Part 1: Japanese Character Recognition

1. NetLin

Final Accuracy:

The final model achieved an accuracy of **70%** on the test set, classifying **6958** out of **10000** images correctly.

Confusion Matrix:

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.828949
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.623921
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.595386
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.597939
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.320547
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.519950
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.671064
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.608339
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.352172
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.672435
<class 'numpy.ndarray'>
[[765.   5.   9.  12.  31.  64.   2.  63.  31.  18.]
 [  7. 668. 110.  18.  28.  22.  59.  12.  25.  51.]
 [  6.  62. 693.  26.  27.  20.  48.  36.  44.  38.]
 [  4.  37.  61. 753.  15.  58.  14.  18.  31.   9.]
 [ 61.  52.  78.  19. 625.  22.  32.  35.  20.  56.]
 [  8.  28. 122.  17.  19. 727.  28.   8.  33.  10.]
 [  5.  22. 146.  10.  26.  24. 723.  22.   9.  13.]
 [ 16.  28.  27.  11.  88.  17.  54. 624.  87.  48.]
 [ 10.  38.  90.  45.   7.  32.  47.   8. 703.  20.]
 [  8.  50.  89.   3.  53.  31.  19.  29.  41. 677.]]

Test set: Average loss: 1.0090, Accuracy: 6958/10000 (70%)
```

The total number of trainable parameters：7850

```
[17] !python3 kuzu_main.py --net lin

     Total number of trainable parameters: 7850
```

2. NetFull

Final Accuracy:

The final model achieved an accuracy of **84.5%** on the test set, classifying **8450 out of 10000** images correctly. The corresponding test loss was **0.5055**.

Confusion Matrix:

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.304192
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.230311
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.262069
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.203069
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.137804
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.246613
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.204706
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.337266
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.136941
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.263106
<class 'numpy.ndarray'>
[[837.   4.   3.   4.  35.  32.   2.  47.  31.   5.]
 [  6. 810.  45.   5.  14.  10.  58.   5.  19.  28.]
 [  7.  10. 852.  38.   9.  19.  21.  13.  19.  12.]
 [  4.  12.  37. 913.   1.  14.   5.   2.   7.   5.]
 [ 40.  28.  28.   8. 807.  10.  30.  16.  19.  14.]
 [ 10.  11.  72.  14.  10. 831.  31.   3.  12.   6.]
 [  3.  10.  57.   8.  16.   5. 888.   6.   2.   5.]
 [ 19.  14.  19.   3.  17.   9.  29. 836.  18.  36.]
 [ 10.  24.  28.  47.   5.  10.  30.   3. 840.   3.]
 [  3.  19.  66.   7.  26.   3.  14.  13.  13. 836.]]

Test set: Average loss: 0.5055, Accuracy: 8450/10000 (84%)
```

The total number of trainable parameters: 101,770

```
!python3 kuzu_main.py --net full
```

```
Total number of trainable parameters: 101770
```

3. NetConv

Final Accuracy:

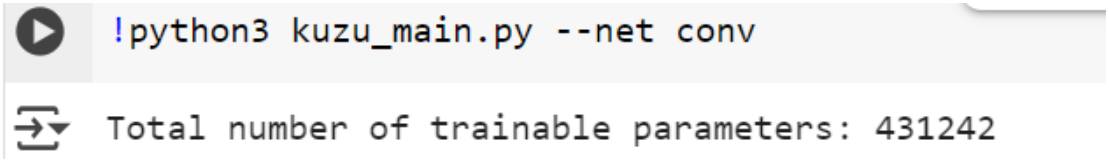
The model achieved a test set accuracy of **94%**, correctly classifying **9356 out of 10000** images. The final test loss was **0.2231**.

Confusion Matrix:

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.131743
Train Epoch: 10 [6400/60000 (11%)]      Loss: 0.104457
Train Epoch: 10 [12800/60000 (21%)]     Loss: 0.162825
Train Epoch: 10 [19200/60000 (32%)]     Loss: 0.140472
Train Epoch: 10 [25600/60000 (43%)]     Loss: 0.101567
Train Epoch: 10 [32000/60000 (53%)]     Loss: 0.108921
Train Epoch: 10 [38400/60000 (64%)]     Loss: 0.121151
Train Epoch: 10 [44800/60000 (75%)]     Loss: 0.316765
Train Epoch: 10 [51200/60000 (85%)]     Loss: 0.163031
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.141472
<class 'numpy.ndarray'>
[[971.   3.   2.   1.  11.   0.   0.   9.   1.   2.]
 [  3. 914.   5.   0.  13.   3.  37.   1.   4.  20.]
 [ 11.   0. 872.  49.   8.   8.  24.   8.   5.  15.]
 [  1.   1.   9. 973.   0.   2.   7.   1.   3.   3.]
 [ 26.   3.   1.  13. 918.   3.  13.   6.  13.   4.]
 [  3.   9.  29.   8.   3. 913.  24.   4.   3.   4.]
 [  3.   4.  11.   4.   3.   0. 970.   2.   1.   2.]
 [  4.   1.   6.   1.   8.   1.   9. 946.   3.  21.]
 [  4.  19.  10.   6.   6.   4.  11.   1. 927.  12.]
 [  4.   2.  10.   5.  10.   2.  10.   3.   2. 952.]]

Test set: Average loss: 0.2231, Accuracy: 9356/10000 (94%)
```

The total number of trainable parameters: 431,242



Discussion:

The three models demonstrate the trade-off between model complexity and classification performance. **NetLinear** has the fewest parameters but limited performance; **NetFull** significantly improves accuracy and is suitable for small and medium-sized tasks; **NetConv** has the most parameters but the highest accuracy and is the preferred choice for image-related tasks.

1. Accuracy Comparison:

1. **NetLinear**: Achieved a test accuracy of around **70%**. This is reasonable for a linear classifier but insufficient for the complex task of handwritten hiragana recognition.
2. **NetFull**: Improved test accuracy to around **84%**. The addition of a hidden layer with non-linearity significantly boosts the model's ability to capture complex patterns.
3. **NetConv**: Achieved the best performance, reaching **94%** accuracy. This meets and even exceeds the standard benchmarks for this dataset, showing the power of CNNs for image-based tasks.

2. Number of Trainable Parameters:

1. **NetLinear** has **7,850** parameters, as it directly maps 784 input pixels to 10 output classes.
2. **NetFull** contains **101,770** parameters, due to the additional hidden layer of 128 neurons.
3. **NetConv** has **431,242** parameters, owing to multiple convolutional filters and dense layers.

3. Confusion Matrix Insights:

The confusion matrix of the **NetLinea**r model demonstrates limited ability to distinguish between structurally similar Hiragana characters. **Major misclassifications occur between classes such as "ki" and "su"/"ma", "ha" and "ki"/"su", as well as "re" and "ki"/"su"/"ha".** For instance, "ki" is incorrectly classified as "su" 108 times, while "ha" is classified as "ki" and "su" 72 and 128 times, respectively. **This indicates that the linear model is unable to capture the complex spatial structures and fine-grained features of the images**, making it difficult to accurately differentiate characters with subtle stroke differences or similar shapes.

Compared to the linear model, the **NetFull model** shows stronger discriminative ability, with nonlinear transformations enhancing feature representation. **However, the confusion matrix still reveals significant misclassification between classes such as "ki", "su", "ha", and "re".** For example, "ki" is misclassified as "su" 45 times, and "ha" is misclassified as "ki" and "su" 72 and 128 times, respectively. **This suggests that, while the model can learn more complex features, the fully connected structure lacks explicit modeling of local spatial information, resulting in continued confusion among characters with similar stroke patterns.**

The **NetConv model** exhibits the best performance in its confusion matrix, **achieving high accuracy across most character classes**. Categories such as "o", "tsu", and "wo" are classified almost perfectly, **with only a small number of misclassifications remaining between highly similar classes, such as "ki" and "su"** (e.g., "ki" is misclassified as "su" only 11 times). **The convolutional network effectively extracts local spatial features and texture information, providing strong discrimination for handwritten details and significantly improving overall recognition accuracy.**

Part 2: Multi-Layer Perceptron

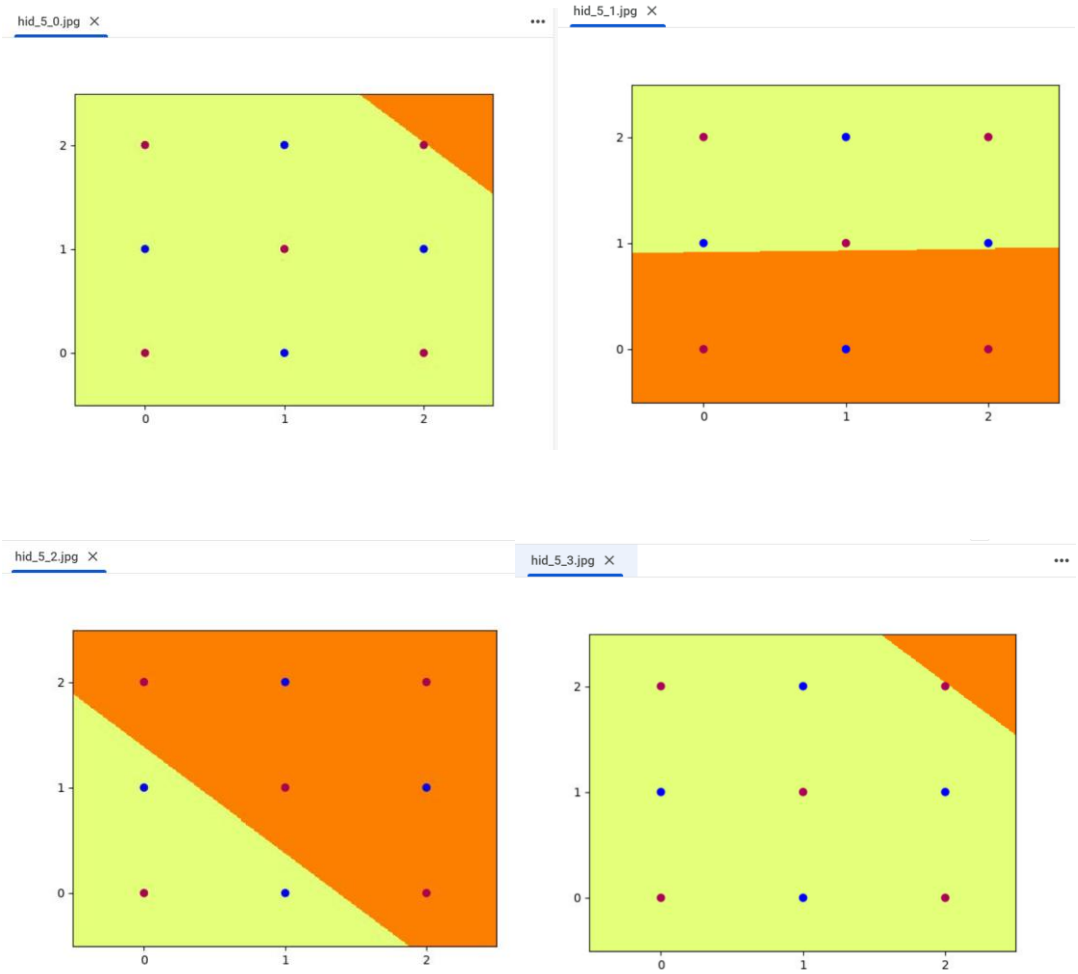
1. Train a 2-layer neural network with 5 hidden nodes, using sigmoid activation at both the hidden and output layer, on the above data.

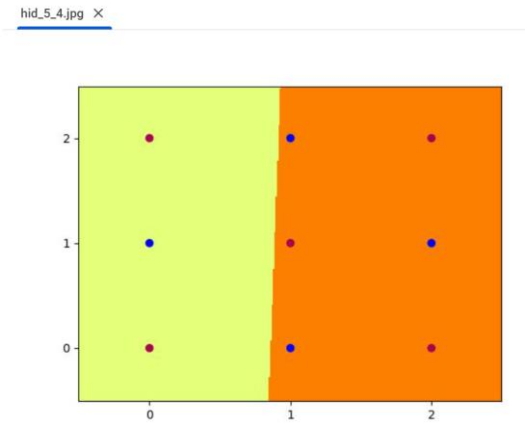
This figure shows the output of a two-layer neural network using a sigmoid activation function after training. The network contains five hidden nodes and was trained on a given dataset to achieve 100% classification accuracy.

Output of Trained 2-Layer Sigmoid Neural Network with 5 Hidden Units

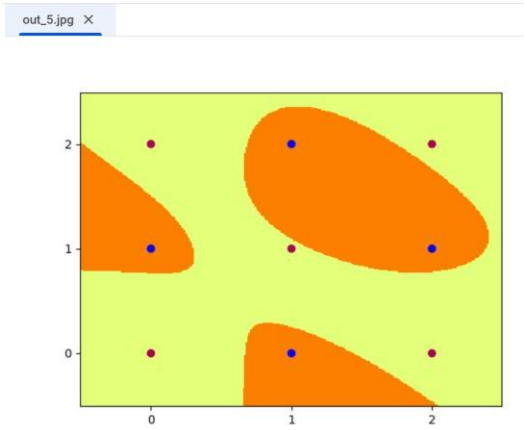
```
Final Weights:
tensor([[ 5.5730, -4.6410],
        [ 4.1997, -6.7148],
        [-7.1554, -7.3003],
        [-5.1939,  6.8265],
        [ 6.7825, -4.3193]])
tensor([ 7.2180, -6.3256,  2.4465,  1.0554, -0.3114])
tensor([[ 6.6335, -6.5291, -6.3886, -5.7748, -5.8014]])
tensor([2.1650])
Final Accuracy:  100.0
```

Decision boundaries learned by each of the 5 hidden units





Final decision boundary of the overall network after combining all hidden node activations



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 above data.

1. Network Structure and Parameters

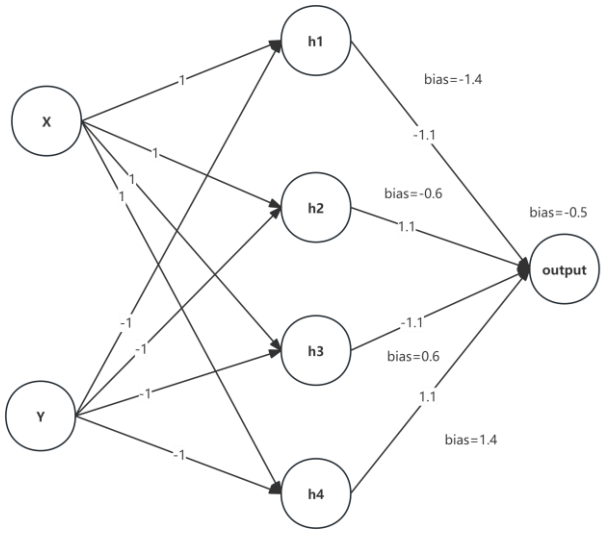
This is a two-layer neural network consisting of an input layer with two nodes (representing x and y), a hidden layer with four nodes where each node uses the Heaviside step activation function, and an output layer with a single node that also uses the Heaviside step activation function.

Parameter values:

```
##### Enter Weights Here ############
    in_hid_weight  = [[1, -1],[1, -1],[1, -1],[1, -1]]
    hid_bias       = [-1.4, -0.6, 0.6, 1.4]
    hid_out_weight = [[-1.1,1.1,-1.1,1.1]]
    out_bias       = [-0.5]

```

A diagram of the network is shown below:



2. Dividing Line Equations

The **dividing line** corresponding to each hidden node is:

1. $x-y-1.4=0 \rightarrow \quad y=x-1.4$
2. $x-y-0.6=0 \rightarrow \quad y=x-0.6$
3. $x-y+0.6=0 \rightarrow \quad y=x+0.6$
4. $x-y+1.4=0 \rightarrow \quad y=x+1.4$

3. Activation Table for Training Samples

Below are the hidden activations and outputs for all 9 points:

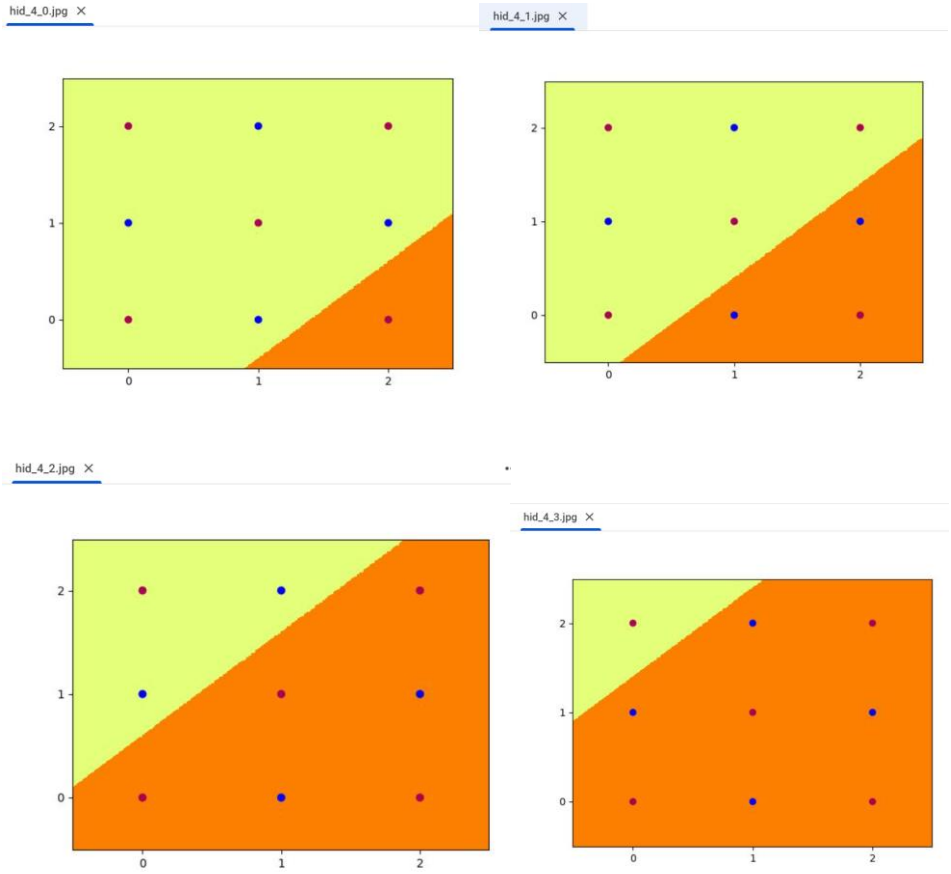
x	y	h1	h2	h3	h4	Output
0	0	0	0	1	1	1
0	1	0	0	0	1	0
0	2	0	0	0	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	2	0	0	0	1	0
2	0	1	1	1	1	1
2	1	1	1	1	1	1
2	2	0	1	1	1	1

3. Now rescale your hand-crafted weights and biases 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.

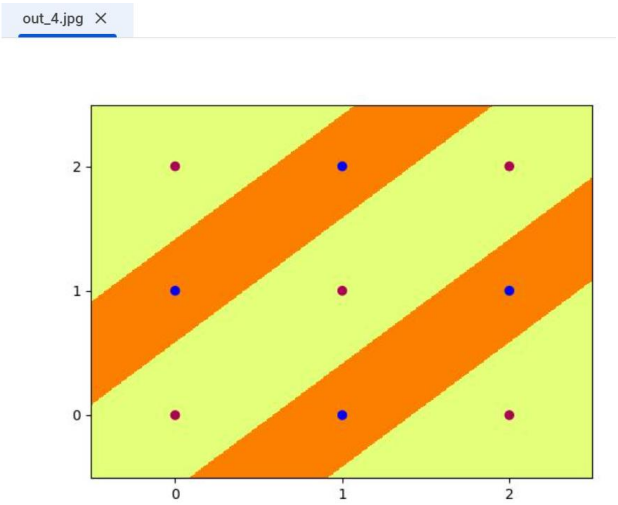
We rescale all hand-crafted weights and biases from Part 2 by a factor of 10. The resulting parameters are:

```
##### Enter Weights Here ############
        in_hid_weight  = [[10, -10],[10, -10],[10, -10],[10, -10]]
        hid_bias       = [-14, -6, 6, 14]
        hid_out_weight = [[-11, 11, -11, 11]]
        out_bias       = [-5]
```

Decision Boundaries of Hidden Nodes with Rescaled Weights Using Sigmoid Activation



Overall Decision Boundary of the Network after Rescaling Weights and Using Sigmoid Activation



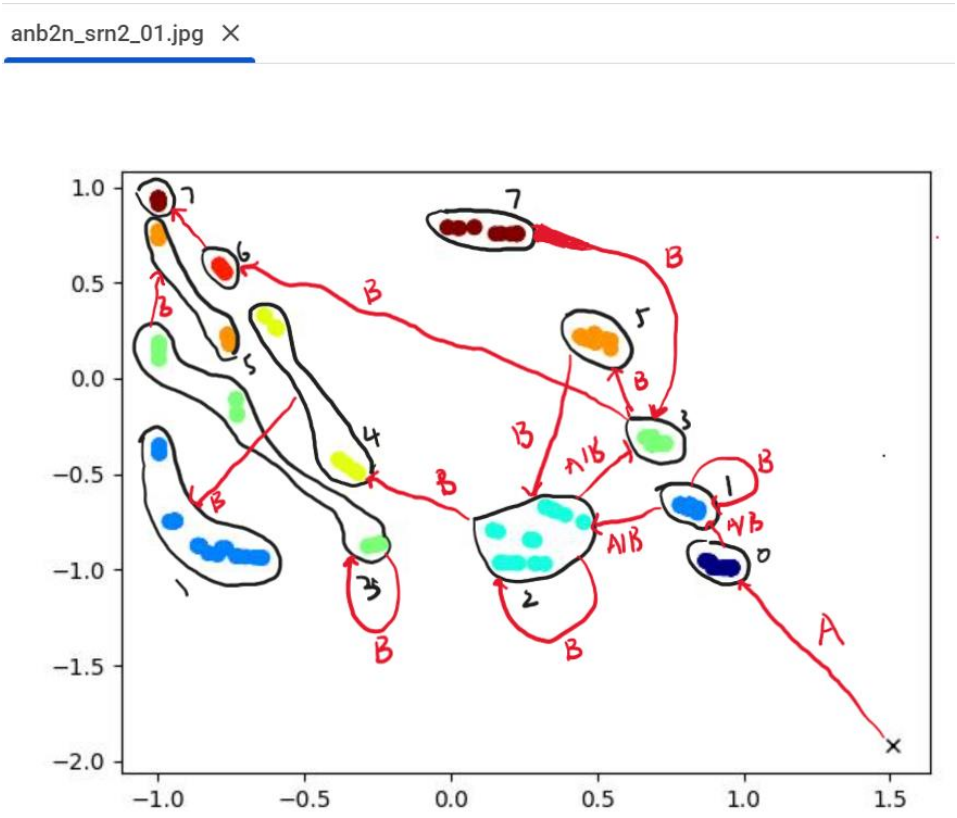
Conclusion:

As shown in the figures, the sigmoid network with rescaled parameters achieves perfect classification, and the decision boundaries are as sharp as those of the step-function network. This confirms that a sufficiently steep sigmoid can approximate the step function in neural network classification.

Part 3: Hidden Unit Dynamics for Recurrent Networks

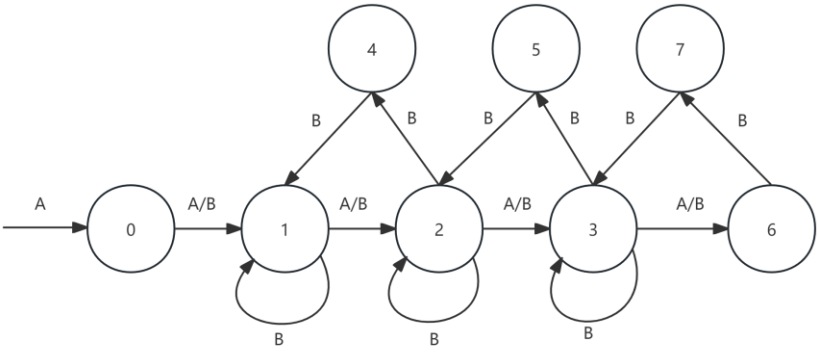
1. Train a Simple Recurrent Network (SRN) with 2 hidden nodes on the $a^n b^{2n}$ language prediction task by typing:

annotated image



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.

picture of a finite state machine



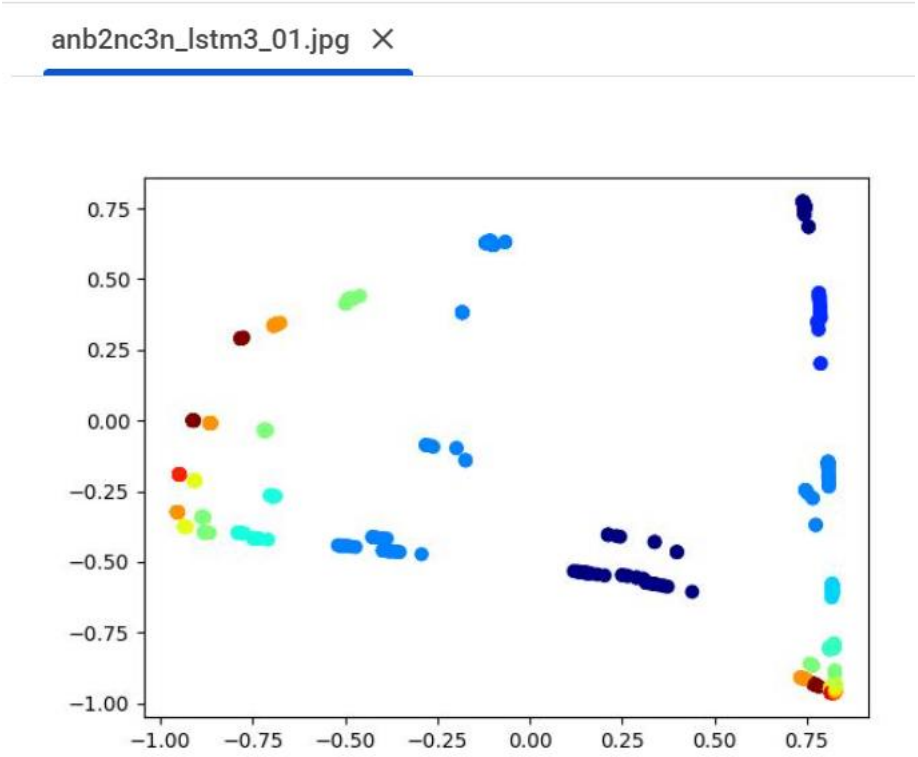
3. Briefly explain how the network accomplishes the $a^n b^{2n}$ task. Specifically, you should describe how the hidden unit activations change as the string is processed, and how it is able to correctly predict all B's after the first B, as well as the initial A following the last B in the sequence

In the anb2n task, the **SRN dynamically encodes the number of A in the sequence through the consecutive activation values of two hidden units**, thereby enabling precise prediction of the number of B. During the initial stage of the sequence, **each time an A is received, the trajectory of the hidden state gradually enters a new activation interval, forming a state distribution with 'counting' significance**. When the first B arrives, the SRN has already recorded the number of A through the hidden state, and **the positions of all subsequent B can be accurately predicted, with output probabilities approaching 1**. Finally, after processing all B, **the SRN returns to its initial state, ready to predict the next sequence of A**.

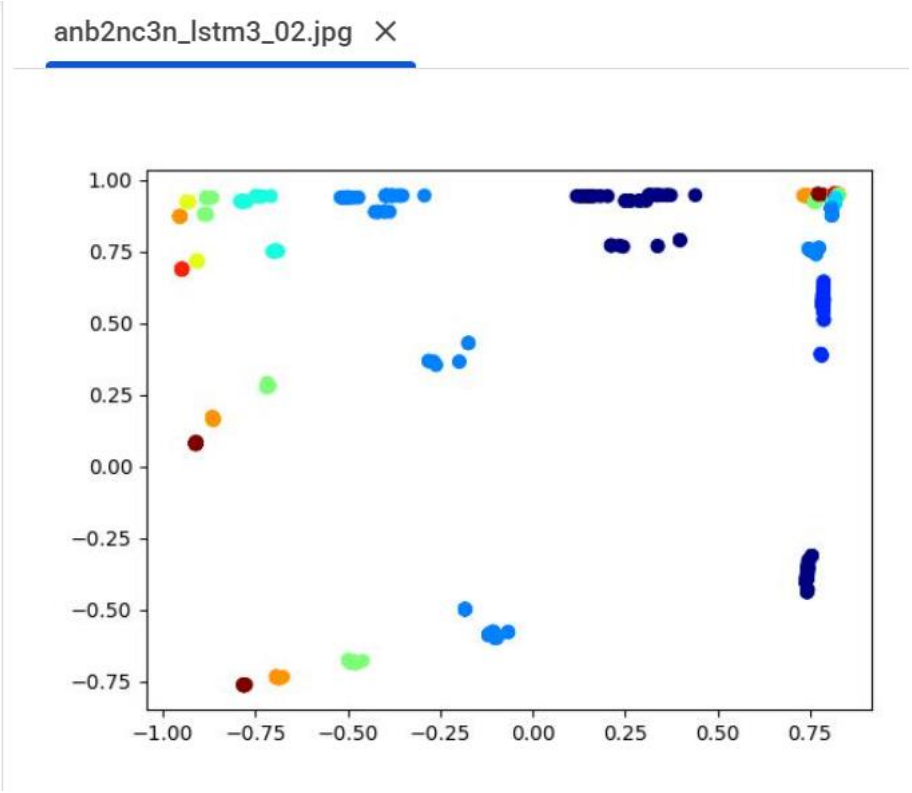
Throughout the process, although the SRN lacks an explicit discrete state transition mechanism, its **hidden units exhibit clear clustering and segmentation characteristics at different stages**, resembling an **implicit state machine**. This enables it to capture the structural patterns in the anb2n sequence and **model and predict long-range dependencies**.

4. This should produce three images from different angles, labeled anb2nc3n_lstm3_??.jpg, as well as an interactive 3D plot which you can rotate to a visually appealing angle and save as an image. Copy these images into your report.

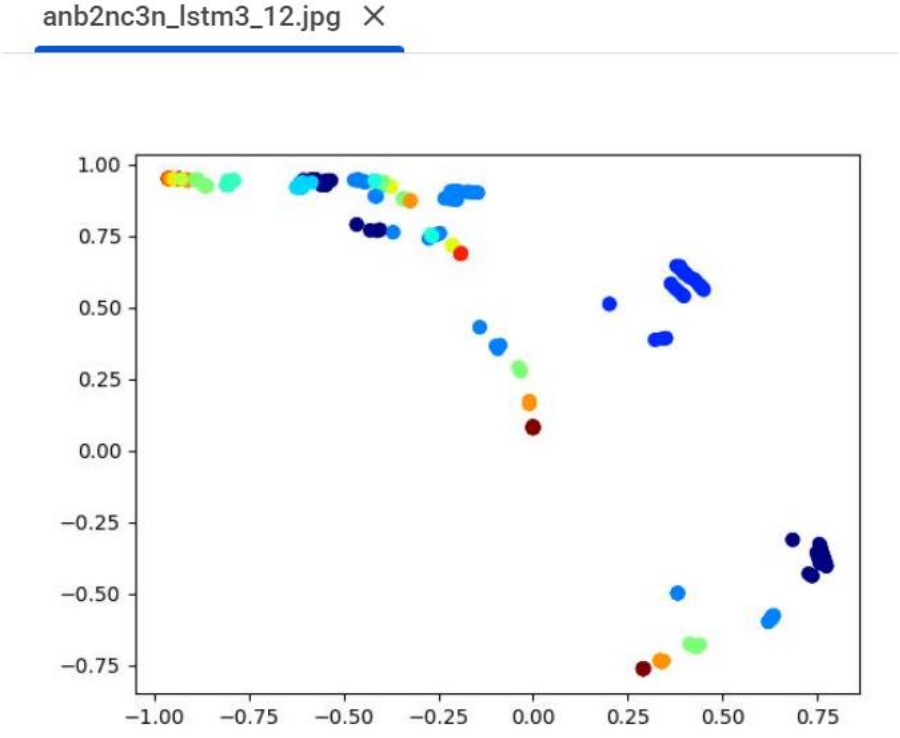
2D Projection of LSTM Hidden State Activations (anb2nc3n_lstm3_01.jpg)



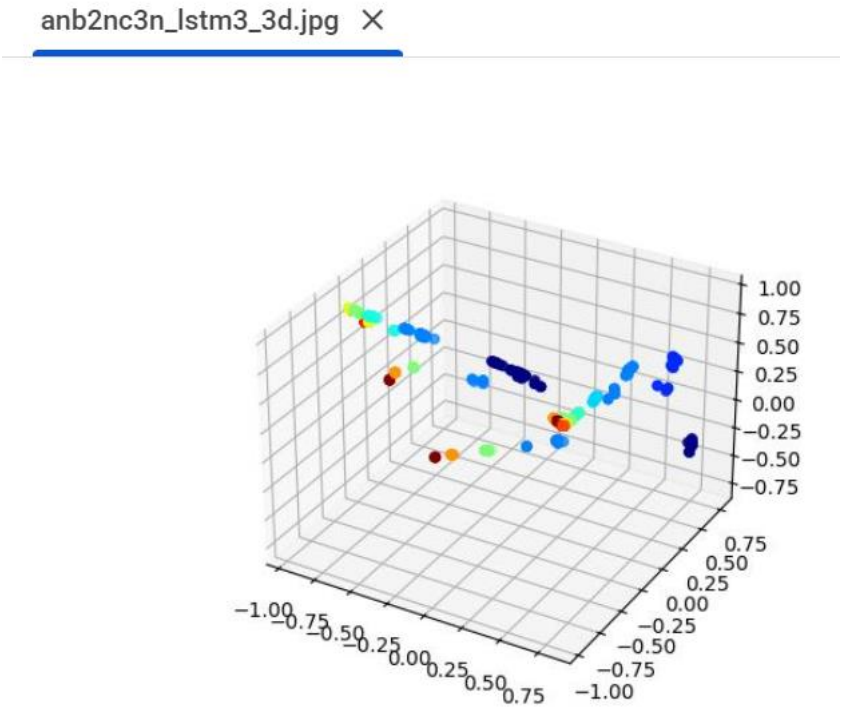
2D Projection of LSTM Hidden State Activations (anb2nc3n_lstm3_02.jpg)



2D Projection of LSTM Hidden State Activations (anb2nc3n_lstm3_12)



3D Visualization of LSTM Hidden State Dynamics(anb2nc3n_lstm3_3d)



Conclusion:

The visualization of hidden unit activations shows distinct clusters and trajectories in the state space, corresponding to different stages of the sequence (A's, B's, C's). This indicates that the LSTM can successfully represent and distinguish different sequence segments, supporting accurate predictions.

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), try to analyse the dynamics of the hidden and/or context units, and explain how the LSTM successfully accomplishes the $a^n b^{2n} c^{3n}$ prediction task (this might involve modifying the code so that it returns and prints out the context units as well as the hidden units).

In the language prediction task of anb2nc3n, the model needs to simultaneously remember the quantitative relationships among the three characters A, B, and C, and be able to accurately switch between different stages. **RNNs, lacking long-term memory capabilities**, often forget previous information when dealing with longer sequences or multiple stage switches, leading to inaccurate predictions of the quantities of B and C and overall suboptimal performance.

LSTMs, in contrast, have a clear advantage. The core of LSTM lies in its **gating mechanism and memory units (cell state)**, which allow the model to selectively 'remember' or 'forget' certain information during the learning process. For example, during the A phase, LSTM can accumulate the quantity of A through the cell state; when entering the B phase, it retains or updates the information in the cell state via the gating mechanism, helping the model accurately predict the expected quantity of B; upon entering the C phase, it similarly relies on

the cell state to perform correct counting and switching. In this way, **LSTM effectively addresses the forgetting issues faced by RNNs in long-term dependencies and phase transitions**.

By observing the **visualisation of the hidden state**, it can be seen that different stages cluster into distinct clusters in the three-dimensional hidden space. When the sequence transitions to a new stage, the hidden state exhibits noticeable jumps or trajectory shifts. This indicates that the **LSTM's hidden state effectively encodes the current stage** of the sequence and dynamically adjusts the state based on input, enabling the model to continuously track and remember patterns.