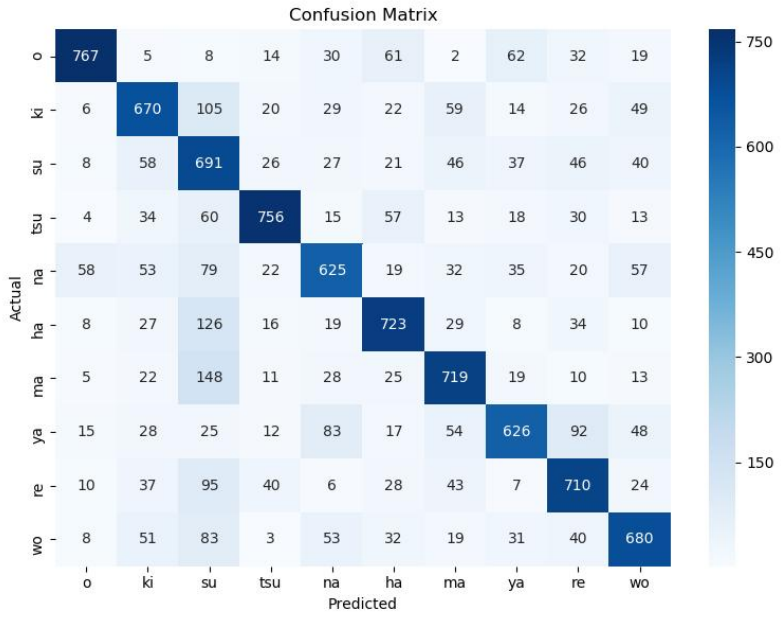
z5564038 Xinyu Li

Part 1: Japanese Character Recognition

1.Netlin:

```
Test set: Average loss: 1.0085, Accuracy: 6967/10000 (70%)
```



From the confusion matrix, the most frequent misclassification is ma to su (148 times), which means ma is most likely to be mistaken for su.

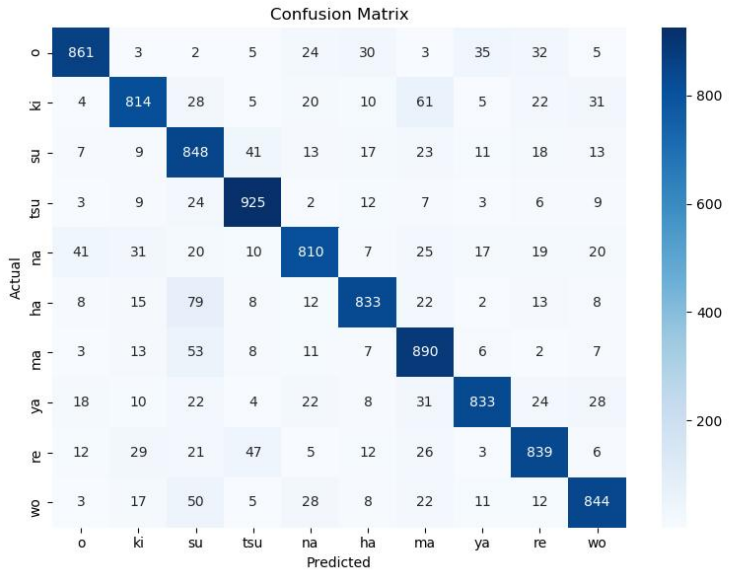
```python
class NetLin(nn.Module):
    # linear function followed by log_softmax
    def __init__(self):
        super(NetLin, self).__init__()
        # INSERT CODE HERE
        self.fc = nn.Linear(28 * 28, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)
        # CHANGE CODE HERE
```

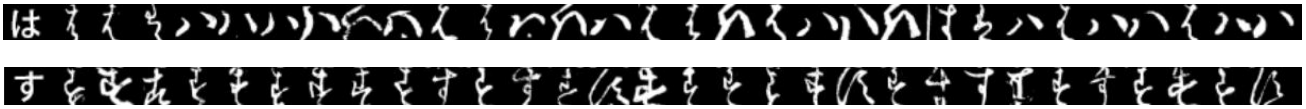
So the total number of independent parameters in the network is 28*28*10+10 = 7850.

2.NetFull:

```
Test set: Average loss: 0.4981, Accuracy: 8497/10000 (85%)
```



From the confusion matrix, the most frequent misclassification is ha to su (79 times), which means ha is most likely to be mistaken for su.



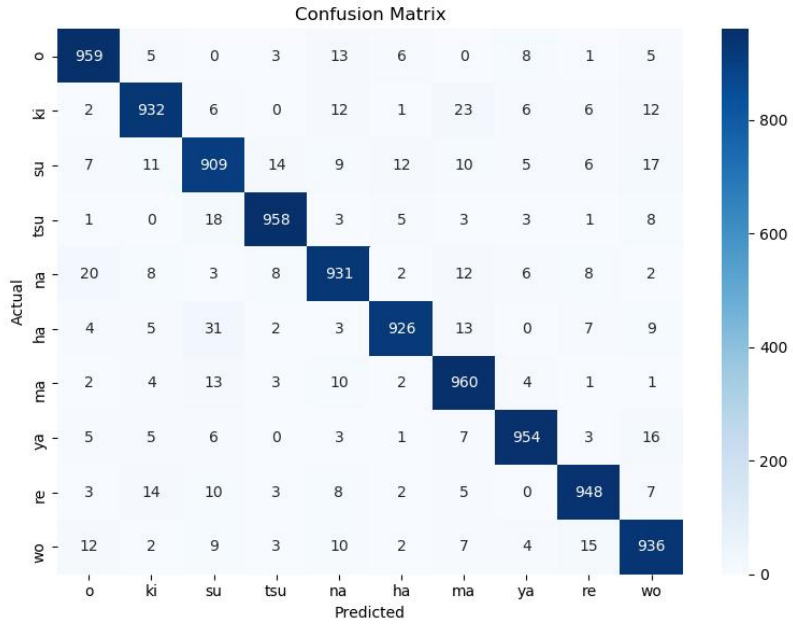
```python
class NetFull(nn.Module):
    # two fully connected tanh layers followed by log softmax
    def __init__(self):
        super(NetFull, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 200)
        self.fc2 = nn.Linear(200, 10)
        # INSERT CODE HERE

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = torch.tanh(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
        # CHANGE CODE HERE
```

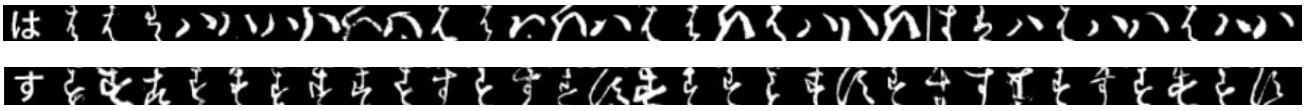
So the total number of independent parameters in the network is (28*28*200+200) + (200*10+10) = 159010.

3.NetConv:

```
Test set: Average loss: 0.2546, Accuracy: 9413/10000 (94%)
```



From the confusion matrix, the most frequent misclassification is ha to su (31 times), which means ha is most likely to be mistaken for su.



```python
class NetConv(nn.Module):
    # two convolutional layers and one fully connected layer,
    # all using relu, followed by log_softmax
    def __init__(self):
        super(NetConv, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 14 * 14, 128)
        self.fc2 = nn.Linear(128, 10)
        # INSERT CODE HERE

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 14 * 14)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
        # CHANGE CODE HERE
```

Conv1: 1*32*3*3 + 32 = 320

Conv2: 32*64*3*3 + 64 = 18496

Fc1: 64*14*14*128 + 128 = 1605760

Fc2: 128*10 + 10 = 1290

So the total number of independent parameters in the network is: 320 + 18496 + 1605760 + 1290 = 1625866.

4.The relative accuracy of the three models:

NetLin gets the accuracy around 70%, by using a single linear layer. NetFull improves the accuracy to 85%, by using a hidden layer with non-linear activation (tanh). NetConv gets 94% accuracy, by using two convolutional layers.

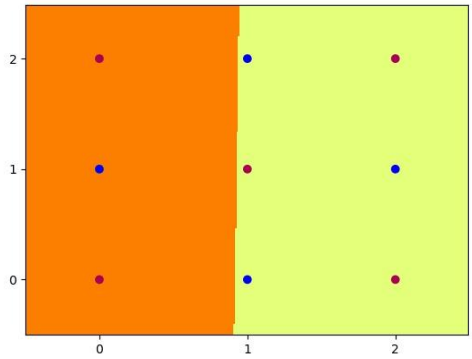
NetFull improves over NetLin by (85-70)/70*100%=21.43%

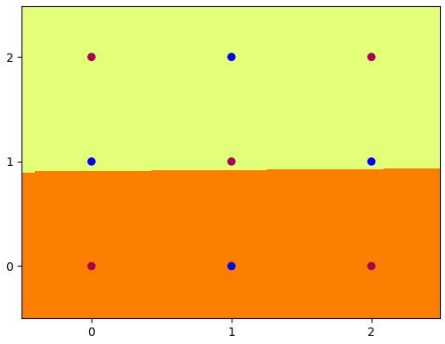
NetConv improves over NetFull by (94-85)/85*100%=10.59%

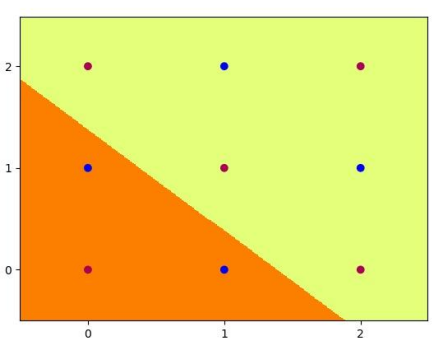
Part 2: Multi-Layer Perceptron

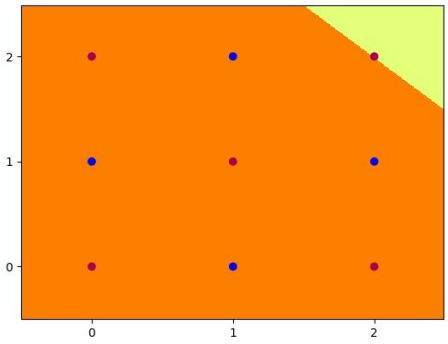
1. hid_5_0-4.jpg & out_5.jpg

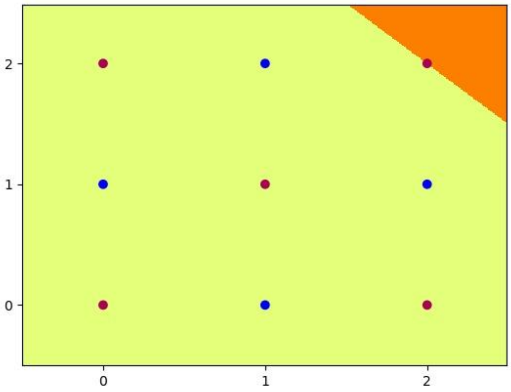
```
ep:24800 loss: 0.1911 acc: 100.00
ep:24900 loss: 0.1861 acc: 100.00
ep:24900 loss: 0.1861 acc: 100.00
ep:25000 loss: 0.1813 acc: 100.00
ep:25100 loss: 0.1765 acc: 100.00
ep:25200 loss: 0.1719 acc: 100.00
ep:25200 loss: 0.1719 acc: 100.00
ep:25300 loss: 0.1675 acc: 100.00
ep:25400 loss: 0.1631 acc: 100.00
ep:25500 loss: 0.1589 acc: 100.00
Final Weights:
tensor([[-4.1443,  0.0476],
        [ 0.0457, -3.7893],
        [-5.4587, -5.4910],
        [-2.7505, -2.7539],
        [ 2.7339,  2.7371]])
tensor([  3.7908,   3.4290,   7.5486,  10.9684, -10.9162])
tensor([[ -9.6418,  -9.8199,   8.9063,   9.0887, -10.3871]])
tensor([-1.8343])
Final Accuracy:  100.0
```

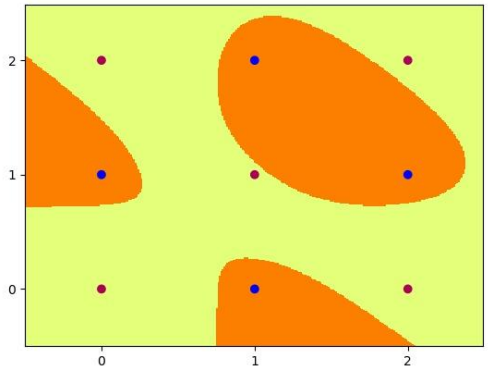






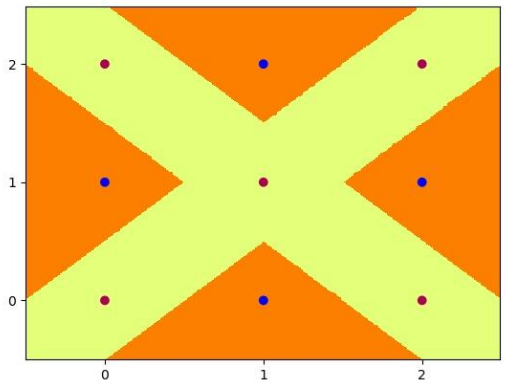




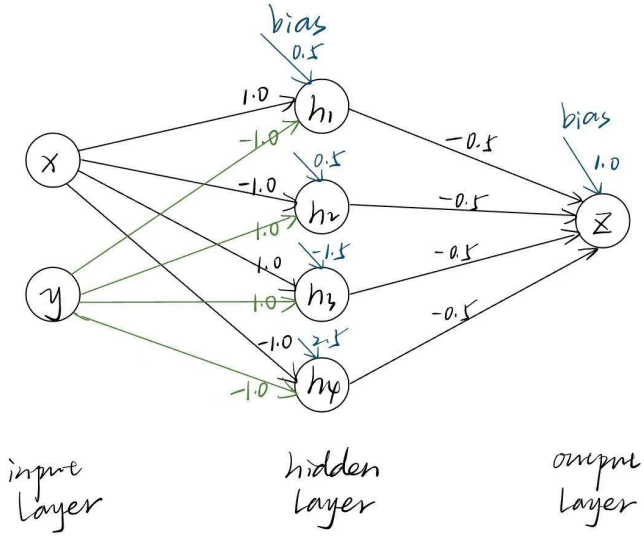


2. 2-layer neural network with 4 hidden nodes:

```
Initial Weights:
tensor([[ 1., -1.],
        [-1.,  1.],
        [ 1.,  1.],
        [-1., -1.]])
tensor([ 0.5000,  0.5000, -1.5000,  2.5000])
tensor([[-0.5000, -0.5000, -0.5000, -0.5000]])
tensor([1.])
Initial Accuracy:  100.0
```



N7



(2) the equations for the dividing line:

∵ $z = w_1 \cdot x + w_2 \cdot y + b$

Heaviside step: $h = step(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases}$

∴ $w_1 \cdot x + w_2 \cdot y + b = 0$

h_1: $1.0 \cdot x - 1.0 \cdot y + 0.5 = 0 \Rightarrow x - y + 0.5 = 0$

h_2: $-1.0 \cdot x + 1.0 \cdot y + 0.5 = 0 \Rightarrow -x + y + 0.5 = 0$

h_3: $1.0 \cdot x + 1.0 \cdot y - 1.5 = 0 \Rightarrow x + y - 1.5 = 0$

h_4: $-1.0x - 1.0 \cdot y + 2.5 = 0 \Rightarrow -x - y + 2.5 = 0$

(3)

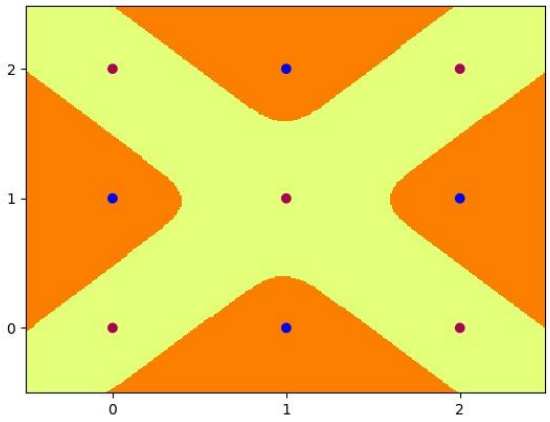
X	Y	H1	H2	H3	H4	Output
0	0	0	0	0	0	0
0	1	-1	1	1	-1	1
0	2	-2	2	2	-2	0
1	0	1	-1	1	-1	1
1	1	0	0	2	-2	0
1	2	-1	1	3	-3	1
2	0	2	-2	2	-2	0

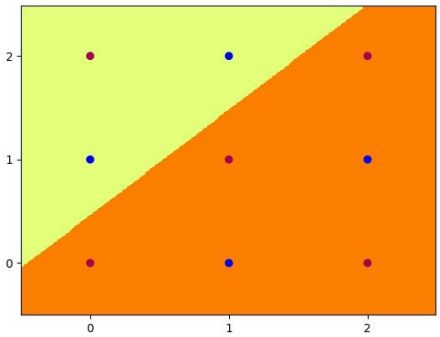
2	1	1	-1	3	-3	1
2	2	0	0	4	-4	0

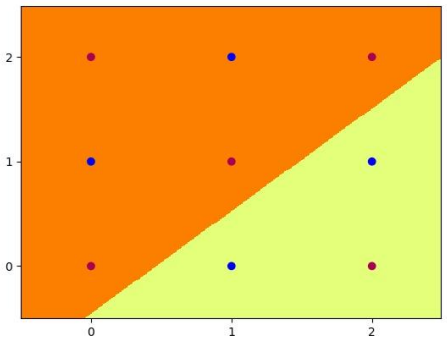
3. Rescale:

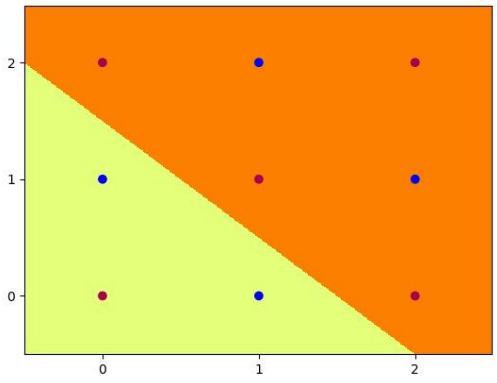
hid_4_0-3.jpg & out_4.jpg

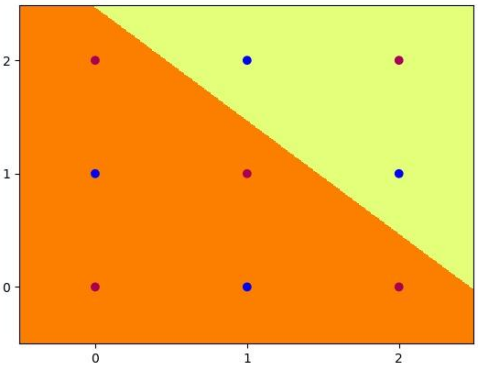
```
ep:  100 loss: 0.2631 acc: 100.00
ep:  200 loss: 0.2153 acc: 100.00
ep:  300 loss: 0.1803 acc: 100.00
ep:  400 loss: 0.1550 acc: 100.00
ep:  500 loss: 0.1371 acc: 100.00
ep:  600 loss: 0.1247 acc: 100.00
ep:  700 loss: 0.1162 acc: 100.00
ep:  800 loss: 0.1106 acc: 100.00
ep:  900 loss: 0.1070 acc: 100.00
ep: 1000 loss: 0.1047 acc: 100.00
ep: 1100 loss: 0.1033 acc: 100.00
ep: 1200 loss: 0.1023 acc: 100.00
ep: 1300 loss: 0.1017 acc: 100.00
ep: 1400 loss: 0.1011 acc: 100.00
ep: 1500 loss: 0.1007 acc: 100.00
ep: 1600 loss: 0.1003 acc: 100.00
ep: 1700 loss: 0.0998 acc: 100.00
ep: 1800 loss: 0.0994 acc: 100.00
ep: 1900 loss: 0.0989 acc: 100.00
ep: 2000 loss: 0.0984 acc: 100.00
Final Weights:
tensor([[ 10.6001, -10.4016],
        [-10.4016,  10.6001],
        [ 10.1827,  10.1827],
        [-10.0516, -10.0516]])
tensor([  4.8107,   4.8107, -15.2398,  24.8158])
tensor([[-4.4274, -4.4274, -4.4287, -4.4278]])
tensor([10.8860])
Final Accuracy:  100.0
```









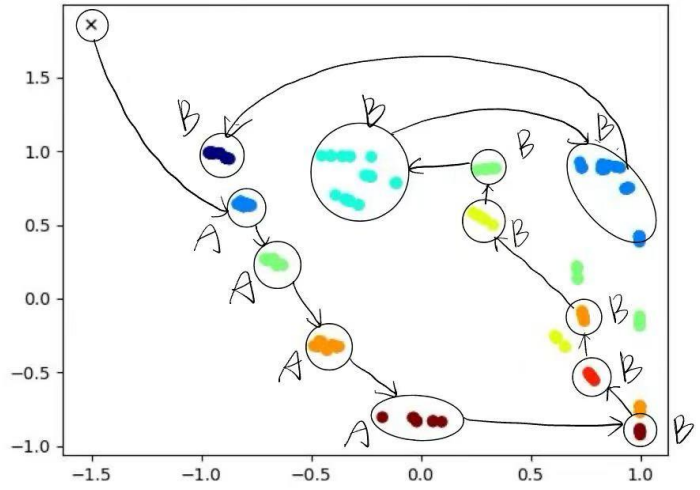


Part 3: Hidden Unit Dynamics for Recurrent Networks

```
-----
symbol= AAAABBBBBBBBAAAABBBBBBBBAABBBBAABBBBAAAABBBBBBBBA
hidden activations and output probabilities:
A [-0.81  0.63] [0.81 0.19]
A [-0.66  0.23] [0.66 0.34]
A [-0.43 -0.35] [0.36 0.64]
A [-0.02 -0.83] [0.04 0.96]
B [ 1.   -0.91] [0. 1.]
B [ 0.78 -0.53] [0. 1.]
B [ 0.74 -0.12] [0. 1.]
B [0.29 0.54] [0. 1.]
B [0.29 0.88] [0. 1.]
B [-0.32  0.96] [0.09 0.91]
B [0.83 0.91] [0. 1.]
B [-0.95  0.99] [0.9 0.1]
A [-0.79  0.64] [0.78 0.22]
A [-0.7   0.27] [0.71 0.29]
A [-0.39 -0.32] [0.3 0.7]
A [-0.16 -0.8 ] [0.11 0.89]
B [ 1.   -0.92] [0. 1.]
B [ 0.79 -0.55] [0. 1.]
B [ 0.74 -0.15] [0. 1.]
```

```
B [0.32 0.5 ] [0. 1.]
B [0.26 0.87] [0. 1.]
B [-0.24  0.96] [0.05 0.95]
B [0.74 0.92] [0. 1.]
B [-0.93  0.99] [0.88 0.12]
A [-0.81  0.65] [0.81 0.19]
A [-0.68  0.27] [0.69 0.31]
B [ 1.   -0.13] [0. 1.]
B [-0.37  0.69] [0.15 0.85]
B [0.95 0.75] [0. 1.]
B [-0.96  0.99] [0.9 0.1]
A [-0.78  0.63] [0.78 0.22]
A [-0.7   0.26] [0.72 0.28]
B [ 1.   -0.16] [0. 1.]
B [-0.32  0.66] [0.11 0.89]
B [0.94 0.75] [0. 1.]
B [-0.95  0.98] [0.9 0.1]
A [-0.78  0.63] [0.78 0.22]
A [-0.7   0.26] [0.72 0.28]
A [-0.38 -0.33] [0.29 0.71]
A [-0.17 -0.81] [0.11 0.89]
B [ 1.   -0.92] [0. 1.]
B [ 0.79 -0.55] [0. 1.]
B [ 0.74 -0.15] [0. 1.]
B [0.33 0.5 ] [0. 1.]
B [0.26 0.87] [0. 1.]
```

1.



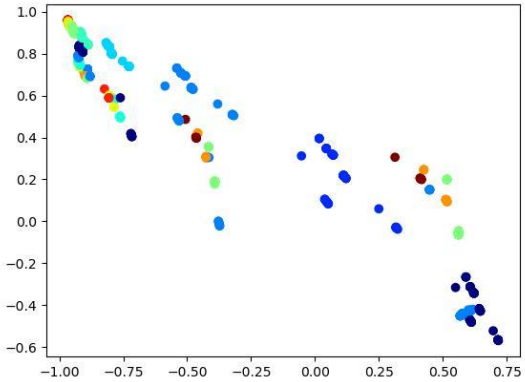
2. FSM:

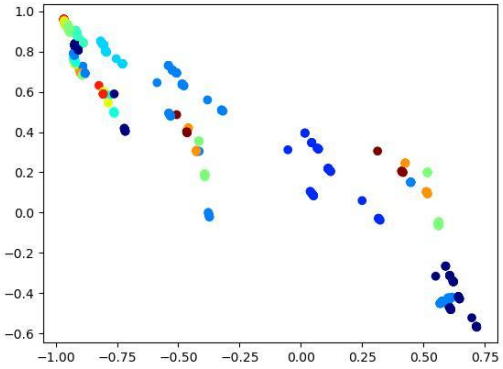
X → A → A → A → A → B → B → B → B → B → B → B → B → B

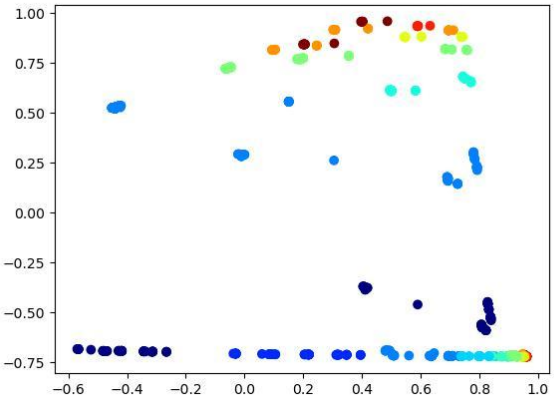
3.The SRN model uses its 2-dimensional hidden state to track how many A's have been seen. While reading A's, the hidden state moves gradually in one direction. As training progresses, we can observe that during the A phase, the predicted probability of A starts to decrease, this means the probability of B increases. When the B appears, the value of n becomes known, and the model can predict 2n B. The SRN is encoding the phase and the count of A's into the location of the hidden state in 2D space. This spatial encoding allows

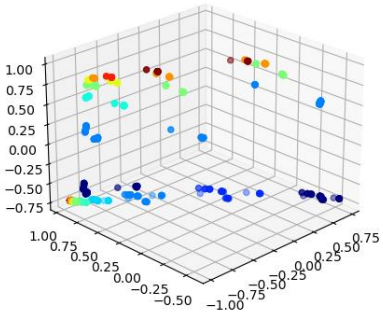
the network to know when the B sequence should end and prepares it for the next input.

4.anb2nc3n_lstm3_01/02/12.jpg & 3D.png





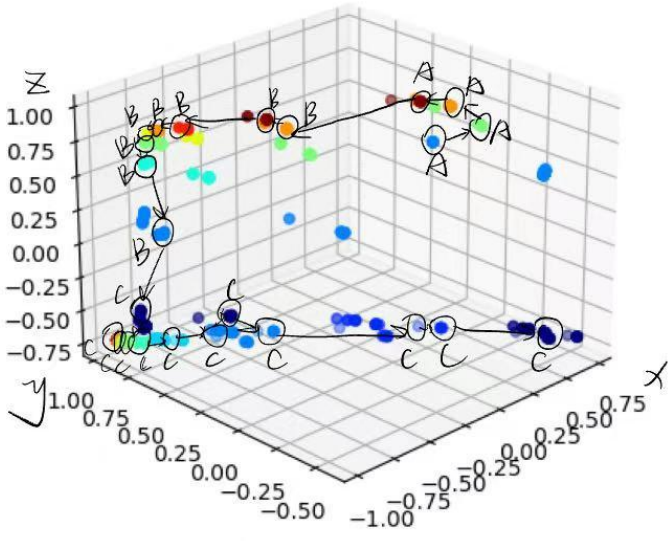




5.

```
symbol= ABBCCCAABBBBCCCCCCAAABBBBBBCCCCCCCCCAABBBBCCCCCCABBCCCA
hidden activations and output probabilities:
A [0.45 0.15 0.56] [0.54 0.45 0.01]
B [-0.41  0.3   0.26] [0.03 0.9  0.07]
B [-0.76  0.59 -0.46] [0.   0.04 0.96]
C [-0.59  0.65 -0.7 ] [0.   0.01 0.99]
C [-0.05  0.31 -0.71] [0.04 0.01 0.95]
C [ 0.55 -0.32 -0.7 ] [0.82 0.   0.17]
A [ 0.6  -0.42  0.53] [0.8 0.2 0. ]
A [ 0.56 -0.05  0.73] [0.58 0.42 0.  ]
B [-0.39  0.19  0.77] [0.01 0.99 0.  ]
B [-0.76  0.5   0.61] [0.   0.99 0.01]
B [-0.88  0.69  0.16] [0.   0.76 0.23]
B [-0.91  0.81 -0.58] [0.   0.01 0.99]
C [-0.95  0.91 -0.71] [0. 0. 1.]
C [-0.91  0.87 -0.72] [0. 0. 1.]
C [-0.79  0.8  -0.72] [0. 0. 1.]
C [-0.48  0.63 -0.72] [0.   0.   0.99]
C [ 0.12  0.21 -0.71] [0.12 0.01 0.88]
C [ 0.65 -0.43 -0.69] [0.91 0.   0.09]
A [ 0.59 -0.44  0.53] [0.79 0.21 0.  ]
A [ 0.56 -0.06  0.73] [0.58 0.42 0.  ]
A [0.51 0.1  0.82] [0.42 0.57 0.  ]
B [-0.43  0.31  0.92] [0. 1. 0.]
B [-0.79  0.55  0.88] [0. 1. 0.]
B [-0.89  0.68  0.82] [0. 1. 0.]
```

```
B [-0.92  0.74  0.68] [0.   0.99 0.01]
B [-0.92  0.78  0.3 ] [0.   0.88 0.12]
B [-0.92  0.83 -0.46] [0.   0.03 0.97]
C [-0.96  0.95 -0.71] [0. 0. 1.]
C [-0.96  0.95 -0.72] [0. 0. 1.]
C [-0.96  0.94 -0.72] [0. 0. 1.]
C [-0.95  0.93 -0.72] [0. 0. 1.]
C [-0.91  0.89 -0.72] [0. 0. 1.]
C [-0.8   0.83 -0.72] [0. 0. 1.]
C [-0.51  0.69 -0.72] [0.   0.   0.99]
C [ 0.07  0.32 -0.71] [0.07 0.01 0.92]
C [ 0.62 -0.34 -0.7 ] [0.87 0.   0.12]
A [ 0.6  -0.43  0.53] [0.8 0.2 0. ]
A [ 0.56 -0.05  0.73] [0.58 0.42 0.  ]
B [-0.39  0.19  0.77] [0.01 0.99 0.  ]
B [-0.76  0.5   0.61] [0.   0.99 0.01]
B [-0.88  0.69  0.17] [0.   0.78 0.22]
B [-0.91  0.81 -0.57] [0.   0.01 0.99]
C [-0.95  0.91 -0.71] [0. 0. 1.]
C [-0.91  0.88 -0.72] [0. 0. 1.]
C [-0.8   0.8  -0.72] [0. 0. 1.]
C [-0.48  0.64 -0.72] [0.   0.   0.99]
C [ 0.11  0.22 -0.71] [0.11 0.01 0.88]
C [ 0.64 -0.42 -0.69] [0.91 0.   0.09]
A [ 0.59 -0.44  0.53] [0.79 0.21 0.  ]
```



LSTM overcomes the gradient vanishing problem of SRN by using memory cells and gate mechanisms. While reading A's, it stores the count in the long-term cell state c_t. As it moves into B and C phases, the gates (forget, input, output) control how much memory to keep and use. This allows LSTM to know which phase it is in, and how many B’s and C’s to output. Unlike SRN, which only uses hidden state positions, LSTM separates memory and control, making it better at handling long dependencies language prediction task.