

Part 1: Japanese Character Recognition

1. NetLin

`self.fc = nn.Linear(28*28, 10)`

input features $28 * 28 = 784$

weights number = input * output = $784 * 10 = 7840$

bias = 10

Total : weights number + bias = 7850

```
<class 'numpy.ndarray'>
[[ 768.   5.   8.  13.  31.  63.   2.  62.  30.  18.]
 [   7. 667. 108.  18.  30.  23.  59.  11.  26.  51.]
 [   6.  59. 690.  26.  28.  20.  48.  37.  47.  39.]
 [   5.  34.  60. 756.  15.  57.  16.  18.  28.  11.]
 [  59.  51.  77.  20. 626.  19.  33.  38.  20.  57.]
 [   7.  26. 121.  17.  19. 726.  27.  11.  35.  11.]
 [   5.  21. 145.  11.  24.  24. 727.  21.  10.  12.]
 [  15.  29.  26.  12.  83.  16.  54. 622.  93.  50.]
 [  10.  39.  97.  40.   7.  31.  46.   7. 702.  21.]
 [   8.  50.  83.   3.  56.  31.  18.  33.  41. 677.]]

Test set: Average loss: 1.0091, Accuracy: 6961/10000 (70%)

Model parameters have :7850
```

2. NetFull

`self.fc1 = nn.Linear(28*28, 110)`

input features $28 * 28 = 784$ output = 110

weights = input * output = 86240 bias = 110

number of independent parameters(fc1) : weights + bias = **86350**

`self.fc2 = nn.Linear(110, 10)`

input = 110 output = 10

weight = input * output = 1100 bias = 10

number of independent parameters(fc2) : weights + bias = **1110**

Total = 86350 + 1110 = 87460

```
<class 'numpy.ndarray'>
[[ 852.   3.   2.   5.  27.  26.   3.  44.  34.   4.]
 [   6. 805.  32.   7.  23.  11.  63.   4.  24.  25.]
 [   8.  14. 830.  37.  17.  20.  21.  12.  26.  15.]
 [   3.   8.  31. 916.   2.  13.   6.   7.   8.   6.]
 [  45.  33.  20.  10. 794.   8.  33.  19.  24.  14.]
 [  10.   9.  78.  15.  14. 837.  21.   2.  10.   4.]
 [   3.   9.  46.   8.  12.   4. 904.   5.   2.   7.]
 [  18.  18.  16.   8.  27.  12.  42. 799.  33.  27.]
 [  14.  24.  25.  42.   2.  10.  28.   3. 845.   7.]
 [   3.  18.  51.   5.  27.   8.  23.  11.  14. 840.]]

Test set: Average loss: 0.5174, Accuracy: 8422/10000 (84%)

Model parameters have :87460
```

3. conv

```
<class 'numpy.ndarray'>
[[962.  5.  1.  2. 21.  1.  0.  6.  1.  1.]
 [ 2. 923. 10.  0. 11.  0. 30.  5.  4. 15.]
 [11. 10. 896. 25.  7.  4. 24.  9.  9.  5.]
 [ 4.  1. 20. 948.  2.  8.  7.  4.  4.  2.]
 [25. 10.  2.  3. 924.  5.  9.  9.  9.  4.]
 [ 4.  6. 49.  9.  5. 878. 21. 16.  2. 10.]
 [ 4.  6. 11.  1.  4.  0. 964.  6.  1.  3.]
 [ 7.  5.  6.  0.  6.  0.  3. 949.  5. 19.]
 [ 7. 14.  7.  4.  5.  3. 10.  5. 933. 12.]
 [ 7.  5.  8.  2. 11.  0.  7.  4.  7. 949.]]
```

Test set: Average loss: 0.2585, Accuracy: 9326/10000 (93%)

Model parameters have :392354

self.conv1 = nn.Conv2d(1, 48, kernel_size=3)

input = 1 output = 48 kernel-size = 3*3

weight = input * output * kernel-size = 432 bias = 48

number of independent parameters(conv1) = weight + bias = 480

self.conv2 = nn.Conv2d(48, 88, kernel_size=3)

input = 48 output = 88 kernel-size = 3*3

weight = input * output * kernel-size = 37904 bias = 48

number of independent parameters(conv2) = weight + bias = 37992

self.fc1 = nn.Linear(88 * 5 * 5, 160) #32 channel 5*5 kernal

input = 88 * 5 * 5 output = 160

weight = input * output = 352000 bias = 160

number of independent parameters(fc1) = weight + bias = 352160

self.fc2 = nn.Linear(160, 10)

input = 160 output = 10

weight = input * output = 1600 bias = 10

number of independent parameters(fc2) = weight + bias = 1610

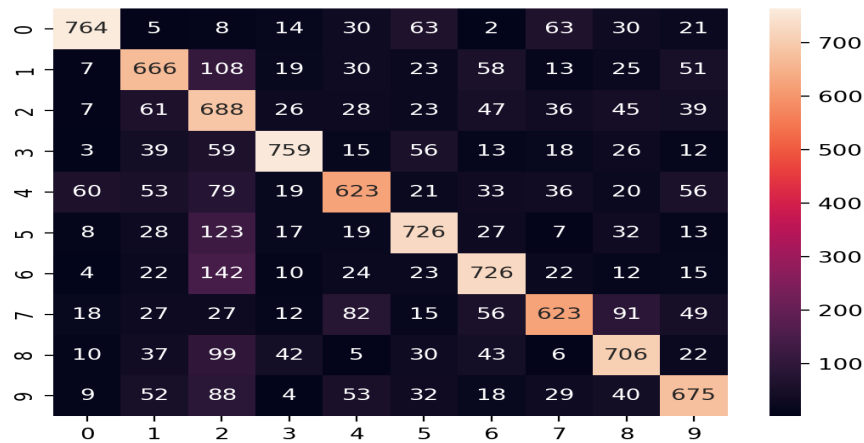
Total = 480 + 37992 + 352160 + 1610 = 392354

4.

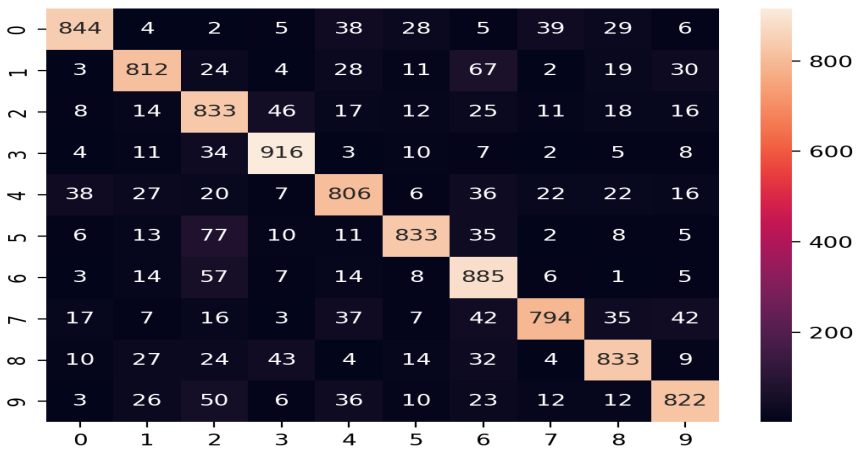
	NetLin	NetFull	NetConv
Accuracy	0.7	0.84	0.93
Total_parameter	7850	87460	392354

C.

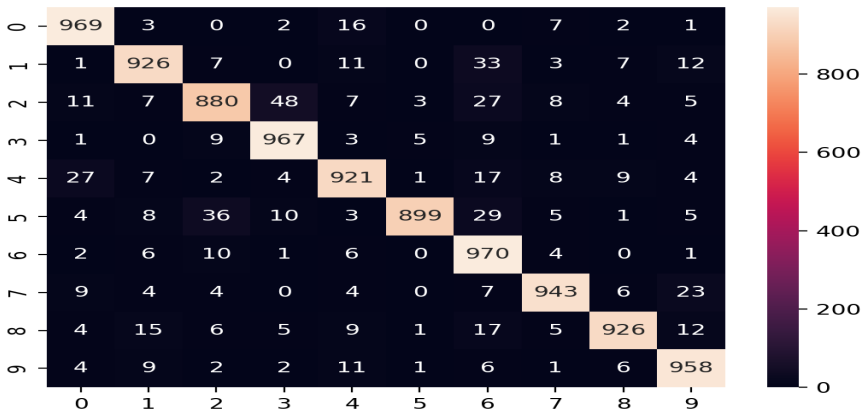
lin.matrix:



full.matrix:



conv.matrix



After visualizing the confusion matrix, I found that **characters 5(real label)** are most likely to be mistaken for **characters 2(predict)** from the confuse matrix

	NeiLin	Netfull	NetConv
--	--------	---------	---------

5 -> 2 samples	123	77	36
----------------	-----	----	----

All models have relatively large error rate in mistaking the 5(label) to 2,
so such situation doesn't attribute to models but the data itself; Indeed,
character 5 are relatively similar to char2 compare with others

Hiragana	Unicode	Samples	Sample Images
お (o)	U+304A	7000	
き (ki)	U+304D	7000	
す (su)	U+3059	7000	
つ (tsu)	U+3064	7000	
な (na)	U+306A	7000	

Hiragana	Unicode	Samples	Sample Images
は (ha)	U+306F	7000	
ま (ma)	U+307E	7000	
や (ya)	U+3084	7000	
れ (re)	U+308C	7000	
を (wo)	U+3092	7000	

Otherwise, models matter in prediction, from character 1(label) to
2(prediction); Netlin performs obviously worst in three models here since it
has a huge rate in making such specified error

	NeiLin	Netfull	NetConv
1-> 2 samples	108	24	7

Part 2: Multi-Layer Perceptron

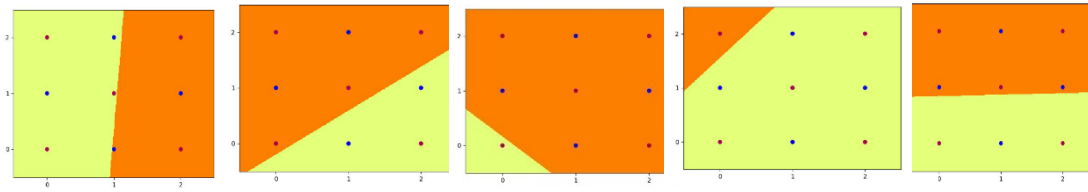
1.

```

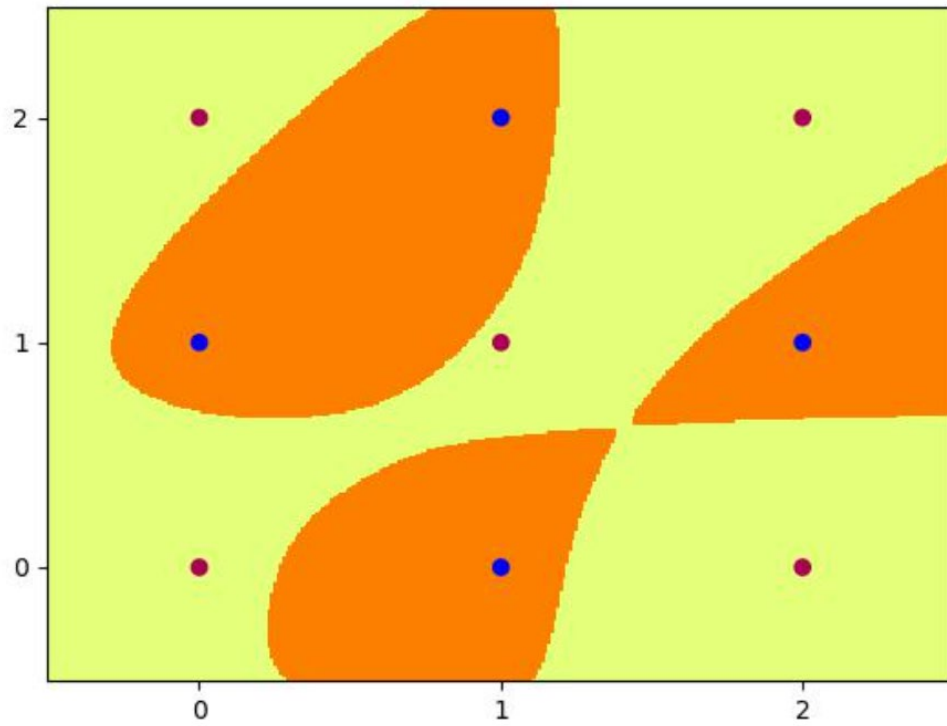
ep: 6100 loss: 0.1574 acc: 100.00
ep: 6200 loss: 0.1535 acc: 100.00
Final Weights:
Final Weights:
tensor([[ 5.0707, -0.3579],
        [-5.8170,  7.3803],
        [ 5.8790,  5.8983],
        [-6.1128,  4.9425],
        [-0.1042,  3.9683]], device='cuda:0')
tensor([-4.9386,  1.4484, -0.9586, -7.6813, -3.3357], device='cuda:0')
tensor([[ -7.4816, -7.5532,  5.8305, -7.6027,  6.4499]], device='cuda:0')
tensor([-0.1933], device='cuda:0')
Final Accuracy: 100.0

```

hidden_5_(1-5).Jpg



out_5.jpg



2.

assume that the diagram is divided as below



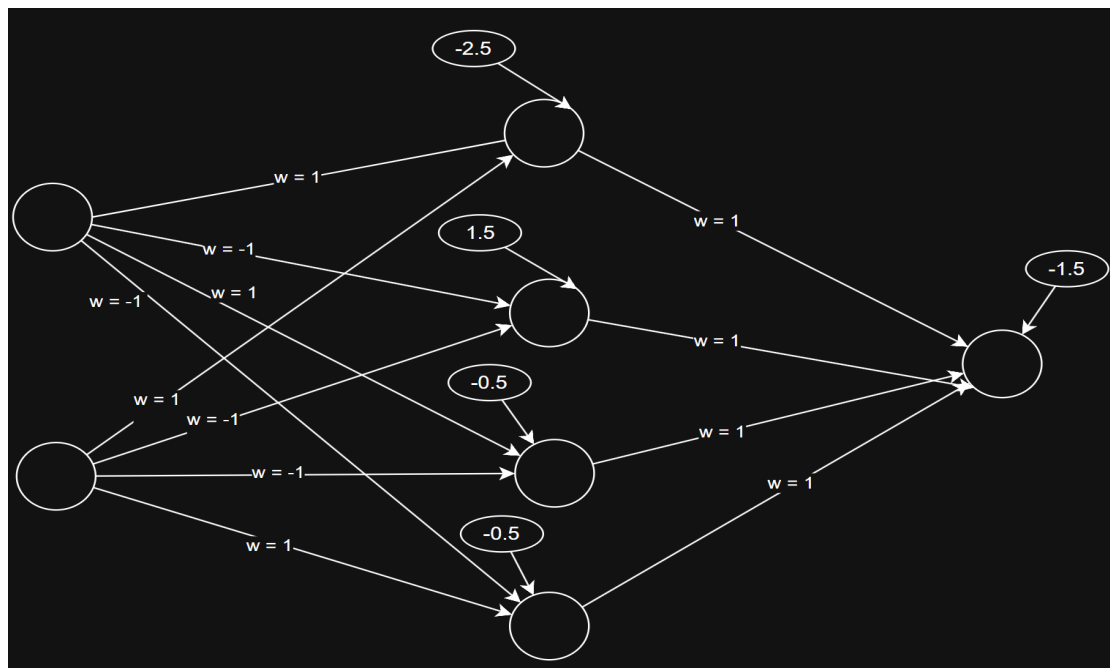
X1	X2	Hidden_layer	y
0	0	0100	0
0	1	0101	1
0	2	0001	0
1	0	0110	1
1	1	0000	0
1	2	1001	1

2	0	0010	0
2	1	1010	1
2	2	1000	0

Hidden layer1 output 1 in (1,2) (2,1) (2,2) set ' $W_1 x_1 + W_2 x_2 + \text{bias} \geq 0$ ' only work in these nodes, calculate bias and weights

Repeat the process until we get

H1	$x_1 + x_2 - 2.5 = 0$
H2	$-x_1 - x_2 + 1.5 = 0$
H3	$x_1 - x_2 - 0.5 = 0$
H4	$-x_1 + x_2 - 0.5 = 0$

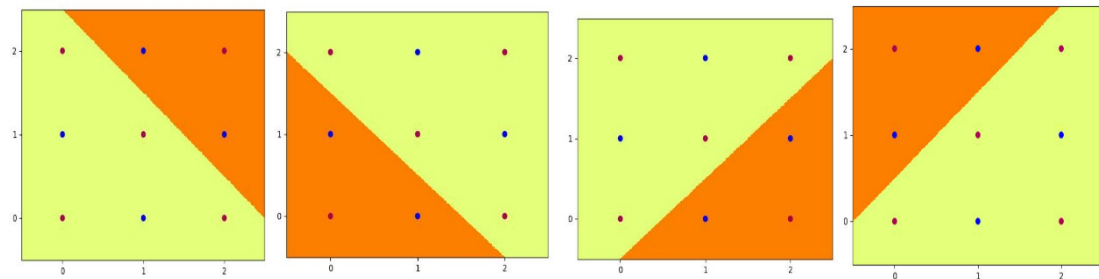


test model::

```
PS D:\Desktop\code\9444_ass1\al> python check_main.py --act step --hid 4 --set_weights
Using device: cuda
Initial Weights:
tensor([[ 1.,  1.],
        [-1., -1.],
        [ 1., -1.],
        [-1.,  1.]], device='cuda:0')
tensor([-2.5000,  1.5000, -0.5000, -0.5000], device='cuda:0')
tensor([[1., 1., 1., 1.]], device='cuda:0')
tensor([-1.5000], device='cuda:0')
Initial Accuracy: 100.0
```

3.

images showing the function computed by each hidden node



```
PS D:\Desktop\code\9444_ass1\al> python check_main.py --act sig --hid 4 --set_weights
Using device: cuda
Initial Weights:
tensor([[ 10., 10.],
        [-10., -10.],
        [ 10., -10.],
        [-10., 10.]], device='cuda:0')
tensor([-25., 15., -5., -5.], device='cuda:0')
tensor([[10., 10., 10., 10.]], device='cuda:0')
tensor([-15.], device='cuda:0')
Initial Accuracy: 100.0
```

images showing the function computed network as a whole in part 2

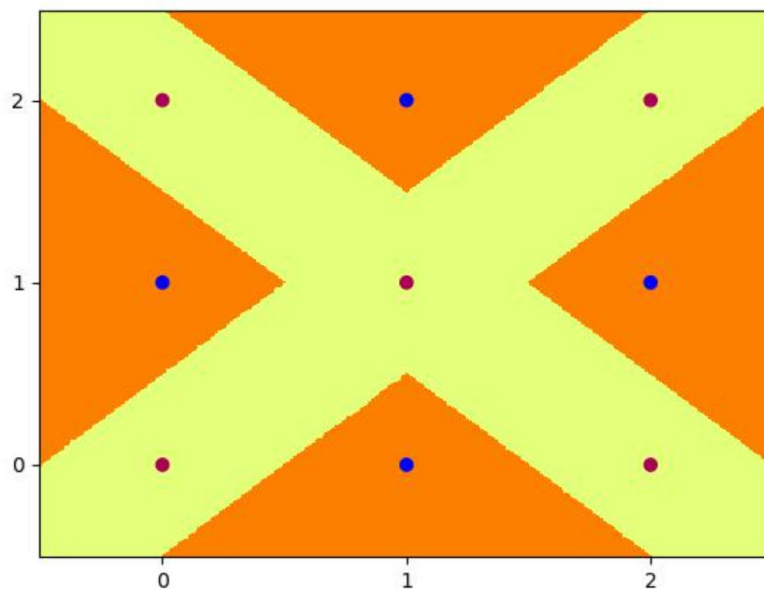
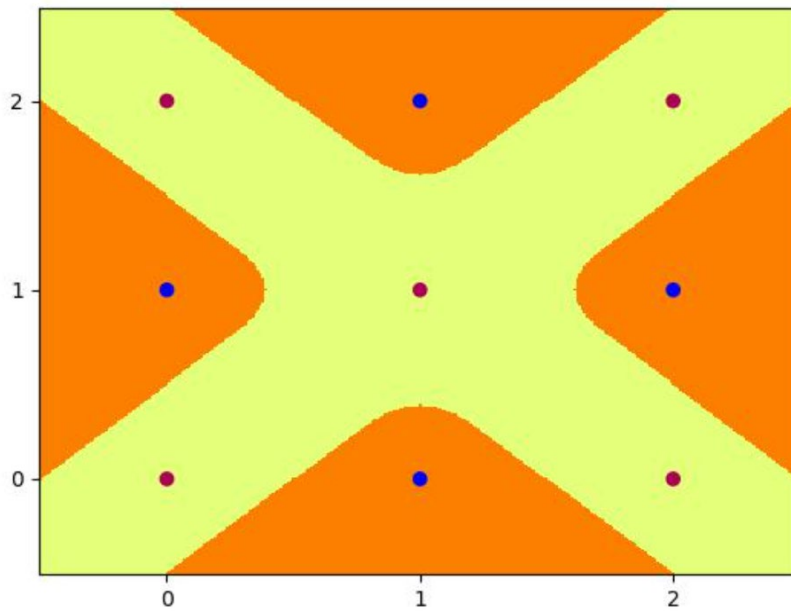


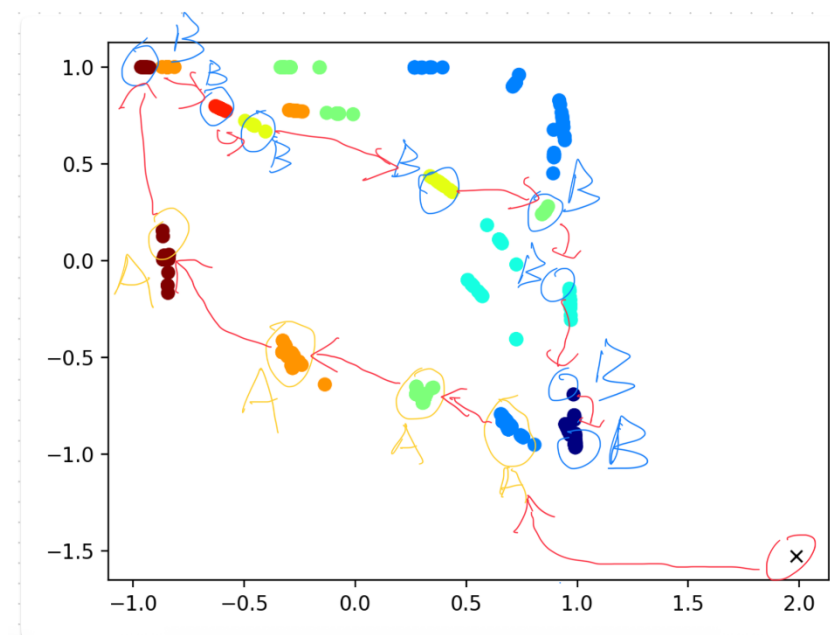
image showing the function computed network as a whole in part 3(with sigmoid)



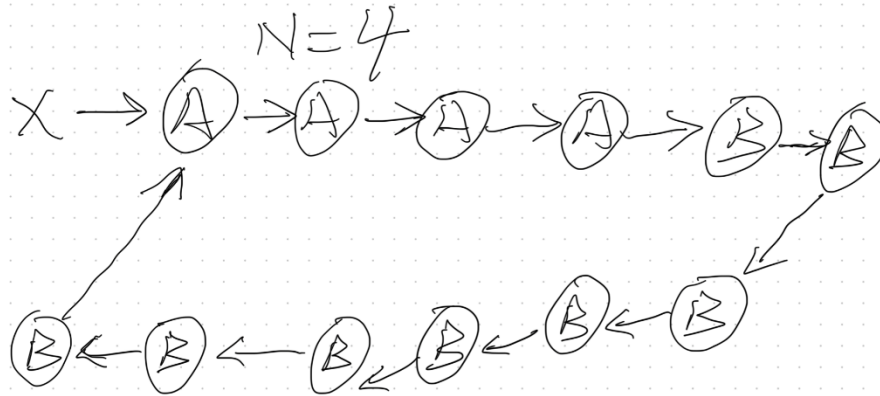
Part 3: Hidden Unit Dynamics for Recurrent Networks

1.

In the diagram , one circulation with $N = 4$ is shown; I wanna show such circulation so the last B does not link to next A



2.

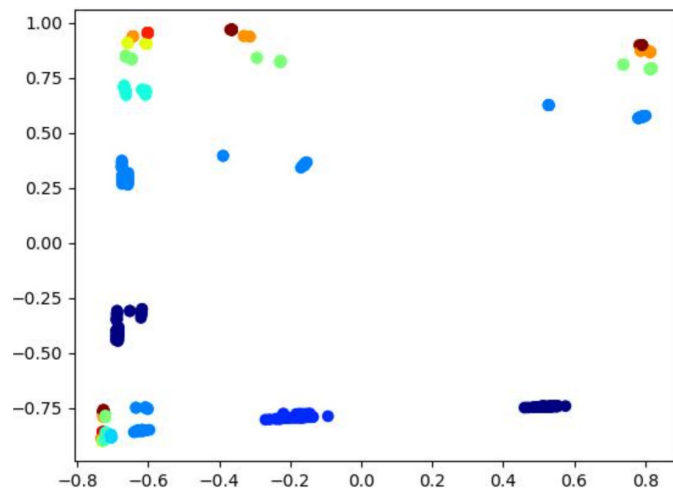


3.

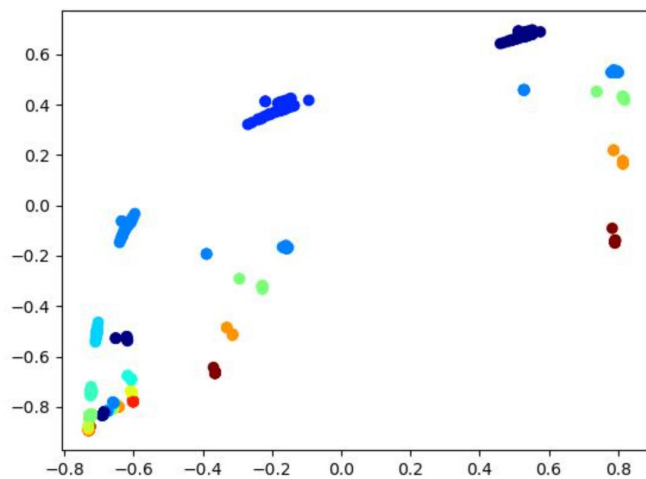
- RNN use the hidden layer activation to restore a hidden state and memories, it counts how many A to determine N, then it output the double-N result B in the next steps
- Network starts from initial state; whenever A is entered, the hidden unit activations repeat regular and directed movement between various states(these states also indicates how many A up to now). Such states can record how many A has been entered up to now (memory ability); Otherwise, the more A appears, the less predicted probability of A in next enter but B more likely appear next time; RNN learns from the relation of time and states with the memory ability
- When the first B input appears, means N is known for the network(the hidden state knows how many B will come now); The network moves along the path of B states clusters and starts counts backwards. With each state counts, the hidden unit activations is one step closer to the initial state. Except the first B input, the following input B should be predicted with a high probability by the network.

•When the last B input is finished(count backward is over), the network state will return to initial state, here the network and its memory has been reset; Under such situation, the network will predict the next input A in a high probability, and is ready to start a new $A^n b^{2^n}$ cycle

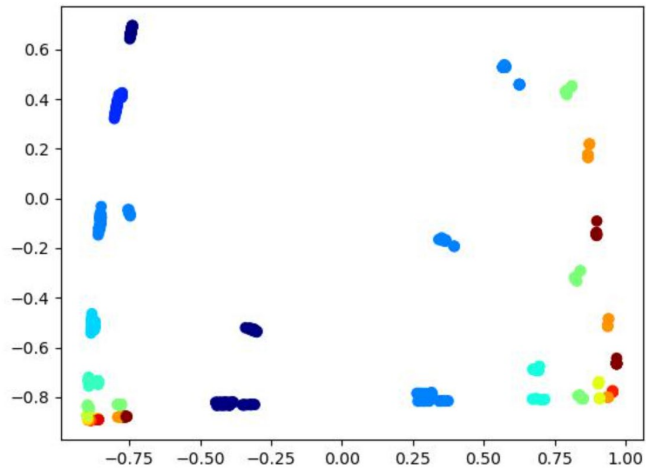
4.



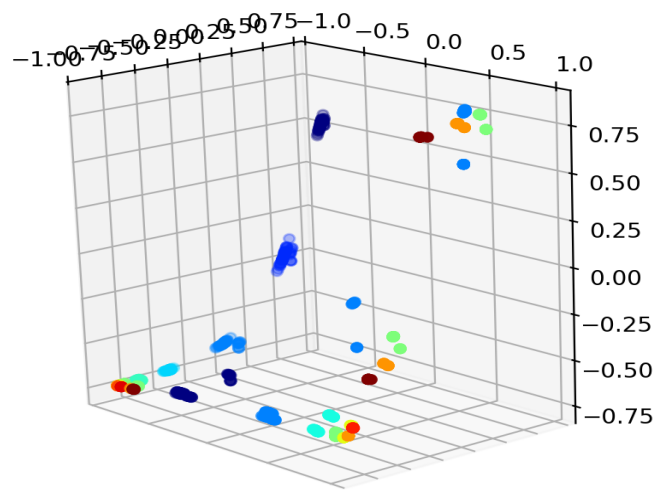
Istm3 01



Istm3 02

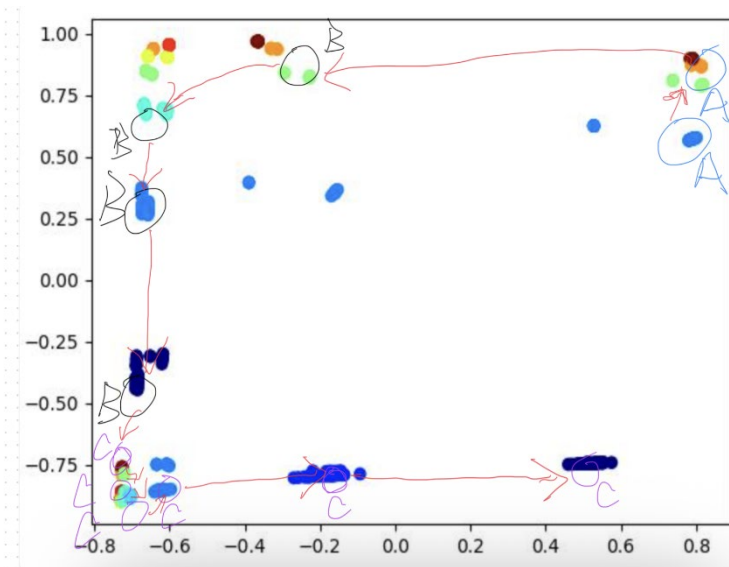


Istm3 12

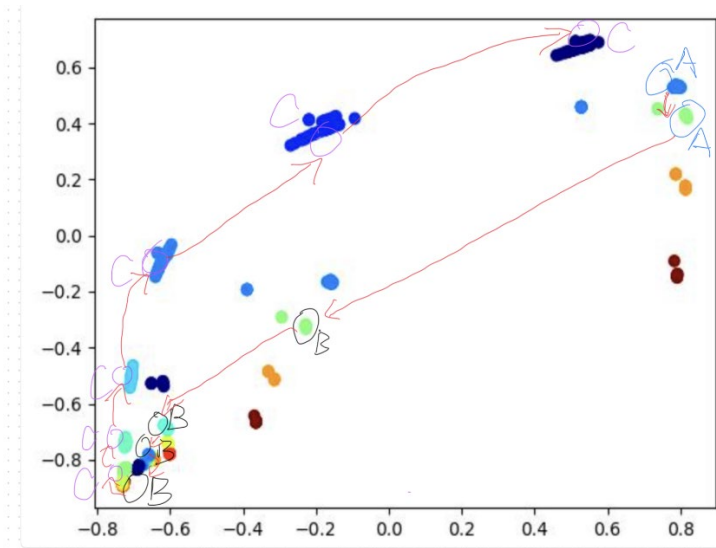


Istm3 3d

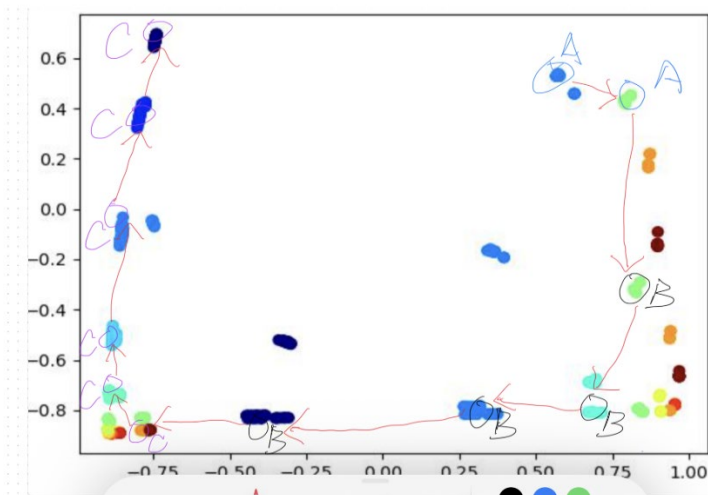
5.



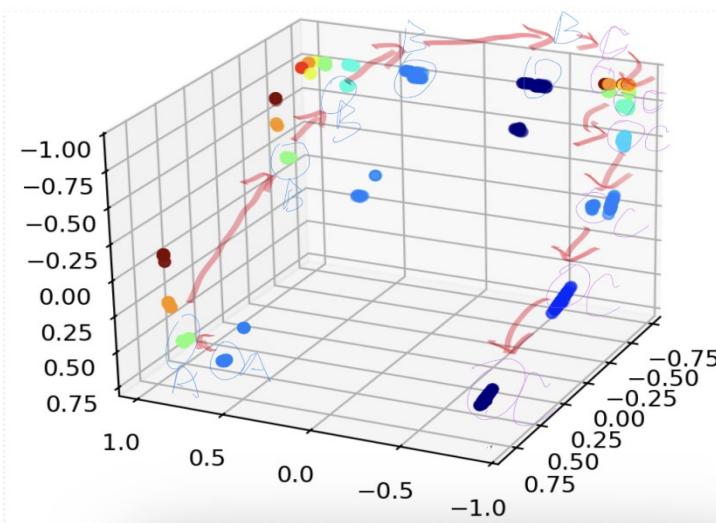
Istm3 01 annotated



Istm3 02 annotated



Istm3 12 annotated



Istm3 3d annotated

In LSTM, it adds a memory cell and several gates:

forget gate determines which historical memories should be saved

input gate determines which new data should be entered

output gate determines what information should be output

memory cell will remember the n value directly, that's why LSTM can output B

by $2n$, C by $3n$ much easily; It creates **a long term memory** for the value, then

Lstm can maintain performance in a long-period run, meanwhile it still exploits

the advantages of the hidden states to make next prediction(**Short-Term**

Memory)

Compare it with RNN, RNN is hard to perform well especially in the phase

of generating C with large N value; Its hidden state might forget the N value

since RNN needs to keep counting N from the above data, it lacks the ability of

LSTM to remember it in a long-term