Exercise 1

Question 4. Briefly discuss the following points:

a. the relative accuracy of the three models

NetLin: 70%; NetFull: 85%; NetConv: 94%

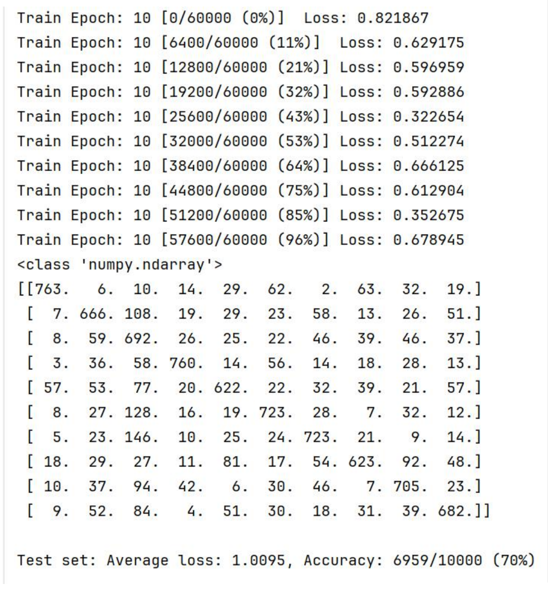
b. the number of independent parameters in each of the three models

NetLin: 784 * 10 + 10 = 7850

NetFull: (784 * 200 + 200) + (200 * 10 + 10) = 159010

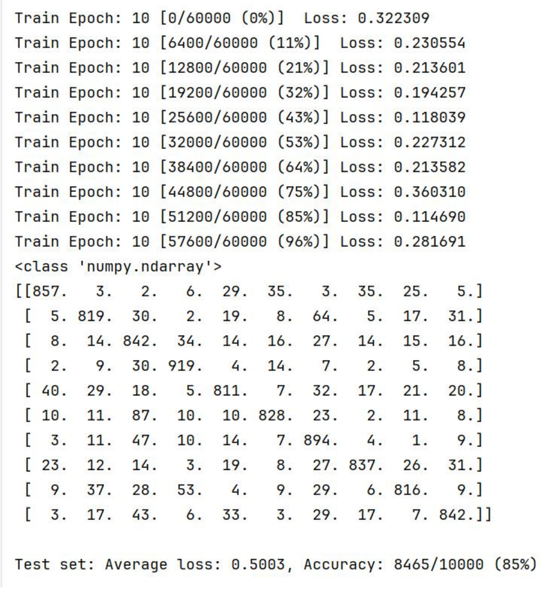
NetConv: (32 * 1 * 3 * 3 + 32) + (64 * 32 * 3 * 3 + 64) + (64 * 7 * 7 * 200 + 200) + (200 * 10 + 10)
= 648226

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



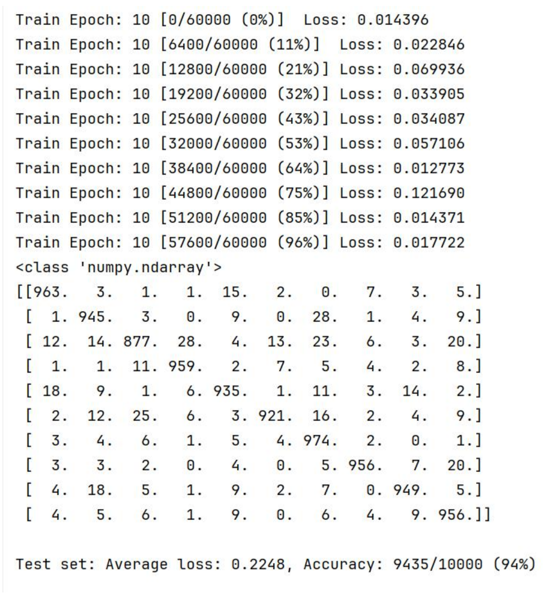
```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.821867
Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.629175
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.596959
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.592886
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.322654
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.512274
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.666125
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.612904
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.352675
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.678945
<class 'numpy.ndarray'>
[[763.   6.  10.  14.  29.  62.   2.  63.  32.  19.]
 [  7. 666. 108.  19.  29.  23.  58.  13.  26.  51.]
 [  8.  59. 692.  26.  25.  22.  46.  39.  46.  37.]
 [  3.  36.  58. 760.  14.  56.  14.  18.  28.  13.]
 [ 57.  53.  77.  20. 622.  22.  32.  39.  21.  57.]
 [  8.  27. 128.  16.  19. 723.  28.   7.  32.  12.]
 [  5.  23. 146.  10.  25.  24. 723.  21.   9.  14.]
 [ 18.  29.  27.  11.  81.  17.  54. 623.  92.  48.]
 [ 10.  37.  94.  42.   6.  30.  46.   7. 705.  23.]
 [  9.  52.  84.   4.  51.  30.  18.  31.  39. 682.]]

Test set: Average loss: 1.0095, Accuracy: 6959/10000 (70%)
```

NetLin



```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.322309
Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.230554
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.213601
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.194257
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.118039
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.227312
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.213582
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.360310
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.114690
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.281691
<class 'numpy.ndarray'>
[[857.   3.   2.   6.  29.  35.   3.  35.  25.   5.]
 [  5. 819.  30.   2.  19.   8.  64.   5.  17.  31.]
 [  8.  14. 842.  34.  14.  16.  27.  14.  15.  16.]
 [  2.   9.  30. 919.   4.  14.   7.   2.   5.   8.]
 [ 40.  29.  18.   5. 811.   7.  32.  17.  21.  20.]
 [ 10.  11.  87.  10.  10. 828.  23.   2.  11.   8.]
 [  3.  11.  47.  10.  14.   7. 894.   4.   1.   9.]
 [ 23.  12.  14.   3.  19.   8.  27. 837.  26.  31.]
 [  9.  37.  28.  53.   4.   9.  29.   6. 816.   9.]
 [  3.  17.  43.   6.  33.   3.  29.  17.   7. 842.]]

Test set: Average loss: 0.5003, Accuracy: 8465/10000 (85%)
```

NetFull



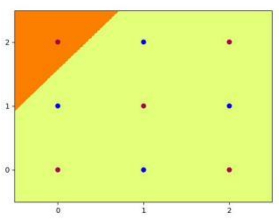
```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.014396
Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.022846
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.069936
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.033905
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.034087
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.057106
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.012773
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.121690
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.014371
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.017722
<class 'numpy.ndarray'>
[[963.   3.   1.   1.  15.   2.   0.   7.   3.   5.]
 [  1. 945.   3.   0.   9.   0.  28.   1.   4.   9.]
 [ 12.  14. 877.  28.   4.  13.  23.   6.   3.  20.]
 [  1.   1.  11. 959.   2.   7.   5.   4.   2.   8.]
 [ 18.   9.   1.   6. 935.   1.  11.   3.  14.   2.]
 [  2.  12.  25.   6.   3. 921.  16.   2.   4.   9.]
 [  3.   4.   6.   1.   5.   4. 974.   2.   0.   1.]
 [  3.   3.   2.   0.   4.   0.   5. 956.   7.  20.]
 [  4.  18.   5.   1.   9.   2.   7.   0. 949.   5.]
 [  4.   5.   6.   1.   9.   0.   6.   4.   9. 956.]]

Test set: Average loss: 0.2248, Accuracy: 9435/10000 (94%)
```

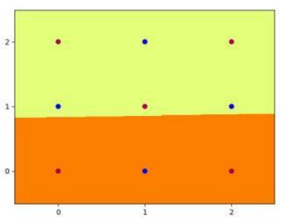
NetConv

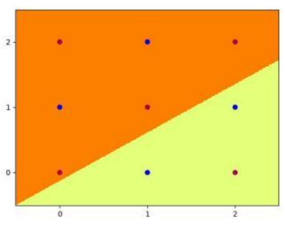
NetLin: class 4 is most easy to be seen as 0, 1, 2, 7, 9, it is a little bit difficult to get the regular pattern so may because the accuracy of this model is not so good. Just one connected layer can not do the classification task better.
NetFull: class 4 is most easy to be seen as 0, 1, 6, it seems that some characters with similar strokes or similar intermediate structures are easy to confuse.
NetConv: class 0 is easy to be seen as 4, class 1 is easy to be seen as 6, class 2 is easy to be seen as 3, 6, 9. That's all because they are really similar in storks and structures, so it is difficult to classify.

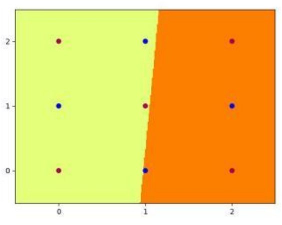
Exercise 2

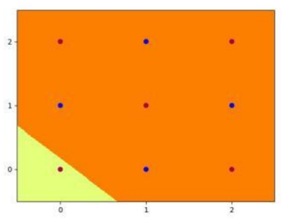
Question 1.

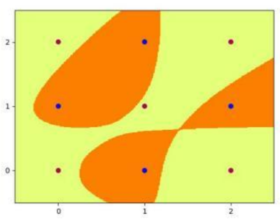

Hid_5_0


Hid_5_1


Hid_5_2

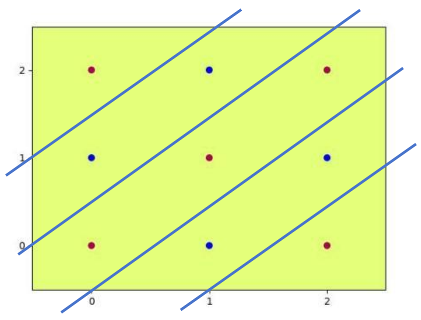

Hid_5_3

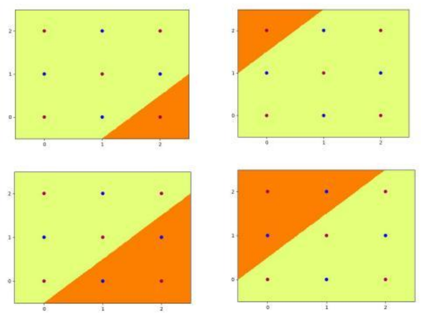

Hid_5_4


Out_5

Question 2.

Net: Input → Linear → Heaviside → Hidden → Linear → Heaviside → Output

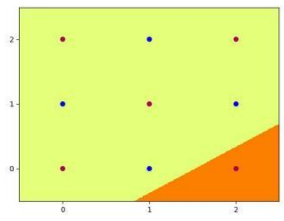




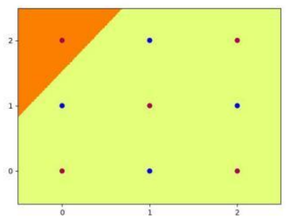
Using this 4 lines to get the correct classification.
In first linear: w1 = [1, -1], b1 = -1.5; w2 = [-1, 1], b2 = -1.5, w3 = [1, -1], b3 = -0.5; w4 = [-1, 1], b4 = -0.5. So x - y - 1.5 = 0; -x + y - 1.5 = 0; x - y - 0.5 = 0; -x + y - 0.5 = 0.
In second linear: keep blue points and delete red points so use -1 in first two weight, then adjust b. hid_out_weight = [[-1,-1,1,1]]   out_bias = [-1].
Here is the results of each linear:

	Input	h1	h2	h3	h4	Output	Target
1	(0.0, 0.0)	0	0	0	0	0	0
2	(1.0, 0.0)	0	0	1	0	1	1
3	(2.0, 0.0)	1	0	1	0	0	0
4	(0.0, 1.0)	0	0	0	1	1	1
5	(1.0, 1.0)	0	0	0	0	0	0
6	(2.0, 1.0)	0	0	1	0	1	1
7	(0.0, 2.0)	0	1	0	1	0	0
8	(1.0, 2.0)	0	0	0	1	1	1
9	(2.0, 2.0)	0	0	0	0	0	0

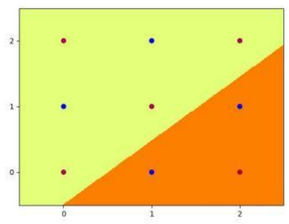
Question 3.



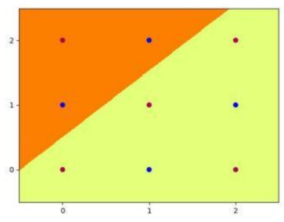
Hid_4_0



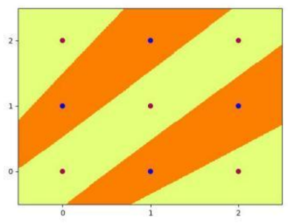
Hid_4_1



Hid_4_2



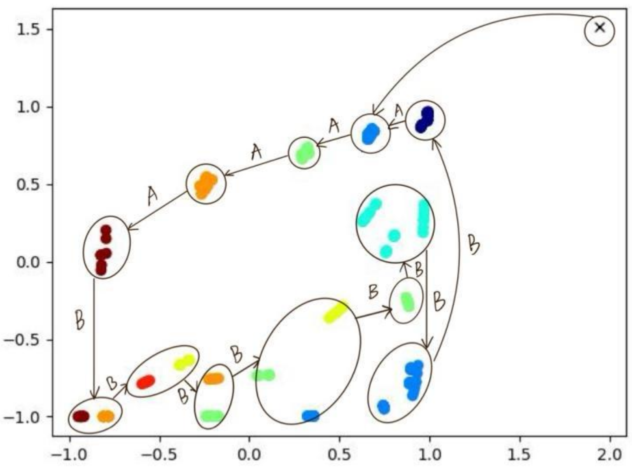
Hid_4_3



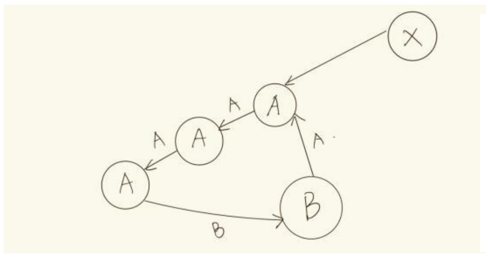
Out_4

Exercise 3

Question 1.



Question 2.



I merged all the predictions which is Determined so each step of the FSM is responsible for executing the current state and producing a prediction for the next state (determining the next state) and finally producing the output

Question 3.

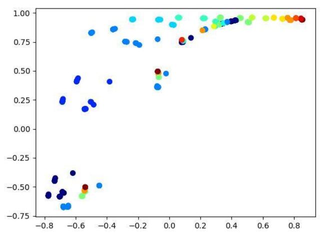
The goal of the task is to predict the probability of the next character given a prefix.

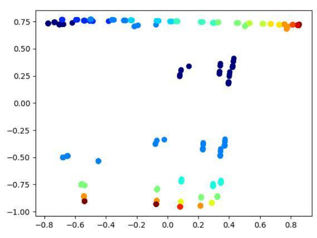
The model will give "self.H0" as the starting "X", use self.W as the weight of input to hidden layer. Then give self.U as weight of last hidden layer and self.hid_bias as its b, self.V is the weight of output, self.out_bias is the b of output.

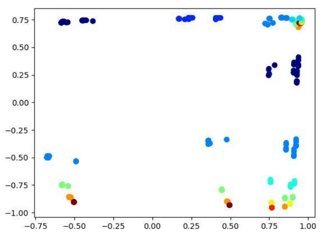
A "tanh" is a predicted pivot, new hidden data is the data c_t go through a tanh, and this C part is three parts: the product of input and its weight; the product of last hidden data and its weight and then plus the b. We will append all of these hidden, and use in next hidden. And the output is the product of all hidden and weight plus its b.

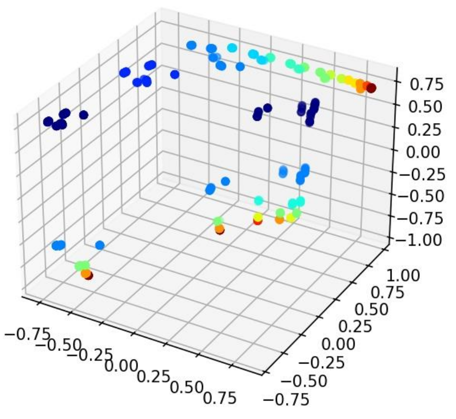
The hidden activations is updated with each character input. After multiple A inputs, the value of a certain direction of the hidden state gradually accumulates which allows the hidden state to indirectly remember the number of A. So after the first B, it knows how many B it need so that it can correctly predict all B after the first B. And after the number of B are already output, it knows now need A, so it can correctly predict the first letter A after the last B in the sequence.

Question 4.







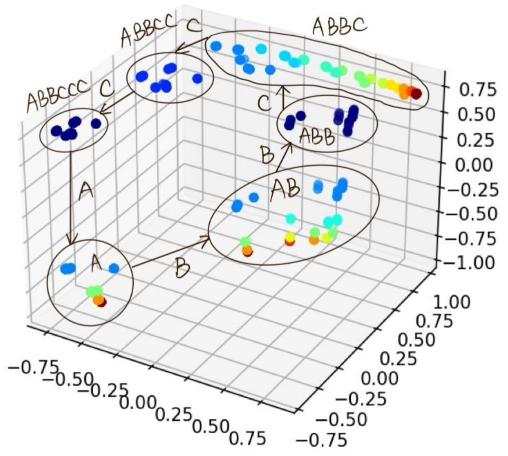


Question 5.

LSTM is very similar to SRN. Their process is to loop through a tanh hub, but LSTM adds four gates to control information, making it less susceptible to the impact of gradients and able to retain more important information of long-distance text.

These four gates will be used in c_t part, c_t = f_t * c_t + i_t * g_t, then go through a tanh as SRN do. Among them, i_t is used to control the input amount (whether to update the memory), f_t determines whether to keep the old memory, g_t controls the current written content and o_t controls the output. So the code is a little different in self.W, self.U, self.hid_bias parts because they need to give space to the four gates(num_hid * 4).

It remembers the number of A: Specifically, each time an A is read in, the LSTM's c_t will accumulate along a certain dimension. The gating mechanism ensures that the memory is updated only when A is seen. (i_t is turned on, content is written, f_t retains the previous memory, g_t is positive → causing c_t to grow). When reading the B sequence, it will output 2N B and remember whether the position of B has reached 2N. At this time, it will no longer write new information, but use the existing c_t to make judgments(f_t tend to 1, i_t tend to 0). Finally, when reading the C sequence, it will output 3N C and predict the start of the next A after the last C.



I simplified this process into 1n prediction and input of A, 2n prediction generation of B and 3n prediction generation of C, analyse the dynamics of the context units and how the input change.