z5567643

COMP9444 A1

Part 1: Japanese Character Recognition

1. My final accuracy was 70%, with an average test loss of 1.0091. The confusion matrix is shown below:

```
[767.   5.   7.  13.  32.  64.   2.  62.  29.  19.]
[  7. 666. 108.  17.  31.  24.  60.  13.  24.  50.]
[  8.  62. 692.  26.  24.  21.  46.  39.  45.  37.]
[  4.  35.  62. 758.  15.  56.  14.  18.  26.  12.]
[ 61.  51.  77.  19. 627.  18.  33.  34.  22.  58.]
[  8.  27. 125.  17.  18. 726.  28.   8.  32.  11.]
[  5.  25. 148.  11.  25.  23. 722.  20.   8.  13.]
[ 17.  29.  26.  11.  84.  15.  51. 629.  90.  48.]
[ 10.  36.  96.  41.   6.  31.  47.   7. 705.  21.]
[  8.  52.  86.   3.  54.  30.  20.  29.  38. 680.]
```

2. My final accuracy was 85%, with an average test loss of 0.4992. The model has 318,010 parameters: (28 * 28 * 400)+400 = 314,000 for the input to hidden layer, and (400 * 10) + 10 = 4,010 for the hidden to output layer. The confusion matrix is shown below:

```
[849.   4.   2.   6.  29.  36.   2.  39.  29.   4.]
[  7. 820.  35.   3.  18.  12.  54.   5.  18.  28.]
[  7.   9. 831.  46.  14.  18.  26.  11.  19.  19.]
[  3.   8.  24. 920.   2.  16.   7.   1.   9.  10.]
[ 37.  25.  16.   6. 822.   7.  30.  20.  22.  15.]
[  8.  13.  75.   9.  13. 841.  19.   1.  16.   5.]
[  3.  14.  42.   7.  15.   9. 892.   8.   1.   9.]
[ 13.  11.  19.   3.  24.  10.  32. 829.  26.  33.]
[ 11.  27.  26.  49.   2.   6.  30.   3. 838.   8.]
[  3.  20.  46.   5.  29.   4.  19.  15.  15. 844.]
```

3. My final accuracy was 94%, with an average test loss of 0.2393. My model has 269,322 parameters in total: (1 * 64 * 5 * 5) + 64 = 1,664 for the first convolutional layer, (64 * 128 * 5 * 5) + 128 = 204,928 for the second convolutional layer, and (128 * 7 * 7) + 10 = 62,730 for the fully connected layer. The confusion matrix is shown below:

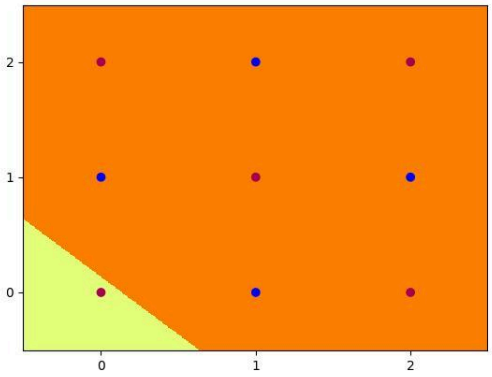
```
[965.   2.   2.   0.  18.   1.   0.   7.   2.   3.]
[  2. 919.   8.   1.  15.   0.  35.   5.   6.   9.]
[ 11.   7. 872.  44.  11.   7.  24.  10.   4.  10.]
[  1.   1.  19. 955.   6.   4.   5.   3.   2.   4.]
[ 18.   2.   5.  10. 926.   0.  13.   9.   9.   8.]
[  4.  12.  50.   4.   5. 894.  21.   6.   2.   2.]
[  4.   2.  16.   2.   8.   0. 964.   1.   1.   2.]
```

```
[ 11.   3.   3.   1.   6.   1.   7. 944.   5.  19.]
[  7.   9.   7.   0.   7.   1.   1.   1. 966.   1.]
[  6.   7.   3.   3.  11.   1.   0.   3.   3. 963.]
```

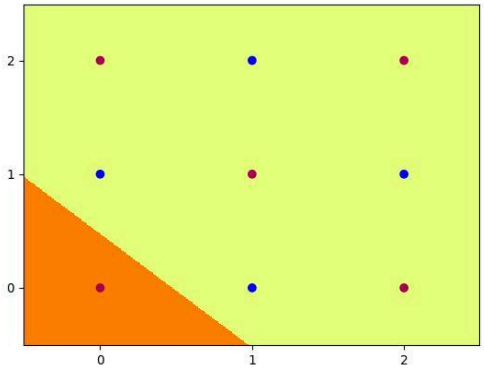
4. The accuracy of the three models roughly scales with the architecture's suitability for the task, rather than parameter size. The convolutional neural network somewhat unsurprisingly has by far the best results, in line with the fact that this architecture underpins most SOTA vision models. Between the other two, the fully connected outperforms the linear model, but at a significant parameter cost.

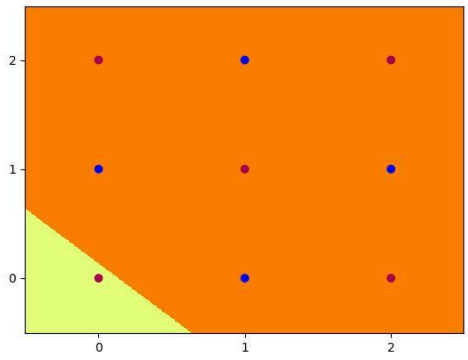
In terms of confusion matrices, the most common false classifications across all three models roughly line up with what humans might perceive as visually similar too. For example, two of the most commonly confused character pairs int the matrices are the pairing き (ki) and ま (ma), both of which share a long stem with two lines through it, and the triplet of す (su), ま (ma) and は (ha), all of which share a similar structure with a large circular loop. It's likely that the models have learnt features that represent these similarities, resulting in them more likely to be confused for one another.

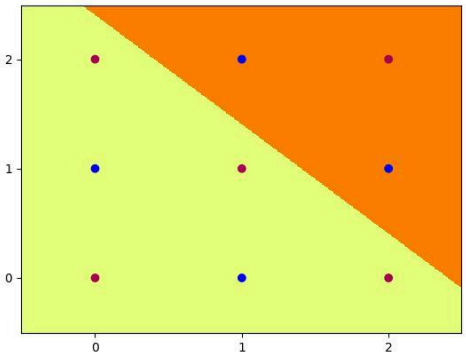
Part 2: Multi-Layer Perceptron

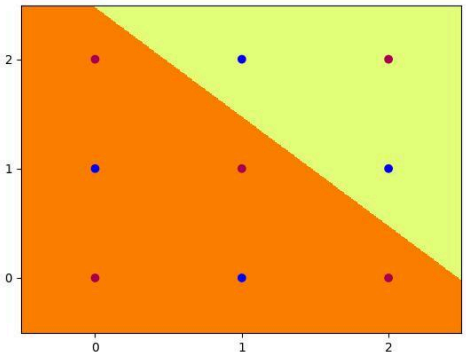


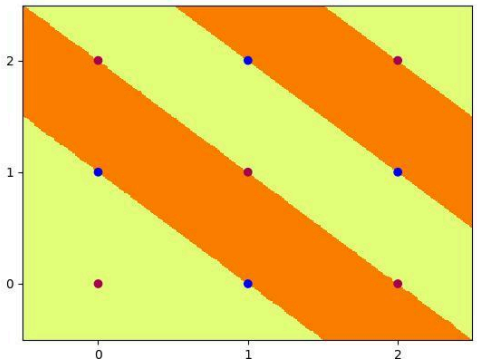
1. My results from running the network are below:











2. The diagram of my hand-crafted neural network is below. The key for me here was the observation that the red class appeared when x + y was even, and blue appeared when x + y was odd. The lines created in the hidden layer are as follows:

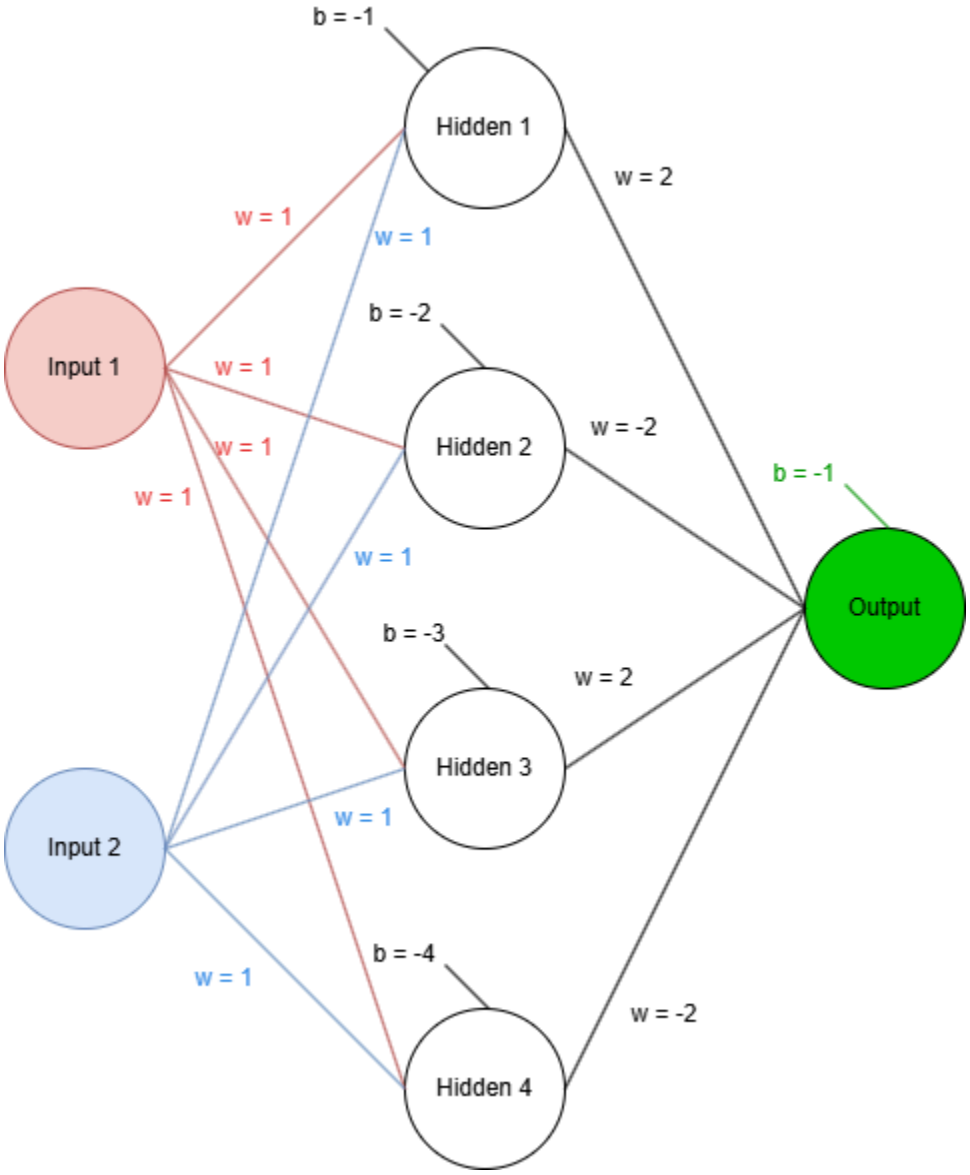
in1 + in2 = 1
in1 + in2 = 2
in1 + in2 = 3
in1 + in2 = 4

The activations of my hidden nodes are as follows:

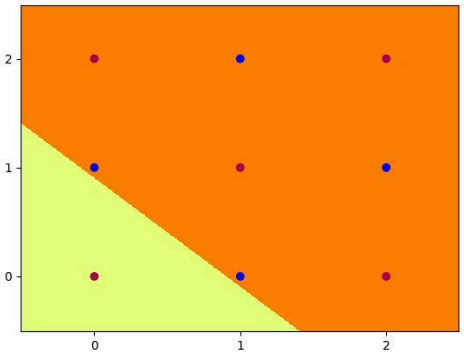
(x, y)	**h1**	**h2**	**h3**	**h4**	**output**
(0,0)	0	0	0	0	0
(1, 0)	1	0	0	0	1
(2, 0)	1	1	0	0	0
(0, 1)	1	0	0	0	1
(1, 1)	1	1	0	0	0
(2, 1)	1	1	1	0	1

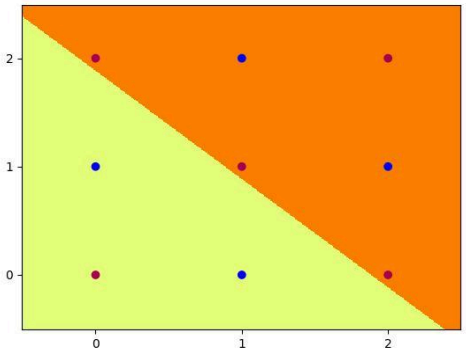
(0, 2)	1	1	0	0	0
(1, 2)	1	1	1	0	1
(2, 2)	1	1	1	1	0

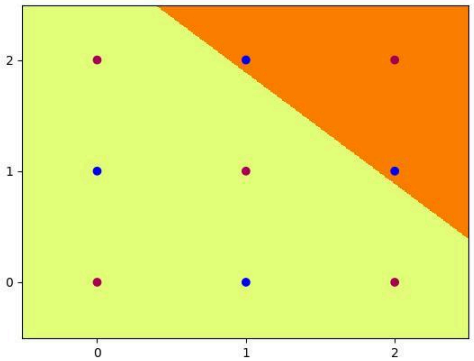
Finally, the diagram of my neural network is below:

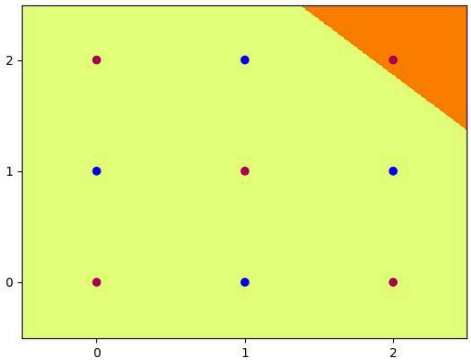


3. I scaled all my weights and biases by 10. The activations of each hidden node are as follows:

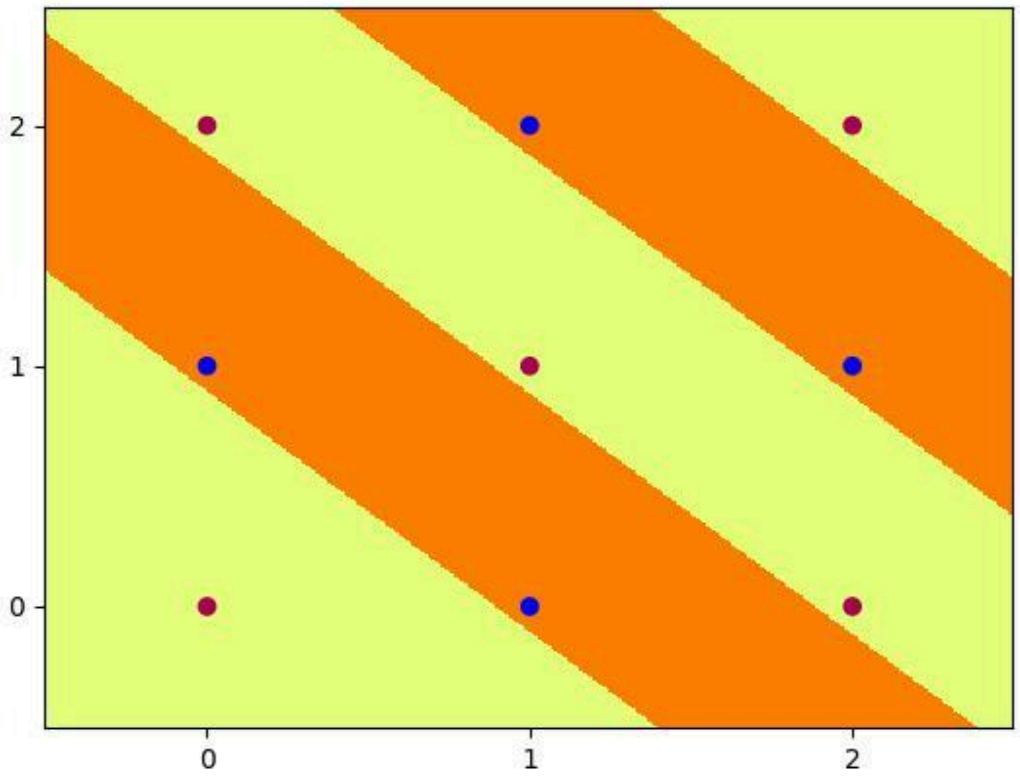

2
1
0
0
1
2


2
1
0
0
1
2


2
1
0
0
1
2

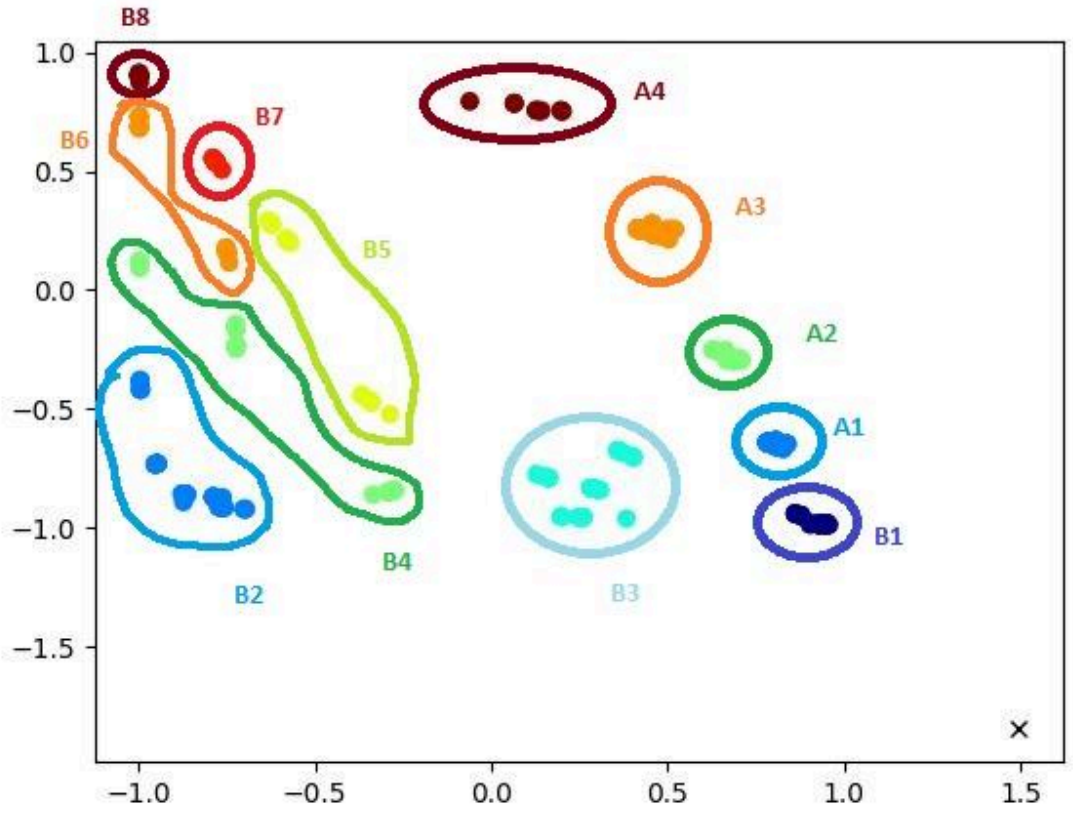

2
1
0
0
1
2

The final classification is below, achieving 100%:

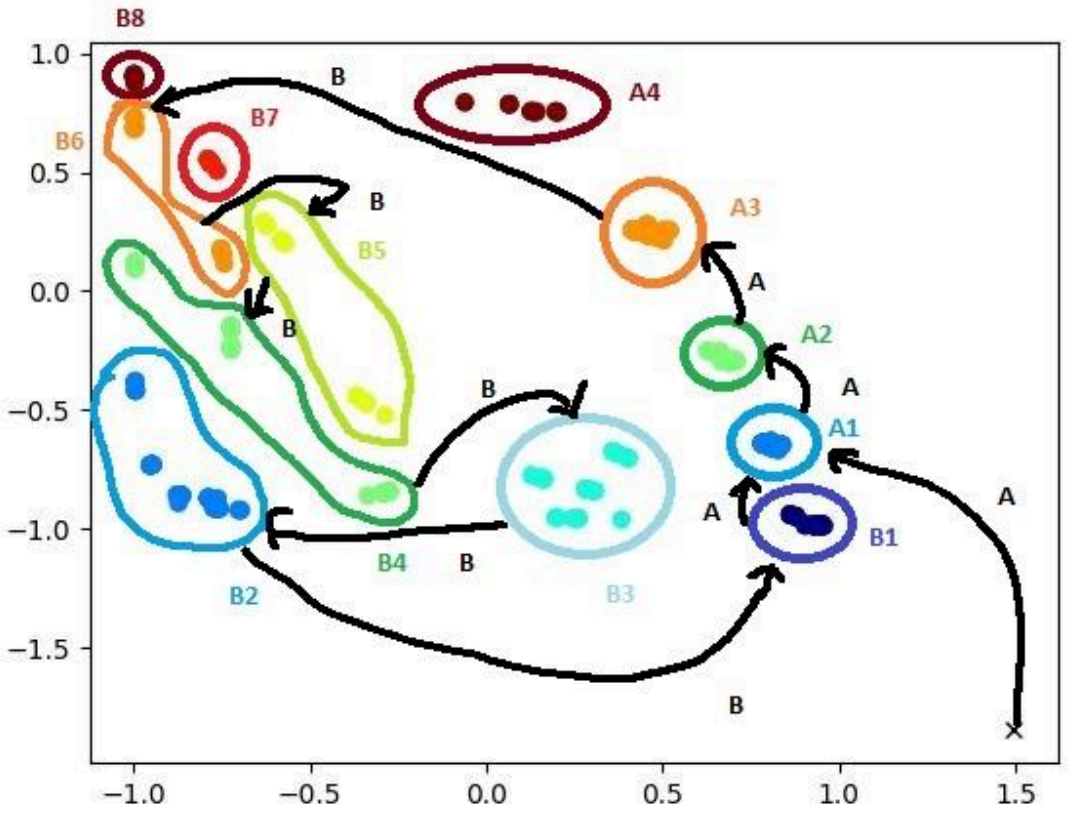


Part 3: Hidden Unit Dynamics for Recurrent Networks

1. For this question, I've done two separate annotations to show some interesting points. The boundaries I've drawn are below.

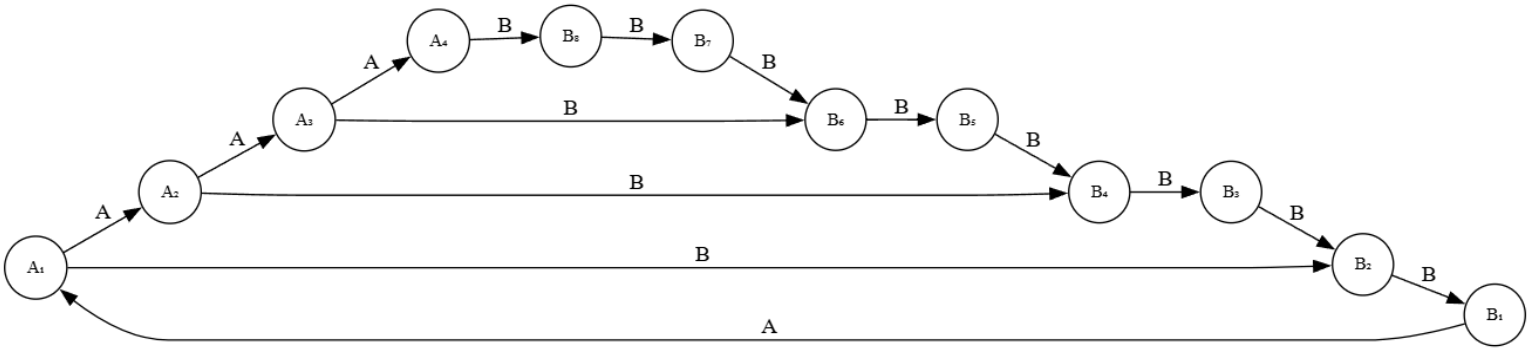


From my annotations you can see the states form a counter. When the first A appears, the state is A1. If there is a second A, the state transitions to A2. However, if there's no second A, it instead transitions to B2, with the '2' indicating that there are two Bs remaining in total. It then moves to B1 for the last B. The same pattern exists for higher numbers. For example, below I've annotated the path the pattern 'AAABBBBBB' takes through the hidden dimensions:



As we can see, it starts from A1, counts up to A3, then switches to B6, counting its way back down through the states.

2. The below is a finite state machine aligned with the transitions in my diagram:

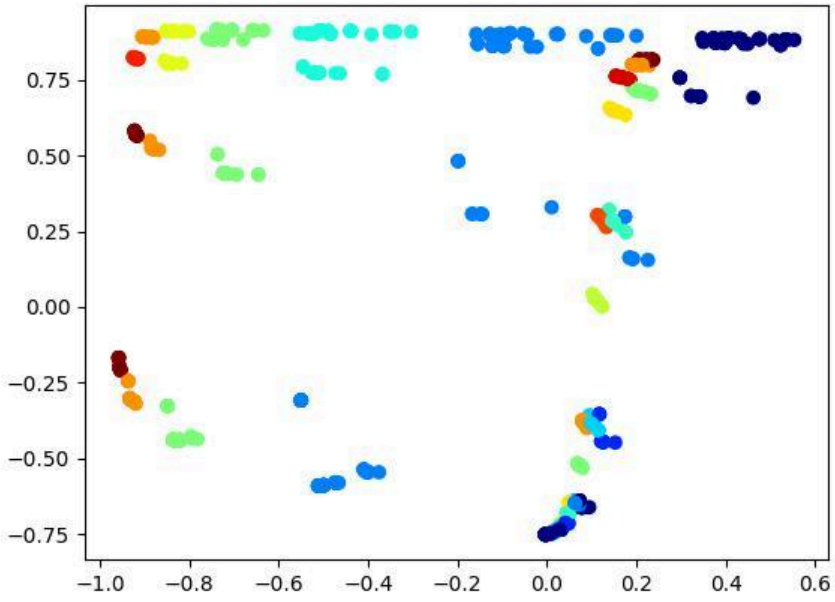


3. The network here seems to have effectively learnt a function in polar space with a radius and an angle (which I'll call *r* and *θ* respectively) radiating outwards from the initial state.*θ* acts as a pointer around what is effectively a loop in the hidden dimension, and the radius *r* encodes the count of each letter. To colour this a bit more I'll list out what's happening at each step:

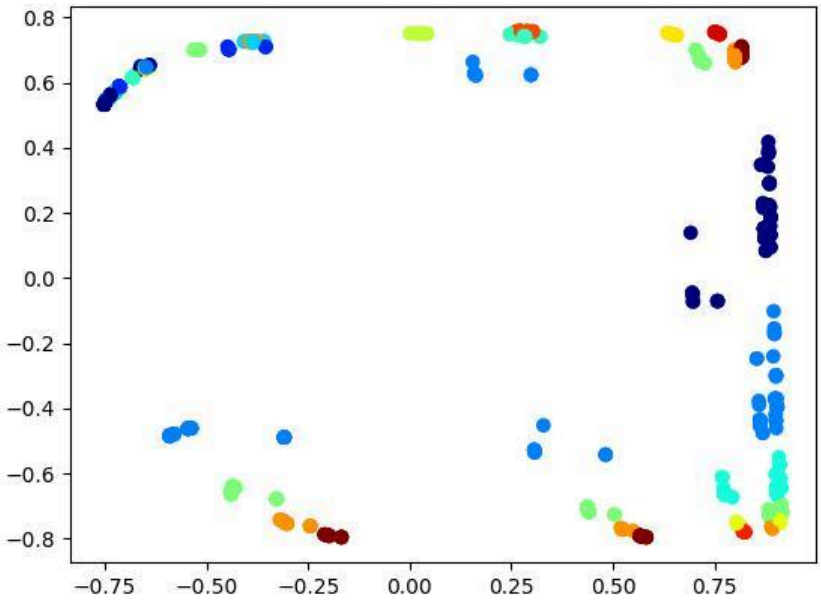
- In the A period (regardless of how many As), the hidden state makes a small, anti-clockwise transition up the right side of the graph each time there's another A. θ is roughly constant across this part, but we can see the number of As scales with the radius r from the initial state.
- When the network sees its first B, θ shifts anticlockwise harshly over to the lefthand side (generally with $h1=-1$). r now encodes how many Bs we have to count down to get back to A - you can see that B8 is very far away from the initial state, but B4 is about halfway down
- After this jump, r slowly shrinks for each B, and θ moves around a bit, but always staying to the left of the A group. This is a fairly predictable process aside from B3 being further to the right, still in the B loop, but out of order in terms of the counter. On top of that, the B3 state also has a lower probability of the next B, despite it being completely deterministic (it's the only state that shows this behaviour too). I don't think there is any obvious reason for this, potentially just an effect of the problem being very complex for two dimensions.
- Eventually we move back to B1, which transitions to the A1 group, and we repeat for the next anb2n pattern.

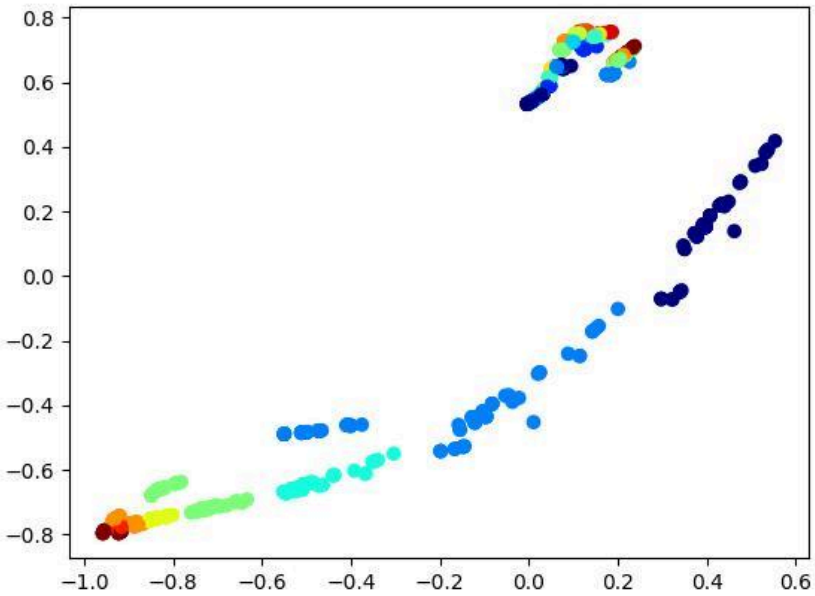
Because every hidden location on the B side maps to exactly one future location, the network's output layer can assign unit probability to the correct symbol for all B's after the first, and to the initial A that follows the last B.

4. Below I've enclosed the 3 generated images, as well as a screenshot of the 3D chart that I believe shows some of the patterns quite well



:

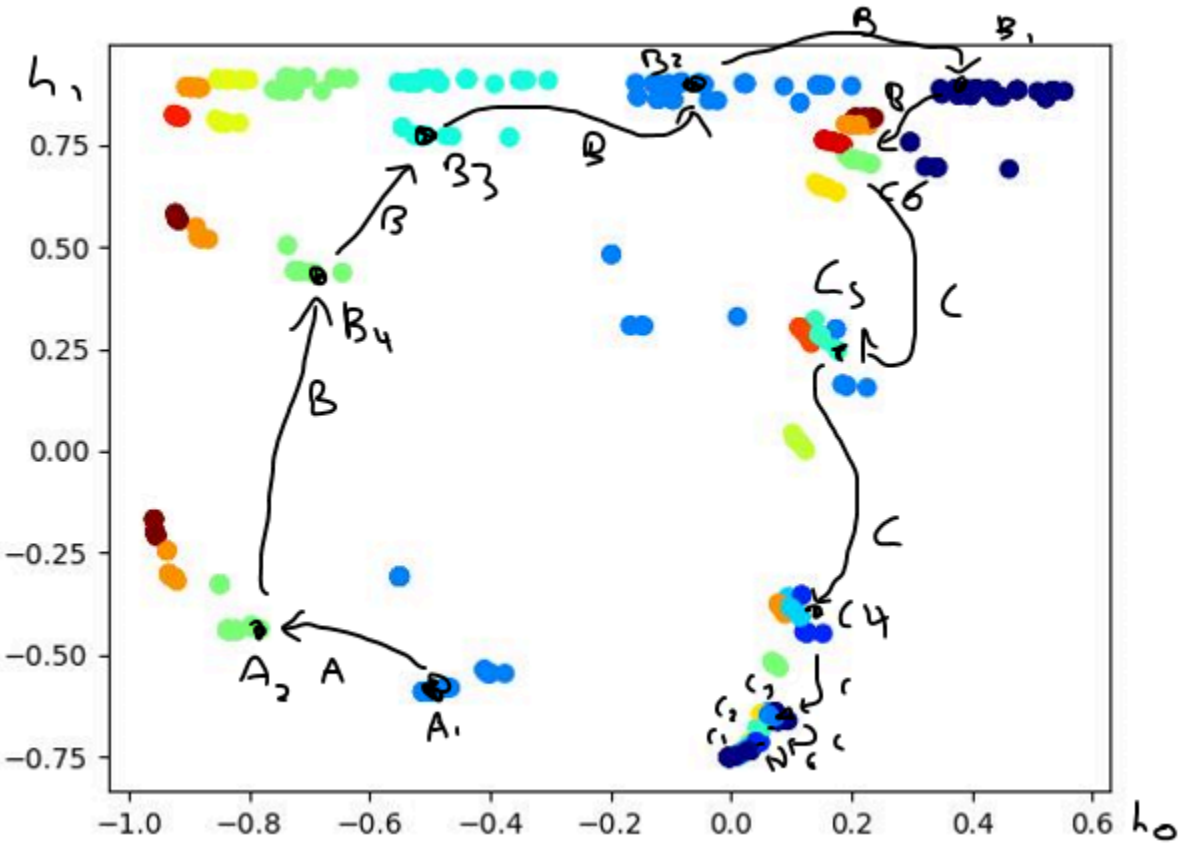


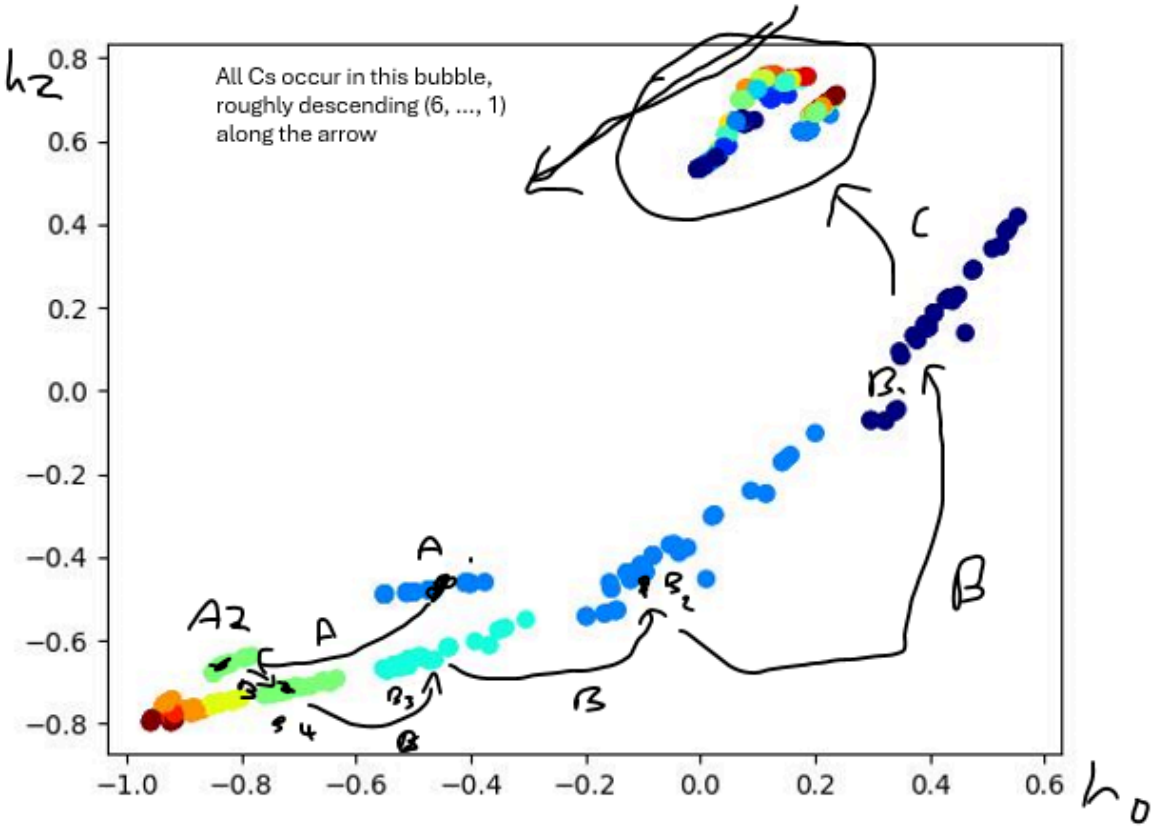


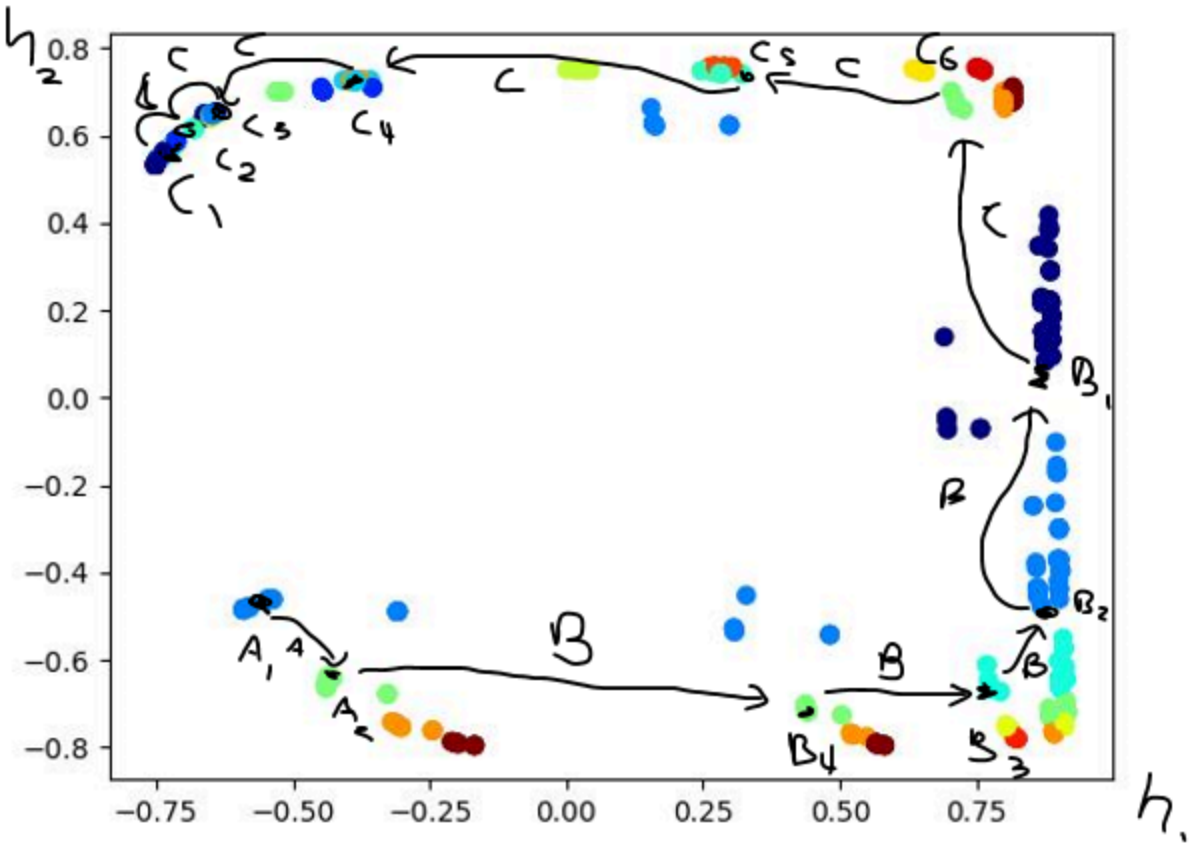
5. In this network, we're once again presented with a 'loop'-like structure that shows how the states change across the input string. Below, I've annotated the hidden activations of the string 'AABBBBCCCCCC', included here for context:

A [-0.51 -0.59 -0.48] [0.81 0.17 0.02]
A [-0.84 -0.44 -0.66] [0.59 0.41 0.]
B [-0.72 0.44 -0.72] [0.02 0.97 0.]

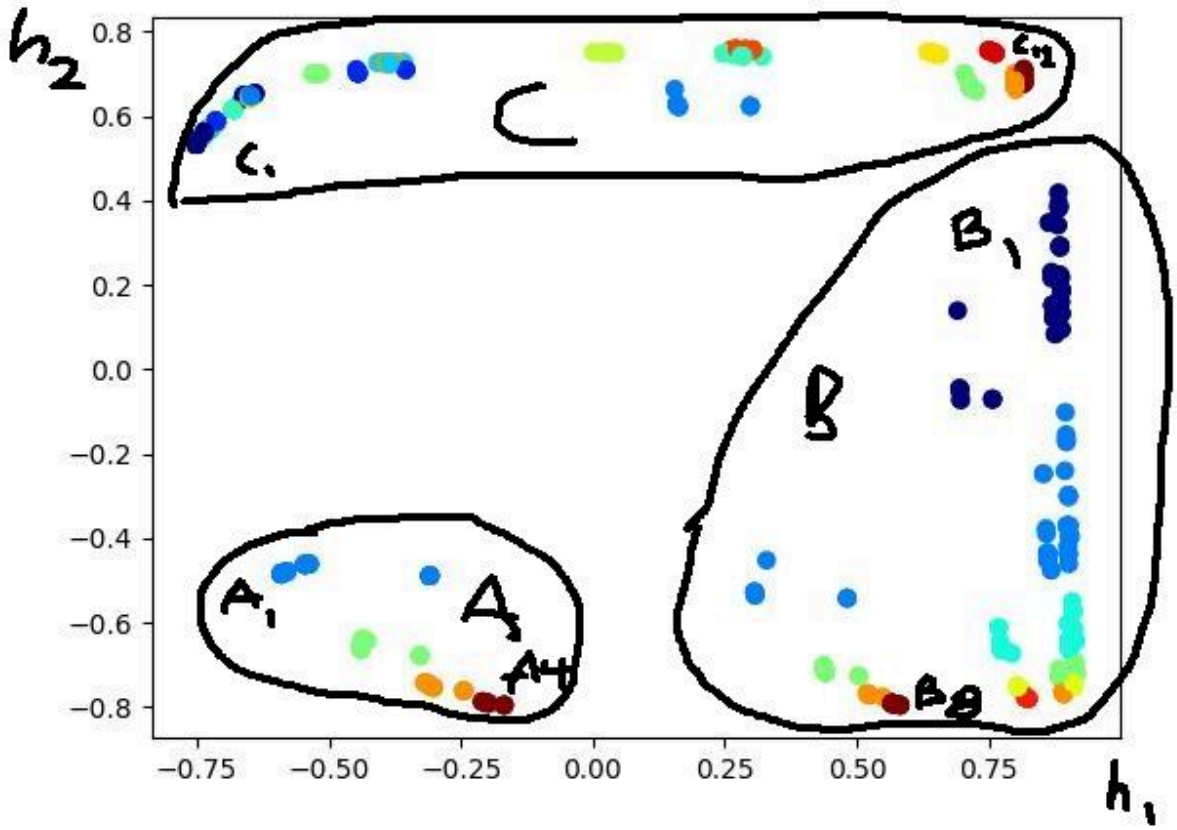
B [-0.53 0.77 -0.66] [0.01 0.99 0.]
B [-0.12 0.86 -0.45] [0.01 0.95 0.05]
B [0.38 0.87 0.12] [0.   0.14 0.86]
C [0.2 0.71 0.67] [0.   0.01 0.98]
C [0.15 0.28 0.74] [0.01 0.01 0.99]
C [0.1 -0.38 0.72] [0.05 0.   0.95]
C [0.06 -0.65 0.65] [0.13 0.   0.86]
C [0.04 -0.71 0.59] [0.2 0. 0.8]
C [0.03 -0.73 0.56] [0.23 0.   0.77]




All Cs occur in this bubble,
roughly descending (6, ..., 1)
along the arrow
h2
h0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
A
A2
A
B
B
B
C



I've also included the below diagram showing the segmentation of the classes. These are visible on all 3 pairs of axes, but I think y & z (i.e. hidden dimensions 1 and 2, 0 omitted) show it best:



Note that here I'm using the same notations (and as I'll explain later, the same logic) as the previous question: The first A of the sequence is A1, ascending up to A4. Then, the B annotations start with a big number and descend to 1. The same goes for C, starting from a maximum possible value of C12 (if A = 4), descending down to C1 again.

With that in mind, we have a fairly similar thing happening here to the previous question:

- The hidden states map out a circuit-like path, moving anti-clockwise at each state change
- When a letter change occurs, the state 'jumps' between groups

It seems once again like the model, despite being a different architecture, has also learnt some sort of function mapping a radius and an angle, but this time in 3D space. Overall, the LSTM has created a counter where the direction around the loop represents what letter we're currently looking at, and the distance seems to represent the counter.