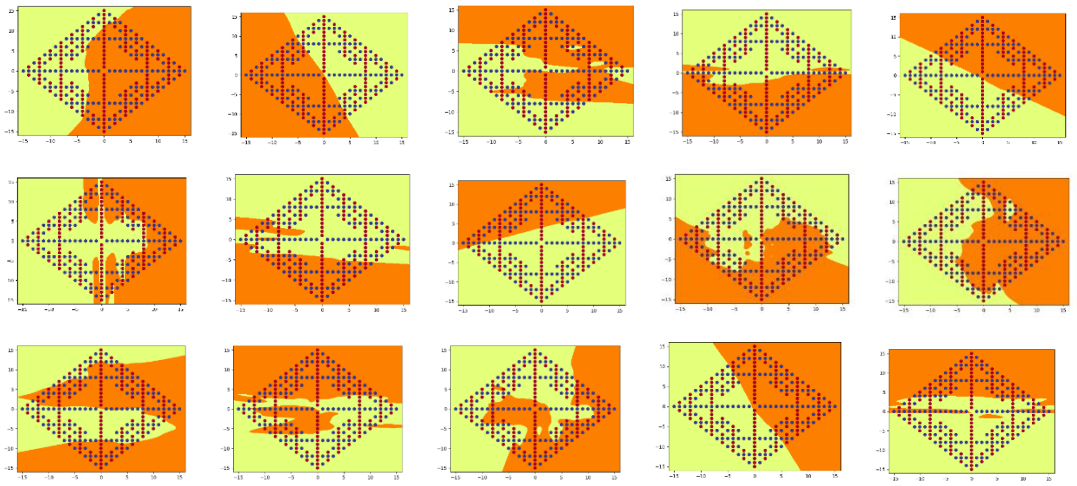
The input data is x and y, which is 2.

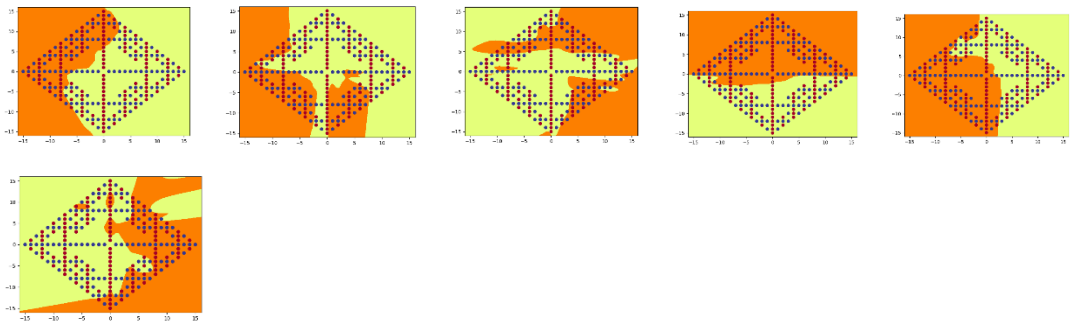
1. python frac_main.py --net full2 --hid 21

   First hidden layer

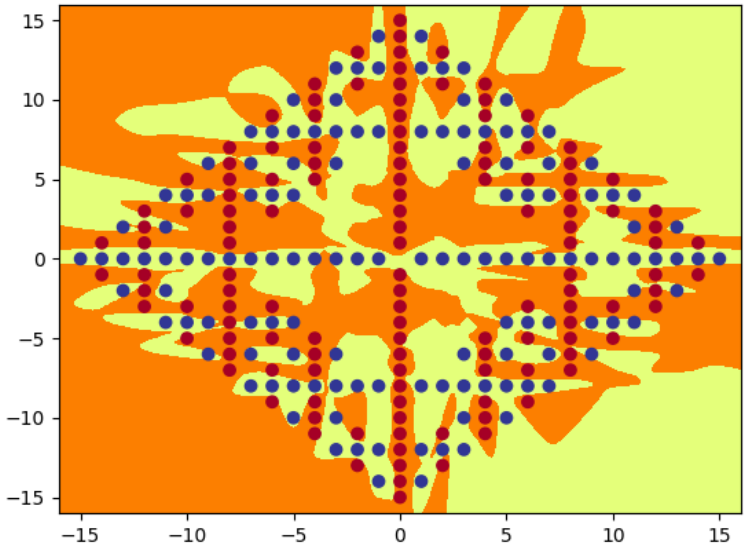


   Second hidden layer





Output



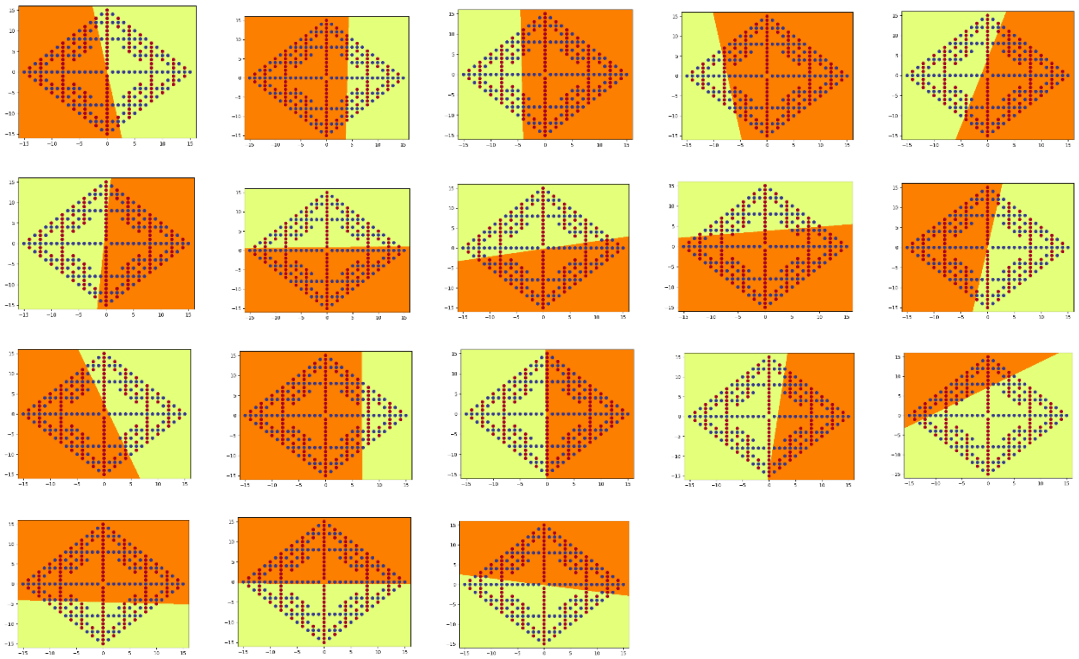
All other parameters remain as default(init=0.15, lr=0.01, epochs=200,000). Althought the smallest number of hidden nodes that can train the net successfully is 18, it does not converge most of the time. So the number of hidden nodes is determined to be 21 since it consistently train successfully. The final result is shown above.
total number of parameters in the network 547 = 63 + 462 + 22,

- first hidden layer: 63 = (1 + 2) * 21, which is 2 (x and y) of the input plus 1 bias and multiplied by the number of hidden nodes (21).
- Second hidden layer: 462 = (1 + 21) * 21, which is the number of hidden nodes from last layer (21) plus 1 bias, and multiplied by the number of hidden nodes in this layer (21).
- output layer: 22 = (1 + 21) * 1, which is the number of hidden nodes from the last layer (21) plus 1 bias, and multiplied by the number of output classification (1 (red or blue)).

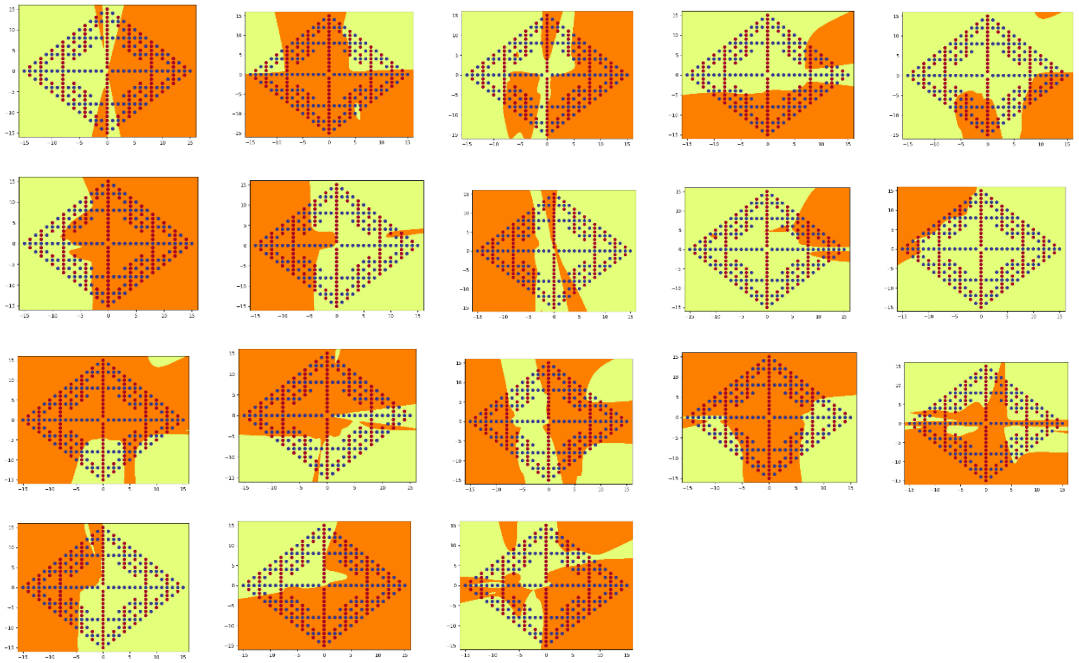
=

2. python frac_main.py --net full3 --hid 18

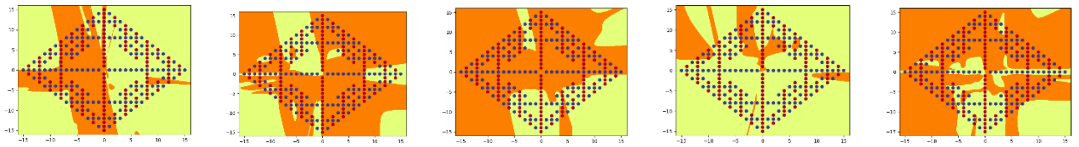
First hidden layer

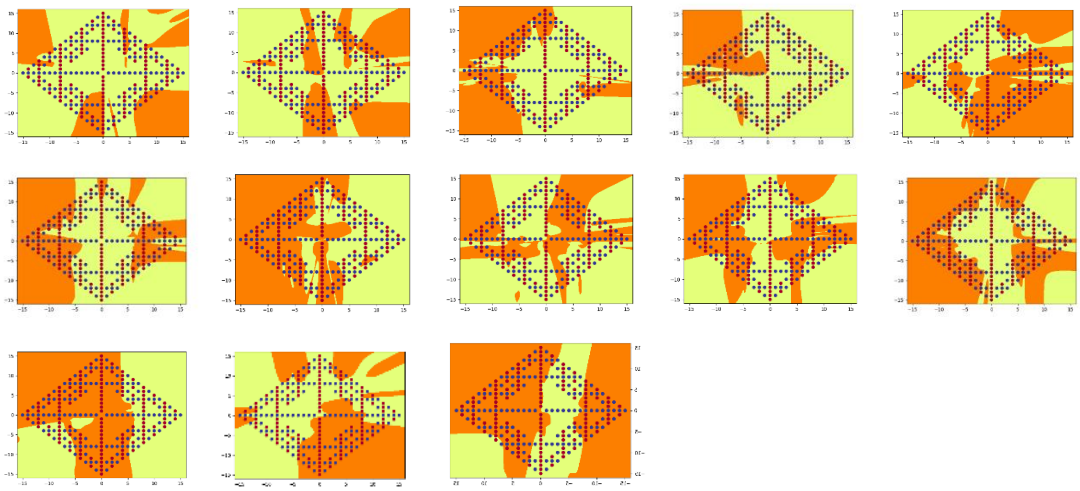


Second hidden layer

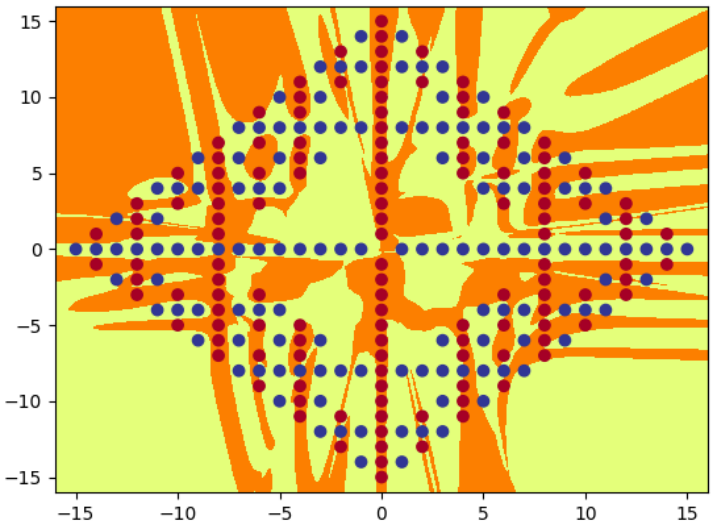


Third hidden layer





Output

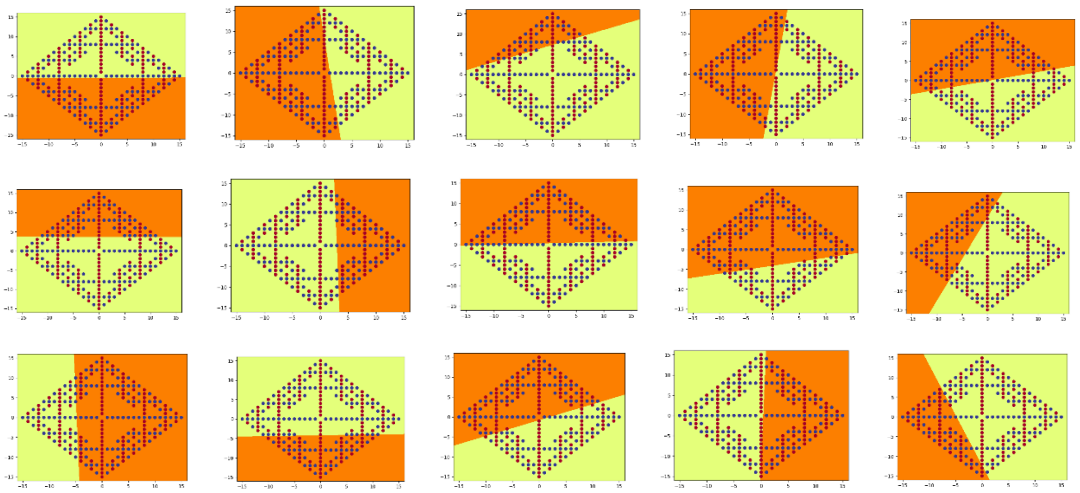


All other parameters remain as default(init=0.15, lr=0.01, epochs=200,000). Althought the smallest number of hidden nodes that can train the net successfully is 14, it does not converge most of the time. So the number of hidden nodes is determined to be 18 since it consistently train successfully within 200,000 epochs. The final result is shown above.
total number of parameters in the network 757 = 54 + 342 + 342 + 19,

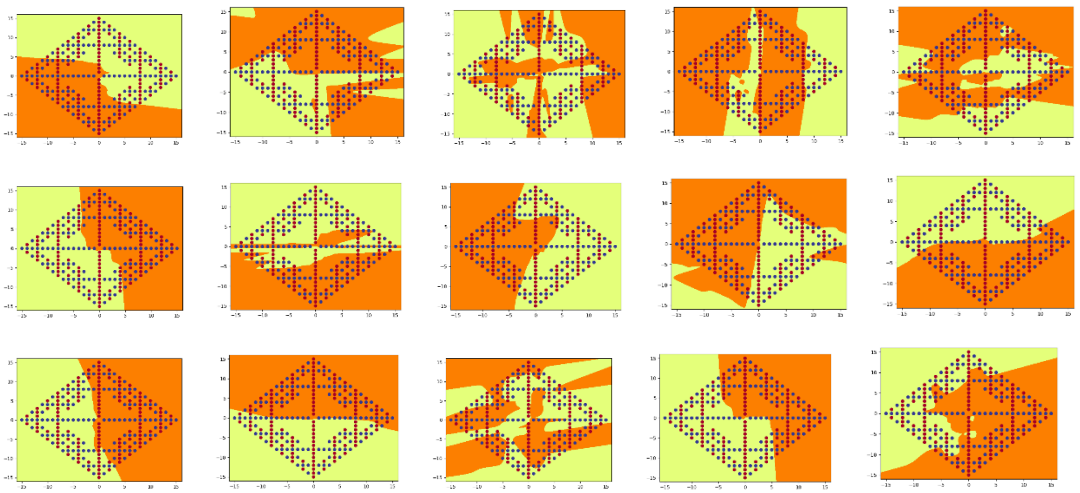
- first hidden layer: 54 = (1 + 2) * 18, which is 2 (x and y) of the input plus 1 bias and multiplied by the number of hidden nodes (20).
- Second hidden layer: 342 = (1 + 18) * 18, which is the number of hidden nodes from last layer (20) plus 1 bias, and multiplied by the number of hidden nodes in this layer (20).
- Third hidden layer: 342 = (1 + 18) * 18, which is the number of hidden nodes from last layer (20) plus 1 bias, and multiplied by the number of hidden nodes in this layer (20).
- output layer: 19 = (1 + 18) * 1, which is the number of hidden nodes from the last layer (20) plus 1 bias, and multiplied by the number of output classification (1 (red or blue)).

3. python frac_main.py --net dense --hid 15

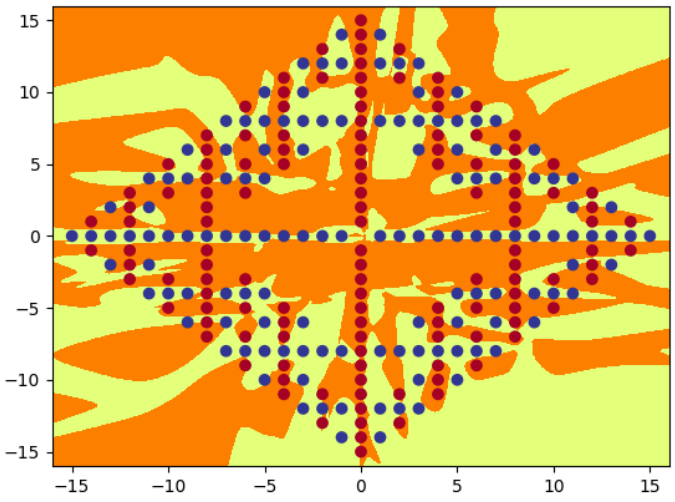
First hidden layer



Second hidden layer



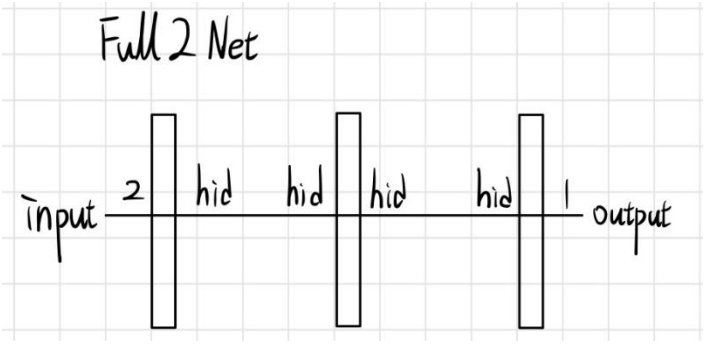
Output



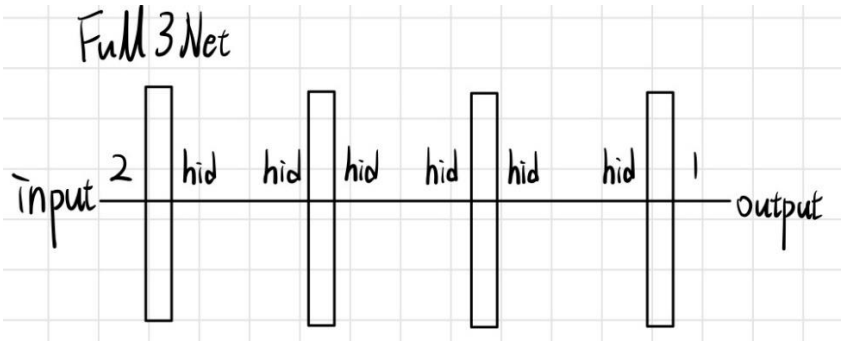
All other parameters remain as default(init=0.15, lr=0.01, epochs=200,000). Althought the smallest number of hidden nodes that can train the net successfully is 12, it does not converge most of the time. So the number of hidden nodes is determined to be 15 since it consistently train successfully within 200,000 epochs. The final result is shown above.
total number of parameters in the network 348 = 45 + 270 + 33,

- first hidden layer: 45 = (1 + 2) * 15, which is 2 (x and y) of the input plus 1 bias and multiplied by the number of hidden nodes (15).
- Second hidden layer: 270 = (1 + 15 + 2) * 15, which is the number of hidden nodes from last layer (15) plus 1 bias, plus the concatenation of 2 of the input, and multiplied by the number of hidden nodes in this layer (15).
- output layer: 33 = (1 + 15 + 15 + 2) * 1, which is the number of hidden nodes from the last layer (15) and plus 1 bias, plus the concatenation of 15 of the activations of first hidden layer and 2 from the input, and multiplied by the number of output classification (1 (red or blue)).

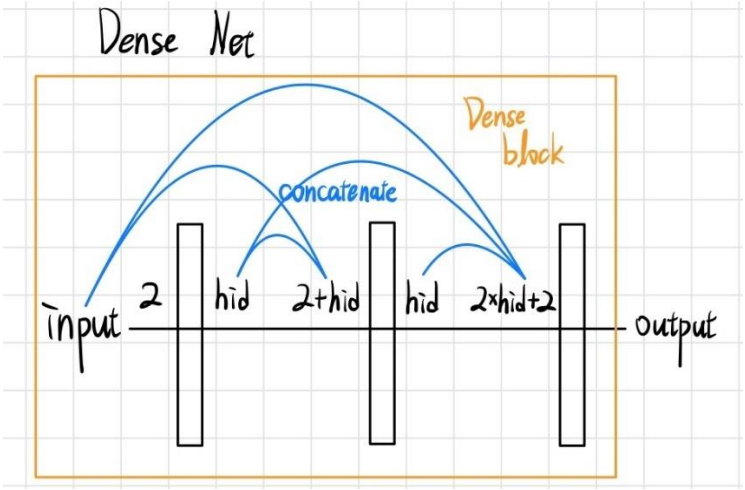
4. Discussion
    a. Full2Net: with 547 independent parameters, around 120000 epochs it trained successfully
    Full3Net: with 757 independent parameters, around 90000 epochs it trained successfully
    DenseNet: with 348 independent parameters,around 70000 epochs it trained successfully
    From the result, Full3Net train with less number of epochs than Full2Net. Althought Full3Net has less hidden units in each layer, the total number of independent parameters is still larger than Full2Net, which requires a large amount of calculation resources. Incomparison, the dense net structure can train the model with the least number of epochs with the least number of independent parameters, which indicates that it can do the same task better with less calculation resources required. In result, the dense net structure can help the learning task perform better.
    b. The Full2Net module is shown below with 2 hidden layers applying tanh activation and a output layer applying sigmoid activation.



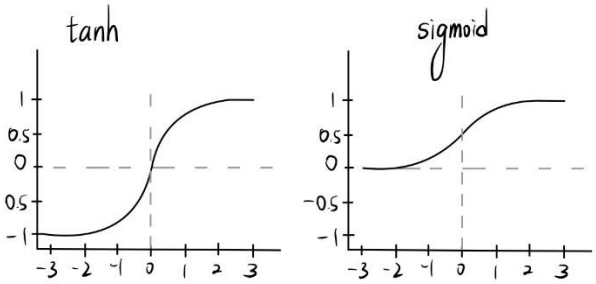
The Full3Net module is shown below with 3 hidden layers applying tanh activation and a output layer applying sigmoid activation.



The Dense Net module is shown below with 2 hidden layers applying tanh activation and a output layer applying sigmoid activation. And the input of the second hidden layer and input of the output layer is differed from the Full2Net module since they concatenates the activations got from the previous layers.



All the hidden layers use tanh function as loss function to backpropagate to modify the weight, and use sigmoid function to differentiate the blue and red dots. The difference between tanh and sigmoid function is that sigmoid range from y(0, 1), while tanh range from y(-1, 1), which makes it able to calculate the loss function with negative value. The number of filters in layer are all the same except DenseNet since it needs to concatentate the results from input and previous layer.



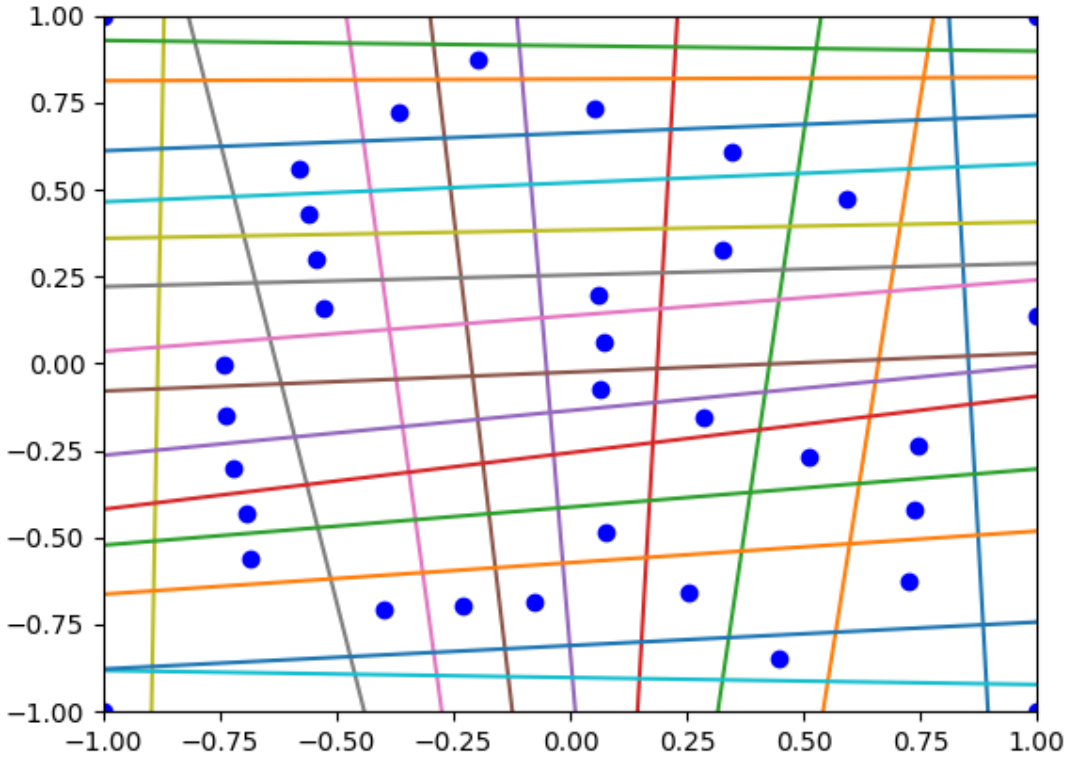
c. From the hidden layer and output figures,
   - In Full2Net, the first hidden layer only give a linear classification to the input data, and the second hidden layer enables the model to fit the input data better.

- In Full3Net, in addition to the first and second hidden layer, it provides a third hidden layer, which should gives more detail information about input data. The output figure is more fit to the input data compares to the output figure produced by the Full2Net
- In Dense net, the hidden layers figures is more complex than second layer of Full2Net while simpler than third hidden layer of Full3Net. The output figure is also more generalised compares to the output figure of Full3Net.

All three network use tanh and sigmoid functions to calculate the hidden layer activations and output layer classification. The differences are Full3Net has one layer more than Full2Net, which makes it able to fit better and quicker to the input data. Dense net has the same number of layer as Full2Net, while it concatenate the input and activations from previous layers. The preceding layer do the “whole” job and later layer provides additional details and corrects the errors from previous layer, making Dense net able to classify the red and blue dots correctly with a more general model.

From the test results, it is showed with less numberx of hidden nodes in each layer and with less independent parameters in total, dense network train better than simply adding one more layer to the network.

Part 3

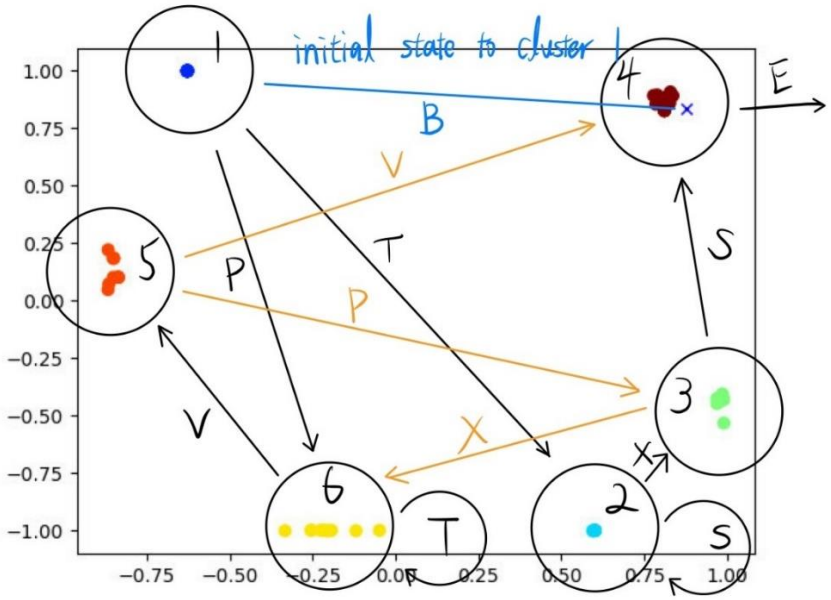

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

Part 4

1. python seq_plot.py --lang reber --epoch 50

```
state = 0122366666654
symbol= BTSXXTTTTVVE
label = 0123311111556
true probabilities:
     B    T    S    X    P    V    E
1 [0.  0.5 0.  0.  0.5 0.  0. ]
2 [0.  0.  0.5 0.5 0.  0.  0. ]
2 [0.  0.  0.5 0.5 0.  0.  0. ]
3 [0.  0.  0.5 0.5 0.  0.  0. ]
6 [0.  0.5 0.  0.  0.  0.5 0. ]
6 [0.  0.5 0.  0.  0.  0.5 0. ]
6 [0.  0.5 0.  0.  0.  0.5 0. ]
6 [0.  0.5 0.  0.  0.  0.5 0. ]
6 [0.  0.5 0.  0.  0.  0.5 0. ]
6 [0.  0.5 0.  0.  0.  0.5 0. ]
5 [0.  0.  0.  0.  0.5 0.5 0. ]
4 [0. 0. 0. 0. 0. 0. 1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-0.63  1.  ] [0.   0.35 0.   0.   0.48 0.16 0.  ]
2 [ 0.6 -1. ] [0.   0.   0.45 0.54 0.   0.   0.01]
2 [ 0.6 -1. ] [0.   0.   0.44 0.54 0.   0.   0.01]
3 [ 0.97 -0.45] [0.   0.   0.61 0.36 0.   0.03 0.  ]
6 [0.84 0.89] [0.   0.44 0.01 0.   0.01 0.53 0.  ]
6 [0.79 0.89] [0.   0.45 0.01 0.   0.02 0.52 0.  ]
6 [0.79 0.86] [0.   0.43 0.01 0.   0.02 0.54 0.  ]
6 [0.81 0.84] [0.   0.41 0.01 0.   0.02 0.56 0.  ]
6 [0.82 0.86] [0.   0.43 0.01 0.   0.02 0.55 0.  ]
6 [0.81 0.87] [0.   0.43 0.01 0.   0.02 0.54 0.  ]
5 [-0.86  0.14] [0.   0.11 0.   0.01 0.47 0.4  0.02]
4 [-0.26 -1.  ] [0.   0.   0.   0.01 0.   0.   0.99]
epoch: 9
error: 0.0018
```

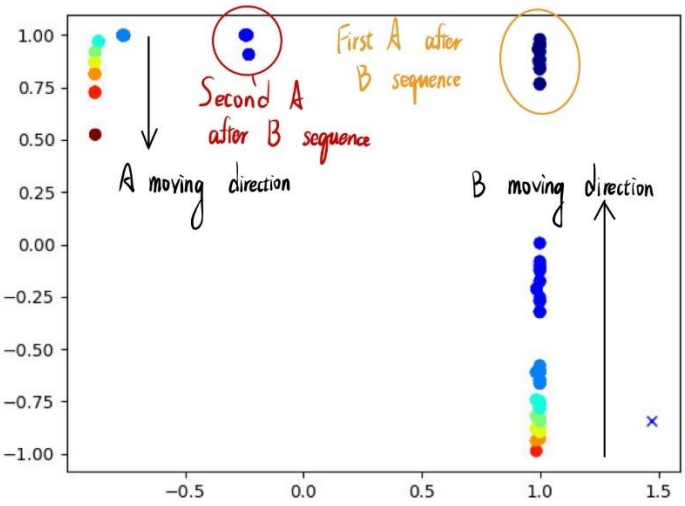
The number of row indicates the states the hidden activations and output probabilities it belongs to. The first 2 number shows the x and y the state is in the hidden unit space, while the 7 numbers afterward is the probabilities this number would go to route BTSXPVE repectively. Within the information, figure is annotated as below,



2. python seq_plot.py --lang anbn --epoch 100

```
color = 0123456787654321012321012345654321012345676543210
symbol= AAAAAAAABBBBBBBBAAABBBAAAAAABBBBBBAAAAAAABBBBBBBABA
label = 0000000011111111000111000000111111000000011111110
hidden activations and output probabilities:
A [-0.23  0.91] [0.85 0.15]
A [-0.76  1.  ] [0.87 0.13]
A [-0.87  0.97] [0.82 0.18]
A [-0.88  0.92] [0.75 0.25]
A [-0.88  0.87] [0.67 0.33]
A [-0.88  0.82] [0.56 0.44]
A [-0.88  0.73] [0.39 0.61]
B [-0.88  0.53] [0.11 0.89]
B [ 0.98 -0.99] [0. 1.]
B [ 1.   -0.93] [0. 1.]
B [ 1.   -0.88] [0. 1.]
B [ 1.   -0.83] [0. 1.]
B [ 1.   -0.75] [0. 1.]
B [ 1.   -0.58] [0. 1.]
B [1.   0.01] [0.02 0.98]
A [1.   0.98] [0.98 0.02]
A [-0.25  1.  ] [0.92 0.08]
A [-0.76  1.  ] [0.87 0.13]
B [-0.87  0.97] [0.82 0.18]
B [ 0.98 -0.61] [0. 1.]
B [ 1.   -0.18] [0. 1.]
A [1.   0.92] [0.97 0.03]
A [-0.24  1.  ] [0.92 0.08]
A [-0.76  1.  ] [0.87 0.13]
A [-0.87  0.97] [0.82 0.18]
A [-0.88  0.92] [0.75 0.25]
A [-0.88  0.87] [0.67 0.33]
B [-0.88  0.81] [0.56 0.44]
B [ 0.98 -0.88] [0. 1.]
B [ 1.   -0.84] [0. 1.]
B [ 1.   -0.78] [0. 1.]
B [ 1.   -0.65] [0. 1.]
B [ 1.   -0.27] [0. 1.]
A [1.   0.84] [0.94 0.06]
A [-0.24  1.  ] [0.92 0.08]
A [-0.76  1.  ] [0.87 0.13]
A [-0.87  0.97] [0.82 0.18]
A [-0.88  0.92] [0.75 0.25]
A [-0.88  0.87] [0.67 0.33]
A [-0.88  0.81] [0.56 0.44]
B [-0.88  0.73] [0.39 0.61]
B [ 0.98 -0.94] [0. 1.]
B [ 1.  -0.9] [0. 1.]
B [ 1.   -0.85] [0. 1.]
B [ 1.   -0.78] [0. 1.]
B [ 1.   -0.66] [0. 1.]
B [ 1.   -0.32] [0. 1.]
A [1.   0.77] [0.89 0.11]
B [-0.24  1.  ] [0.92 0.08]
A [0.99 0.94] [0.97 0.03]
epoch: 9
error: 0.0228
```

x

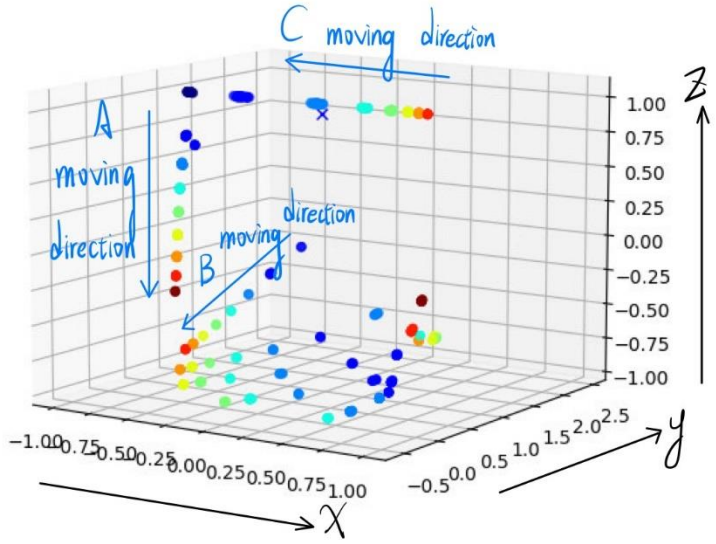


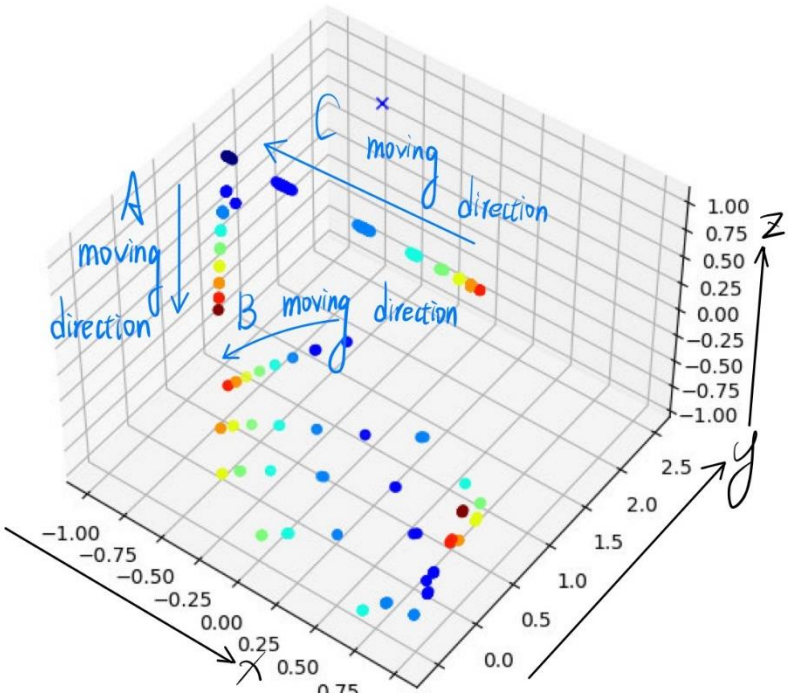
With the output hidden activations, we know that besides the first and second A after B sequence, A are the dots on the top left, and B are the dots at the bottom right. The hidden activation goes downward as the next A is received (monotonic). When the first B arrived, the hidden activation goes to the location it is corresponded in B region, and moving upward as the next B is received (monotonic). In this way, the module knows the next A is coming if

the hidden activation is in the specific zone (in the example, the zone is around (x, y) = (1, 0.65)).

3. python seq_plot.py --lang anbncn --epoch 200

```
color = 012121012343214321011012345678765432187654321012121O
symbol= AABBCCAAAABBBBCCCCABCAAAAAAAABBBBBBBBCCCCCCCCAABBCCA
label = 001122000011111222201200000000111111112222222220011220
hidden activations and output probabilities:
A [-0.87  0.98  0.62] [0.87 0.13 0.  ]
B [-0.95  0.98  0.47] [0.88 0.12 0.  ]
B [-0.29  0.84 -0.23] [0.01 0.98 0.01]
C [ 0.45  0.75 -0.4 ] [0. 0. 1.]
C [-0.56  1.    0.99] [0.01 0.   0.99]
A [-0.92  1.    1.  ] [0.94 0.03 0.02]
A [-0.93  0.99  0.68] [0.92 0.08 0.  ]
A [-0.96  0.98  0.47] [0.88 0.12 0.  ]
A [-0.97  0.97  0.29] [0.83 0.17 0.  ]
B [-0.98  0.97  0.12] [0.75 0.25 0.  ]
B [-0.53  0.78 -0.52] [0.01 0.99 0.  ]
B [-0.17  0.6  -0.72] [0. 1. 0.]
B [ 0.4   0.47 -0.72] [0.   0.96 0.04]
C [ 0.91  0.52 -0.47] [0. 0. 1.]
C [0.25 1.   1.  ] [0. 0. 1.]
C [-0.07  1.    1.  ] [0. 0. 1.]
C [-0.57  1.    1.  ] [0.01 0.   0.99]
A [-0.92  1.    1.  ] [0.94 0.03 0.02]
B [-0.93  0.99  0.68] [0.92 0.08 0.  ]
C [-0.11  0.87 -0.02] [0.   0.17 0.83]
A [-0.89  1.    0.99] [0.92 0.04 0.04]
A [-0.93  0.99  0.69] [0.92 0.08 0.  ]
A [-0.96  0.98  0.48] [0.88 0.12 0.  ]
A [-0.97  0.97  0.3 ] [0.83 0.17 0.  ]
A [-0.98  0.97  0.13] [0.75 0.25 0.  ]
A [-0.98  0.96 -0.04] [0.66 0.34 0.  ]
A [-0.99  0.96 -0.2 ] [0.55 0.45 0.  ]
A [-0.99  0.95 -0.35] [0.44 0.56 0.  ]
B [-0.99  0.94 -0.46] [0.36 0.64 0.  ]
B [-0.77  0.63 -0.81] [0.01 0.99 0.  ]
B [-0.65  0.29 -0.89] [0. 1. 0.]
B [-0.46 -0.07 -0.92] [0. 1. 0.]
B [-0.03 -0.39 -0.92] [0. 1. 0.]
B [ 0.69 -0.54 -0.89] [0. 1. 0.]
B [ 0.98 -0.48 -0.72] [0. 1. 0.]
B [ 0.99 -0.26 -0.45] [0.   0.92 0.08]
C [ 1.    0.1  -0.13] [0. 0. 1.]
C [0.66 0.99 1.  ] [0. 0. 1.]
C [0.6 1.  1. ] [0. 0. 1.]
C [0.53 1.   1.  ] [0. 0. 1.]
C [0.43 1.   1.  ] [0. 0. 1.]
C [0.26 1.   1.  ] [0. 0. 1.]
C [-0.05  1.    1.  ] [0. 0. 1.]
C [-0.55  1.    1.  ] [0.01 0.   0.99]
A [-0.91  1.    1.  ] [0.94 0.03 0.03]
A [-0.93  0.99  0.68] [0.92 0.08 0.  ]
B [-0.96  0.98  0.47] [0.88 0.12 0.  ]
B [-0.29  0.84 -0.23] [0.01 0.98 0.01]
C [ 0.44  0.75 -0.4 ] [0. 0. 1.]
C [-0.57  1.    0.99] [0.01 0.   0.99]
A [-0.92  1.    1.  ] [0.94 0.03 0.02]
epoch: 9
error: 0.0072
```





With the output hidden activations, it is showed this network predicts the next C and A with hidden activations moving monotonically. The more A it received, the more decreasement in z direction, but x and y do not change much. Once the first B is received, the hidden activations moving with x increased, y decreased and z decreased. When the first C is received, the hidden activations' x is decreasing, while y and z do not change much. Similar to the anbn, the network predict the incoming by letting hidden activations hit the specific zone (not obvious when predicting C; predict the next one is A when the hidden activation of C is around -0.55). Hidden activations of B do not follow a specific line, the reason is perhaps that it contains the information about how much A has gone which it has to go to correspond to A. Also the information that needs to be passed on to hidden activations in C to let it know how far it should go.

4. python seq_train.py --lang reber --embed True --model lstm --hid 4

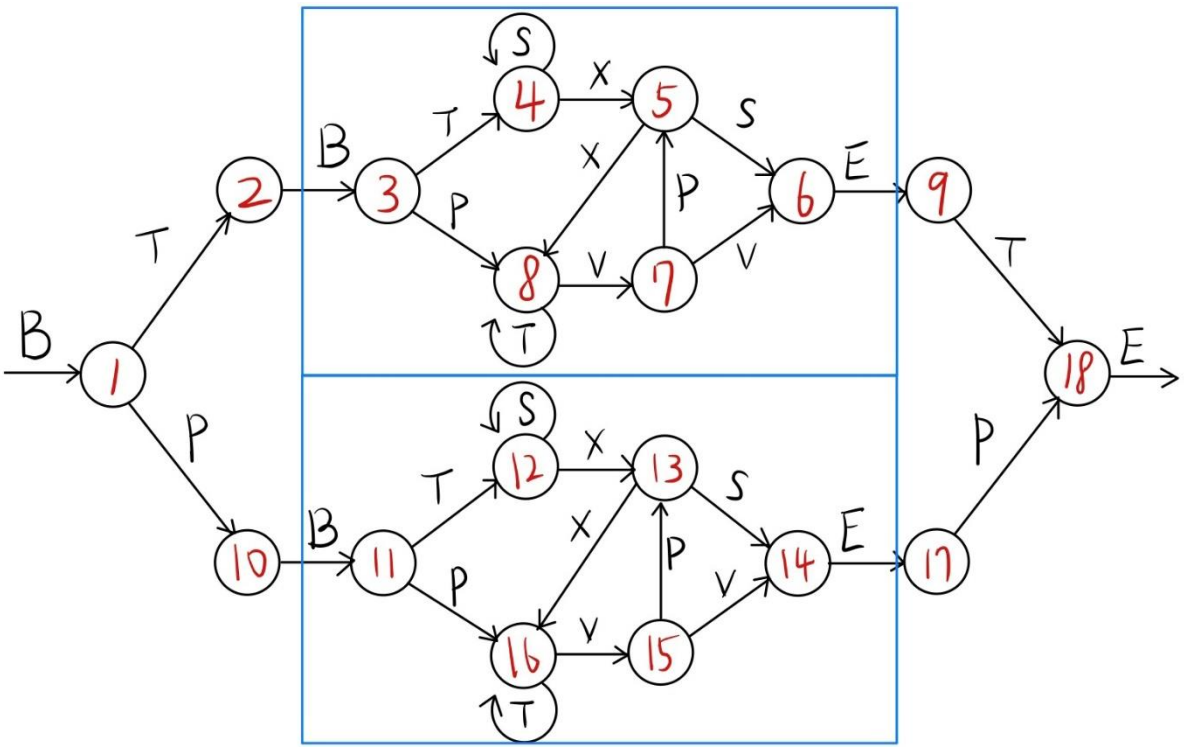
```
state =  0 1 2 3 4 4 4 4 5 6 9 18
symbol= BTBTSSSXSETE
label = 010122232616
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
2 [ 1.   0.   0.   0.   0.   0.   0.]
3 [ 0.   0.5  0.   0.   0.5  0.   0. ]
4 [ 0.   0.   0.5  0.5  0.   0.   0. ]
4 [ 0.   0.   0.5  0.5  0.   0.   0. ]
4 [ 0.   0.   0.5  0.5  0.   0.   0. ]
5 [ 0.   0.   0.5  0.5  0.   0.   0. ]
6 [ 0.   0.   0.   0.   0.   0.   1.]
9 [ 0.   1.   0.   0.   0.   0.   0.]
18 [ 0.   0.   0.   0.   0.   0.   1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-0.07 -0.75  0.23 -0.16] [ 0.    0.53  0.    0.    0.47  0.    0.  ]
2 [-0.63  0.61  0.95 -0.32] [ 1.    0.    0.    0.    0.    0.    0.]
3 [ 0.52 -0.75  0.1   0.27] [ 0.    0.51  0.    0.    0.48  0.01  0.  ]
4 [ 0.21  0.55  0.98 -0.2 ] [ 0.    0.    0.46  0.53  0.    0.    0.  ]
4 [ 0.11  0.92  0.97 -0.29] [ 0.01  0.    0.46  0.52  0.    0.    0.01]
4 [ 0.14  0.98  0.86 -0.23] [ 0.01  0.    0.48  0.51  0.    0.    0.01]
4 [ 0.2   0.99  0.82 -0.28] [ 0.    0.    0.47  0.51  0.    0.    0.01]
5 [ 0.45  0.86  0.59  0.54] [ 0.    0.    0.6   0.4   0.    0.    0. ]
6 [ 0.38  0.93  0.17 -0.67] [ 0.    0.    0.01  0.02  0.    0.    0.97]
9 [-0.47 -0.73 -0.23 -0.05] [ 0.    0.97  0.    0.    0.03  0.    0.  ]
18 [-0.54  0.52  0.13 -0.75] [ 0.    0.    0.    0.    0.    0.    1.]
epoch: 50000
error: 0.0006
final: 0.0003
```

```
state =  0 1 10 11 16 16 15 13 14 17 18
symbol= BPBPTVPSEPE
label = 04041542646
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
10 [ 1.   0.   0.   0.   0.   0.   0.]
11 [ 0.   0.5  0.   0.   0.5  0.   0. ]
16 [ 0.   0.5  0.   0.   0.   0.5  0. ]
16 [ 0.   0.5  0.   0.   0.   0.5  0. ]
15 [ 0.   0.   0.   0.   0.5  0.5  0. ]
13 [ 0.   0.   0.5  0.5  0.   0.   0. ]
14 [ 0.   0.   0.   0.   0.   0.   1.]
17 [ 0.   0.   0.   0.   1.   0.   0.]
18 [ 0.   0.   0.   0.   0.   0.   1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-0.07 -0.75  0.23 -0.15] [ 0.    0.53  0.    0.    0.46  0.    0.  ]
10 [-0.71  0.18  0.54  0.55] [ 1.    0.    0.    0.    0.    0.    0.]
11 [ 0.74 -0.73  0.67  0.54] [ 0.    0.48  0.    0.    0.51  0.    0.  ]
16 [ 0.94 -0.01 -0.63  0.89] [ 0.    0.41  0.    0.    0.02  0.57  0.  ]
16 [ 0.47 -0.03 -0.93  0.73] [ 0.    0.44  0.    0.    0.    0.55  0.  ]
15 [ 0.94  0.05 -0.97 -0.35] [ 0.    0.    0.    0.    0.42  0.57  0.  ]
13 [ 0.54  0.81 -0.02  0.22] [ 0.    0.    0.54  0.44  0.    0.01  0.  ]
14 [ 0.16  0.94 -0.8  -0.65] [ 0.    0.    0.    0.    0.    0.    1.]
17 [ 0.65 -0.74 -0.03 -0.12] [ 0.    0.05  0.    0.    0.95  0.    0.  ]
18 [-0.03  0.22 -0.64 -0.85] [ 0.    0.    0.    0.    0.    0.    1.]
epoch: 48000
error: 0.0010
final: 0.0007
```

```
state =  0 1 10 11 12 12 12 13 14 17 18
symbol= BPBTSSXSEPE
label = 04012232646
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
10 [ 1.   0.   0.   0.   0.   0.   0.]
11 [ 0.   0.5  0.   0.   0.5  0.   0. ]
12 [ 0.   0.   0.5  0.5  0.   0.   0. ]
12 [ 0.   0.   0.5  0.5  0.   0.   0. ]
12 [ 0.   0.   0.5  0.5  0.   0.   0. ]
13 [ 0.   0.   0.5  0.5  0.   0.   0. ]
14 [ 0.   0.   0.   0.   0.   0.   1.]
17 [ 0.   0.   0.   0.   1.   0.   0.]
18 [ 0.   0.   0.   0.   0.   0.   1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-0.05 -0.75  0.23 -0.16] [ 0.    0.48  0.    0.    0.51  0.    0.  ]
10 [-0.7   0.18  0.52  0.52] [ 1.    0.    0.    0.    0.    0.    0.]
11 [ 0.74 -0.74  0.65  0.53] [ 0.    0.46  0.    0.    0.53  0.    0.  ]
12 [ 0.86  0.53  0.97 -0.09] [ 0.    0.    0.46  0.52  0.02  0.    0.  ]
12 [ 0.54  0.92  0.94 -0.37] [ 0.    0.    0.45  0.55  0.    0.    0.  ]
12 [ 0.45  0.98  0.81 -0.21] [ 0.    0.    0.48  0.51  0.    0.    0.  ]
13 [ 0.75  0.82  0.06  0.56] [ 0.    0.01  0.61  0.37  0.    0.02  0.  ]
14 [ 0.19  0.91 -0.7  -0.7 ] [ 0.    0.    0.    0.    0.    0.    1.]
17 [ 0.63 -0.74 -0.03 -0.1 ] [ 0.    0.06  0.    0.    0.93  0.    0.  ]
18 [-0.03  0.21 -0.64 -0.84] [ 0.    0.    0.    0.    0.    0.    1.]
epoch: 47000
error: 0.0011
final: 0.0012
```

```
state =  0 1 2 3 8 8 7 6 9 18
symbol= BTBPTVVETE
label = 0104155616
true probabilities:
     B    T    S    X    P    V    E
1 [ 0.   0.5  0.   0.   0.5  0.   0. ]
2 [ 1.   0.   0.   0.   0.   0.   0.]
3 [ 0.   0.5  0.   0.   0.5  0.   0. ]
8 [ 0.   0.5  0.   0.   0.   0.5  0. ]
8 [ 0.   0.5  0.   0.   0.   0.5  0. ]
7 [ 0.   0.   0.   0.   0.5  0.5  0. ]
6 [ 0.   0.   0.   0.   0.   0.   1.]
9 [ 0.   1.   0.   0.   0.   0.   0.]
18 [ 0.   0.   0.   0.   0.   0.   1.]
hidden activations and output probabilities [BTSXPVE]:
1 [-0.09 -0.75  0.21 -0.13] [ 0.    0.59  0.    0.    0.4   0.    0.  ]
2 [-0.62  0.58  0.95 -0.3 ] [ 1.    0.    0.    0.    0.    0.    0.]
3 [ 0.53 -0.74  0.11  0.27] [ 0.    0.52  0.    0.    0.48  0.01  0.  ]
8 [ 0.64  0.11 -0.65  0.78] [ 0.    0.44  0.01  0.    0.01  0.54  0.  ]
8 [ 0.41 -0.09 -0.93  0.7 ] [ 0.    0.5   0.    0.    0.    0.5   0. ]
7 [ 0.85  0.06 -0.97 -0.39] [ 0.    0.01  0.    0.    0.36  0.63  0.  ]
6 [ 0.38  0.73 -0.53 -0.85] [ 0.    0.    0.    0.    0.    0.    1.]
9 [-0.5  -0.74 -0.06 -0.05] [ 0.    0.96  0.    0.    0.04  0.    0.  ]
18 [-0.57  0.5  -0.86 -0.79] [ 0.    0.    0.    0.    0.    0.    1.]
epoch: 44000
error: 0.0015
final: 0.0005
```

With the input symbols and the output probabilities, the Embedded Reber Grammar with numbers corresponding to the states in the hidden activations and output probabilites can be graphed,



The long short term memory has context units which is able to store the input information long ago. The scale of how much the information from the past will be forgot and how much in the information currently received will be stored in the context units is determined by gates (forget gate, input gate, update values and output gate.) And the gates values are determined by the hidden units (state) and the input. In this way, when the input sequence hits states inside blue boxes (reber grammers), the output values are determined mainly by the hidden units. When the input sequence hits states outside the blue boxes, the output values are determined mainly by the context units. The context units enable long short term memory model to handle long range dependencies.

```
0
context unit:tensor([[ 0.8065, -0.9761,  0.7006,  0.8299]], grad_fn=<AddBackward0>)
hidden unit:tensor([[ 0.6492, -0.7494,  0.5834,  0.6066]], grad_fn=<MulBackward0>)
----
1
context unit:tensor([[ 1.0958,  0.4813, -0.6211,  1.5372]], grad_fn=<AddBackward0>)
hidden unit:tensor([[ 0.7948,  0.4443, -0.5393,  0.0311]], grad_fn=<MulBackward0>)
----
2
context unit:tensor([[ 0.6382, -0.6913, -0.0728,  2.0139]], grad_fn=<AddBackward0>)
hidden unit:tensor([[ 0.5535, -0.5866, -0.0474,  0.9461]], grad_fn=<MulBackward0>)
----
3
context unit:tensor([[-0.4911, -0.9709, -0.9622,  2.7131]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.4539, -0.7469, -0.7359,  0.1609]], grad_fn=<MulBackward0>)
----
4
context unit:tensor([[-0.9844, -1.4554, -0.8496,  2.6839]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.7171, -0.8952, -0.6894,  0.5654]], grad_fn=<MulBackward0>)
----
5
context unit:tensor([[-1.6774, -0.6619,  0.3740,  3.6194]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.9185, -0.5789,  0.3570,  0.9903]], grad_fn=<MulBackward0>)
----
6
context unit:tensor([[-2.5730,  0.9680,  1.2605,  4.5363]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.9345,  0.7474,  0.8508,  0.8272]], grad_fn=<MulBackward0>)
0
context unit:tensor([[ 0.8067, -0.9761,  0.7008,  0.8298]], grad_fn=<AddBackward0>)
hidden unit:tensor([[ 0.6493, -0.7494,  0.5835,  0.6066]], grad_fn=<MulBackward0>)
----
1
context unit:tensor([[ 1.0734,  0.9210, -0.5261,  1.3823]], grad_fn=<AddBackward0>)
hidden unit:tensor([[ 0.7308,  0.7211, -0.4768,  0.0187]], grad_fn=<MulBackward0>)
----
2
context unit:tensor([[ 0.6806, -0.5431, -0.0825,  1.5842]], grad_fn=<AddBackward0>)
hidden unit:tensor([[ 0.5749, -0.4830, -0.0478,  0.9002]], grad_fn=<MulBackward0>)
----
3
context unit:tensor([[-0.4482, -0.9515, -0.9632,  2.2293]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.4191, -0.7376, -0.7330,  0.1693]], grad_fn=<MulBackward0>)
----
4
context unit:tensor([[-1.2816, -0.9437,  0.3579,  3.1421]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.8401, -0.7321,  0.3408,  0.9911]], grad_fn=<MulBackward0>)
----
5
context unit:tensor([[-1.8439,  0.2212, -0.2812,  3.9433]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.9442,  0.2176, -0.2740,  0.0498]], grad_fn=<MulBackward0>)
----
6
context unit:tensor([[-2.1638, -0.7497, -0.7887,  0.5391]], grad_fn=<AddBackward0>)
hidden unit:tensor([[-0.8713, -0.6318, -0.6543,  0.3758]], grad_fn=<MulBackward0>)
```