z5270707   Yuemeng Yin

COMP9444 Assignment 1

Part 1: Japanese Character Recognition

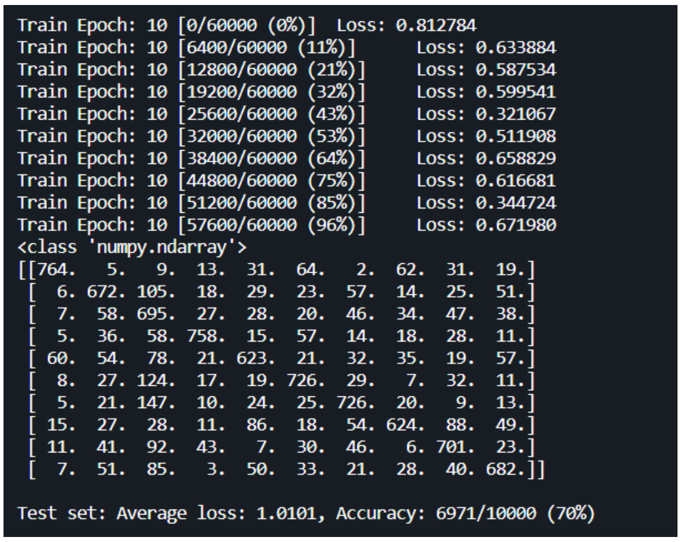
Part 1.1: Final accuracy and confusion matrix for NetLin model

Confusion matrix:

```
[[764.   5.   9.  13.  31.  64.   2.  62.  31.  19.]
 [  6. 672. 105.  18.  29.  23.  57.  14.  25.  51.]
 [  7.  58. 695.  27.  28.  20.  46.  34.  47.  38.]
 [  5.  36.  58. 758.  15.  57.  14.  18.  28.  11.]
 [ 60.  54.  78.  21. 623.  21.  32.  35.  19.  57.]
 [  8.  27. 124.  17.  19. 726.  29.   7.  32.  11.]
 [  5.  21. 147.  10.  24.  25. 726.  20.   9.  13.]
 [ 15.  27.  28.  11.  86.  18.  54. 624.  88.  49.]
 [ 11.  41.  92.  43.   7.  30.  46.   6. 701.  23.]
 [  7.  51.  85.   3.  50.  33.  21.  28.  40. 682.]]
```

Final accuracy:

Test set: Average loss: 1.0101, Accuracy: 6971/10000 (70%)



```
Train Epoch: 10 [0/60000 (0%)]	Loss: 0.812784
Train Epoch: 10 [6400/60000 (11%)]	Loss: 0.633884
Train Epoch: 10 [12800/60000 (21%)]	Loss: 0.587534
Train Epoch: 10 [19200/60000 (32%)]	Loss: 0.599541
Train Epoch: 10 [25600/60000 (43%)]	Loss: 0.321067
Train Epoch: 10 [32000/60000 (53%)]	Loss: 0.511908
Train Epoch: 10 [38400/60000 (64%)]	Loss: 0.658829
Train Epoch: 10 [44800/60000 (75%)]	Loss: 0.616681
Train Epoch: 10 [51200/60000 (85%)]	Loss: 0.344724
Train Epoch: 10 [57600/60000 (96%)]	Loss: 0.671980
<class 'numpy.ndarray'>
[[764.   5.   9.  13.  31.  64.   2.  62.  31.  19.]
 [  6. 672. 105.  18.  29.  23.  57.  14.  25.  51.]
 [  7.  58. 695.  27.  28.  20.  46.  34.  47.  38.]
 [  5.  36.  58. 758.  15.  57.  14.  18.  28.  11.]
 [ 60.  54.  78.  21. 623.  21.  32.  35.  19.  57.]
 [  8.  27. 124.  17.  19. 726.  29.   7.  32.  11.]
 [  5.  21. 147.  10.  24.  25. 726.  20.   9.  13.]
 [ 15.  27.  28.  11.  86.  18.  54. 624.  88.  49.]
 [ 11.  41.  92.  43.   7.  30.  46.   6. 701.  23.]
 [  7.  51.  85.   3.  50.  33.  21.  28.  40. 682.]]

Test set: Average loss: 1.0101, Accuracy: 6971/10000 (70%)
```

z5270707   Yuemeng Yin

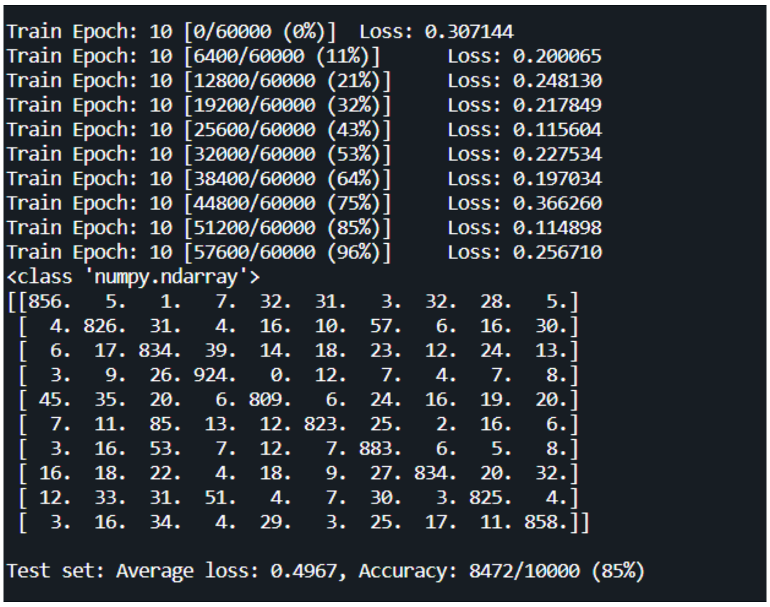
Part 1.2: Final accuracy and confusion matrix for NetFull model

Confusion matrix:

```
[[856.   5.   1.   7.  32.  31.   3.  32.  28.   5.]
 [  4. 826.  31.   4.  16.  10.  57.   6.  16.  30.]
 [  6.  17. 834.  39.  14.  18.  23.  12.  24.  13.]
 [  3.   9.  26. 924.   0.  12.   7.   4.   7.   8.]
 [ 45.  35.  20.   6. 809.   6.  24.  16.  19.  20.]
 [  7.  11.  85.  13.  12. 823.  25.   2.  16.   6.]
 [  3.  16.  53.   7.  12.   7. 883.   6.   5.   8.]
 [ 16.  18.  22.   4.  18.   9.  27. 834.  20.  32.]
 [ 12.  33.  31.  51.   4.   7.  30.   3. 825.   4.]
 [  3.  16.  34.   4.  29.   3.  25.  17.  11. 858.]]
```

Final accuracy:

Test set: Average loss: 0.4967, Accuracy: 8472/10000 (85%)



```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.307144
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.200065
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.248130
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.217849
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.115604
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.227534
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.197034
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.366260
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.114898
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.256710
<class 'numpy.ndarray'>
[[856.   5.   1.   7.  32.  31.   3.  32.  28.   5.]
 [  4. 826.  31.   4.  16.  10.  57.   6.  16.  30.]
 [  6.  17. 834.  39.  14.  18.  23.  12.  24.  13.]
 [  3.   9.  26. 924.   0.  12.   7.   4.   7.   8.]
 [ 45.  35.  20.   6. 809.   6.  24.  16.  19.  20.]
 [  7.  11.  85.  13.  12. 823.  25.   2.  16.   6.]
 [  3.  16.  53.   7.  12.   7. 883.   6.   5.   8.]
 [ 16.  18.  22.   4.  18.   9.  27. 834.  20.  32.]
 [ 12.  33.  31.  51.   4.   7.  30.   3. 825.   4.]
 [  3.  16.  34.   4.  29.   3.  25.  17.  11. 858.]]

Test set: Average loss: 0.4967, Accuracy: 8472/10000 (85%)
```

z5270707 Yuemeng Yin

Calculation of the total number of independent parameters in the NetFull network:

Since we set the number of hidden nodes as 250, we calculate the total number of independent parameters as following:

Total number of independent parameters = (number of inputs + bias) * number of hidden nodes + (number of hidden nodes + bias) *number of outputs = (784+1)*250 + (250+1)*10 = 198760

Thus, the total number of independent parameters in this network is **198760**.

z5270707 Yuemeng Yin

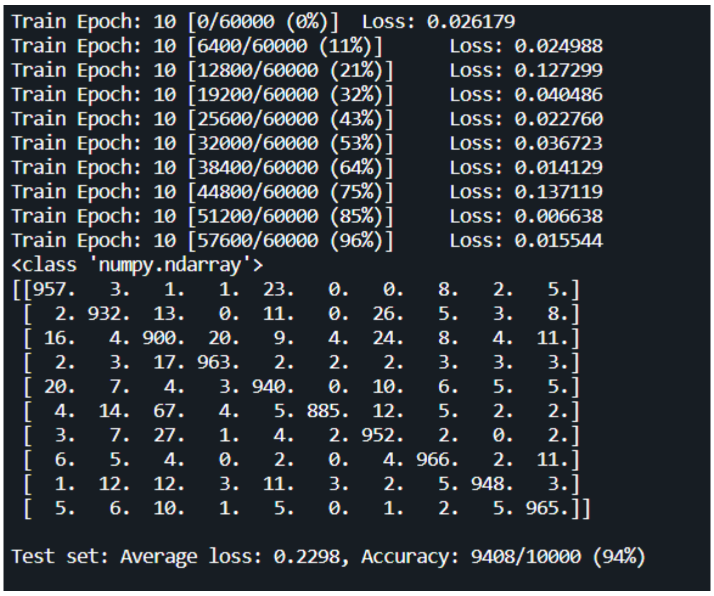
Part 1.3: Final accuracy and confusion matrix for NetConv model

Confusion matrix:

```
[[957.   3.   1.   1.  23.   0.   0.   8.   2.   5.]
 [  2. 932.  13.   0.  11.   0.  26.   5.   3.   8.]
 [ 16.   4. 900.  20.   9.   4.  24.   8.   4.  11.]
 [  2.   3.  17. 963.   2.   2.   2.   3.   3.   3.]
 [ 20.   7.   4.   3. 940.   0.  10.   6.   5.   5.]
 [  4.  14.  67.   4.   5. 885.  12.   5.   2.   2.]
 [  3.   7.  27.   1.   4.   2. 952.   2.   0.   2.]
 [  6.   5.   4.   0.   2.   0.   4. 966.   2.  11.]
 [  1.  12.  12.   3.  11.   3.   2.   5. 948.   3.]
 [  5.   6.  10.   1.   5.   0.   1.   2.   5. 965.]]
```

Final accuracy:

Test set: Average loss: 0.2298, Accuracy: 9408/10000 (94%)


```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.026179
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.024988
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.127299
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.040486
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.022760
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.036723
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.014129
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.137119
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.006638
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.015544
<class 'numpy.ndarray'>
[[957.   3.   1.   1.  23.   0.   0.   8.   2.   5.]
 [  2. 932.  13.   0.  11.   0.  26.   5.   3.   8.]
 [ 16.   4. 900.  20.   9.   4.  24.   8.   4.  11.]
 [  2.   3.  17. 963.   2.   2.   2.   3.   3.   3.]
 [ 20.   7.   4.   3. 940.   0.  10.   6.   5.   5.]
 [  4.  14.  67.   4.   5. 885.  12.   5.   2.   2.]
 [  3.   7.  27.   1.   4.   2. 952.   2.   0.   2.]
 [  6.   5.   4.   0.   2.   0.   4. 966.   2.  11.]
 [  1.  12.  12.   3.  11.   3.   2.   5. 948.   3.]
 [  5.   6.  10.   1.   5.   0.   1.   2.   5. 965.]]

Test set: Average loss: 0.2298, Accuracy: 9408/10000 (94%)
```

z5270707   Yuemeng Yin

Calculation of the total number of independent parameters in the NetConv network:

The network has two convolutional layers, one fully connected layer and the output layer.

The first convolutional layer has 1 in channels, 32 out channels, 5 as kernal size; the other convolutional layer has 32 in channels, 128 out channels, 5 as kernal size. The fully connected layer has 2048 inputs, 512 outputs; the output layer has 512 inputs, 10 outputs.

we calculate the total number of independent parameters as following:

Total number of independent parameters = (kernal size ^2 * number of in channels + 1) * number of out channels + (kernal size ^2 * number of in channels + 1) * number of out channels + (number of inputs + bias) * number of hidden nodes + (number of hidden nodes + bias) *number of outputs = (5^2 * 1 + 1) *32 + (5^2 *32 +1) *128 + (2048+1)*512 + (512+1)*10 = 832 + 102528 + 1049088 + 5130 = 1157578

Thus, the total number of independent parameters in this network is **1157578**.

z5270707   Yuemeng Yin

Part 1.4: Briefly discuss the following points:

a) the relative accuracy of the three models,

As can be observed from the above results, the NetConv model (with two convolutional layers and one fully connected layer achieved the best performance), with accuracy of **94%**. The next to NetConv is the NetFull model (with two fully connected tanh layers), which has accuracy of **85%**. Then the worst performing model compared to others is the NetLin model (with linear function followed by log_softmax), which has accuracy of **70%**.

In summary, the NetConv model outperformed the NetFull and NetLin models for Japanese character recognition. The high accuracy of the NetConv model is due to the use of convolutional layers for capturing visual patterns and features effectively through filters. As the aim of the model is to classify Japanese characters written in various styles, NetConv model would fit the purpose better.

The NetFull model has 2 fully connected layers. It shows better accuracy over the NetLin model due to its ability to capture more complex relationships and learn non-linear mappings between input and output (using non-linear activation function tanh). However, it may struggle to capture complex visual features (curves, strokes, etc) and patterns as NetConv does, leading to lower accuracy compared to NetConv. The NetLin model which uses linear function and a simple activation function, is not suitable for capture the complex features of Japanese characters and classify them linearly into different characters, resulting in the lowest accuracy among the three models.

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

The number of independent parameters in the NetLin, NetFull, NetConv models are 7850 ((784+1) *10), 198760, and 1157578 respectively.

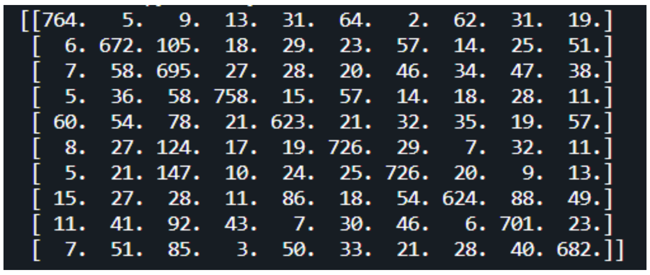
The NetLin model has 7850 independent parameters. This can be calculated by considering the input layer with 784 units (the input size is 28x28 pixels), plus one bias term (for each of the 10 output classes) in the fully connected layer. The calculation of two other models had been explained in pervious questions. Overall, the numbers of independent parameters in these models are closely related to their capacity to learn complex representations and relationships.

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

As described in the question, the rows of the confusion matrix indicate the target character, while the columns indicate the one chosen by the network. (0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo"). We analyse the confusion matrix for each model below.

z5270707   Yuemeng Yin

NetLin model:



```
[[764.   5.   9.  13.  31.  64.   2.  62.  31.  19.]
 [  6. 672. 105.  18.  29.  23.  57.  14.  25.  51.]
 [  7.  58. 695.  27.  28.  20.  46.  34.  47.  38.]
 [  5.  36.  58. 758.  15.  57.  14.  18.  28.  11.]
 [ 60.  54.  78.  21. 623.  21.  32.  35.  19.  57.]
 [  8.  27. 124.  17.  19. 726.  29.   7.  32.  11.]
 [  5.  21. 147.  10.  24.  25. 726.  20.   9.  13.]
 [ 15.  27.  28.  11.  86.  18.  54. 624.  88.  49.]
 [ 11.  41.  92.  43.   7.  30.  46.   6. 701.  23.]
 [  7.  51.  85.   3.  50.  33.  21.  28.  40. 682.]]
```

In this confusion matrix, we observe from diagonal element (row 2, column 2) that the character “su” is correctly recognized with count of 695. This character is most often mistaken for the character "ma" and “ha” with a count of mistaken times 147 (row 2, column 6) and 125 (row 2, column 5) respectively. This confusion could be due to similarities in the strokes and structure of "su", "ma" and “ha”, as can be seen in the following picture: 10 classes of Kuzushiji-MNIST. They all have a vertical line and a horizontal stroke pattern.

Moreover, the character “re” is most often mistaken for “ya”, with a count of 88 (row 8, column 7) as shown in matrix. The reason as can be observed from the figure below, could be due to their shared components, such as the short vertical line and the curved stroke. These shared components can make it difficult to differentiate between the two characters, especially in handwritten or stylized forms.

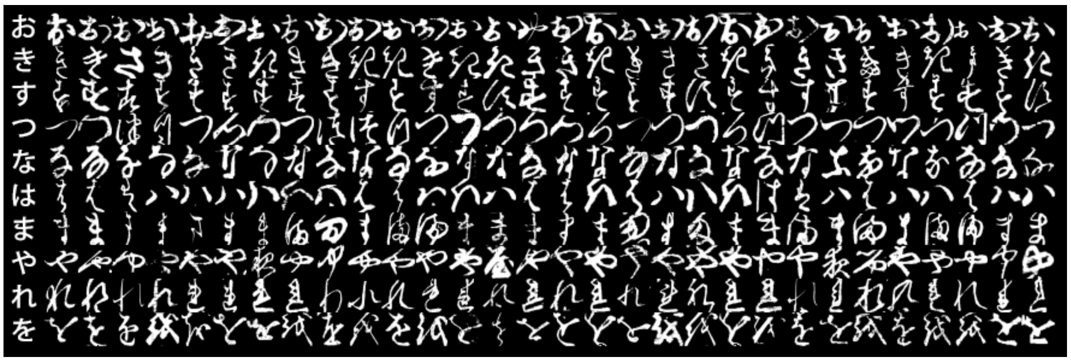
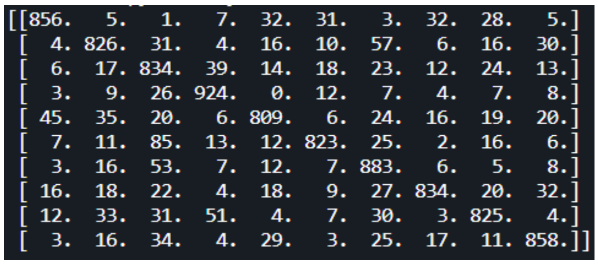


Figure cited from: https://arxiv.org/pdf/1812.01718.pdf

z5270707   Yuemeng Yin

NetFull model:



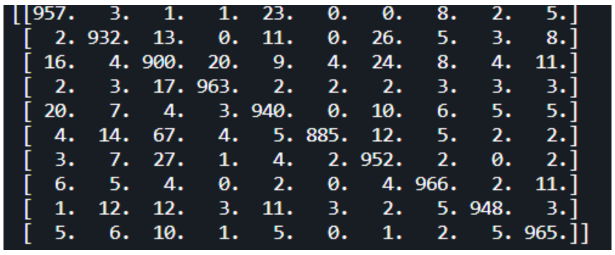
```
[[856.   5.   1.   7.  32.  31.   3.  32.  28.   5.]
 [  4. 826.  31.   4.  16.  10.  57.   6.  16.  30.]
 [  6.  17. 834.  39.  14.  18.  23.  12.  24.  13.]
 [  3.   9.  26. 924.   0.  12.   7.   4.   7.   8.]
 [ 45.  35.  20.   6. 809.   6.  24.  16.  19.  20.]
 [  7.  11.  85.  13.  12. 823.  25.   2.  16.   6.]
 [  3.  16.  53.   7.  12.   7. 883.   6.   5.   8.]
 [ 16.  18.  22.   4.  18.   9.  27. 834.  20.  32.]
 [ 12.  33.  31.  51.   4.   7.  30.   3. 825.   4.]
 [  3.  16.  34.   4.  29.   3.  25.  17.  11. 858.]]
```

In this confusion matrix, we observe through diagonal elements that the accuracy improved greatly compared to NetLin model. The character “su” is still most often mistaken for the character "ha" and “ma” with a count of mistaken times 85 (row 2, column 5) and 53 (row 2, column 6) respectively. They have a similar stroke pattern, especially the horizontal stroke component could be easily mistaken in some written styles.

Moreover, the character “ma” is often mistaken for “ki”, as indicated by the count of 57 (row 6, column 1). This is due to similar visual features and structures of these Japanese characters in certain writing styles.

z5270707   Yuemeng Yin

NetConv model:

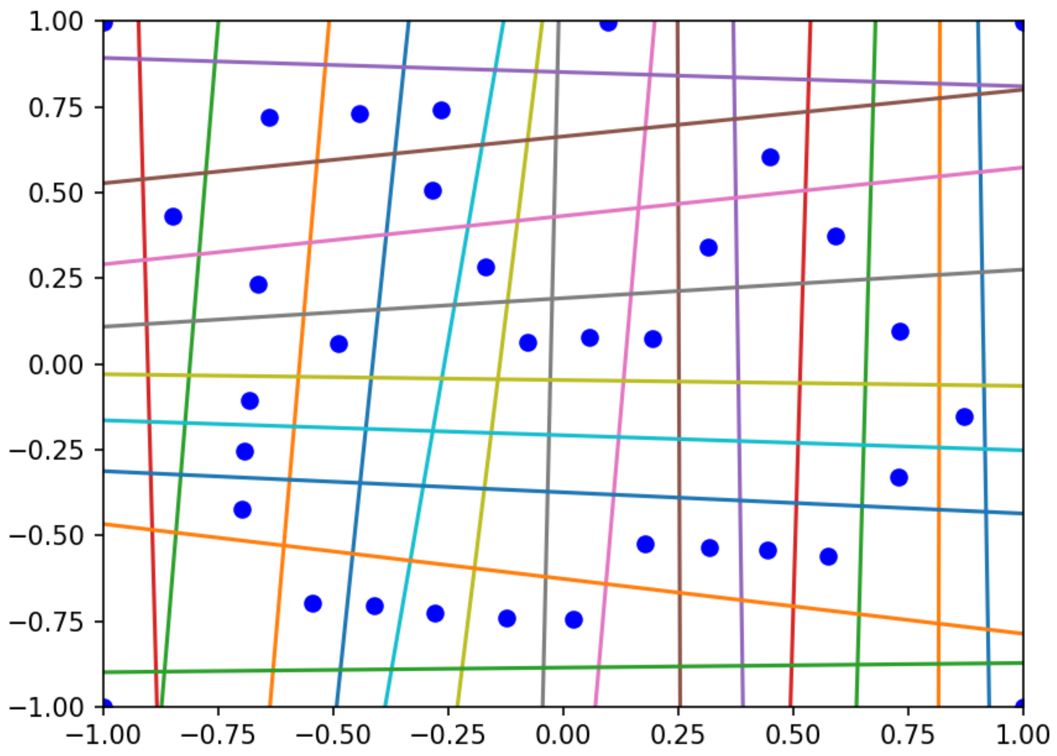


```
[[957.   3.   1.   1.  23.   0.   0.   8.   2.   5.]
 [  2. 932.  13.   0.  11.   0.  26.   5.   3.   8.]
 [ 16.   4. 900.  20.   9.   4.  24.   8.   4.  11.]
 [  2.   3.  17. 963.   2.   2.   2.   3.   3.   3.]
 [ 20.   7.   4.   3. 940.   0.  10.   6.   5.   5.]
 [  4.  14.  67.   4.   5. 885.  12.   5.   2.   2.]
 [  3.   7.  27.   1.   4.   2. 952.   2.   0.   2.]
 [  6.   5.   4.   0.   2.   0.   4. 966.   2.  11.]
 [  1.  12.  12.   3.  11.   3.   2.   5. 948.   3.]
 [  5.   6.  10.   1.   5.   0.   1.   2.   5. 965.]]
```

In this confusion matrix, we observe that the performance significantly improved from previous models. There is one pair of characters that is still often mistakenly classified. The character “su” is still often mistaken for the character “ha”, as indicated by the count of 67 misclassifications (row 2, column 5). The reason why these 2 characters are often mistakenly classified has been explained in above 2 sections.

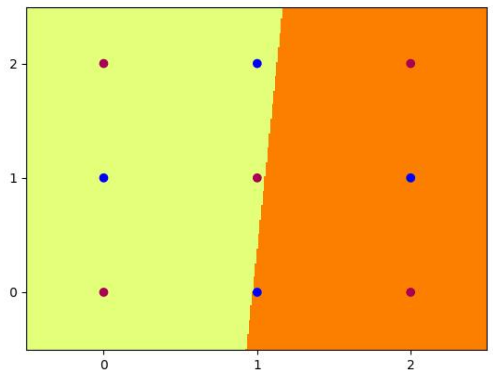
Part 2: Encoder Networks

The image produced:

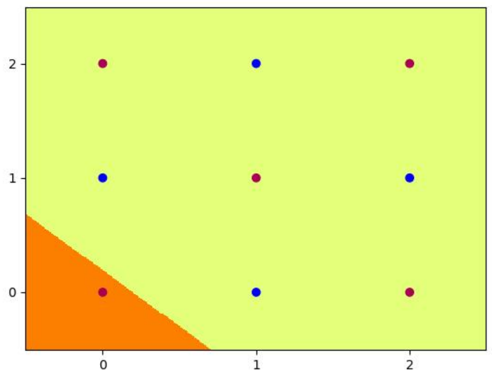


Part 2: Multi-Layer Perceptron

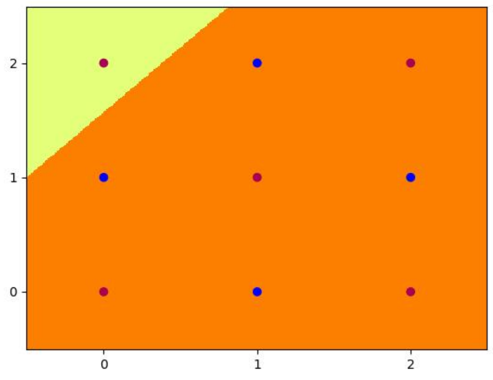
Part 2, Step 1. `python3 check_main.py --act sig --hid 5`



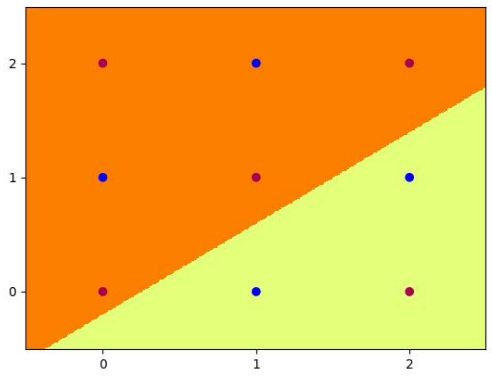
Hidden Unit 1



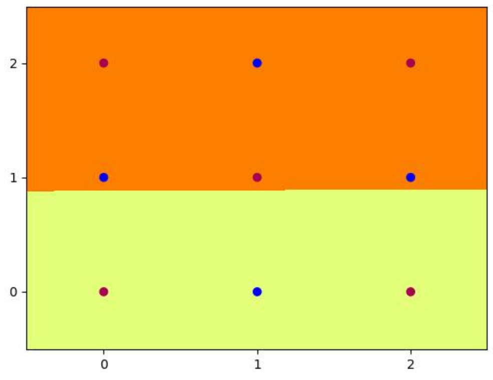
Hidden Unit 2



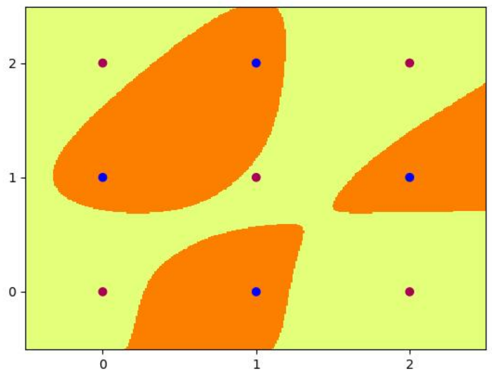
Hidden Unit 3



Hidden Unit 4

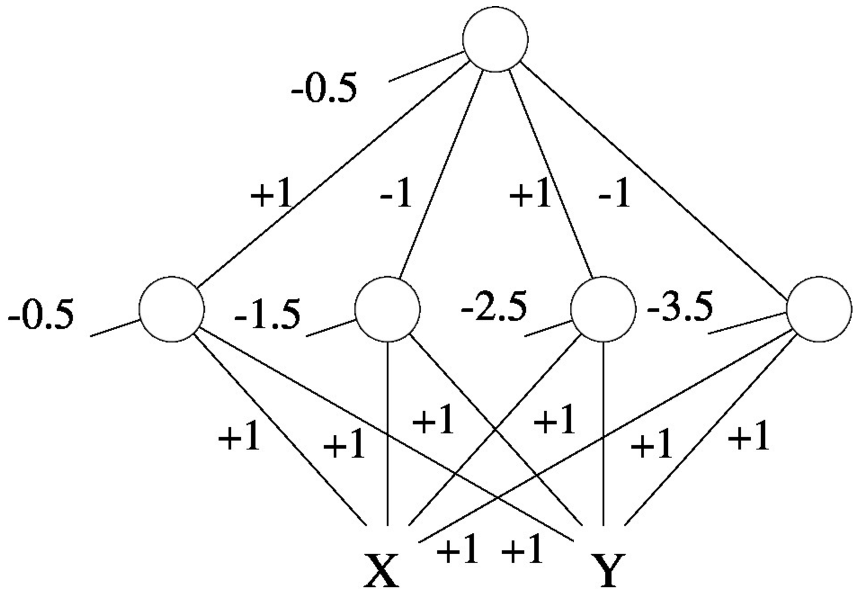


Hidden Unit 5



Output Unit

Part 2, Step 2.
Design by hand a 2-layer neural network with 4 hidden nodes, using the Heaviside (step) activation function at both the hidden and output layer, which correctly classifies the data.



Write the equations for the dividing line determined by each hidden node.

Hidden Unit 1:	$X + Y - 0.5 > 0$
Hidden Unit 2:	$X + Y - 1.5 > 0$
Hidden Unit 3:	$X + Y - 2.5 > 0$
Hidden Unit 4:	$X + Y - 3.5 > 0$

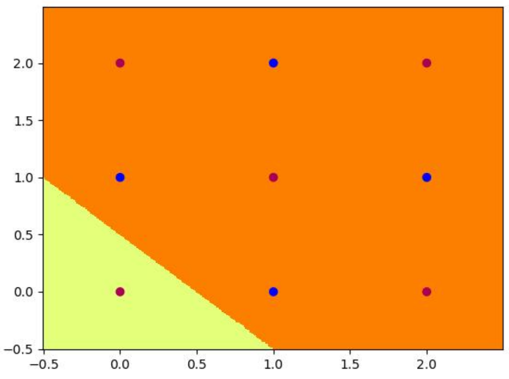
Create a table showing the activations of all the hidden nodes and the output node, for each of the 9 training items, and include it in your report.

X	Y	H1	H2	H3	H4	Out
0	0	0	0	0	0	0
0	1	1	0	0	0	1
0	2	1	1	0	0	0
1	0	1	0	0	0	1
1	1	1	1	0	0	0
1	2	1	1	1	0	1
2	0	1	1	0	0	0
2	1	1	1	1	0	1
2	2	1	1	1	1	0

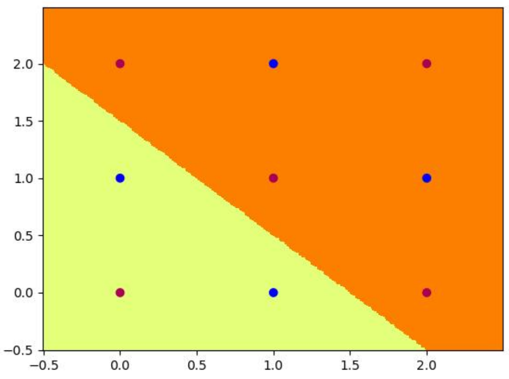
Part 2, Step 3.
Now rescale your hand-crafted weights from Part 2 by multiplying all of them by a large (fixed) number (for example, 10) so that the combination of rescaling followed by sigmoid will mimic the effect of the step function.

```
in_hid_weight  = [[10,10],[10,10],[10,10],[10,10]]
hid_bias       = [-5,-15,-25,-35]
hid_out_weight = [[10,-10,10,-10]]
out_bias       = [-5]
```

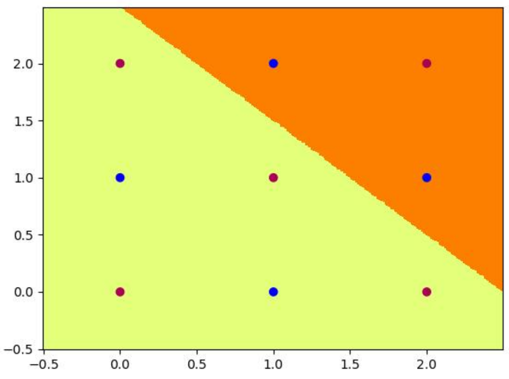
```
python3 check_main.py --act sig --hid 4 --set_weights
```



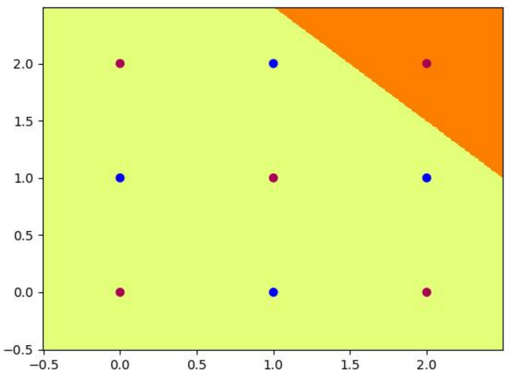
Hidden Unit 1



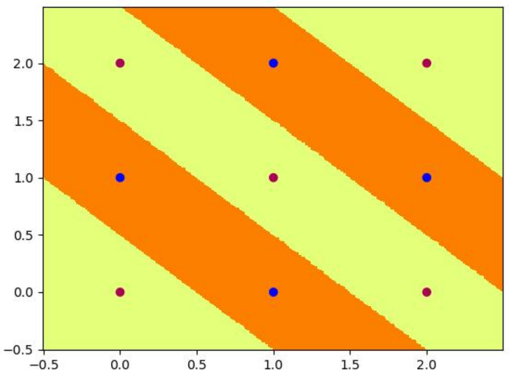
Hidden Unit 2



Hidden Unit 3



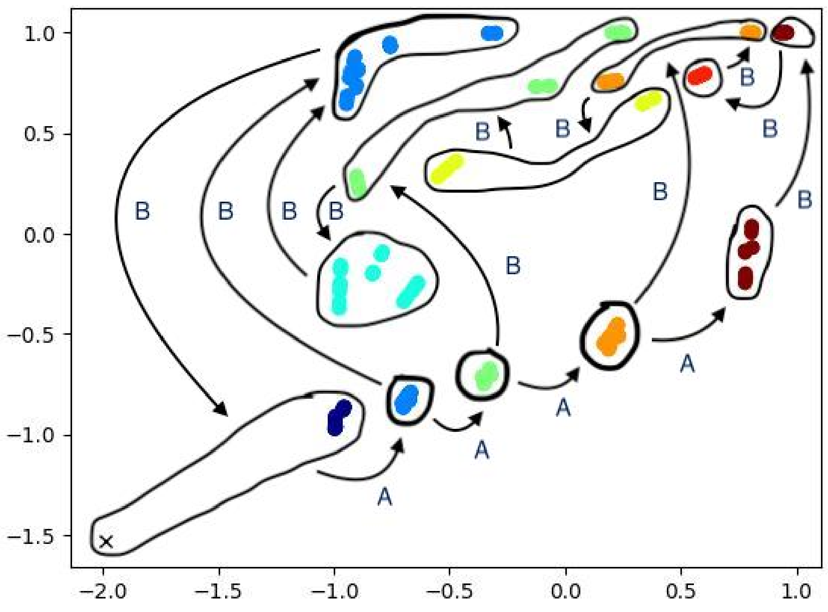
Hidden Unit 4



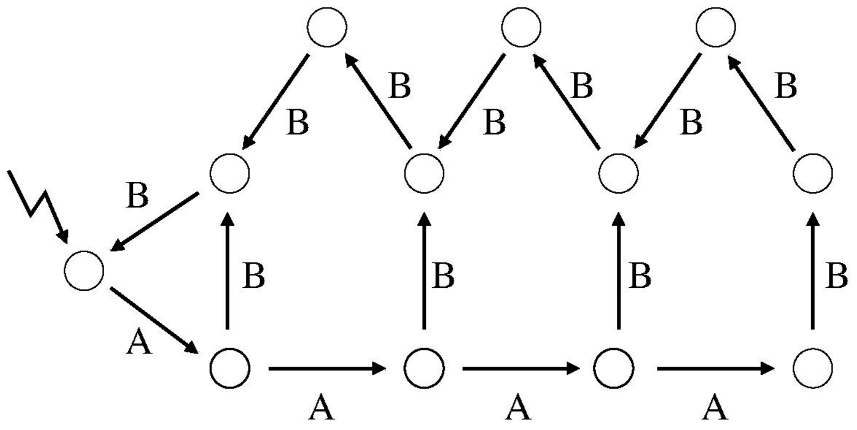
Output Unit

Part 3: Hidden Unit Dynamics for Recurrent Networks

Part 3, Step 1. After examining the hidden unit activations, annotate this image (either electronically, or with a pen on a printout) by grouping clusters of points together to form "state". Draw a boundary around each state, and draw arrows between the states labeled 'A' or 'B'.



Part 3, Step 2. Draw a picture of a finite state machine, using circles and arrows, which is equivalent to the finite state machine in your annotated image from Step 1.

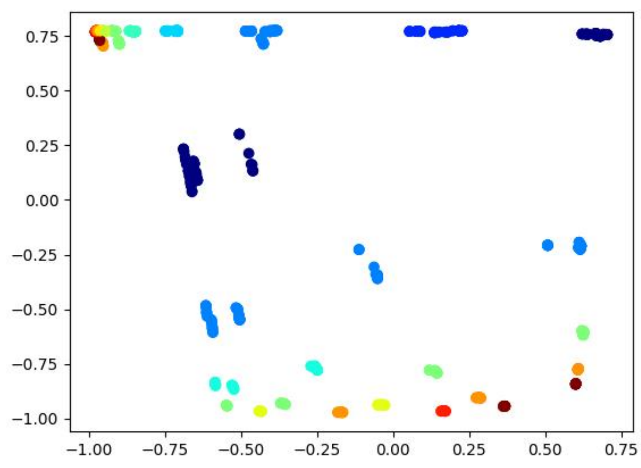


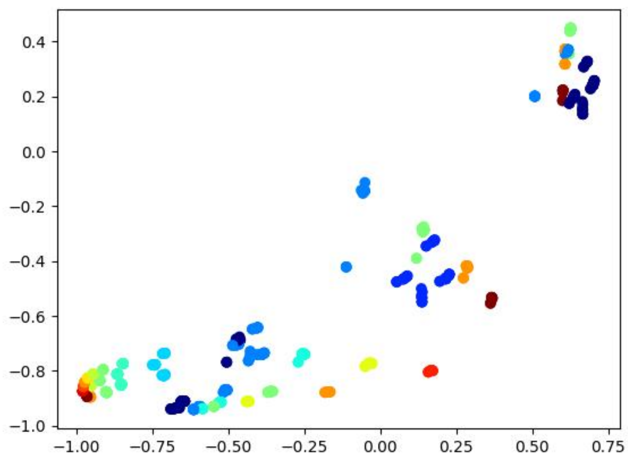
Part 3, Step 3.

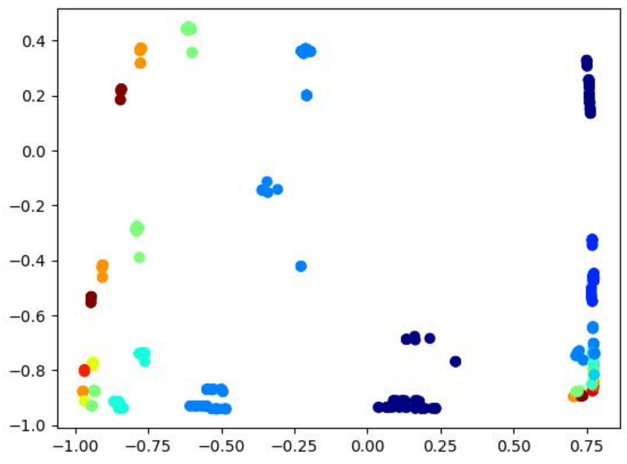
Briefly explain how the network accomplishes the anb2n task.

The initial hidden unit activation is in the lower left corner. As the network processes the A's, the activations move to the right, through the 'A' states in the lower part of the figure. When the first B is encountered, the activation moves upward to a 'B' state of the same color. It then progresses to the left through the states in the upper part of the figure, visiting twice as many 'B' states compared to the number of 'A' states, before returning to the initial cluster, thus allowing the subsequent A to be correctly predicted.

Part 3, Step 4.
```
python3 seq_plot.py --lang anb2nc3n --model lstm --epoch 50
```



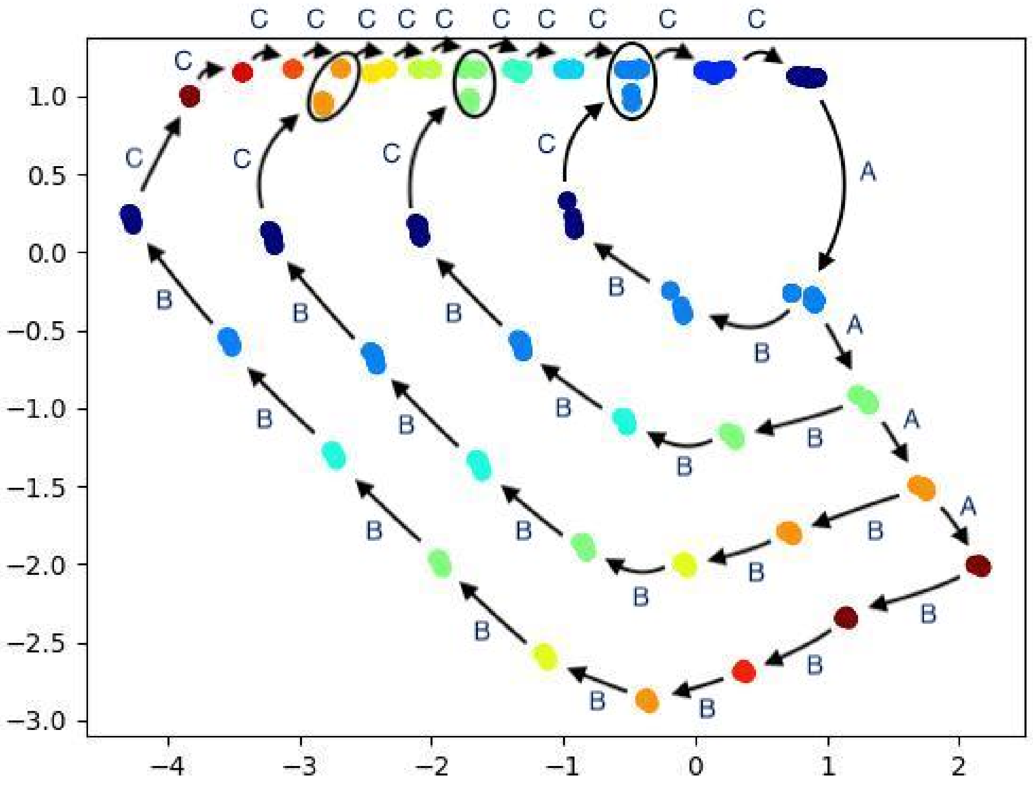




Part 3, Step 5.

This question is intended to be a bit more challenging. By annotating the generated images from Step 4 (or others of your own choosing), explain how the LSTM successfully accomplishes the $a^n b^{2n} c^{3n}$ task (this might involve modifying the code so that it prints out the context units as well as the hidden units).

After modifying the code to print the context units, we obtain the following figure for the first two context units:



The initial activation is in the top right corner. As the A's are processed, the activation moves down through the states near the right edge of the figure. As the B's are processed, the activation moves left and up through a different sequence of states, depending on how many A's were observed, thus allowing twice that number of B's to be accepted before entering the sequence of 12 'C' states along the top edge of the figure at either the 1st, 4th, 7th or 10th cluster, depending on how many A's were initially observed. This enables the correct number of C's to be accepted before returning to the initial state, ready to predict the next A.