Project Report

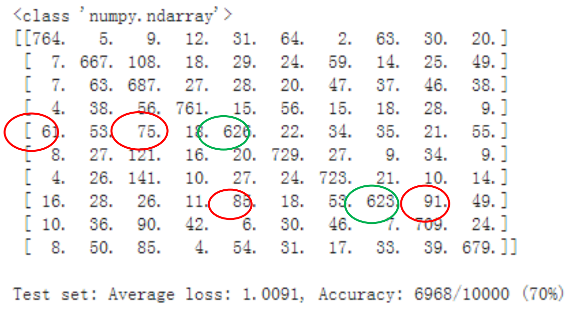
Z5325417 Guanzhu Hou

The versions of numpy, sklearn and pytorch are all the same as these in course outline, and the version of python is 3.7.12. This project is completed on the collab platform.

Part 1

1. Implement a model NetLin which computes a linear function of the pixels in the image, followed by log softmax.

   The final accuracy and confusion matrix are shown in picture 1-1.



```
<class 'numpy.ndarray'>
[[764.   5.   9.  12.  31.  64.   2.  63.  30.  20.]
 [  7. 667. 108.  18.  29.  24.  59.  14.  25.  49.]
 [  7.  63. 687.  27.  28.  20.  47.  37.  46.  38.]
 [  4.  38.  56. 761.  15.  56.  15.  18.  28.   9.]
 [ 61.  53.  75.  18. 626.  22.  34.  35.  21.  55.]
 [  8.  27. 121.  16.  20. 729.  27.   9.  34.   9.]
 [  4.  26. 141.  10.  27.  24. 723.  21.  10.  14.]
 [ 16.  28.  26.  11.  85.  18.  53. 623.  91.  49.]
 [ 10.  36.  90.  42.   6.  30.  46.   7. 709.  24.]
 [  8.  50.  85.   4.  54.  31.  17.  33.  39. 679.]]

Test set: Average loss: 1.0091, Accuracy: 6968/10000 (70%)
```

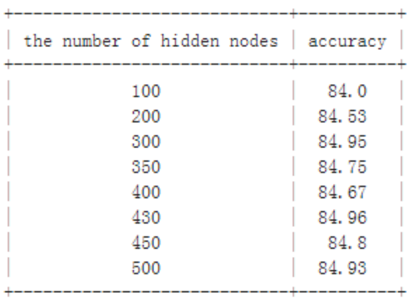
Picture 1-1 final accuracy and confusion matrix of NetLin

2. 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. 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.

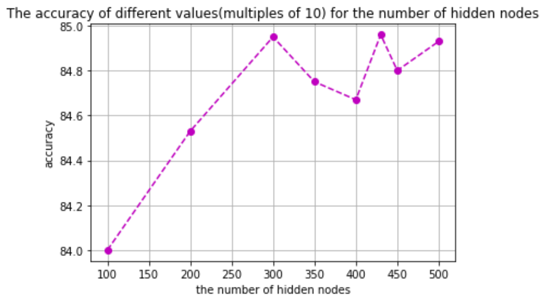
```
self.full_1 = nn.Linear(in_features = 28*28, out_features = 400)
self.full_2 = nn.Linear(in_features = 400, out_features = 10)
```

Picture 1-2 the structure of NetFull layers

   The results of testing different values for the number of hidden nodes are shown in picture 1-2.

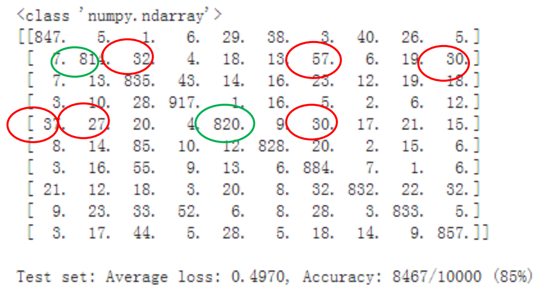


the number of hidden nodes	accuracy
100	84.0
200	84.53
300	84.95
350	84.75
400	84.67
430	84.96
450	84.8
500	84.93



Picture 1-3 The accuracy of different values for the number of hidden nodes

Pick 400 hidden nodes which achieve accuracy 84.47% and the confusion matrix is shown in picture 1-3.



```
<class 'numpy.ndarray'>
[[847.   5.   1.   6.  29.  38.   3.  40.  26.   5.]
 [  7. 814.  32.   4.  18.  13.  57.   6.  19.  30.]
 [  7.  13. 835.  43.  14.  16.  23.  12.  19.  18.]
 [  3.  10.  28. 917.   1.  16.   5.   2.   6.  12.]
 [ 37.  27.  20.   4. 820.   9.  30.  17.  21.  15.]
 [  8.  14.  85.  10.  12. 828.  20.   2.  15.   6.]
 [  3.  16.  55.   9.  13.   6. 884.   7.   1.   6.]
 [ 21.  12.  18.   3.  20.   8.  32. 832.  22.  32.]
 [  9.  23.  33.  52.   6.   8.  28.   3. 833.   5.]
 [  3.  17.  44.   5.  28.   5.  18.  14.   9. 857.]]

Test set: Average loss: 0.4970, Accuracy: 8467/10000 (85%)
```

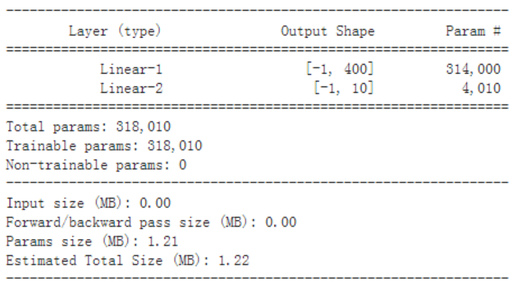
Picture 1-4 The accuracy and confusion matrix of 400 hidden nodes NetFull

$$(28 * 28 + 1) \times 400 \ = \ 314{,}000$$

$$(400 + 1) \times 10 = 4{,}010$$

$$314{,}000 + 4010 \ = \ 318{,}010$$

With the checking of summary function, the total free parameters is 318,010.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 400]         314,000
            Linear-2                   [-1, 10]           4,010
================================================================
Total params: 318,010
Trainable params: 318,010
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 1.21
Estimated Total Size (MB): 1.22
----------------------------------------------------------------
```

Picture 1-5 The summary of 400 hidden nodes NetFull

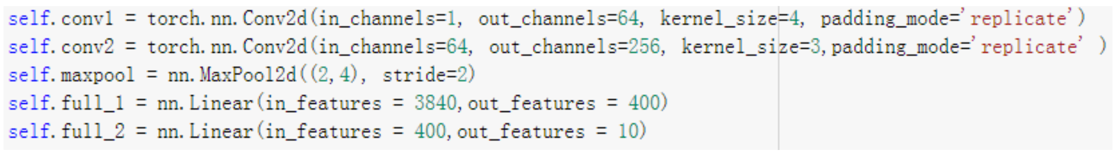
Additional checking: why do we have different results using same number of hidden nodes?

Start to use the GPU of colab this time, but find there is a slight difference compared with not using GPU. For example, for 400 hidden nodes, the result with GPU is 85.00%, and the result without GPU is 84.93%. In order to check whether it is because of the usage of GPU, redo the code with GPU twice, the result is 84.80% and 84.47%. What'more, is definitely not because of 'drop out', as we already use 'model. eval()'.

Guess there could be some randomness in the main code, but not find it yet.

Truly appreciate Alan's help in resolving this problem. He illustrates that it is because of the different chosen initial weight values when running the code.

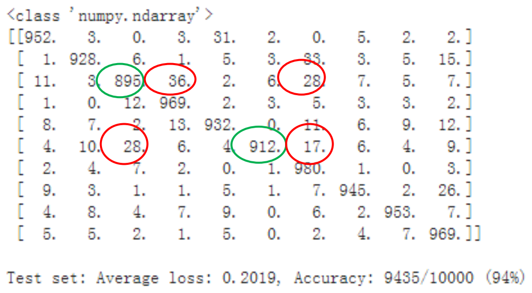
3. Implement a convolutional network called NetConv, with two convolutional layers plus one fully connected layer, all using relu activation function, followed by the output layer, using log softmax. Copy the final accuracy and confusion matrix into your report, and include a calculation of the total number of independent parameters in the network.



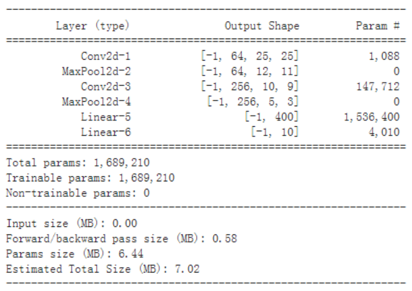
```
self.conv1 = torch.nn.Conv2d(in_channels=1, out_channels=64, kernel_size=4, padding_mode='replicate')
self.conv2 = torch.nn.Conv2d(in_channels=64, out_channels=256, kernel_size=3,padding_mode='replicate' )
self.maxpool = nn.MaxPool2d((2,4), stride=2)
self.full_1 = nn.Linear(in_features = 3840,out_features = 400)
self.full_2 = nn.Linear(in_features = 400,out_features = 10)
```

Picture 1-6 the structure of NetConv layers

The accuracy of NetConv is 94.35% and the number of independent parameters is 1,689,210.



```
<class 'numpy.ndarray'>
[[952.   3.   0.   3.  31.   2.   0.   5.   2.   2.]
 [  1. 928.   6.   1.   5.   3.  33.   3.   5.  15.]
 [ 11.   3. 895.  36.   2.   6.  28.   7.   5.   7.]
 [  1.   0.  12. 969.   2.   3.   5.   3.   3.   2.]
 [  8.   7.   2.  13. 932.   0.  11.   6.   9.  12.]
 [  4.  10.  28.   6.   4. 912.  17.   6.   4.   9.]
 [  2.   4.   7.   2.   0.   1. 980.   1.   0.   3.]
 [  9.   3.   1.   1.   5.   1.   7. 945.   2.  26.]
 [  4.   8.   4.   7.   9.   0.   6.   2. 953.   7.]
 [  5.   5.   2.   1.   5.   0.   2.   4.   7. 969.]]

Test set: Average loss: 0.2019, Accuracy: 9435/10000 (94%)
```



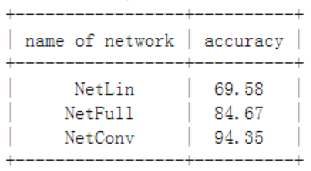
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 25, 25]           1,088
         MaxPool2d-2           [-1, 64, 12, 11]               0
            Conv2d-3           [-1, 256, 10, 9]         147,712
         MaxPool2d-4            [-1, 256, 5, 3]               0
            Linear-5                  [-1, 400]       1,536,400
            Linear-6                   [-1, 10]           4,010
================================================================
Total params: 1,689,210
Trainable params: 1,689,210
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.58
Params size (MB): 6.44
Estimated Total Size (MB): 7.02
----------------------------------------------------------------
```

Picture 1-5 The confusion matrix and summary of NetConv

4. Briefly discuss the following points:

a) The relative accuracy of the three models

NetConv has highest accuracy, followed by NetFull and finally NetLin.



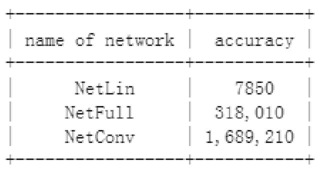
```
+-----------------+----------+
| name of network | accuracy |
+-----------------+----------+
|      NetLin     |  69.58   |
|     NetFull     |  84.67   |
|     NetConv     |  94.35   |
+-----------------+----------+
```

Picture 1-6 the relative accuracy of the three models

Between NetLin and NetFull, the main advances in the accuracy chiefly from the increasing layer, where the 2-layer network NetFull can compute not linearly separable function like the Exclusive OR or XOR.

Between NetConv and NetFull, the convolutional network NetConv has the advantages of local connection, weight sharing, and max pooling, versus fully-connected network NetFull. What's more, NetConv also replaces Tanh with ReLU, which could avoid gradients vanish or explode

b) The number of independent parameters in each of the three models



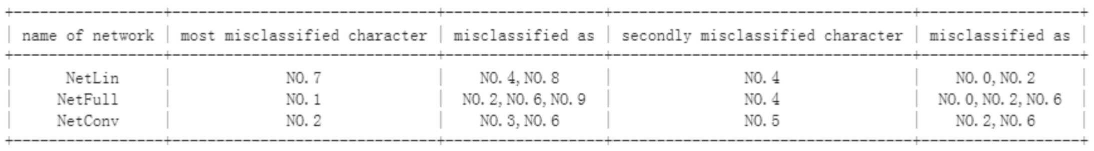
```
+-----------------+------------+
| name of network |  accuracy  |
+-----------------+------------+
|      NetLin     |    7850    |
|     NetFull     |  318,010   |
|     NetConv     | 1,689,210  |
+-----------------+------------+
```

Picture 1-7 the number of independent parameters of the three models

NetConv has the most independent parameters, followed by NetFull and finally NetLin, while the training speed of the models is opposite. The complexity of the network causes the rise of several independent parameters.

However, it can be seen from the discussion of hidden node numbers of NetFull in picture 1-2, that with the rise of independent parameters, there is a fluctuation of accuracy instead of an upward trend. So if the independent parameters are increased blindly, it does not fundamentally increase the accuracy.

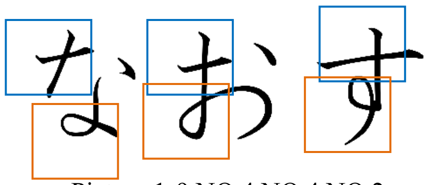
c) The confusion matrix for each model: which characters are most likely to be mistaken for which other characters, and why?



```
+-----------------+---------------------------+--------------------+-------------------------------+-------------------+
| name of network | most misclassified character | misclassified as | secondly misclassified character | misclassified as |
+-----------------+---------------------------+--------------------+-------------------------------+-------------------+
|      NetLin     |           NO.7            |     NO.4, NO.8     |              NO.4             |     NO.0, NO.2    |
|     NetFull     |           NO.1            | NO.2, NO.6, NO.9   |              NO.4             | NO.0, NO.2, NO.6  |
|     NetConv     |           NO.2            |     NO.3, NO.6     |              NO.5             |     NO.2, NO.6    |
+-----------------+---------------------------+--------------------+-------------------------------+-------------------+
```

Picture 1-8 misclassified characters of the three models

For NetLin and NetFull, they are both easy to misclassified NO.4 as NO.0 and NO.2. From picture 1-9, the similarity of character structure in blue squares and orange squares is easy to be found. In addition, the training set and test set are 28*28 pixel cursive and handwritten pictures in ancient Japanese, which will also increase the difficulty of classification.



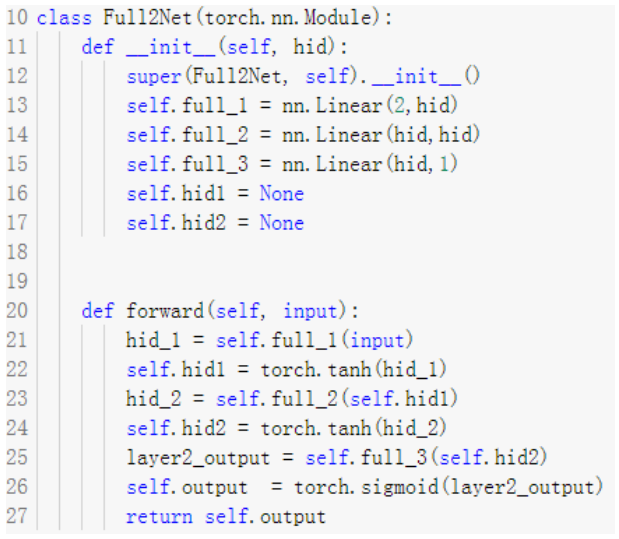
Picture 1-9 NO.4 NO.4 NO.2

Compared with NetLin and NetFull, NetConv works better on NO.4, and the other most likely misclassified character. Though the similarity of character structure also troubles NetConv like picture 1-10 is shown, it is indicated that NetConv has stronger expression ability and can deal with more complex classification problems.

Picture 1-10 NO.2 NO.3 NO.6

Part 2

1) Provide code for a Pytorch Module called Full2Net

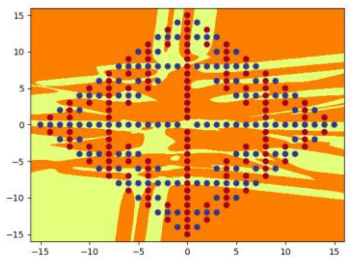


```
class Full2Net(torch.nn.Module):
    def __init__(self, hid):
        super(Full2Net, self).__init__()
        self.full_1 = nn.Linear(2,hid)
        self.full_2 = nn.Linear(hid,hid)
        self.full_3 = nn.Linear(hid,1)
        self.hid1 = None
        self.hid2 = None


    def forward(self, input):
        hid_1 = self.full_1(input)
        self.hid1 = torch.tanh(hid_1)
        hid_2 = self.full_2(self.hid1)
        self.hid2 = torch.tanh(hid_2)
        layer2_output = self.full_3(self.hid2)
        self.output  = torch.sigmoid(layer2_output)
        return self.output
```

Picture 2-1 The code for Full2Net

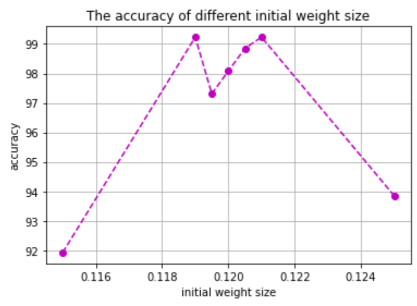
2) Try to determine a number of hidden nodes close to the mininum required for the network to be trained successfully.

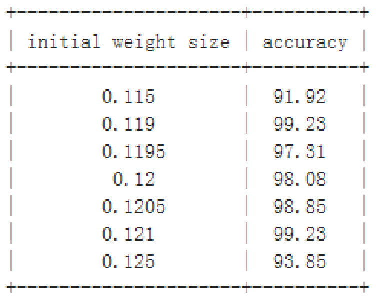
Firstly, choose 15 number hidden nodes and find it is closely to 100% accuracy.



Picture 2-2 15 with initial weight size 0.125 accuracy 93.58%

Change the initial weight size to find a convergence picture, however, it seems that 15 is not a good choice.



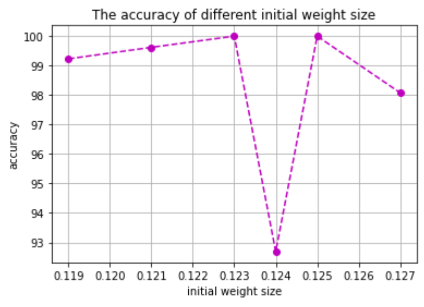


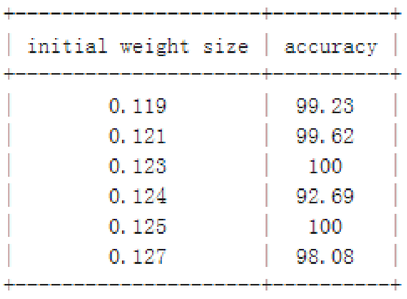
initial weight size	accuracy
0.115	91.92
0.119	99.23
0.1195	97.31
0.12	98.08
0.1205	98.85
0.121	99.23
0.125	93.85

Picture 2-3 The accuracy of different initial weight size when hidden nodes is 15

Then we have a look at 14 and 16. 14 with 0.119 initial weight size has the accuracy of 92.69, while 16 with 0.125 initial weight size can separate blue and red dots and gets a 100%-accuracy picture at epoch 200,000.

Adjust the initial weight size since 0.119 with 16 hidden nodes to see whether we can have a 100%-accuracy convergence picture, which will stop the loop before 200,000 epochs. From the picture 2-5, we can find it reached 100% accuracy when initial weight size equal to 0.123 or 0.125.

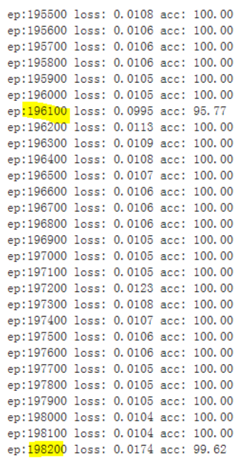




initial weight size	accuracy
0.119	99.23
0.121	99.62
0.123	100
0.124	92.69
0.125	100
0.127	98.08

Picture 2-4 The accuracy of different initial weight size when hidden nodes is 16

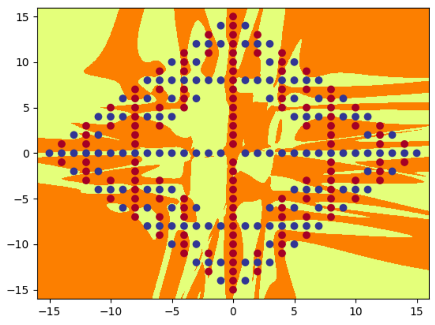
But, as we can see in the picture 2-6, it always met with some no-100%-accuracy disturbs like what happened at epoch 196100 and continue to loop with 100% accuracy. As it is very close to the standard, which is a count less than 2,000, take the initial weight size as 0.125.

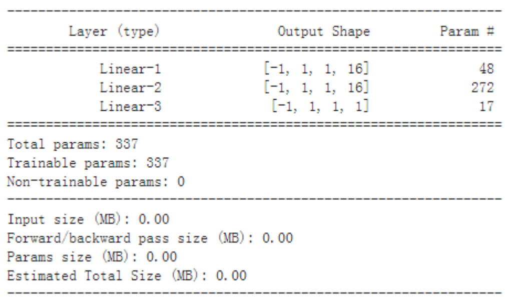


```
ep:195500 loss: 0.0108 acc: 100.00
ep:195600 loss: 0.0106 acc: 100.00
ep:195700 loss: 0.0106 acc: 100.00
ep:195800 loss: 0.0106 acc: 100.00
ep:195900 loss: 0.0105 acc: 100.00
ep:196000 loss: 0.0105 acc: 100.00
ep:196100 loss: 0.0995 acc: 95.77
ep:196200 loss: 0.0113 acc: 100.00
ep:196300 loss: 0.0109 acc: 100.00
ep:196400 loss: 0.0108 acc: 100.00
ep:196500 loss: 0.0107 acc: 100.00
ep:196600 loss: 0.0106 acc: 100.00
ep:196700 loss: 0.0106 acc: 100.00
ep:196800 loss: 0.0106 acc: 100.00
ep:196900 loss: 0.0105 acc: 100.00
ep:197000 loss: 0.0105 acc: 100.00
ep:197100 loss: 0.0105 acc: 100.00
ep:197200 loss: 0.0123 acc: 100.00
ep:197300 loss: 0.0108 acc: 100.00
ep:197400 loss: 0.0107 acc: 100.00
ep:197500 loss: 0.0106 acc: 100.00
ep:197600 loss: 0.0106 acc: 100.00
ep:197700 loss: 0.0105 acc: 100.00
ep:197800 loss: 0.0105 acc: 100.00
ep:197900 loss: 0.0105 acc: 100.00
ep:198000 loss: 0.0104 acc: 100.00
ep:198100 loss: 0.0104 acc: 100.00
ep:198200 loss: 0.0174 acc: 99.62
```

Picture 2-5 The disturbs of accuracy with initial weight size 0.125

So finally, the number of hidden nodes is 16 and the initial weight size is 0.125. The out_full2_16.png is shown as below and the independent parameters are 337.

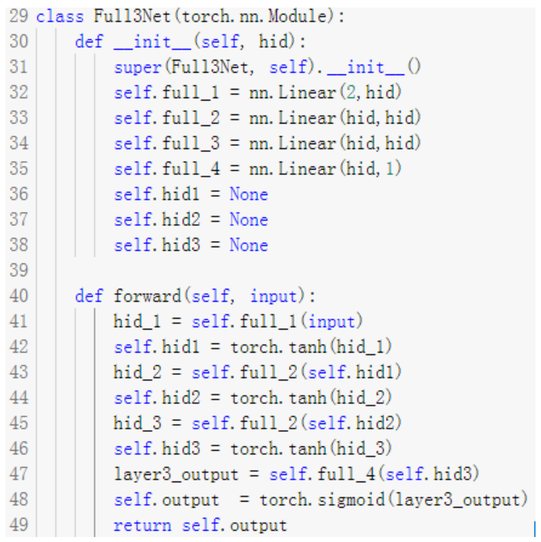




```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1             [-1, 1, 1, 16]              48
            Linear-2             [-1, 1, 1, 16]             272
            Linear-3              [-1, 1, 1, 1]              17
================================================================
Total params: 337
Trainable params: 337
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

Picture 2-6 the out_full2_16.png and the summary of 16 with 0.125 initial weight size

3) Provide code for a Pytorch Module called Full3Net



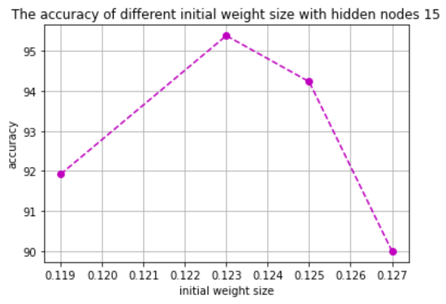
```python
class Full3Net(torch.nn.Module):
    def __init__(self, hid):
        super(Full3Net, self).__init__()
        self.full_1 = nn.Linear(2,hid)
        self.full_2 = nn.Linear(hid,hid)
        self.full_3 = nn.Linear(hid,hid)
        self.full_4 = nn.Linear(hid,1)
        self.hid1 = None
        self.hid2 = None
        self.hid3 = None

    def forward(self, input):
        hid_1 = self.full_1(input)
        self.hid1 = torch.tanh(hid_1)
        hid_2 = self.full_2(self.hid1)
        self.hid2 = torch.tanh(hid_2)
        hid_3 = self.full_2(self.hid2)
        self.hid3 = torch.tanh(hid_3)
        layer3_output = self.full_4(self.hid3)
        self.output  = torch.sigmoid(layer3_output)
        return self.output
```

Picture 2-6 The code for Full3Net

4) Try to determine a number of hidden nodes close to the mininum required for the network to be trained successfully.

Starting from 15 hidden nodes, find the 100% accuracy picture by adjusting the initial weight size.

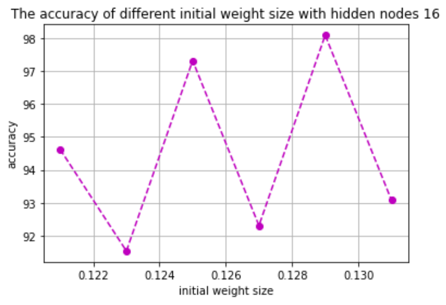
The result of 15 hidden nodes is shown in picture 2-7.



```
+----------------------+----------+
| initial weight size  | accuracy |
+----------------------+----------+
|        0.119         |  91.92   |
|        0.123         |  95.38   |
|        0.125         |  94.23   |
|        0.127         |   90.0   |
+----------------------+----------+
```

Picture 2-7 The accuracy of different initial weight size when hidden nodes is 15

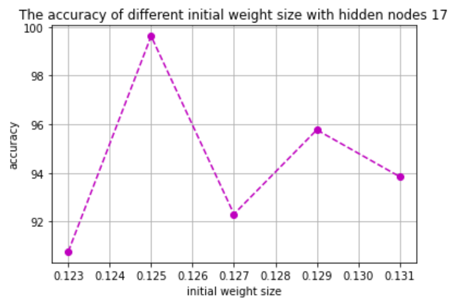
The result of 16 hidden nodes is shown in picture 2-8.



```
+----------------------+----------+
| initial weight size  | accuracy |
+----------------------+----------+
|        0.121         |  94.62   |
|        0.123         |  91.54   |
|        0.125         |  97.31   |
|        0.127         |  92.31   |
|        0.129         |  98.08   |
|        0.131         |  93.08   |
+----------------------+----------+
```

Picture 2-8 The accuracy of different initial weight size when hidden nodes is 16

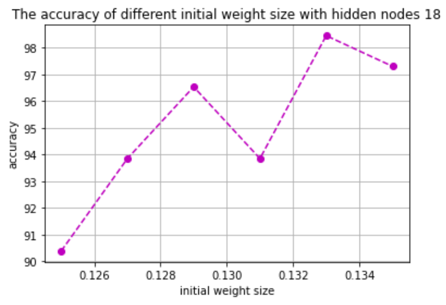
The result of 17 hidden nodes is shown in picture 2-9.



```
+----------------------+----------+
| initial weight size  | accuracy |
+----------------------+----------+
|        0.123         |  90.77   |
|        0.125         |  99.62   |
|        0.127         |  92.31   |
|        0.129         |  95.77   |
|        0.131         |  93.85   |
+----------------------+----------+
```

Picture 2-9 The accuracy of different initial weight size when hidden nodes is 17

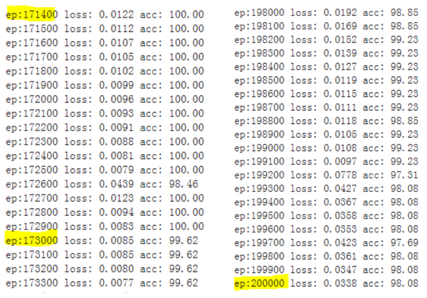
The result of 18 hidden nodes is shown in picture 2-10.



```
+----------------------+----------+
| initial weight size  | accuracy |
+----------------------+----------+
|        0.125         |  90.38   |
|        0.127         |  93.85   |
|        0.129         |  96.54   |
|        0.131         |  93.85   |
|        0.133         |  98.46   |
|        0.135         |  97.31   |
+----------------------+----------+
```

Picture 2-10 The accuracy of different initial weight size when hidden nodes is 18

However, as what is shown in picture 2-11, when the initial weight size is 0.131, it will get a 100% accuracy result at epoch 171,400, but then decline since epoch 173,000 and end up with 98.08% at epoch 200,000.

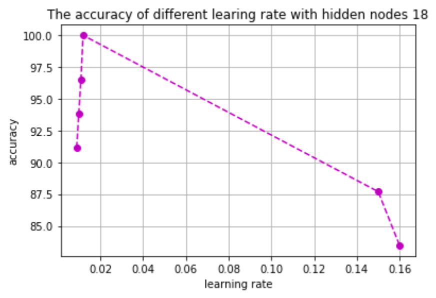


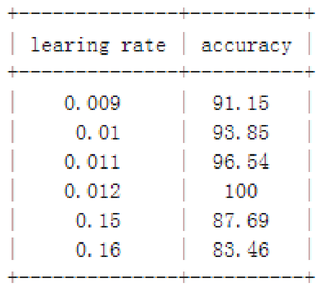
```
ep:171400 loss: 0.0122 acc: 100.00      ep:198000 loss: 0.0192 acc: 98.85
ep:171500 loss: 0.0112 acc: 100.00      ep:198100 loss: 0.0169 acc: 98.85
ep:171600 loss: 0.0107 acc: 100.00      ep:198200 loss: 0.0152 acc: 99.23
ep:171700 loss: 0.0105 acc: 100.00      ep:198300 loss: 0.0139 acc: 99.23
ep:171800 loss: 0.0102 acc: 100.00      ep:198400 loss: 0.0127 acc: 99.23
ep:171900 loss: 0.0099 acc: 100.00      ep:198500 loss: 0.0119 acc: 99.23
ep:172000 loss: 0.0096 acc: 100.00      ep:198600 loss: 0.0115 acc: 99.23
ep:172100 loss: 0.0093 acc: 100.00      ep:198700 loss: 0.0111 acc: 99.23
ep:172200 loss: 0.0091 acc: 100.00      ep:198800 loss: 0.0118 acc: 98.85
ep:172300 loss: 0.0088 acc: 100.00      ep:198900 loss: 0.0105 acc: 99.23
ep:172400 loss: 0.0081 acc: 100.00      ep:199000 loss: 0.0108 acc: 99.23
ep:172500 loss: 0.0079 acc: 100.00      ep:199100 loss: 0.0097 acc: 99.23
ep:172600 loss: 0.0439 acc: 98.46       ep:199200 loss: 0.0778 acc: 97.31
ep:172700 loss: 0.0123 acc: 100.00      ep:199300 loss: 0.0427 acc: 98.08
ep:172800 loss: 0.0094 acc: 100.00      ep:199400 loss: 0.0367 acc: 98.08
ep:172900 loss: 0.0083 acc: 100.00      ep:199500 loss: 0.0358 acc: 98.08
ep:173000 loss: 0.0085 acc: 99.62       ep:199600 loss: 0.0353 acc: 98.08
ep:173100 loss: 0.0085 acc: 99.62       ep:199700 loss: 0.0423 acc: 97.69
ep:173200 loss: 0.0080 acc: 99.62       ep:199800 loss: 0.0361 acc: 98.08
ep:173300 loss: 0.0077 acc: 99.62       ep:199900 loss: 0.0347 acc: 98.08
                                        ep:200000 loss: 0.0338 acc: 98.08
```

Picture 2-11 acc is 100% at epoch 171400 but ends with 98.08%

As what we have learnt in 3b Hidden Unit Dynamics, this could be solved by New Activation Functions, Weight Initialization or Residential Network obey their number of layers.

In this turn we adjust the learning rate in picture 2-12, and get a convergence result with learning rate 0.012.

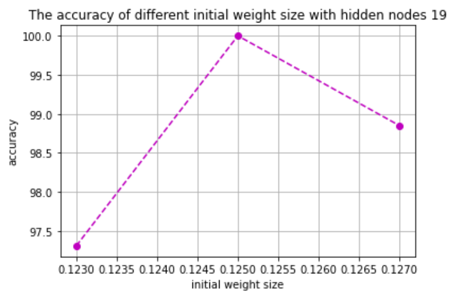


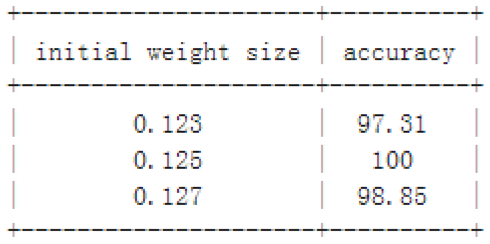


learing rate	accuracy
0.009	91.15
0.01	93.85
0.011	96.54
0.012	100
0.15	87.69
0.16	83.46

Picture 2-12 The accuracy of different learning rate when initial weight size is 0.131

We also test 19 hidden nodes and the result is shown in picture 2-13.

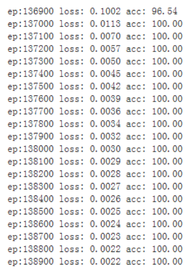




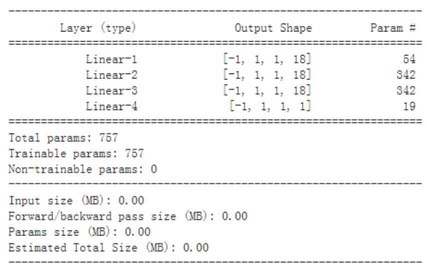
initial weight size	accuracy
0.123	97.31
0.125	100
0.127	98.85

Picture 2-13 The accuracy of different initial weight size when hidden nodes is 19

So, the number of hidden nodes close to the minimum required for the network to be trained successfully is 18, with initial weight size 0.131, learning rate 0.012 and free parameters 757. And it will converge on epoch 138900 as what is in picture 2-14.



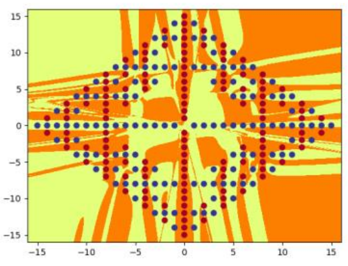
```
ep:136900 loss: 0.1002 acc: 96.54
ep:137000 loss: 0.0113 acc: 100.00
ep:137100 loss: 0.0070 acc: 100.00
ep:137200 loss: 0.0057 acc: 100.00
ep:137300 loss: 0.0050 acc: 100.00
ep:137400 loss: 0.0045 acc: 100.00
ep:137500 loss: 0.0042 acc: 100.00
ep:137600 loss: 0.0039 acc: 100.00
ep:137700 loss: 0.0036 acc: 100.00
ep:137800 loss: 0.0034 acc: 100.00
ep:137900 loss: 0.0032 acc: 100.00
ep:138000 loss: 0.0030 acc: 100.00
ep:138100 loss: 0.0029 acc: 100.00
ep:138200 loss: 0.0028 acc: 100.00
ep:138300 loss: 0.0027 acc: 100.00
ep:138400 loss: 0.0026 acc: 100.00
ep:138500 loss: 0.0025 acc: 100.00
ep:138600 loss: 0.0024 acc: 100.00
ep:138700 loss: 0.0023 acc: 100.00
ep:138800 loss: 0.0022 acc: 100.00
ep:138900 loss: 0.0022 acc: 100.00
```



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1             [-1, 1, 1, 18]              54
            Linear-2             [-1, 1, 1, 18]             342
            Linear-3             [-1, 1, 1, 18]             342
            Linear-4              [-1, 1, 1, 1]              19
================================================================
Total params: 757
Trainable params: 757
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.00
Estimated Total Size (MB): 0.00
----------------------------------------------------------------
```

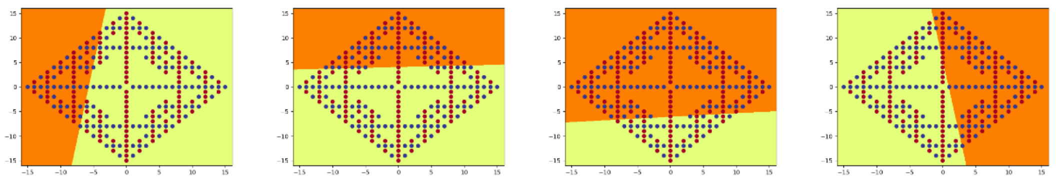
Picture 2-14 The final converged result of Full3Net with 18 hidden nodes

The plot of the output and the plots of all the hidden units in all three layers are as bellow from picture 2-15 to 2-28

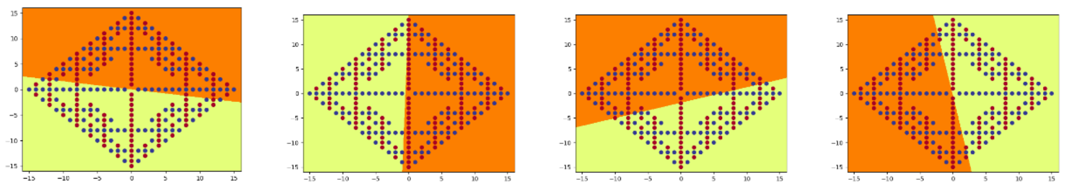


Picture 2-15 the out_full3_18.png

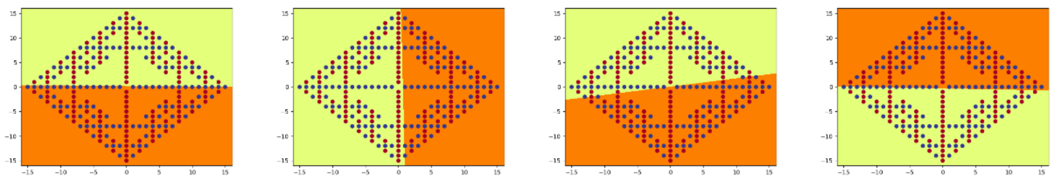
Layer 1:



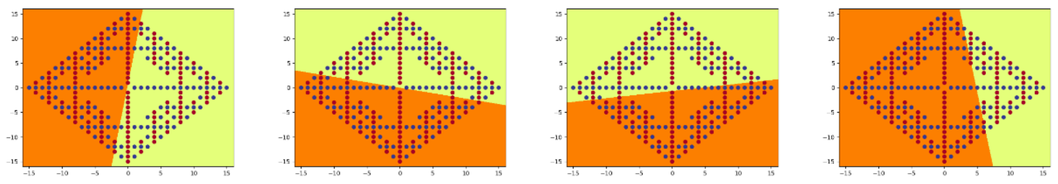
Picture 2-16 layer1 18_1_0 to 18_1_3



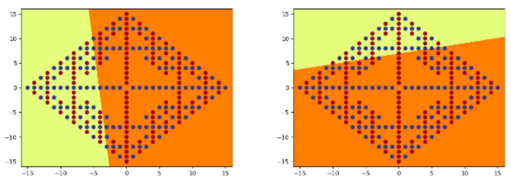
Picture 2-17 layer1 18_1_4 to 18_1_7



Picture 2-17 layer1 18_1_8 to 18_1_11

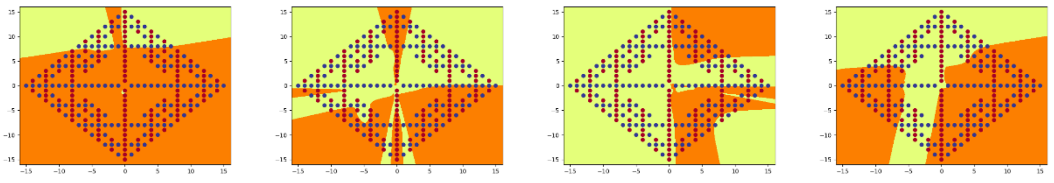


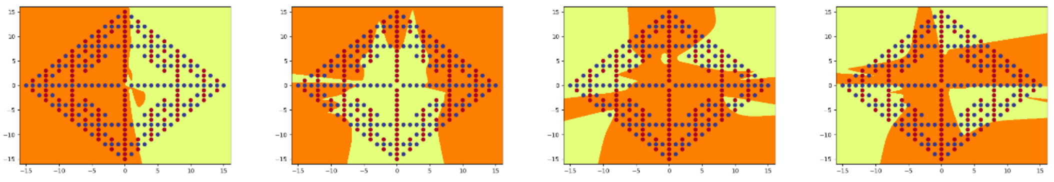
Picture 2-18 layer1 18_1_12 to 18_1_15

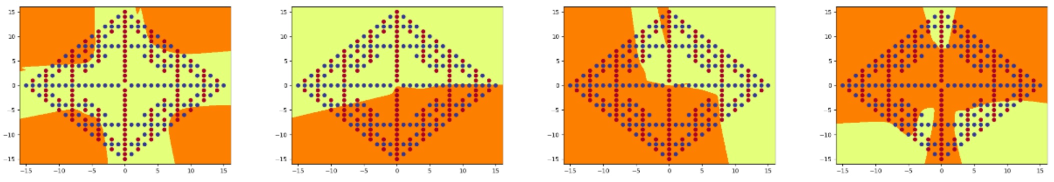


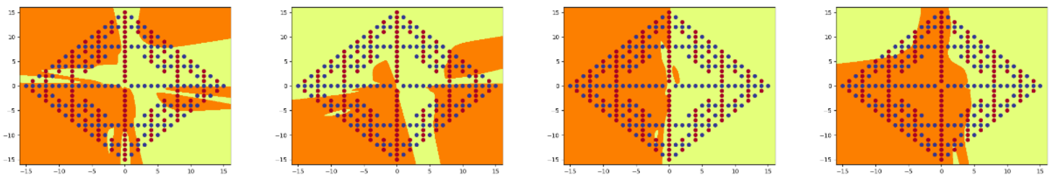
Picture 2-19 layer1 18_1_16 to 18_1_17

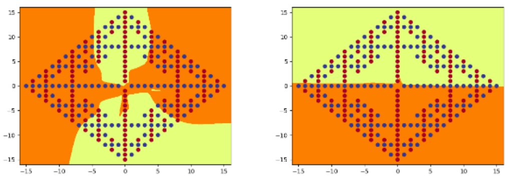
Layer 2:


Picture 2-20 layer1 18_2_0 to 18_2_3

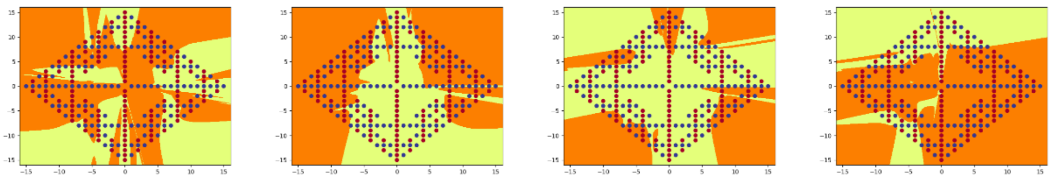

Picture 2-21 layer1 18_2_4 to 18_2_7

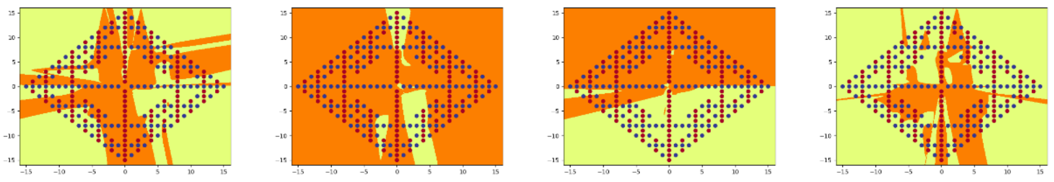

Picture 2-22 layer1 18_2_8 to 18_2_11

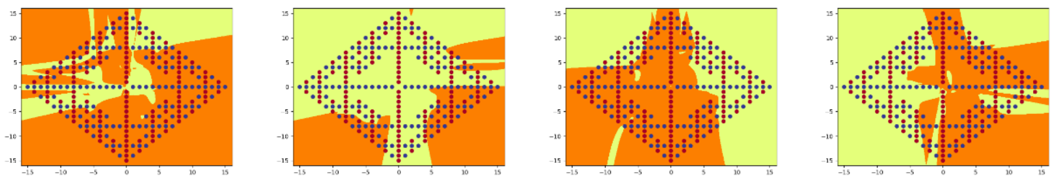

Picture 2-22 layer1 18_2_12 to 18_2_15


Picture 2-23 layer1 18_2_16 to 18_2_17

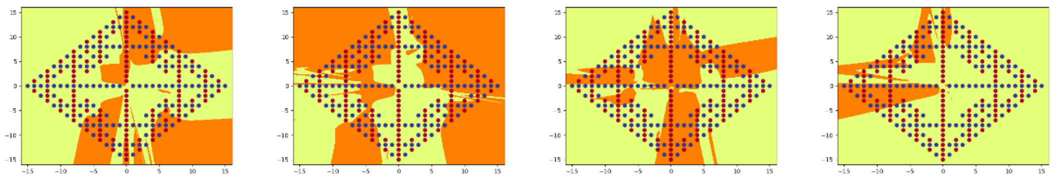
Layer 3:


Picture 2-24 layer1 18_3_0 to 18_3_3

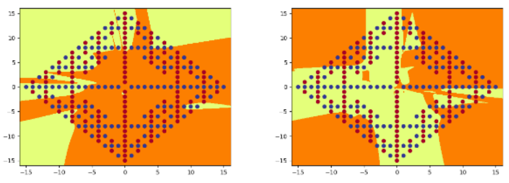

Picture 2-25 layer1 18_3_4 to 18_3_7



Picture 2-26 layer1 18_3_8 to 18_3_11

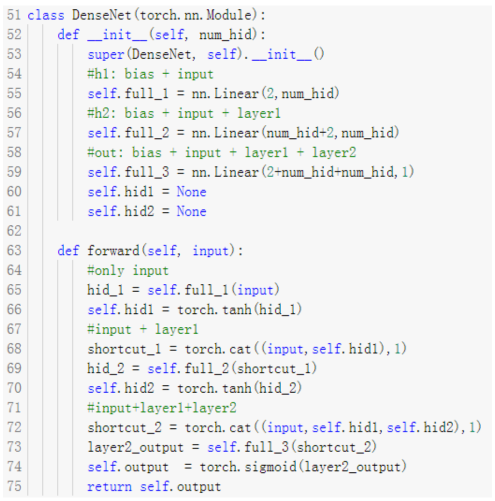


Picture 2-27 layer1 18_3_12 to 18_3_15



Picture 2-28 layer1 18_3_16 to 18_3_17

5) Provide code for a Pytorch Module called DenseNet which implements a 3-layer densely connected neural network.

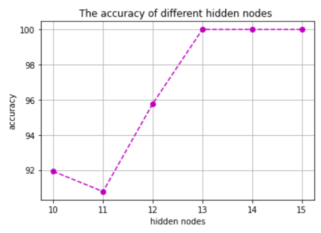


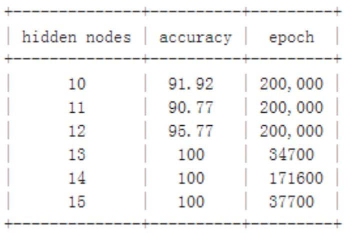
```python
class DenseNet(torch.nn.Module):
    def __init__(self, num_hid):
        super(DenseNet, self).__init__()
        #h1: bias + input
        self.full_1 = nn.Linear(2,num_hid)
        #h2: bias + input + layer1
        self.full_2 = nn.Linear(num_hid+2,num_hid)
        #out: bias + input + layer1 + layer2
        self.full_3 = nn.Linear(2+num_hid+num_hid,1)
        self.hid1 = None
        self.hid2 = None

    def forward(self, input):
        #only input
        hid_1 = self.full_1(input)
        self.hid1 = torch.tanh(hid_1)
        #input + layer1
        shortcut_1 = torch.cat((input,self.hid1),1)
        hid_2 = self.full_2(shortcut_1)
        self.hid2 = torch.tanh(hid_2)
        #input+layer1+layer2
        shortcut_2 = torch.cat((input,self.hid1,self.hid2),1)
        layer2_output = self.full_3(shortcut_2)
        self.output  = torch.sigmoid(layer2_output)
        return self.output
```

Picture 2-29 The code for DenseNet

6) Try to determine a number of hidden nodes close to the mininum required for the network to be trained successfully.

As what is given in picture 2-30, the number of minimum hidden nodes is 13 with the original 0.125 initial weight size and 0.01 learning rate.

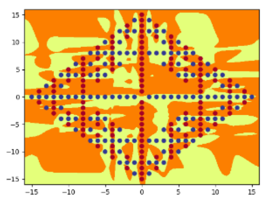
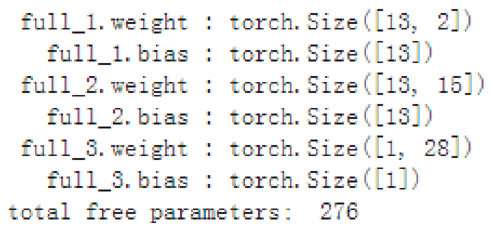




```
+--------------+-----------+----------+
| hidden nodes |  accuracy |  epoch   |
+--------------+-----------+----------+
|      10      |   91.92   | 200,000  |
|      11      |   90.77   | 200,000  |
|      12      |   95.77   | 200,000  |
|      13      |    100    |  34700   |
|      14      |    100    |  171600  |
|      15      |    100    |  37700   |
+--------------+-----------+----------+
```

Picture 2-30 The accuracy of different hidden nodes

The output image of 13 hidden nodes and its structure are shown in picture 2-31. The free parameters are 276.

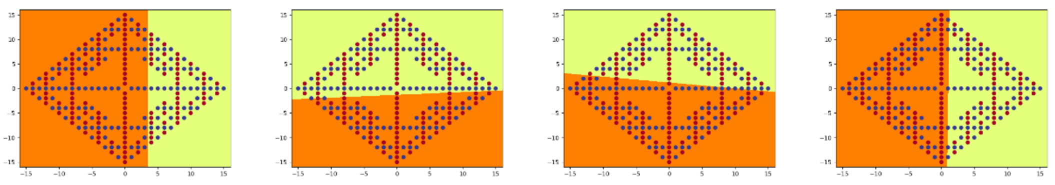



```
full_1.weight : torch.Size([13, 2])
  full_1.bias : torch.Size([13])
full_2.weight : torch.Size([13, 15])
  full_2.bias : torch.Size([13])
full_3.weight : torch.Size([1, 28])
  full_3.bias : torch.Size([1])
total free parameters:  276
```

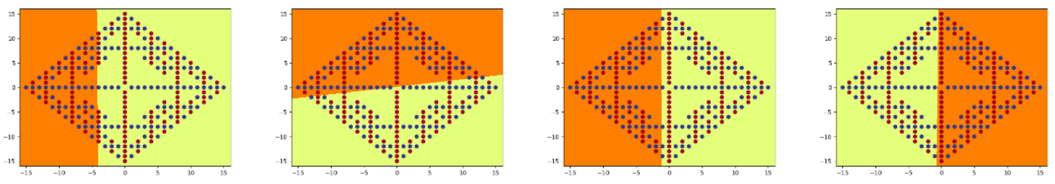
Picture 2-31 out_dense_13 and the structure of 13 hidden nodes

The plots of all the hidden units in all two layers are as bellow from picture 2-31 to 2-39.

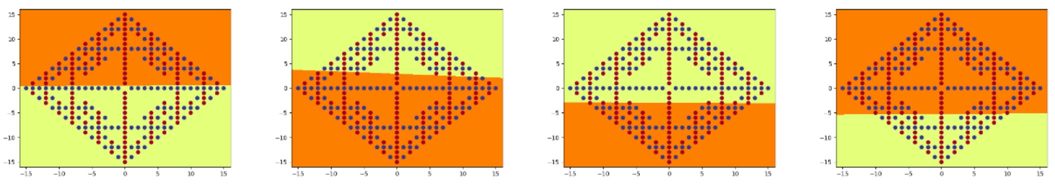
Layer 1：



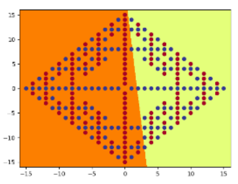
Picture 2-32 hidden_dense_13_1_0 to 13_1_3



Picture 2-33 hidden_dense_13_1_4 to 13_1_7

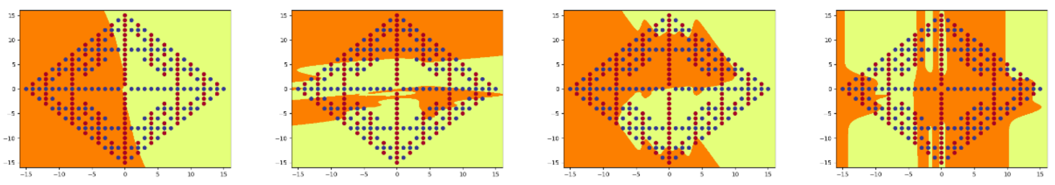


Picture 2-34 hidden_dense_13_1_8 to 13_1_11

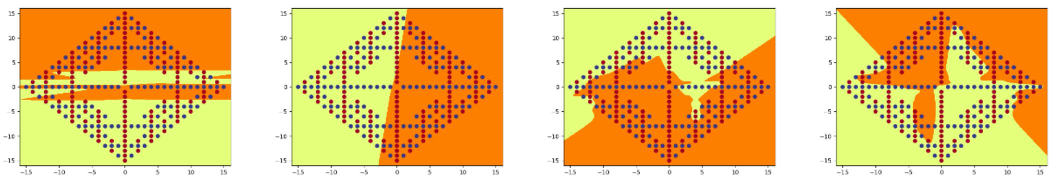


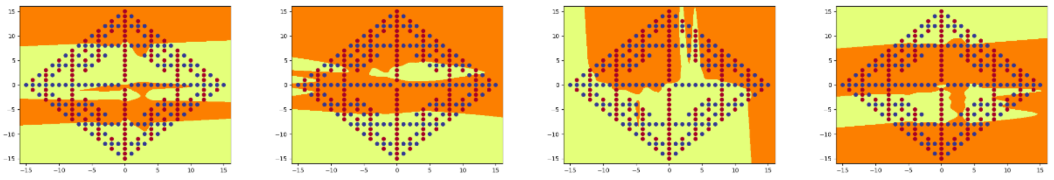
Picture 2-35 hidden_dense_13_1_12

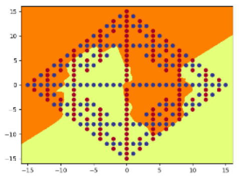
Layer 2:



Picture 2-36 hidden_dense_13_2_0 to 13_2_3


Picture 2-37 hidden_dense_13_2_4 to 13_2_7

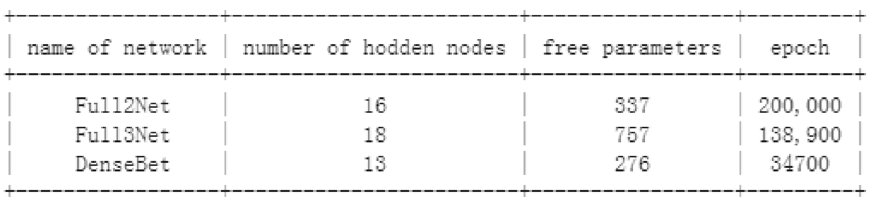

Picture 2-38 hidden_dense_13_2_8 to 13_2_11


Picture 2-39 hidden_dense_13_2_12

7) Briefly discuss the following points:

a) the total number of independent parameters in each of the three networks (using the number of hidden nodes determined by your experiments) and the approximate number of epochs required to train each type of network

As what is shown in picture 2-40, after adding a hidden layer to Full2Net, Full3Net has more free parameters and can converge more quickly. But after shifting the full connect network to a dense network, DenseNet has fewer free parameters and is the most quickly convergence network.


```
+-----------------+-----------------------+-----------------+---------+
| name of network | number of hodden nodes| free parameters |  epoch  |
+-----------------+-----------------------+-----------------+---------+
|    Full2Net     |          16           |       337       | 200,000 |
|    Full3Net     |          18           |       757       | 138,900 |
|    DenseBet     |          13           |       276       |  34700  |
+-----------------+-----------------------+-----------------+---------+
```

Picture 2-40 free parameters and number of epochs of each network

b) a qualitative description of the functions computed by the different layers of Full3Net and DenseNet.

Full3Net: There are 4 layers in Full3Net, which are 3 hidden layers and 1 output layer. Hidden layer 1 learns features that are linearly separable and the other 2 hidden layers complete more complex nonlinear classes, while the output layer learns the target function.

DenseNet: There are 3 layers in DenseNet, which are 2 hidden layers and 1 output layer. Layer 1 also separates pictures linearly and the other layer for irregular classification, while the output layer learns the target function. This network is also a dense network that has a shortcut connection between each layer. With this connection, each layer has direct access to the gradients from the loss function and the original input signal, leading to implicit deep supervision.[1]

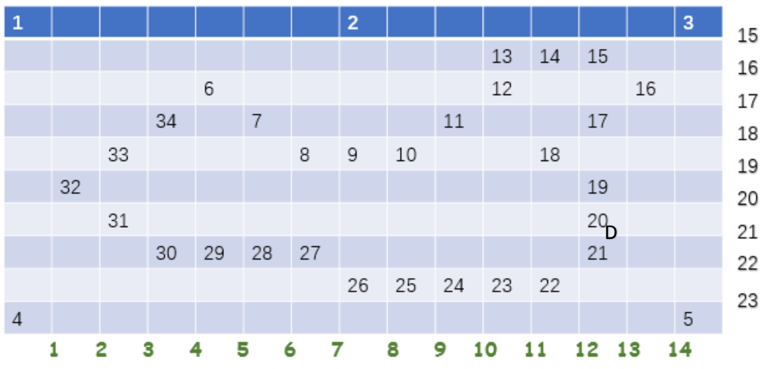
c) the qualitative difference, if any, between the overall function (i.e. output as a function of input) computed by the three networks.

After adding a hidden layer, Full3Net can learn features quicker than Full2Net, but it is also more sensitive to gradient vanish.

After introducing direct connections from any layer to all subsequent layers, DenseNet can avoid gradient vanish and converging 100% accuracy quickest than the other 2 networks. DenseNet also uses the features more efficiently than the other 2 networks.

Part3:

Firstly, draw the position of each dot and label the sequence of each line. Then replace each dot with a number, which is their row number in the matrix. There are 34 columns in the matrix, so there are 23 lines in picture 3-1. There are 23 columns in the matrix, so there are 23 lines in picture 3-1.



1							2							3	
										13	14	15			15
				6						12			16		16
			34		7				11			17			17
		33				8	9	10			18				18
	32											19			19
		31										20D			20
			30	29	28	27						21			21
							26	25	24	23	22				22
4														5	23
1	2	3	4	5	6	7	8	9	10	11	12	13	14		

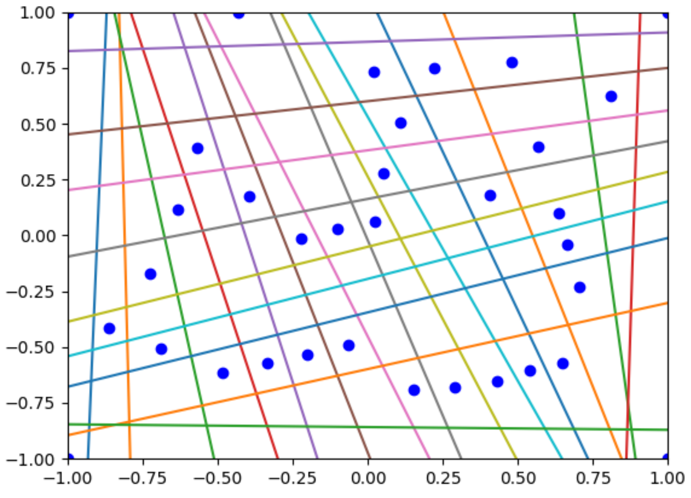
Picture 3-1 Dots in the matrix

The encoder network follows 2 rules:

For a verticle line, its left dot is 0 while its right dot is 1;

For a horizontal line, its upward dot is 0 while its downward dot is 1.

So the output is like :



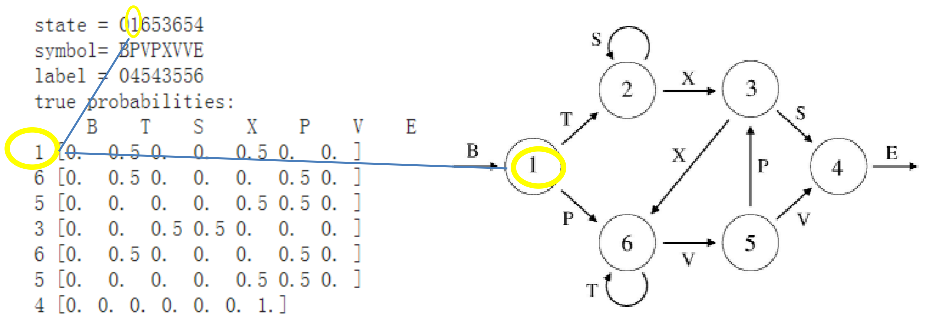
Picture 3-2 Final image of ch34

[1] Gao Huang, Gao Huang, Gao Huang. Densely Connected Convolutional Networks. In CVPR, 2017

Part 4: Hidden Unit Dynamics for Recurrent Networks

1) Based on this colormap, annotate your figure (either electronically, or with a pen on a printout) by drawing a circle around the cluster of points corresponding to each state in the state machine, and drawing arrows between the states, with each arrow labeled with its corresponding symbol. Include the annotated figure in your report.

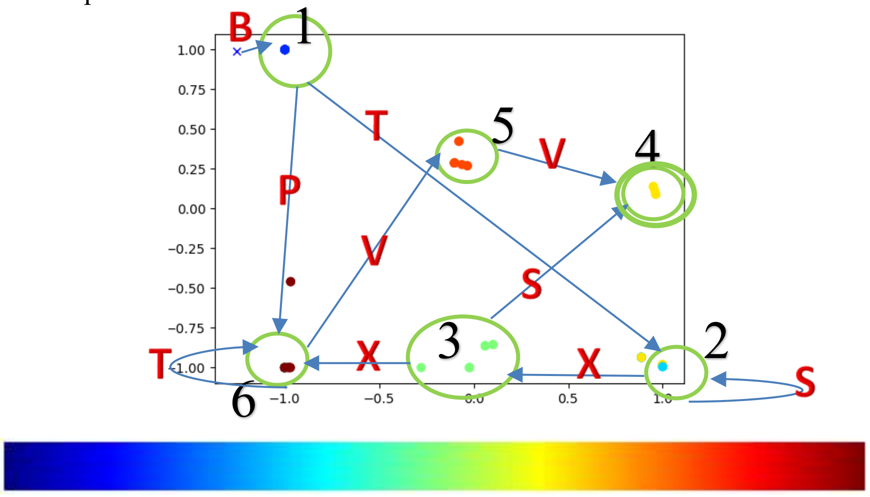
The output in console fits the Reber Grammar model, that the state shows the number of state in Reber Grammar model and the symbol shows what string of characters will you get by running along the state. 'BTSXPVE' corresponds to 0123456 one by one in the label. For the true probabilities, the probability of the next state is the same, and each branch is 0.5.



```
state = 01653654
symbol= BPVPXVVE
label = 04543556
true probabilities:
   B    T    S    X    P    V    E
1 [0.   0.5  0.   0    0.5  0.   0. ]
6 [0.   0.5  0.   0.   0.   0.5  0. ]
5 [0.   0.   0.   0.   0.5  0.5  0. ]
3 [0.   0.   0.5  0.5  0.   0.   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.]
```

Picture 4-1 The output in console

The result picture is as below.

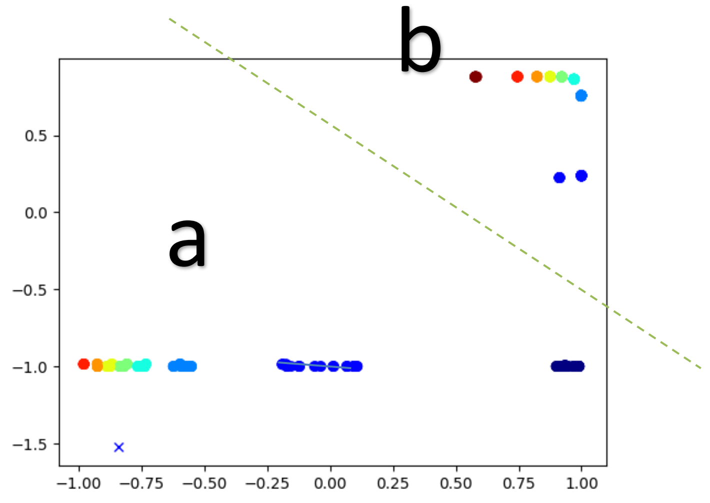


Picture 4-2 the cluster of points and its color map 'jet'

From week 5-a, we know that the cross indicates the initial state while the dots indicate all the hidden states. Each cluster of hidden unit states is converted into a single state, the transitions between clusters are converted into arrows, and the initial state is indicated by an arrow with no origin.

Determine the one-to-one correspondence between the state and cluster of points: follow the order of the color map.

2) Train an SRN on the anbn language prediction task.



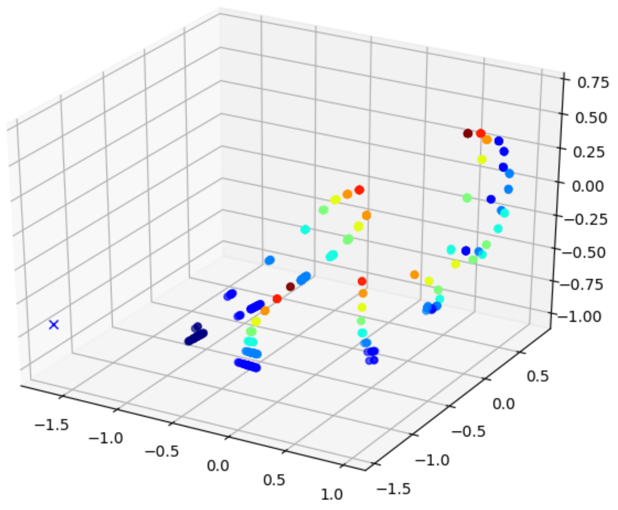
Picture 4-3 the resulting figure of an SRN on the anbn language prediction

3) Briefly explain how the anbn prediction task is achieved by the network, based on the figure you generated in Question 2. Specifically, you should describe how the hidden unit activations change as the string is processed, and how it is able to correctly predict the last B in each sequence as well as the following A.

The network does not implement a Finite State Machine but instead makes use of two fixed points in activation space — one attracting, the other repelling (Wiles & Elman, 1995). It uses evolution rather than backpropagation to train the weight, which is more stable.

As picture 4-3, when it begins to process ‘a’, it moves back and forth and forms some points in the area (a). And when it seizes the first ‘b’, it crosses the dotted line boundary and jumps to area (b). When it seizes more ‘b’s, it begins to oscillate outwards from the first ‘b’. After processing the same number ‘b’s as ‘a’s, it’s enough for the last ‘b’ to cross the boundary.

4) Train an SRN on the anbncn language prediction task



Picture 4-4 the resulting figure of an SRN on the anbncn language prediction

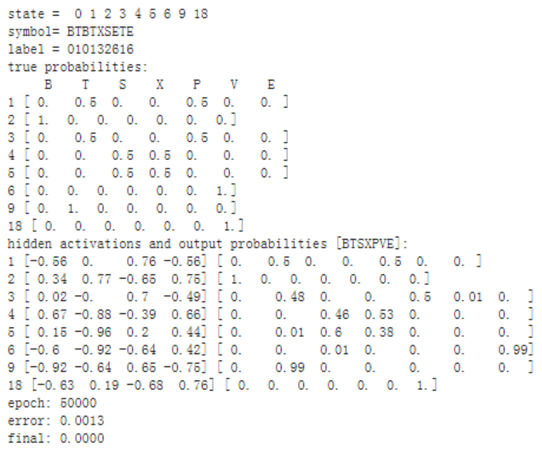
5) Briefly explain how the anbncn prediction task is achieved by the network, based on

the figure you generated in Question 4. Specifically, you should describe how the hidden unit activations change as the string is processed, and how it is able to correctly predict the last B in each sequence as well as all of the C's and the following A.

The network counts down in one direction while simultaneously counting up in another direction, thus producing a star-shaped pattern.

Similar to the anbn task, when processing ‘a’s, it also oscillates from the fixed point. And when processing ‘b’s, in one direction, it’s attracting the fixed point and in the other direction, it’s repelling the fixed point. And this produces a nice star shape. The same thing for the ‘c’s.

6) This question is intended to be more challenging. Train an LSTM network to predict the Embedded Reber Grammar



```
state =  0 1 2 3 4 5 6 9 18
symbol= BTBTXSETE
label = 010132616
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. ]
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 [-0.56  0.    0.76 -0.56] [ 0.   0.5  0.   0.   0.5  0.   0. ]
2 [ 0.34  0.77 -0.65  0.75] [ 1.  0.  0.  0.  0.  0.  0.]
3 [ 0.02 -0.    0.7  -0.49] [ 0.    0.48  0.    0.    0.5   0.01  0.  ]
4 [ 0.67 -0.88 -0.39  0.66] [ 0.    0.    0.46  0.53  0.    0.    0.  ]
5 [ 0.15 -0.96  0.2   0.44] [ 0.    0.01  0.6   0.38  0.    0.    0.  ]
6 [-0.6  -0.92 -0.64  0.42] [ 0.    0.    0.01  0.    0.    0.    0.99]
9 [-0.92 -0.64  0.65 -0.75] [ 0.    0.99  0.    0.    0.    0.    0.  ]
18 [-0.63  0.19 -0.68  0.76] [ 0.  0.  0.  0.  0.  0.  1.]
epoch: 50000
error: 0.0013
final: 0.0000
```

Picture 4-5 the resulting figure of an LSTM on the Embedded Reber Grammar

The LSTM can get a convergence result quickly with a small error. This LSTM has a context layer where we will compute a sequence of hidden units and cell states follow the formulation:

$$\begin{aligned}
\mathbf{f}_t &= \sigma\left(U_f \mathbf{h}_{t-1} + W_f \mathbf{x}_t + \mathbf{b}_f\right) \\
\mathbf{i}_t &= \sigma\left(U_i \mathbf{h}_{t-1} + W_i \mathbf{x}_t + \mathbf{b}_i\right) \\
\mathbf{o}_t &= \sigma\left(U_o \mathbf{h}_{t-1} + W_o \mathbf{x}_t + \mathbf{b}_o\right)
\end{aligned}$$

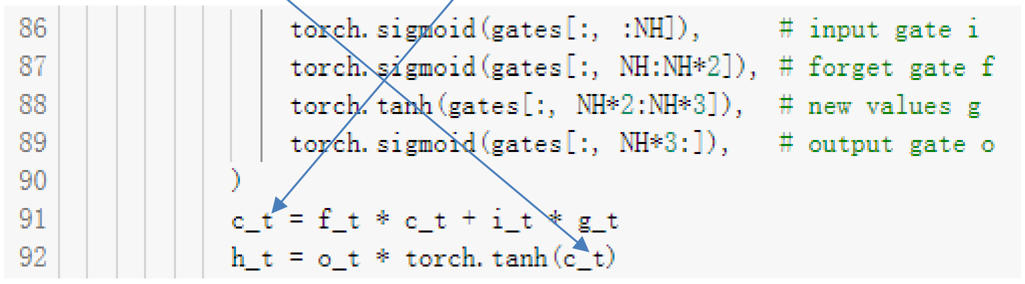
$$\mathbf{g}_t = \tanh\left(U_g \mathbf{h}_{t-1} + W_g \mathbf{x}_t + \mathbf{b}_g\right)$$

Picture 4-6 The calculation of each timestep(week5-b)

And for this Embedded Reber Grammar task, when a context unit is assigned to retain the knowledge of an arbitrary character, and this knowledge is preserved by appropriately high and low values for the forget gate and the input and output gate, respectively. (week5-b)

Beginning with the forget gate (f), it will control whether we forget or remember the previous cell state. Then the input gate (i) combining with the update value (g) will control what new character to be stored in the cell. After confirming these parameters, we update the old cell state to a new one. Finally, the output gate (o) will be determined to generate the next hidden layer value and we may get this value from a filtered cell state.

From picture 4-7, we may determine the ranges of the gates in the code, that besides update gate (g), gate (i), (f) and (o) are between 0 and 1.



```
        torch.sigmoid(gates[:, :NH]),      # input gate i
        torch.sigmoid(gates[:, NH:NH*2]), # forget gate f
        torch.tanh(gates[:, NH*2:NH*3]),  # new values g
        torch.sigmoid(gates[:, NH*3:]),   # output gate o
    )
    c_t = f_t * c_t + i_t * g_t
    h_t = o_t * torch.tanh(c_t)
```

Picture 4-7 The ranges of the gates.

In this way we will finally learn the Embedded Reber Grammar model.