25T2 COMP9444 Assignment 1

Part 1: Japanese Character Recognition

1.

Because the size of images is 28*28, so we need to create 784 inputs. Then ten number, so we have 10 outputs. Before that, we need to use “Flatten” to flatten the image. Finally, use SoftMax algorithm to get the final 10 outputs.

```
class NetLin(nn.Module):
    # linear function followed by log_softmax
    def __init__(self):
        super(NetLin, self).__init__()
        # INSERT CODE HERE
        self.flat = nn.Flatten()
        self.linear1 = nn.Linear(784,10)
        # input 784, output 10



    def forward(self, x):
        x = self.flat(x)
        x = self.linear1(x)

        x = F.log_softmax(x, dim=1)
        # add softmax algrithm



        return x # CHANGE CODE HERE
```

Here is the final accuracy and confusion matrix:

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.822064
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.635364
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.596002
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.600355
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.322878
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.520681
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.660333
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.605380
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.353989
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.670523
<class 'numpy.ndarray'>
[[769.   5.   9.  12.  29.  65.   2.  62.  29.  18.]
 [  7. 668. 107.  17.  28.  24.  58.  13.  27.  51.]
 [  8.  59. 690.  27.  25.  20.  46.  38.  47.  40.]
 [  3.  34.  61. 757.  16.  57.  16.  18.  26.  12.]
 [ 59.  54.  81.  20. 623.  23.  32.  32.  20.  56.]
 [  8.  29. 124.  16.  19. 726.  27.   9.  33.   9.]
 [  5.  24. 149.   9.  25.  24. 720.  20.  10.  14.]
 [ 17.  30.  27.  12.  80.  15.  52. 627.  91.  49.]
 [ 10.  38.  97.  39.   7.  31.  43.   7. 703.  25.]
 [  8.  51.  91.   3.  52.  31.  19.  32.  41. 672.]]

Test set: Average loss: 1.0105, Accuracy: 6955/10000 (70%)

/content/drive/MyDri
```

Emm, 70%.

2.

In this part we need to create linear2, and after linear1, we should let the data go through tanh. Let's start from 10 hidden nodes:

```
class NetFull(nn.Module):
    # two fully connected tanh layers followed by log softmax
    def __init__(self):
        super(NetFull, self).__init__()
        # INSERT CODE HERE
        self.flat = nn.Flatten()
        self.linear1 = nn.Linear(784, 10)
        self.linear2 = nn.Linear(10, 10)

    def forward(self, x):
        x = self.flat(x)
        x = self.linear1(x)
        x = torch.tanh(x)
        # F.tanh: old version, not recommended.
        x = self.linear2(x)
        x = F.log_softmax(x, dim=1)
```

Number of independent parameters: (784*10)+10 + (10*10)+10 = 7960

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.757184
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.532084
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.634130
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.615233
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.289636
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.515823
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.585829
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.676338
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.392871
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.429112
<class 'numpy.ndarray'>
[[730.   5.   1.   6.  33.  43.   2.  65.  62.  53.]
 [  3. 620. 108.  12.  38.  14. 101.   4.  40.  60.]
 [  5.  38. 681.  61.  39.  10.  54.  15.  45.  52.]
 [  4.  16.  60. 830.  15.  14.  10.  16.  20.  15.]
 [ 65.  41.  58.  19. 637.  13.  41.  28.  30.  68.]
 [  9.  15. 151.  30.  14. 709.  32.   3.  27.  10.]
 [  3.  19. 150.  20.  38.   7. 728.  12.  10.  13.]
 [ 13.   9.  35.  11. 142.  11.  69. 528.  96.  86.]
 [ 10.  21.  67.  49.  10.  27.  57.   3. 744.  12.]
 [ 18.  30. 100.   3.  72.  13.  27.  33.  36. 668.]]

Test set: Average loss: 1.0244, Accuracy: 6875/10000 (69%)

/content/drive/MyDrive/a1#
```

Not so good

20:

Number of independent parameters: (784*20+20)+(20*10+10) = 15910

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.564381
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.478359
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.431270
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.400858
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.234999
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.355383
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.450135
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.473295
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.274705
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.410157
<class 'numpy.ndarray'>
[[777.   3.   2.   9.  37.  49.   3.  66.  40.  14.]
 [  5. 736.  67.  13.  37.  10.  60.   6.  33.  33.]
 [  5.  17. 779.  51.  23.   7.  37.  17.  40.  24.]
 [  6.  14.  77. 787.  13.  39.  10.  12.  34.   8.]
 [ 68.  41.  50.  13. 678.  21.  35.  20.  30.  44.]
 [  8.  13. 135.  15.  17. 752.  28.   5.  19.   8.]
 [  4.  16. 135.   6.  24.   7. 777.  10.  14.   7.]
 [ 23.  21.  24.   9.  90.  22.  43. 658.  72.  38.]
 [  9.  20.  41.  34.  11.  23.  30.   5. 817.  10.]
 [  8.  27.  59.   2.  64.  32.  24.  20.  31. 733.]]

Test set: Average loss: 0.8248, Accuracy: 7494/10000 (75%)

/content/drive/MyDrive/a1# ■
```

75%, better!

30:

Number of independent parameters: (784*30+30) +(30*10+10) = 23860

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.459345
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.420819
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.362843
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.342439
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.164767
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.337474
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.342502
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.475261
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.268352
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.364648
<class 'numpy.ndarray'>
[[807.   4.   3.   6.  34.  35.   3.  60.  41.   7.]
 [  7. 770.  56.   6.  38.  12.  55.   6.  23.  27.]
 [  5.  21. 823.  33.  21.  12.  31.  19.  25.  10.]
 [  5.  16.  47. 850.   7.  28.   8.  11.  19.   9.]
 [ 56.  38.  46.  13. 729.  14.  21.  23.  23.  37.]
 [  7.  20. 104.  12.  16. 778.  24.   5.  26.   8.]
 [  3.  18. 101.   6.  16.  17. 807.  16.   7.   9.]
 [ 10.  15.  24.  10.  68.  11.  32. 735.  55.  40.]
 [  5.  21.  32.  29.  10.  19.  35.   7. 833.   9.]
 [  1.  34.  75.   6.  51.  14.  25.  27.  25. 742.]]

Test set: Average loss: 0.7008, Accuracy: 7874/10000 (79%)
```

79%, In progress!

40:

Number of independent parameters: (784*40+40) +(40*10+10)=31810

```
Test set: Average loss: 0.6495, Accuracy: 8005/10000 (80%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.374004
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.312142
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.270533
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.311401
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.149985
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.312580
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.373133
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.350606
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.191140
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.348486
<class 'numpy.ndarray'>
[[841.   5.   1.   7.  34.  28.   4.  39.  28.  13.]
 [  5. 773.  33.   7.  23.  14.  86.   4.  25.  30.]
 [  6.  16. 829.  31.  13.  12.  40.  14.  21.  18.]
 [  4.  22.  39. 871.   7.  19.  10.  13.   6.   9.]
 [ 60.  39.  23.   8. 744.  13.  49.  18.  24.  22.]
 [  8.  23.  93.  14.  15. 781.  37.   3.  15.  11.]
 [  3.  17.  66.   5.  21.   7. 860.  13.   2.   6.]
 [ 16.  17.  30.   9.  33.  10.  62. 739.  36.  48.]
 [ 12.  29.  24.  39.   2.  14.  37.   7. 827.   9.]
 [  3.  32.  57.   2.  37.  11.  33.  15.  12. 798.]]

Test set: Average loss: 0.6386, Accuracy: 8063/10000 (81%)

/content/drive/MyDrive/a1# 
```

81%

50:

Number of independent parameters: (784*50+50) +(50*10+10)=39760

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.323327
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.328267
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.289418
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.270430
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.140613
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.376634
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.284339
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.361505
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.162795
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.368380
<class 'numpy.ndarray'>
[[828.   7.   2.   5.  40.  35.   3.  48.  25.   7.]
 [  4. 783.  32.   9.  24.  11.  70.   7.  26.  34.]
 [  7.  32. 792.  45.  13.  22.  36.   9.  23.  21.]
 [  7.  10.  47. 883.   2.  18.  13.   1.  11.   8.]
 [ 54.  34.  25.  13. 750.   8.  36.  22.  21.  37.]
 [ 10.  15.  77.  14.   6. 817.  35.   3.  16.   7.]
 [  2.  17.  70.   7.  14.   7. 861.  10.   2.  10.]
 [ 21.  20.  19.   4.  43.  12.  40. 773.  26.  42.]
 [ 10.  23.  25.  46.   3.  16.  29.   7. 829.  12.]
 [  3.  20.  70.   9.  34.   9.  21.  20.  16. 798.]]

Test set: Average loss: 0.6006, Accuracy: 8114/10000 (81%)
```

Still 81%

60:

Number of independent parameters: (784*60+60)+(60*10+10) = 47710

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.363458
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.287466
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.253485
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.259589
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.128334
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.315069
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.234374
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.387143
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.173091
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.259589
<class 'numpy.ndarray'>
[[822.   8.   2.   1.  34.  44.   3.  41.  38.   7.]
 [  5. 792.  42.   5.  25.   8.  72.   3.  19.  29.]
 [  6.  13. 839.  33.  17.  13.  28.  11.  23.  17.]
 [  4.  11.  41. 883.   3.  20.  10.   7.  10.  11.]
 [ 44.  32.  25.  11. 781.  11.  33.  15.  21.  27.]
 [  6.  10.  88.  11.   9. 818.  30.   3.  16.   9.]
 [  3.   7.  66.   8.  15.   5. 872.  10.   4.  10.]
 [ 12.   7.  14.   5.  34.  12.  44. 793.  39.  40.]
 [ 10.  23.  31.  41.   4.  14.  36.   4. 827.  10.]
 [  4.  15.  50.   4.  32.  14.  27.  14.  16. 824.]]

Test set: Average loss: 0.5716, Accuracy: 8251/10000 (83%)
```

83%! A big step forward!

70:

Number of independent parameters: (784*70+70)+(70*10+10) =55660

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.363229
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.291528
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.229270
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.204265
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.114644
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.282085
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.276202
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.365121
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.139230
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.340074
<class 'numpy.ndarray'>
[[854.   4.   1.   6.  28.  33.   2.  40.  26.   6.]
 [  4. 796.  39.   7.  23.   9.  63.   4.  24.  31.]
 [  6.  13. 847.  29.  13.  11.  29.  10.  24.  18.]
 [  5.   6.  38. 900.   5.  18.  10.   3.   5.  10.]
 [ 49.  31.  26.  13. 777.   6.  34.  13.  25.  26.]
 [  6.  10.  87.  14.  15. 814.  26.   4.  12.  12.]
 [  3.  14.  67.  11.  14.   9. 867.   8.   3.   4.]
 [ 20.   9.  16.   7.  27.  13.  41. 795.  35.  37.]
 [  9.  17.  29.  57.   5.  12.  31.   8. 824.   8.]
 [  3.  18.  61.   4.  26.  13.  22.  17.  13. 823.]]

Test set: Average loss: 0.5617, Accuracy: 8297/10000 (83%)
```

Still 83%, sad…

80:

Number of independent parameters: (784*80+80)+(80*10+10)=63610

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.340102
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.238114
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.261182
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.258547
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.124793
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.263122
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.210502
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.356416
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.133135
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.252261
<class 'numpy.ndarray'>
[[851.   5.   1.   4.  27.  37.   1.  42.  28.   4.]
 [  6. 821.  35.   5.  16.   8.  54.   4.  24.  27.]
 [  5.  12. 850.  35.  12.  17.  22.  15.  20.  12.]
 [  3.   7.  37. 907.   2.  17.   8.   4.   7.   8.]
 [ 44.  36.  18.   8. 797.   7.  29.  18.  22.  21.]
 [  7.  11.  75.  13.  13. 824.  29.   1.  19.   8.]
 [  3.  15.  60.   9.  15.   6. 878.   6.   3.   5.]
 [ 18.  18.  19.   5.  31.  11.  43. 793.  24.  38.]
 [ 11.  28.  31.  45.   4.  11.  30.   3. 829.   8.]
 [  3.  17.  45.   3.  29.  16.  24.  15.  15. 833.]]

Test set: Average loss: 0.5319, Accuracy: 8383/10000 (84%)
```

Goal! Touch the target!

Because of my curiosity, let's test 100:

Number of independent parameters: (784*100+100)+(100*10+10)=79510

```
Test set: Average loss: 0.5411, Accuracy: 8354/10000 (84%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.301974
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.268440
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.236818
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.231402
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.121280
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.257735
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.229914
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.361864
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.143310
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.291465
<class 'numpy.ndarray'>
[[855.   3.   2.   6.  31.  32.   2.  34.  29.   6.]
 [  7. 798.  30.   4.  25.  11.  66.   4.  20.  35.]
 [  9.  16. 824.  39.  13.  20.  25.  13.  26.  15.]
 [  4.  12.  35. 909.   1.  13.   7.   4.   5.  10.]
 [ 38.  32.  17.   7. 808.   7.  28.  22.  21.  20.]
 [ 10.  12.  63.   9.  10. 842.  27.   4.  17.   6.]
 [  3.   9.  44.  10.  14.   6. 892.   7.   3.  12.]
 [ 13.  17.  14.   3.  35.   7.  39. 807.  30.  35.]
 [ 11.  24.  29.  48.   6.  11.  31.   3. 825.  12.]
 [  3.  18.  40.   5.  32.   3.  22.  10.  13. 854.]]

Test set: Average loss: 0.5229, Accuracy: 8414/10000 (84%)
```

I think 84% is almost the limit. Let's still use 80 hidden nodes.

3)

Ok, first test:

```
    # all using relu, followed by log_softmax
    def __init__(self):
        super(NetConv, self).__init__()
        self.convolution1 = nn.Conv2d(1, 8, 4)

        self.convolution2 = nn.Conv2d(8, 16, 4)
        self.flat = nn.Flatten()
        self.linear1 = nn.Linear(1296, 80)
        self.linear2 = nn.Linear(80, 10)

        # INSERT CODE HERE

    def forward(self, x):
        x = self.convolution1(x)
        # 25
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        # 12
        x = self.convolution2(x)
        # 9
        x = F.relu(x)

        x = self.flat(x)
        # 1296
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        x = F.log_softmax(x, dim=1)
```

:

Then the result:

```
Test set: Average loss: 0.3569, Accuracy: 9063/10000 (91%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.120852
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.021567
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.128868
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.163073
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.053365
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.087052
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.045010
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.198383
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.021494
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.143668
<class 'numpy.ndarray'>
[[933.   4.   1.   1.  42.   3.   3.   7.   5.   1.]
 [  4. 908.  12.   0.   8.   0.  34.   6.   4.  24.]
 [ 11.  10. 882.  34.   4.   8.  21.   8.   9.  13.]
 [  4.   5.  18. 953.   2.  10.   3.   2.   1.   2.]
 [ 27.  19.   4.   4. 888.   3.  19.  14.   7.  15.]
 [  4.  10.  84.   8.   8. 832.  29.   7.   3.  15.]
 [  3.   5.  31.   3.  10.   1. 939.   3.   2.   3.]
 [  3.   4.   6.   0.   5.   1.  17. 921.   4.  39.]
 [  5.  28.  23.  18.   9.   2.  10.   5. 895.   5.]
 [ 10.   9.  14.   1.   5.   1.   7.   7.   1. 945.]]

Test set: Average loss: 0.3531, Accuracy: 9096/10000 (91%)

/content/drive/MyDri
```

Not so good,

I will try to let the system catch more features:

Second test:

```
    # all using relu, followed by log_softmax
    def __init__(self):
        super(NetConv, self).__init__()
        self.convolution1 = nn.Conv2d(1, 16, 4)

        self.convolution2 = nn.Conv2d(16, 32, 4)
        self.flat = nn.Flatten()
        self.linear1 = nn.Linear(2592, 80)
        self.linear2 = nn.Linear(80, 10)

        # INSERT CODE HERE

    def forward(self, x):
        x = self.convolution1(x)
        # 25
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        # 12
        x = self.convolution2(x)
        # 9
        x = F.relu(x)

        x = self.flat(x)
        # 2592
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        x = F.log_softmax(x, dim=1)
```

Result:

```
Test set: Average loss: 0.2993, Accuracy: 9224/10000 (92%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.054432
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.042590
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.095222
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.051923
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.092755
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.055769
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.017958
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.075562
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.030694
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.051751
<class 'numpy.ndarray'>
[[948.   4.   5.   0.  28.   1.   2.   7.   2.   3.]
 [  2. 921.   3.   0.   9.   2.  39.   5.   5.  14.]
 [ 11.  10. 877.  35.   9.   5.  19.   5.  12.  17.]
 [  2.   4.  15. 953.   4.   9.   7.   2.   1.   3.]
 [ 25.   7.   7.   2. 917.   2.  15.  11.   8.   6.]
 [  4.  16.  49.   7.   8. 881.  26.   5.   2.   2.]
 [  2.  10.  20.   1.   6.   2. 949.   5.   2.   3.]
 [  1.   8.   6.   1.  11.   1.  10. 935.   7.  20.]
 [  4.  20.  11.   8.  14.   2.   7.   2. 922.  10.]
 [ 10.   7.  13.   2.   8.   2.   4.   9.   6. 939.]]

Test set: Average loss: 0.3010, Accuracy: 9242/10000 (92%)
```

So close!

Then I tried to increase the number of hidden nodes:

```
    # all using relu, followed by log_softmax
    def __init__(self):
        super(NetConv, self).__init__()
        self.convolution1 = nn.Conv2d(1, 16, 4)

        self.convolution2 = nn.Conv2d(16, 32, 4)
        self.flat = nn.Flatten()
        self.linear1 = nn.Linear(2592, 100)
        self.linear2 = nn.Linear(100, 10)

        # INSERT CODE HERE

    def forward(self, x):
        x = self.convolution1(x)
        # 25
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        # 12
        x = self.convolution2(x)
        # 9
        x = F.relu(x)

        x = self.flat(x)
        # 2592
        x = self.linear1(x)
        x = F.relu(x)
        x = self.linear2(x)
        x = F.log_softmax(x, dim=1)
```

Result:

```
Test set: Average loss: 0.2900, Accuracy: 9236/10000 (92%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.062627
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.008003
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.033813
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.036134
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.043149
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.054075
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.026167
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.062762
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.011960
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.025910
<class 'numpy.ndarray'>
[[951.   3.   3.   0.  25.   2.   0.   9.   1.   6.]
 [  2. 940.   7.   0.   8.   0.  29.   2.   4.   8.]
 [ 10.   9. 862.  45.   8.   6.  27.  11.   6.  16.]
 [  4.   4.  13. 960.   1.   6.   4.   2.   1.   5.]
 [ 24.  15.   3.   5. 914.   1.  13.   8.   7.  10.]
 [  4.  15.  67.   9.   5. 872.  16.   3.   2.   7.]
 [  6.   9.  27.   1.   4.   3. 938.   5.   1.   6.]
 [  6.   5.   7.   0.   5.   1.   7. 953.   6.  10.]
 [  6.  26.  11.  10.  10.   4.  13.   4. 910.   6.]
 [ 10.  10.  13.   2.   6.   0.   3.   2.   1. 953.]]

Test set: Average loss: 0.2953, Accuracy: 9253/10000 (93%)

/content/drive/MyDri
/content/drive/MyDrive/a1# ■
```

Here, we achieved our target.

Then the number of the parameters:

Converlution1(with bias): (4*4*1+1)*16 = 272

Converlution2(with bias): (4*4*16+1)*32=8224

Linear1(with bias): 2592*100+100 = 259300

Linear2(with bias): 100*10+10= 1010

Total: 268806

4.

the relative accuracy of the three models:
NetLin: 70%
Netfull: 84%
NetConv: 93%
netConv works best.
Characters are not linearly separable, so NetLin is not so good.
Compared with NetConv, NetFull cannot capture features. So, the accuracy rate is not as high as the accuracy of NetConv.

the number of independent parameters in each of the three models:
NetLin: 748*10+10 = 7490
NetFull: 63610
netConv: 268806
netConv has the best effect with the most independent parameters

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

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.822064
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.635364
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.596002
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.600355
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.322878
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.520681
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.660333
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.605380
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.353989
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.670523
<class 'numpy.ndarray'>
[[769.   5.   9.  12.  29.  65.   2.  62.  29.  18.]
 [  7. 668. 107.  17.  28.  24.  58.  13.  27.  51.]
 [  8.  59. 690.  27.  25.  20.  46.  38.  47.  40.]
 [  3.  34.  61. 757.  16.  57.  16.  18.  26.  12.]
 [ 59.  54.  81.  20. 623.  23.  32.  32.  20.  56.]
 [  8.  29. 124.  16.  19. 726.  27.   9.  33.   9.]
 [  5.  24. 149.   9.  25.  24. 720.  20.  10.  14.]
 [ 17.  30.  27.  12.  80.  15.  52. 627.  91.  49.]
 [ 10.  38.  97.  39.   7.  31.  43.   7. 703.  25.]
 [  8.  51.  91.   3.  52.  31.  19.  32.  41. 672.]]

Test set: Average loss: 1.0105, Accuracy: 6955/10000 (70%)

/content/drive/MyDri
```

Netfull:

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.340102
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.238114
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.261182
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.258547
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.124793
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.263122
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.210502
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.356416
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.133135
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.252261
<class 'numpy.ndarray'>
[[851.   5.   1.   4.  27.  37.   1.  42.  28.   4.]
 [  6. 821.  35.   5.  16.   8.  54.   4.  24.  27.]
 [  5.  12. 850.  35.  12.  17.  22.  15.  20.  12.]
 [  3.   7.  37. 907.   2.  17.   8.   4.   7.   8.]
 [ 44.  36.  18.   8. 797.   7.  29.  18.  22.  21.]
 [  7.  11.  75.  13.  13. 824.  29.   1.  19.   8.]
 [  3.  15.  60.   9.  15.   6. 878.   6.   3.   5.]
 [ 18.  18.  19.   5.  31.  11.  43. 793.  24.  38.]
 [ 11.  28.  31.  45.   4.  11.  30.   3. 829.   8.]
 [  3.  17.  45.   3.  29.  16.  24.  15.  15. 833.]]

Test set: Average loss: 0.5319, Accuracy: 8383/10000 (84%)
```

NetConv:

```
Test set: Average loss: 0.2900, Accuracy: 9236/10000 (92%)

Train Epoch: 10 [0/60000 (0%)]  Loss: 0.062627
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.008003
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.033813
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.036134
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.043149
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.054075
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.026167
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.062762
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.011960
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.025910
<class 'numpy.ndarray'>
[[951.   3.   3.   0.  25.   2.   0.   9.   1.   6.]
 [  2. 940.   7.   0.   8.   0.  29.   2.   4.   8.]
 [ 10.   9. 862.  45.   8.   6.  27.  11.   6.  16.]
 [  4.   4.  13. 960.   1.   6.   4.   2.   1.   5.]
 [ 24.  15.   3.   5. 914.   1.  13.   8.   7.  10.]
 [  4.  15.  67.   9.   5. 872.  16.   3.   2.   7.]
 [  6.   9.  27.   1.   4.   3. 938.   5.   1.   6.]
 [  6.   5.   7.   0.   5.   1.   7. 953.   6.  10.]
 [  6.  26.  11.  10.  10.   4.  13.   4. 910.   6.]
 [ 10.  10.  13.   2.   6.   0.   3.   2.   1. 953.]]

Test set: Average loss: 0.2953, Accuracy: 9253/10000 (93%)

/content/drive/MyDri
/content/drive/MyDrive/a1# ■
```

No matter which model it is, the number of times the model judges 5 as 2 is quite high.
(But what is strange is the system doesn't easily mistake 2 for 5, cool).
Let's have a look at 5 and 2:
5

2:

Some of the handwriting is indeed very similar.
Although I have no idea why 2 is not easily misjudged as 5, I haven't found any example with

a higher number of misjudgments than this one (5 was judged as 2) in every model.

That's a good example.

So, 5 is most likely to be mistaken for 2.

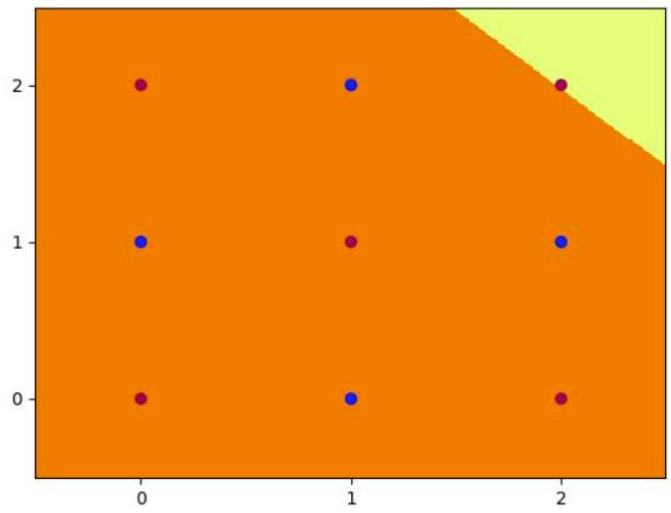
Part 2: Multi-Layer Perceptron

1.

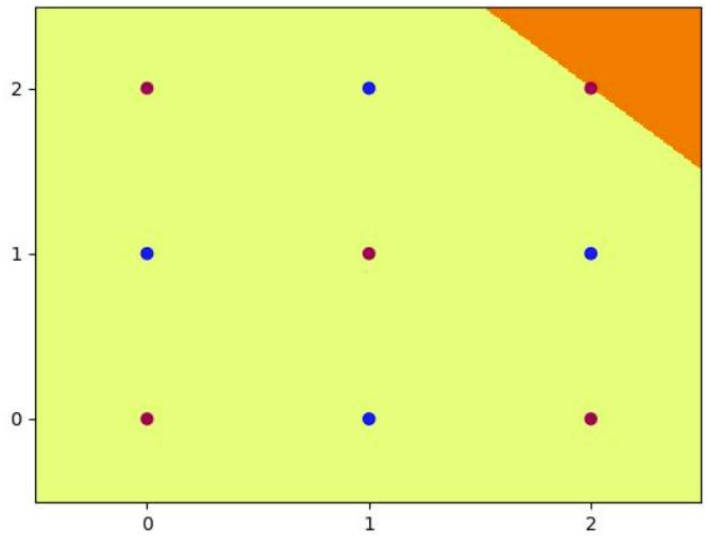
```
ep:23300 loss: 0.1594 acc: 100.00
ep:23400 loss: 0.1553 acc: 100.00
Final Weights:
tensor([[-2.7832, -2.7863],
        [ 2.7103,  2.7132],
        [-4.1317,  0.0531],
        [-5.4807, -5.4593],
        [ 0.0491, -3.8742]])
tensor([ 11.0824, -10.8552,   3.7695,   7.5402,   3.5180])
tensor([[  9.2906, -10.3320,  -9.6734,   8.8993,  -9.7715]])
tensor([-2.0332])
Final Accuracy:  100.0
/content/drive/MyDri
```

Get the accuracy of 100%.

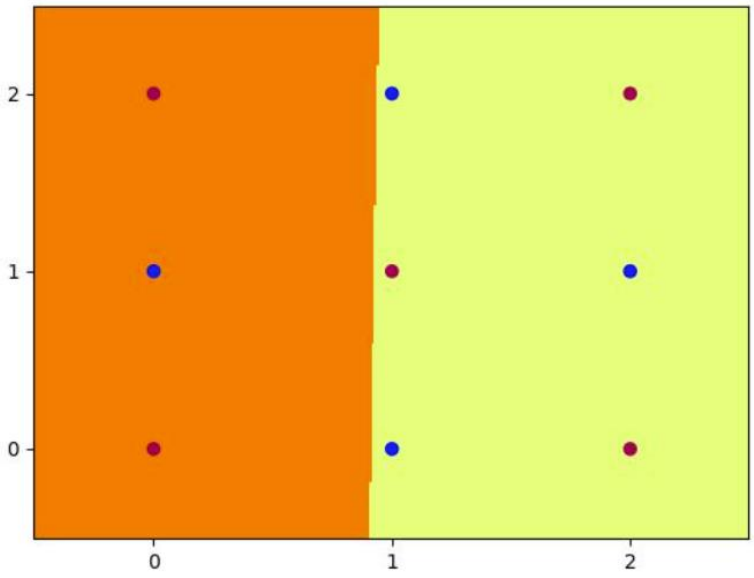
Hid_5_0:



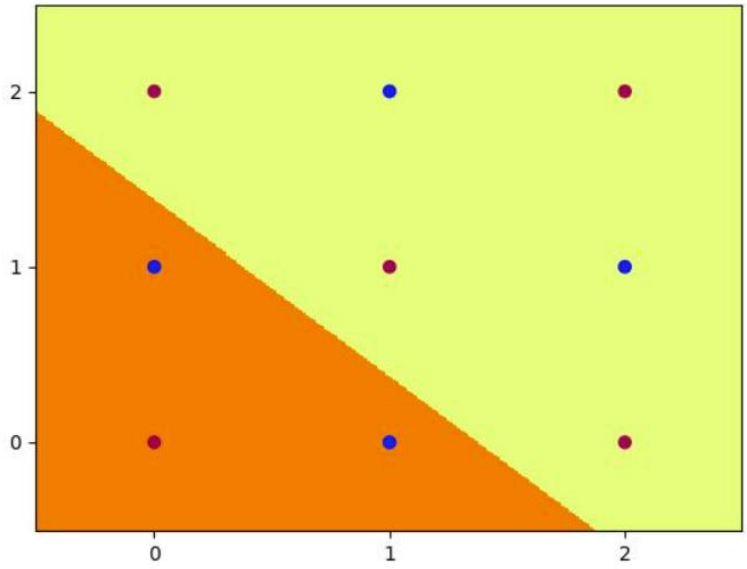
Hid_5_1:



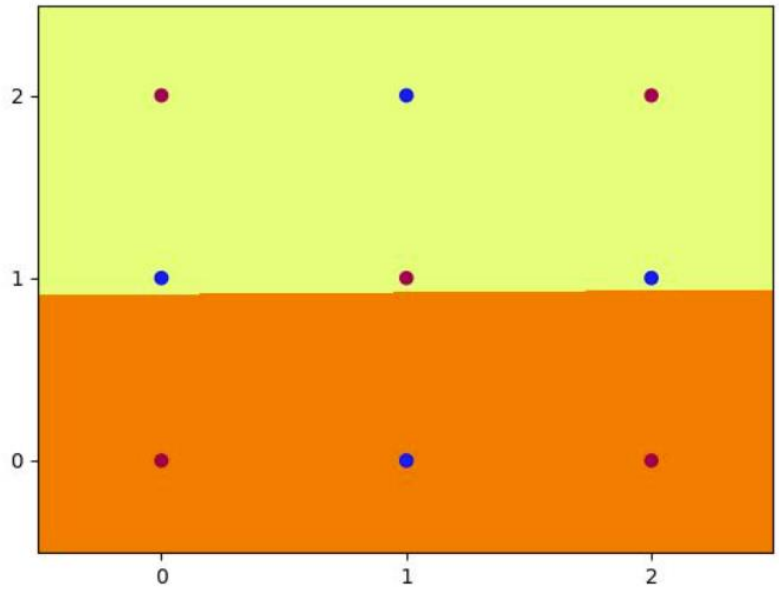
Hid_5_2



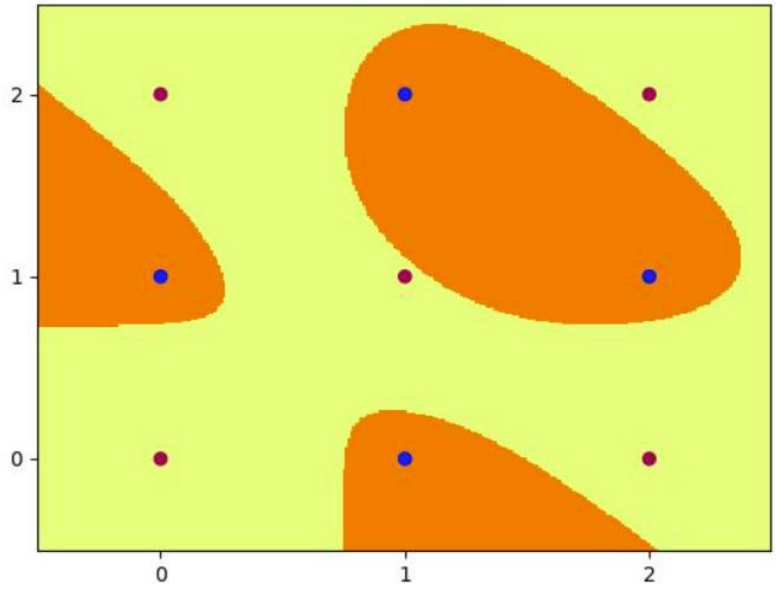
Hid_5_3



Hid_5_4



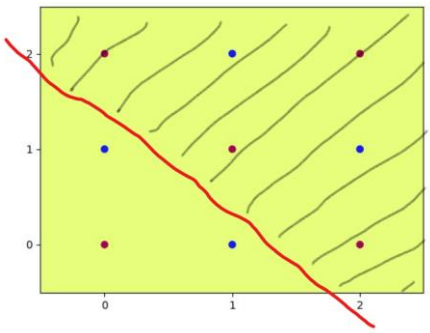
Out_5:



2.

OK, we have four hidden layers, so, we can create four lines:

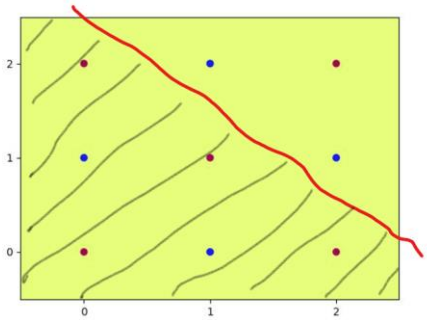
1)



We need large x1 and large x2

y = x1+x2-1.5

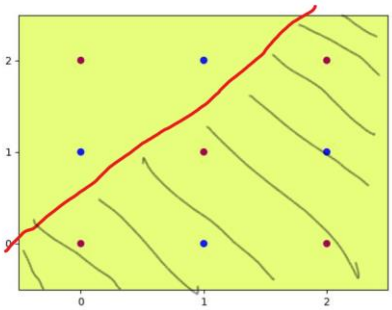
2)



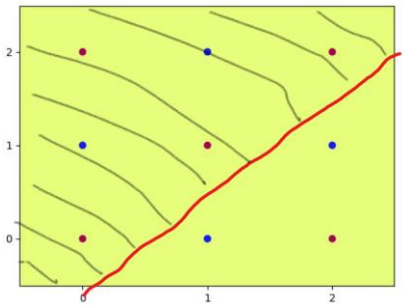
We need small x1 and small x2

y = -x1-x2+2.5

3)

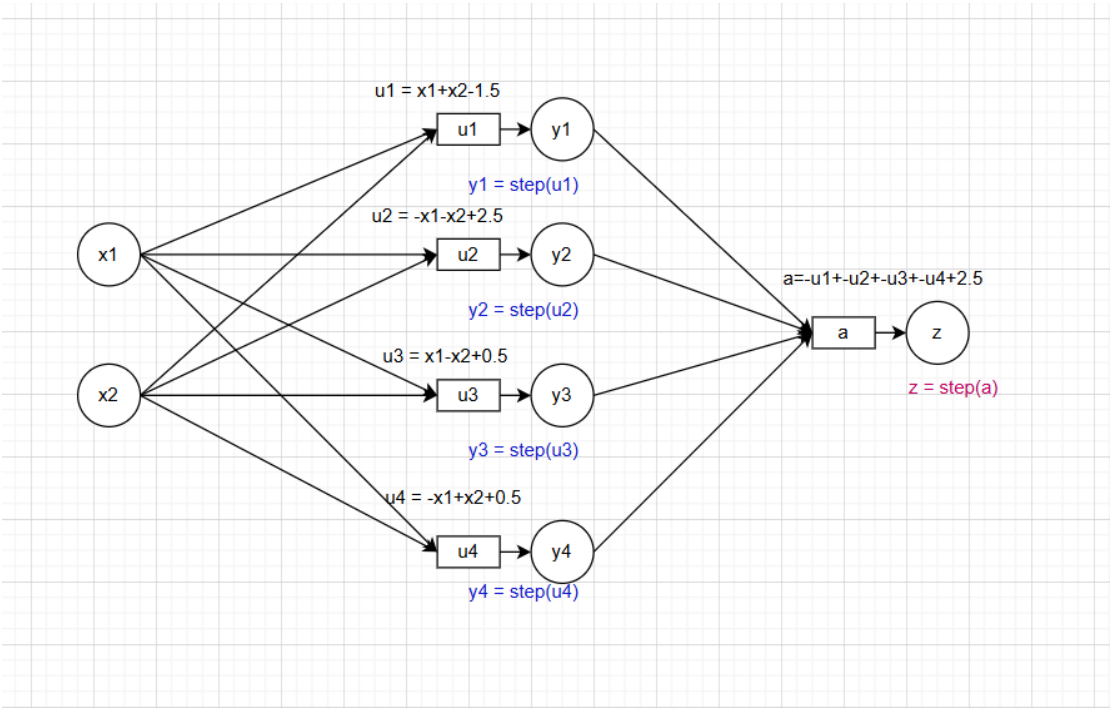


We need large x1 and small x2
y = x1-x2+0.5
4)



We need small x1 and large x2
y = -x1+x2+0.5

Every target point (blue points) can just activate 2 hidden nodes. However, every red point can activate at least 3 hidden nodes. That is the approach we design the output:
z = -y1-y2-y3-y4+2.5. If there are three or more hidden nodes activated, z will be negative.
Here is the diagram:



Also, don't forget step function.

OK, I will show the activation table:

Node 1: y = x1+x2-1.5

Node 2: y = -x1-x2+2.5

Node3: y = x1-x2+0.5

Node4: y = -x1+x2+0.5

Output: z = -y1-y2-y3-y4+2.5

Active: 1

Inactive: 0

	Node1	If-act-1	Node2	If-act-2	Node3	If-act-3	Node4	If-act-4	output	If-act-out
(0,0)	-1.5	0	2.5	1	0.5	1	0.5	1	-0.5	0
(0,1)	-0.5	0	1.5	1	-0.5	0	1.5	1	0.5	1
(0.2)	0.5	1	0.5	1	-1.5	0	2.5	1	-0.5	0
(1,0)	-0.5	0	1.5	1	1.5	1	-0.5	0	0.5	1
(1,1)	0.5	1	0.5	1	0.5	1	0.5	1	-1.5	0
(1,2)	1.5	1	-0.5	0	-0.5	0	1.5	1	0.5	1
(2,0)	0.5	1	0.5	1	2.5	1	-1.5	0	-0.5	0
(2,1)	1.5	1	-0.5	0	1.5	1	-0.5	0	0.5	1
(2,2)	2.5	1	-1.5	0	0.5	1	0.5	1	0.5	1

Here are the parameters in the code:

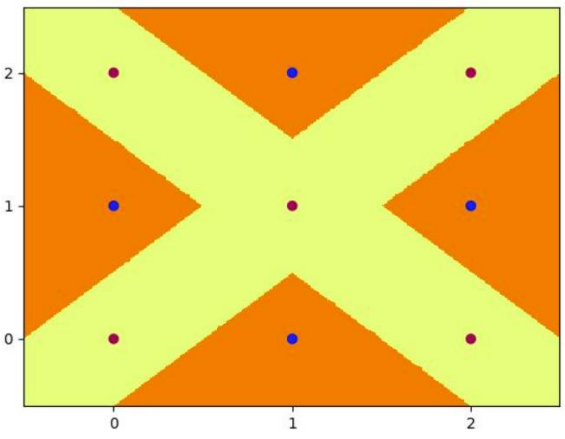
```
                return  torch.sigmoid(self.hid_out(self.hid))

    def  set_weights(self):
##### Enter  Weights  Here  ###########
            in_hid_weight    =  [[1,1],[-1,-1],[1,-1],[-1,1]]
            hid_bias             =  [-1.5,2.5,0.5,0.5]
            hid_out_weight  =  [[-1,-1,-1,-1]]
            out_bias             =  [2.5]
#####################################
            self.in_hid.weight.data  =  torch.tensor(
                    in_hid_weight,  dtype=torch.float32)
```

Then, we can have a test:

```
/content/drive/MyDrive# cd a1
/content/drive/MyDrive/a1# python3 check_main.py --act step --hid 4 --set_weights
Initial Weights:
tensor([[ 1.,  1.],
        [-1., -1.],
        [ 1., -1.],
        [-1.,  1.]])
tensor([-1.5000,  2.5000,  0.5000,  0.5000])
tensor([[-1., -1., -1., -1.]])
tensor([2.5000])
Initial Accuracy:  100.0
/content/drive/MyDrive/a1#
```

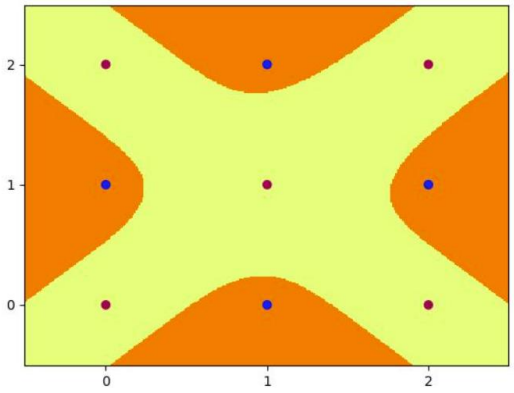
Success! Here is the check.jpg:



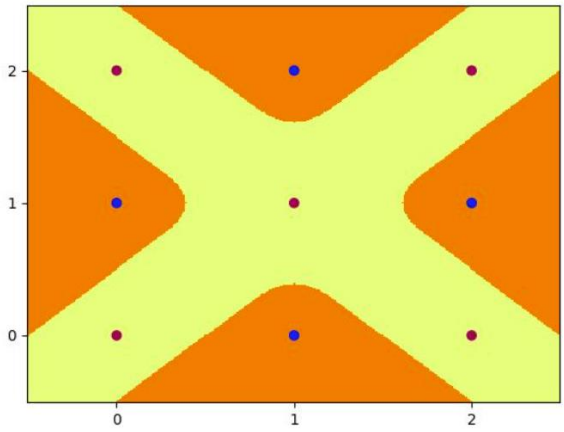
3)

Now we need to replace step function with sigmoid function.

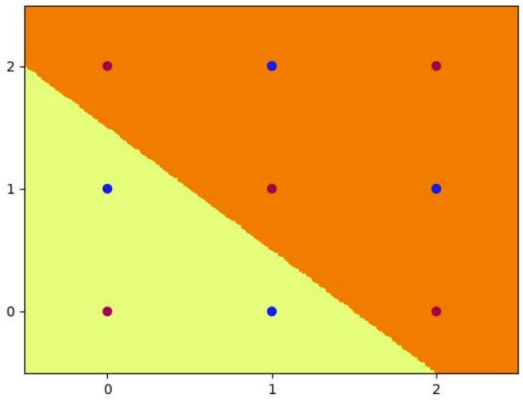
Multiplying weight and bias with a large number can make sigmoid function more like step function: if I use the original weights and bias, it needs training. here are the produce images:



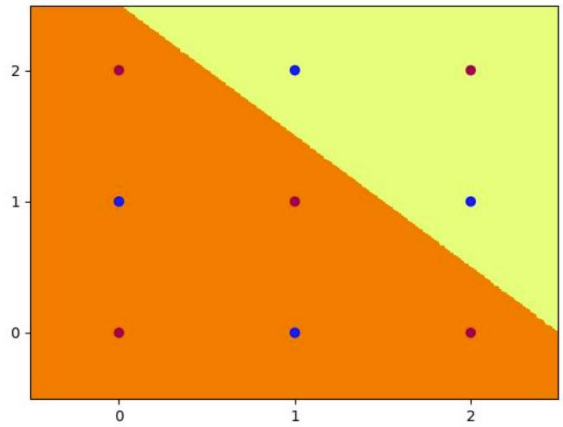
Then multiply all the weights and bias with 10:
Out_4:



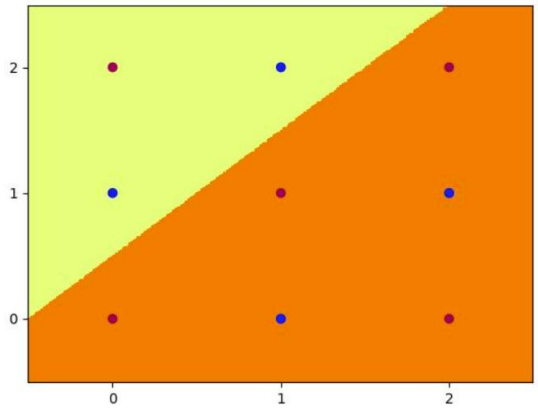
Hid_4_0:



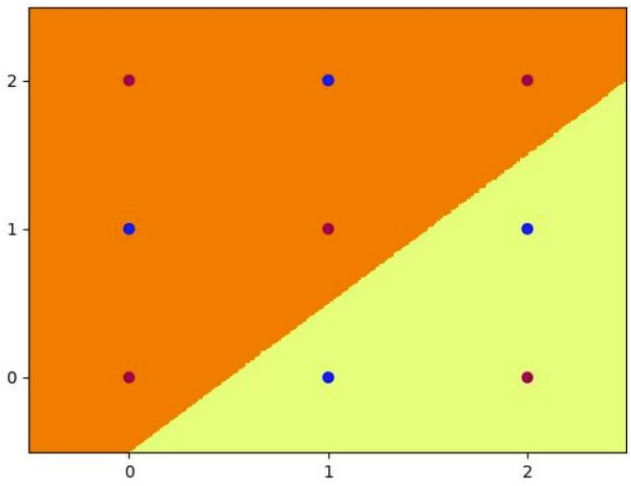
Hid_4_1:



Hid_4_2



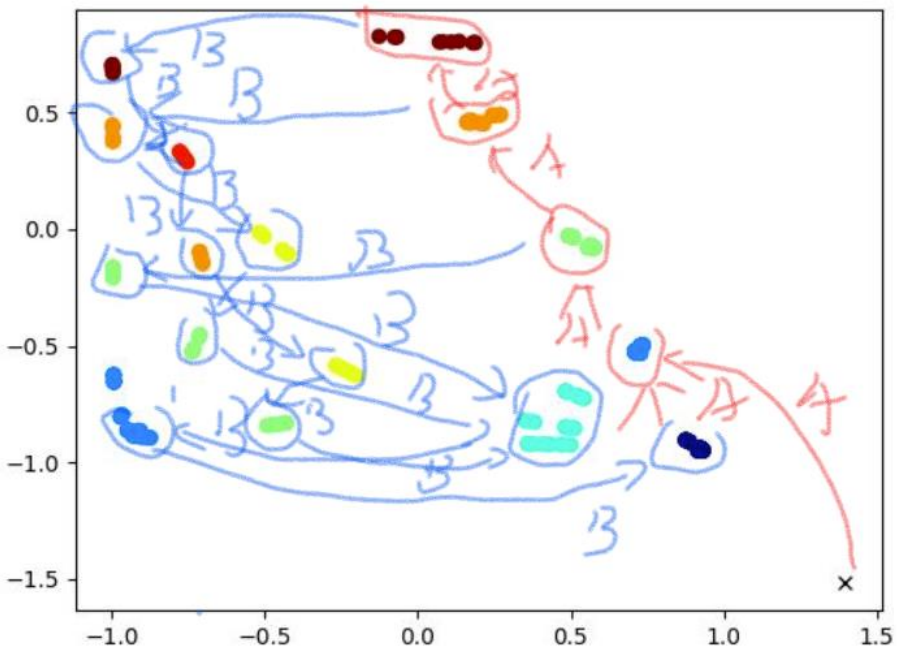
Hid_4_3:



Part 3: Hidden Unit Dynamics for Recurrent Networks

1.

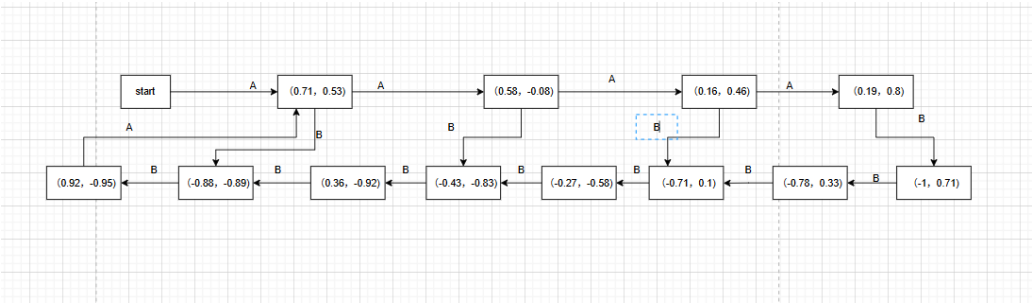
Here are the image, states and arrows:



If n equals to 4, the system will go through 4 states with red colors(feature vectors).

2.

Here is the finite state machine. The vector in the rectangle is one example of that state, and we use it to represent that state.



3.

Just as I demonstrated in the finite state machine, after receiving different numbers of A, the model receives B, and the model will jump to different states based on the number of A received. In other words, every time the model receives an "A", it jumps to a new state. Let's temporarily call these states that jump to due to receiving A the "A" state (the "B" state is the same concept). When different "A" states receive B, they will jump to the corresponding "B" state.

The jumps between these "A" states and "B" states are all very fixed, because the number of A determines the number of B. So, the "A" state before the jump determines the "B" state after the jump. Different “B” states indicate how many predictable Bs there will be later.

However, in any case, after the last B input, the model will jump into a fixed state. This state

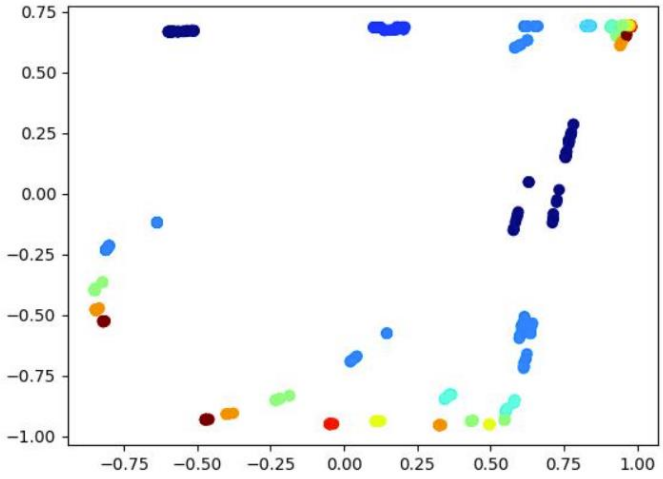
indicates that the next input must be A.

Now we come to the most important question: Why does inputting the same "A" lead to different "A" states? Because RNN has "memory", this "memory" refers to the state the model jumped into after the last input. Perhaps it would be more appropriate for us to call these states "feature vectors". The feature vector after the last input will be calculated together with the newly received letters in the hidden unit to obtain the new feature vector.

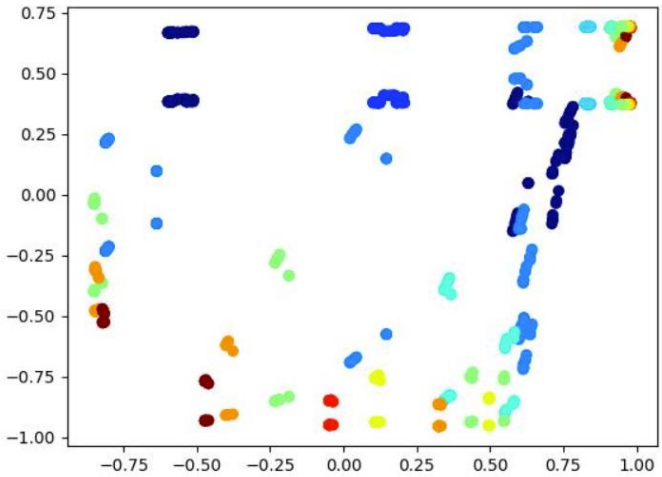
4.

OK, it took a long time to train, here are the three images:

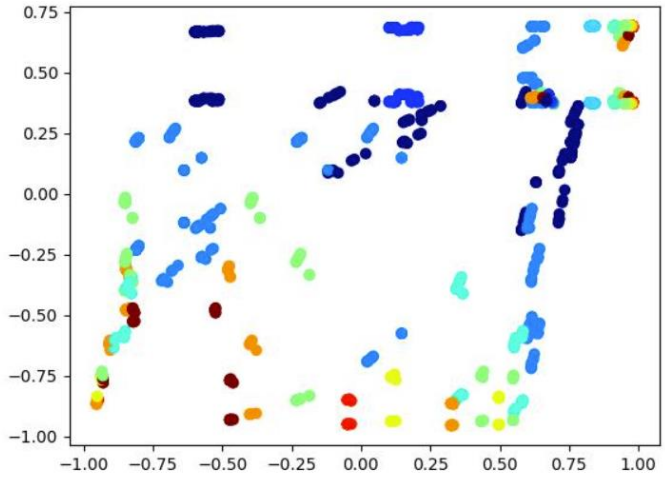
anb2nc3n_lstm3_01.jpg



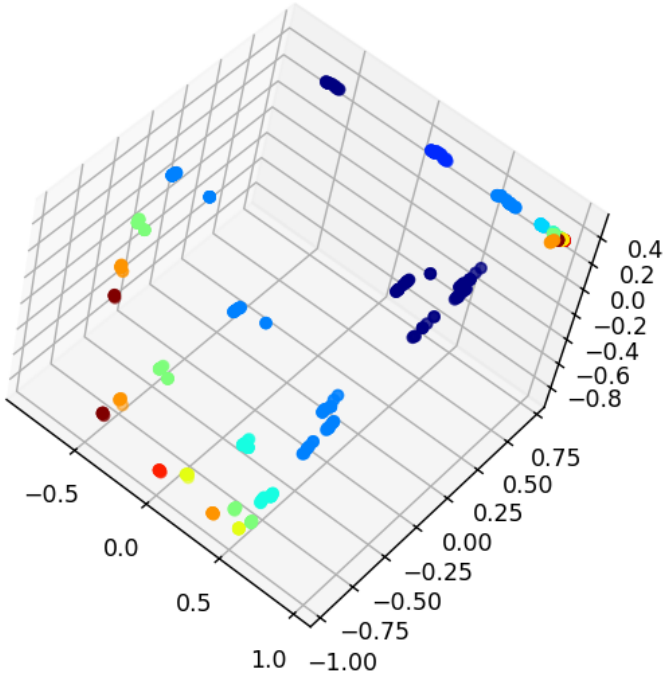
anb2nc3n_lstm3_02.jpg



anb2nc3n_lstm3_12.jpg

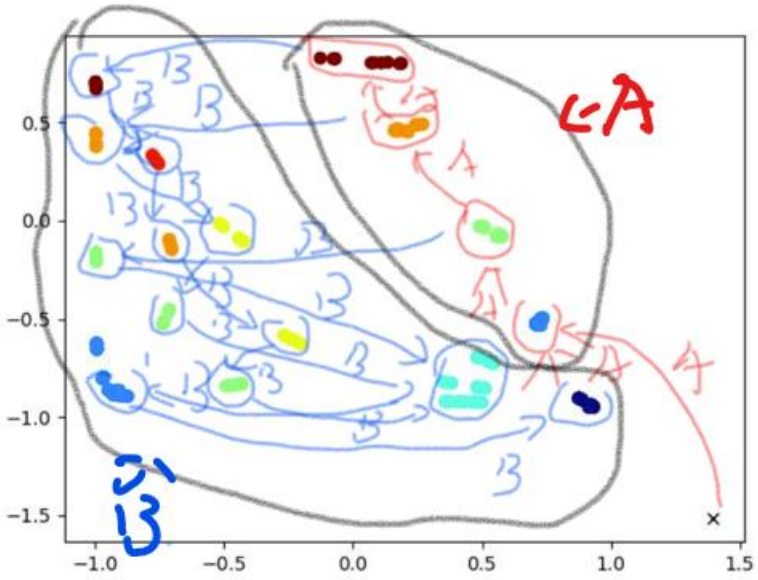


3D:



5.

It's little hard to draw circles and arrows to show how the state (feature vector) of model changes. Because this is actually a three-dimensional system, not a two-dimensional one. But I think the principle and form should be similar to anb2n. in anb2n:



There is one part for states of “last input is A”, and one part for states of “last input is B”. and if we connect the states of the whole loop (for example: AABBBBA), there will be a cycle. There should be a similar condition in anb2nc3n. However, compared with SRN, LSTM can control the input, forgetting and output of memories. This enables LSTM to have a stronger memory capacity. So LSTM can remember anb2nc3n, but SRN may not have such a high accuracy rate. So, in three-dimensional space, it should also be a similar condition like: