ZID: z5518601

Name: Tianxiong Wu

Part1 : Japanese Character Recognition

1. Implement a model NetLin which computes a linear function of the pixels in the image, followed by log softmax. Run the code by typing:
   `python3 kuzu_main.py --net lin`

Final Accuracy&Confusion Matrix

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.825311
Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.632141
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.590211
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.599470
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.316702
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.516328
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.660213
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.604590
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.348631
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.665734
<class 'numpy.ndarray'>
[[766.   5.   7.  12.  31.  65.   2.  63.  30.  19.]
 [  6. 671. 106.  18.  26.  23.  59.  13.  26.  52.]
 [  7.  60. 686.  27.  26.  22.  48.  37.  46.  41.]
 [  4.  34.  62. 758.  15.  55.  15.  18.  28.  11.]
 [ 59.  52.  77.  19. 629.  20.  32.  36.  20.  56.]
 [  8.  28. 126.  16.  20. 726.  26.   7.  33.  10.]
 [  4.  22. 145.  10.  24.  24. 725.  21.   9.  16.]
 [ 18.  30.  26.  11.  81.  17.  54. 627.  89.  47.]
 [ 10.  37.  94.  41.   8.  30.  44.   8. 704.  24.]
 [  9.  51.  85.   3.  52.  30.  21.  30.  41. 678.]]

Test set: Average loss: 1.0080, Accuracy: 6970/10000 (70%)
```

Code

```python
class NetLin(nn.Module):
    def __init__(self):
        super(NetLin, self).__init__()
        self.fc = nn.Linear(28 * 28,  out_features: 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Flatten the image
        x = self.fc(x)
        return F.log_softmax(x, dim=1)
```

2. Implement a fully connected -layer network NetFull (i.e. one hidden layer, plus the output layer), using tanh at the hidden nodes and log softmax at the output node. Run the code by typing: `python3 kuzu_main.py --net full`

Final Accuracy&Confusion Matrix

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.421333
Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.291304
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.250057
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.218298
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.120913
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.256959
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.208207
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.327782
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.119604
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.298960
<class 'numpy.ndarray'>
[[838.   3.   2.   5.  32.  31.   3.  45.  33.   8.]
 [  5. 800.  33.   4.  19.  13.  64.   4.  22.  36.]
 [  7.  12. 840.  34.  13.  17.  28.  10.  24.  15.]
 [  4.  10.  29. 912.   1.  12.   6.   3.   9.  14.]
 [ 35.  26.  18.   5. 815.   9.  32.  18.  23.  19.]
 [  7.  17.  74.  17.  12. 813.  29.   1.  20.  10.]
 [  3.  15.  45.   7.  10.   8. 893.   7.   2.  10.]
 [ 19.  13.  19.   2.  21.  11.  38. 808.  29.  40.]
 [ 12.  25.  28.  45.   3.  11.  30.   4. 836.   6.]
 [  4.  27.  36.   7.  31.   6.  23.  17.  14. 835.]]

Test set: Average loss: 0.5244, Accuracy: 8390/10000 (84%)
```

Code

```python
# Fully-connected model with one hidden layer
3 个用法
class NetFull(nn.Module):
    def __init__(self, hidden_size=100):  # hidden_size
        super(NetFull, self).__init__()
        self.fc1 = nn.Linear(28 * 28, hidden_size)
        self.fc2 = nn.Linear(hidden_size, out_features: 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = torch.tanh(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Commentary

Model Description

We implemented the NetFull model, a fully connected feedforward neural network with:

- One hidden layer of **100** nodes (using tanh activation)
- An output layer with 10 nodes (using log_softmax)

Number of Parameters

- **Input to hidden layer**: 784 × 100 weights + 100 biases = 78500
- **Hidden to output layer**: 100 × 10 weights + 10 biases = 1010
  So, total parameters = 795 × 100 + 10 = 79510

3. Implement a convolutional network called NetConv, with two convolutional layers plus one fully connected layer, all using relu activation function, followed by the output layer, using log softmax. You are free to choose for yourself the number and size of the filters, metaparameter values (learning rate and momentum), and whether to use max pooling or a fully convolutional architecture. Run the code by typing: `python3 kuzu_main.py --net conv`

Final Accuracy&Confusion Matrix

```
Train Epoch: 10 [0/60000 (0%)]  Loss: 0.048140
Train Epoch: 10 [6400/60000 (11%)]  Loss: 0.036944
Train Epoch: 10 [12800/60000 (21%)] Loss: 0.046686
Train Epoch: 10 [19200/60000 (32%)] Loss: 0.100232
Train Epoch: 10 [25600/60000 (43%)] Loss: 0.055060
Train Epoch: 10 [32000/60000 (53%)] Loss: 0.041183
Train Epoch: 10 [38400/60000 (64%)] Loss: 0.018379
Train Epoch: 10 [44800/60000 (75%)] Loss: 0.159896
Train Epoch: 10 [51200/60000 (85%)] Loss: 0.010133
Train Epoch: 10 [57600/60000 (96%)] Loss: 0.026855
<class 'numpy.ndarray'>
[[947.   4.   5.   0.  19.   3.   0.  15.   1.   6.]
 [  2. 923.   5.   1.   6.   3.  30.  13.   4.  13.]
 [ 10.  11. 877.  47.  13.   8.  19.   8.   1.   6.]
 [  0.   2.  17. 959.   3.   4.   2.   5.   3.   5.]
 [ 25.   4.   4.   5. 912.   3.  12.  19.  11.   5.]
 [  7.  15.  34.   8.   3. 905.  14.   8.   1.   5.]
 [  2.  12.   9.   5.   4.   1. 958.   8.   0.   1.]
 [  8.   6.   1.   0.   9.   1.   9. 951.   1.  14.]
 [  8.  15.  13.   9.  13.   5.   9.  11. 915.   2.]
 [ 13.  14.   7.   4.  16.   3.   2.  10.   9. 922.]]

Test set: Average loss: 0.2692, Accuracy: 9269/10000 (93%)
```

✓ **Commentary**

Model Description

The NetConv model consists of two convolutional layers, each followed by ReLU activation and max pooling, and two fully connected layers. The structure is:

- **Conv Layer 1**: Conv2d(1, 16, 5) + BatchNorm2d(16) + ReLU + MaxPool(2×2)
- **Conv Layer 2**: Conv2d(16, 32, 5) + BatchNorm2d(32) + ReLU + MaxPool(2×2)
- **Fully Connected Layer 1**: Linear(512, 128) + ReLU
- **Output Layer**: Linear(128, 10) + LogSoftmax

Parameter Count Calculation

- Conv1: 1×16×5×5 + 16 = 400 + 16 = **416**
- Conv2: 16×32×5×5 + 32 = 12,800 + 32 = **12,832**

- BN1: 32
- BN2: 64
- FC1: 512×128 + 128 = **65664**
- FC2: 128×10 + 10 = **1,290**

  Total Parameters = 80298

4. Comparative Analysis of the Three Models

Relative Accuracy

Model	Accuracy	Reason
NetLin	70%	only applies a single linear transformation to flattened pixels, lacking the capacity to capture spatial or hierarchical features.
NetFull	84%	introducing a hidden layer with nonlinear activation (tanh), enabling more complex representations.
NetConv	93%	extract spatial features using convolutional filters, normalize activation distributions via batch normalization, and downsample irrelevant detail via max pooling.

This shows a clear trend: **the more structurally aware and deeper the model, the higher its classification accuracy**.

Number of Parameters

Although NetConv has fewer parameters than NetFull, it achieves better performance due to better feature extraction via convolution and shared weights.

Confusion Matrix Analysis

From the confusion matrices observed for each model:

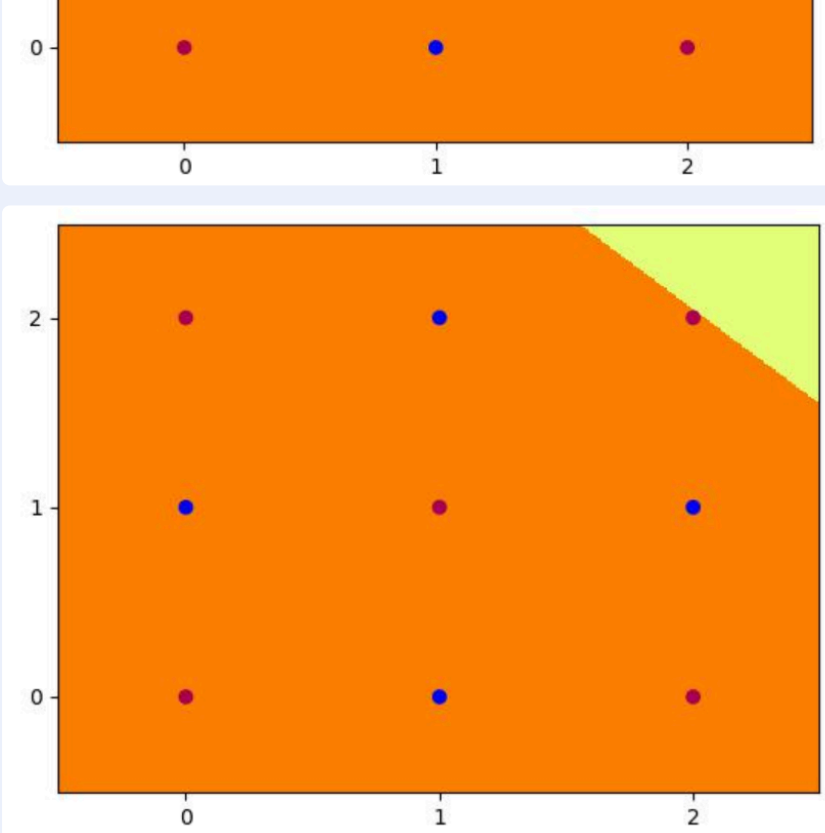
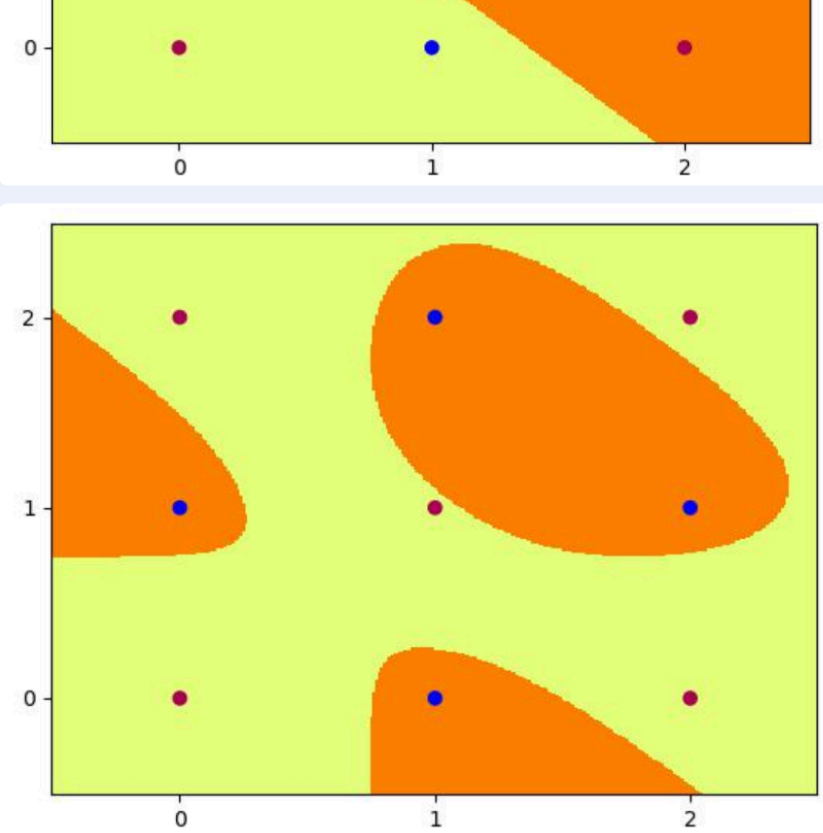
- **NetLin (70%)**:
  - Characters such as "su" (2), "ha" (5), and "ma" (6) were frequently confused with each other.
  - Likely due to their similar stroke density and layout when flattened, which linear models fail to distinguish.
- **NetFull (84%)**:
  - Improved separation of most classes, but still confusion between "ha" (5) and "ma" (6), or "re" (8) and "wo" (9).
  - These pairs have subtle shape differences requiring spatial awareness, which dense layers partially capture.
- **NetConv (93%)**:
  - Most confusion is resolved, but occasional mix-ups remain between "na" (4) vs "re" (8), and "ki" (1) vs "su" (2).
  - These misclassifications are often caused by similar handwritten styles or overlapping radicals.

Part 2: Multi-Layer Perceptron

1. Train a -layer neural network with hidden nodes, using sigmoid activation at both the hidden and output layer, on the above data, by typing: `python3 check_main.py --act sig --hid 5`

Output Images

```
Final Weights:
tensor([[-4.1024,  0.0817],
        [ 2.7719,  2.7717],
        [ 0.0823, -4.0399],
        [ 5.5446,  5.5624],
        [-2.7555, -2.7558]])
tensor([  3.7196, -11.1921,   3.6527,  -7.6602,  11.1430])
tensor([[ -9.5561, -10.3537,  -9.5889,  -8.6482,  10.8272]])
tensor([5.0024])
Final Accuracy:  100.0
```

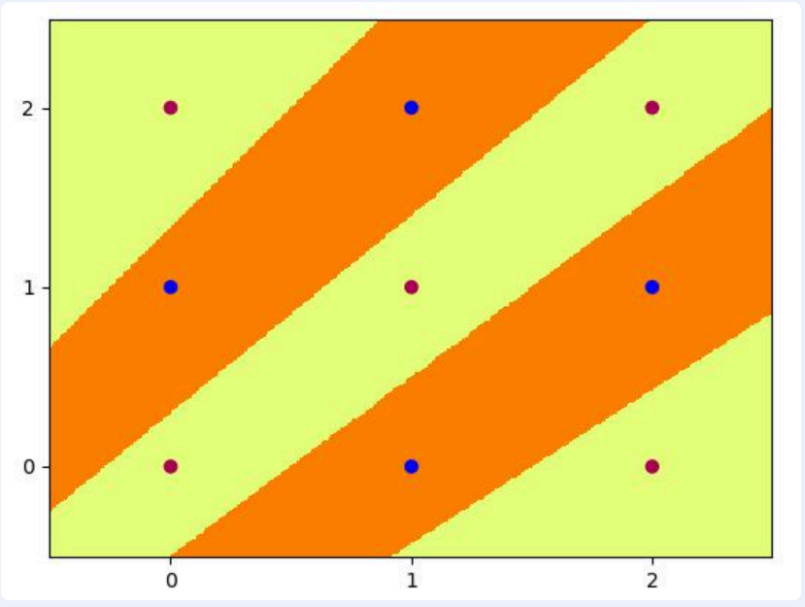



2. Design by hand a -layer neural network with hidden nodes, using the Heaviside (step) activation function at both the hidden and output layer, which correctly classifies the above data. Include a diagram of the network in your report, clearly showing the value of all the weights and biases. Write the equations for the dividing line determined by each hidden node. Create a table showing the activations of all the hidden

nodes and the output node, for each of the training items, and include it in your report. You can check that your weights are correct by entering them in the section of check.py where it says "Enter Weights Here", and typing: `python3 check_main.py --act step --hid 4 --set_weights`

Output Images

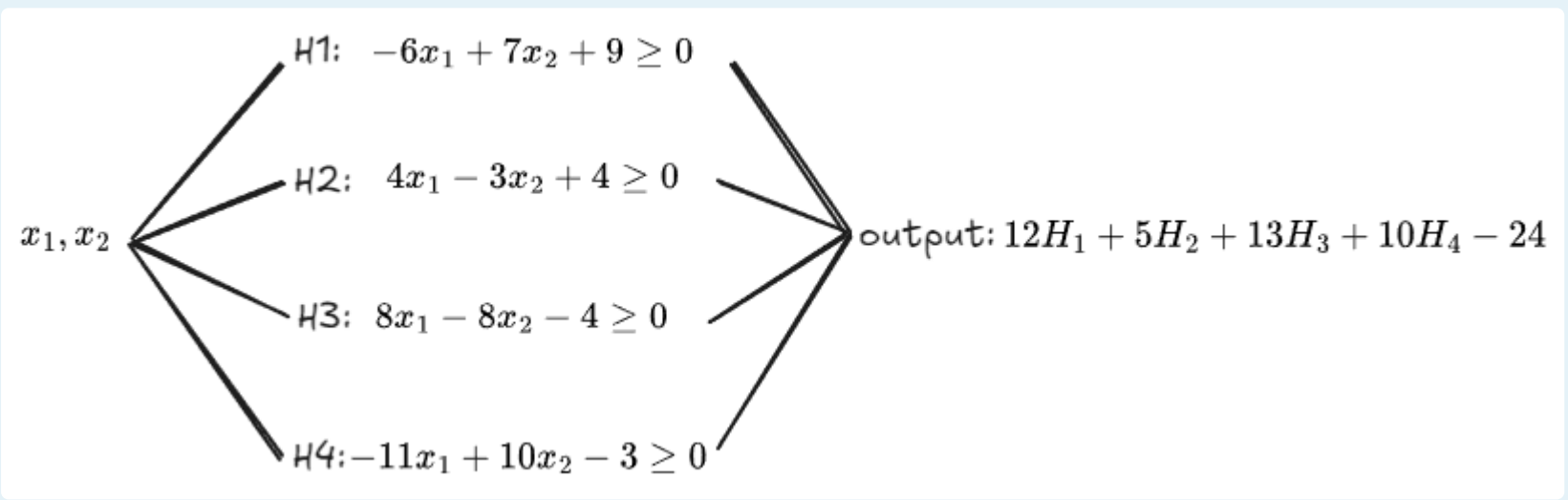
```
Initial Weights:
tensor([[ -6.,   7.],
        [  4.,  -3.],
        [  8.,  -8.],
        [-11.,  10.]])
tensor([ 9.,  4., -4., -3.])
tensor([[12.,  5., 13., 10.]])
tensor([-24.])
Initial Accuracy:  100.0
```



Network Diagram

We designed a 2-layer neural network with:

- **Input layer**: 2 inputs (x_1, x_2)
- **Hidden layer**: 4 hidden nodes
- **Output layer**: 1 output node
- All nodes use the **Heaviside (step)** activation function



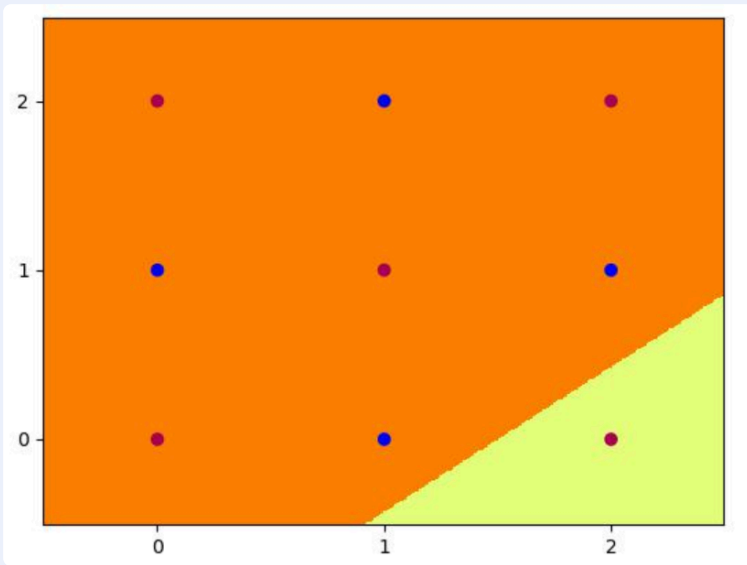
Activation Table

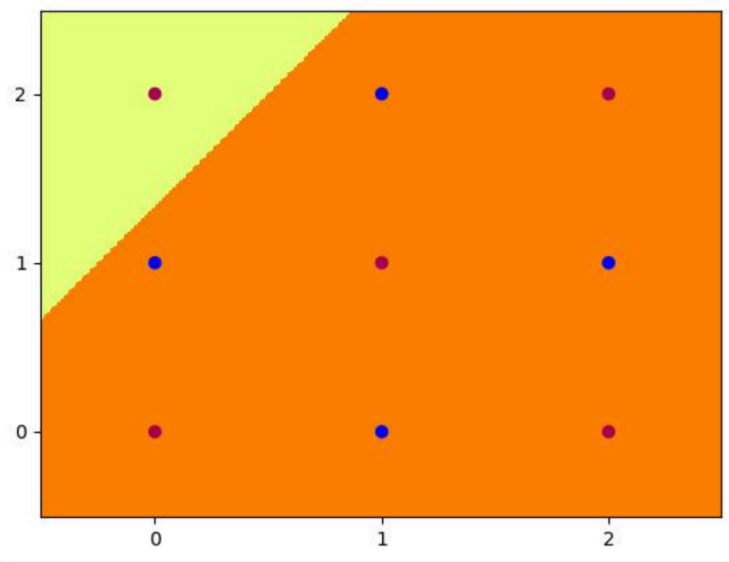
(x_1, x_2)	H1	H2	H3	H4	Output
(0, 0)	1	1	0	0	0
(1, 0)	1	1	1	0	1
(2, 0)	0	1	1	0	0
(0, 1)	1	1	0	1	1
(1, 1)	1	1	0	0	0
(2, 1)	1	1	1	0	1
(0, 2)	1	0	0	1	0
(1, 2)	1	1	0	1	1
(2, 2)	1	1	0	0	0

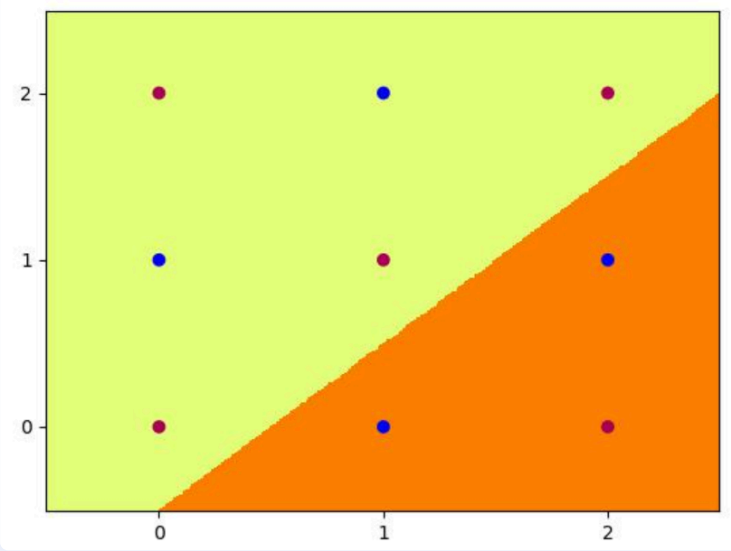
3. Now rescale your hand-crafted weights and biases from Part by multiplying all of them by a large (fixed) number (for example,) so that the combination of rescaling followed by sigmoid will mimic the effect of the step function. With these re-scaled weights and biases, the data should be correctly classified by the sigmoid network as well as the step function network. Verify that this is true by typing: `python3 check_main.py --act sig --hid 4 --set_weights`

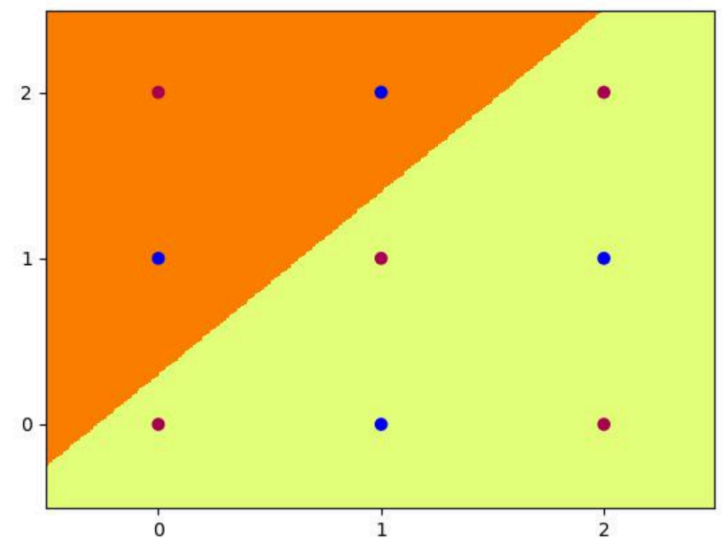
Output Images

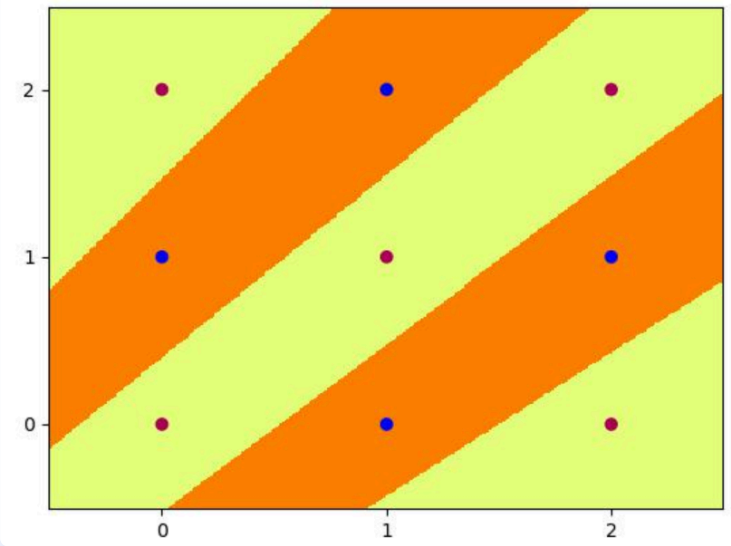
```
Initial Weights:
tensor([[ -60.,   70.],
        [  40.,  -30.],
        [  80.,  -80.],
        [-110.,  100.]])
tensor([ 90.,  40., -40., -30.])
tensor([[120.,  50., 130., 100.]])
tensor([-240.])
Initial Accuracy:  100.0
```







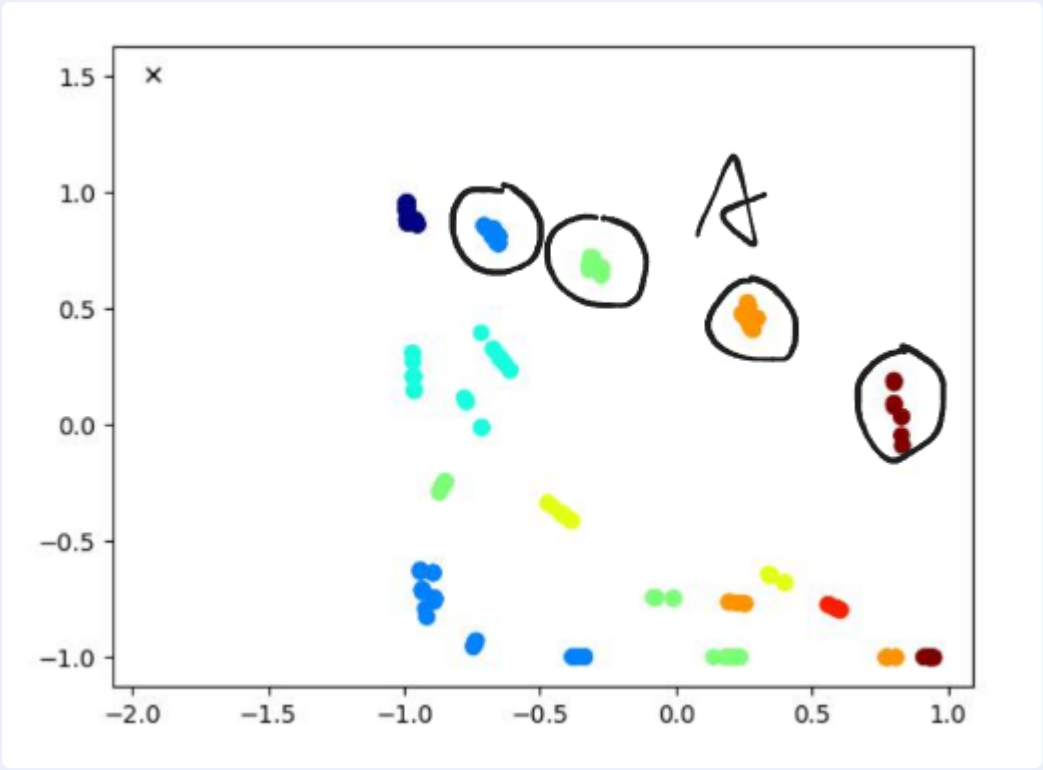




Part 3: Hidden Unit Dynamics for Recurrent Networks

1. Train a Simple Recurrent Network (SRN) with hidden nodes on the a nb n language prediction task by typing: `python3 seq_train.py --lang anb2n`

Hidden Unit Activation




```
A [ 0.27 -0.25] [ 0.46  0.54]
B [-0.98  0.85] [ 0.01  0.99]
B [-0.41 -0.76] [ 0.4   0.6]
A [ 0.66 -0.5 ] [ 0.75  0.25]
A [ 0.18 -0.2 ] [ 0.39  0.61]
A [ 0.65 -0.72] [ 0.82  0.18]
A [ 0.02  0.08] [ 0.21  0.79]
B [-0.94  0.58] [ 0.01  0.99]
B [-0.58 -0.56] [ 0.24  0.76]
B [-0.95  0.81] [ 0.01  0.99]
B [-0.46 -0.73] [ 0.37  0.63]
B [-0.97  0.89] [ 0.01  0.99]
B [-0.39 -0.78] [ 0.42  0.58]
B [-0.97  0.91] [ 0.01  0.99]
B [-0.37 -0.79] [ 0.43  0.57]
A [ 0.63 -0.46] [ 0.72  0.28]
A [ 0.23 -0.28] [ 0.45  0.55]
A [ 0.59 -0.65] [ 0.78  0.22]
B [-0.99  0.96] [ 0.01  0.99]
B [-0.33 -0.82] [ 0.47  0.53]
B [-0.98  0.93] [ 0.01  0.99]
B [-0.36 -0.8 ] [ 0.44  0.56]
B [-0.97  0.92] [ 0.01  0.99]
B [-0.37 -0.8 ] [ 0.44  0.56]
A [ 0.63 -0.45] [ 0.71  0.29]
B [-0.99  0.94] [ 0.01  0.99]
B [-0.34 -0.81] [ 0.46  0.54]
A [ 0.61 -0.42] [ 0.69  0.31]
A [ 0.28 -0.34] [ 0.5   0.5]
B [-0.98  0.88] [ 0.01  0.99]
B [-0.39 -0.78] [ 0.42  0.58]
B [-0.97  0.91] [ 0.01  0.99]
B [-0.37 -0.79] [ 0.43  0.57]
```

```
epoch: 100000
 loss: 0.0323
```

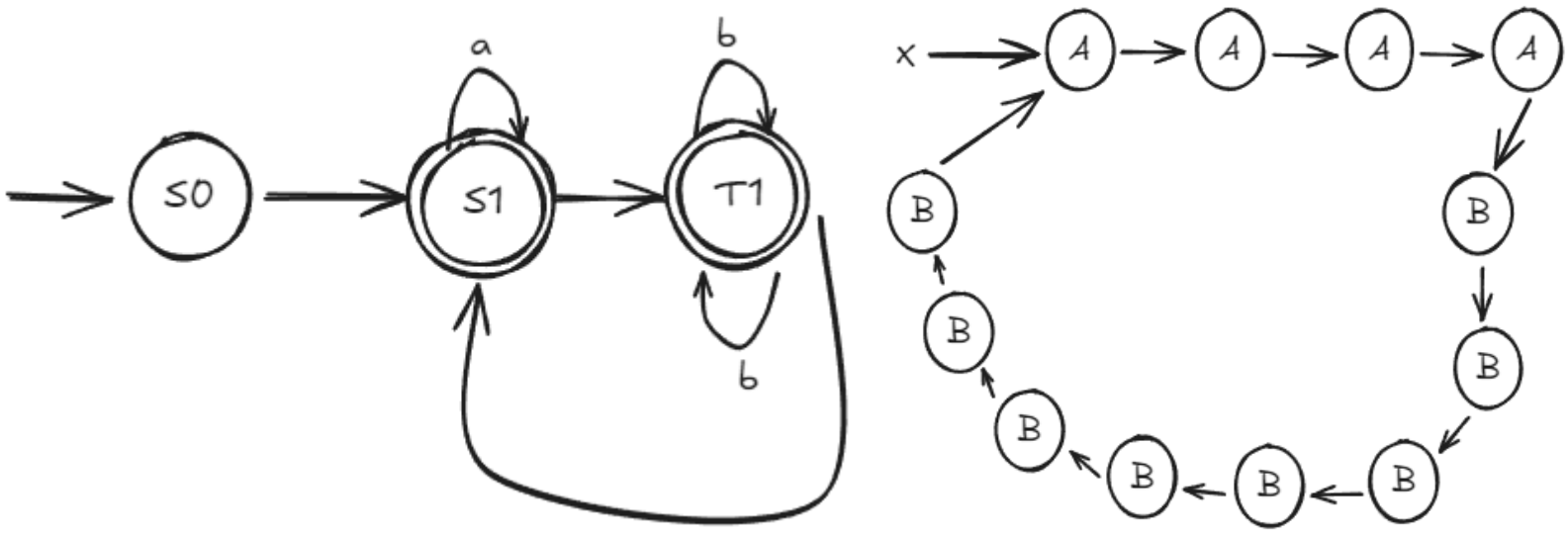
✓ **Observation**

After training for **100,000 epochs**, the network achieved a final loss of **0.0323**, indicating strong convergence and learning.
We examined the hidden layer activations and output probabilities on a long concatenated sequence:
`Input: AAABBBBBBAAAABBBBBBBBAABBBBABBAAABBBBBBA`

Output prediction highlights:

- **Initial A's** in each subsequence were correctly classified with high confidence.
  Example:
  `A [ 0.85 −0.69] → [0.83, 0.17]`
- **Final B** of each subsequence was predicted correctly, even after a long history.
  `B [−1.00 0.84] → [0.00, 1.00]`
  The network showed **transitions in hidden states** between the A and B phases:
  - A-phase activations: positive on first dimension (e.g., `[0.8, −0.6]`)
  - B-phase activations: negative on first dimension (e.g., `[−1.0, 0.8]`)
  - Transitions were **smooth and consistent**, confirming the network's ability to encode sequence structure.
- For most subsequences, both the **last B** and **first A of the next subsequence** were correctly classified, which is the key challenge in modeling `anb2n` patterns.

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 .



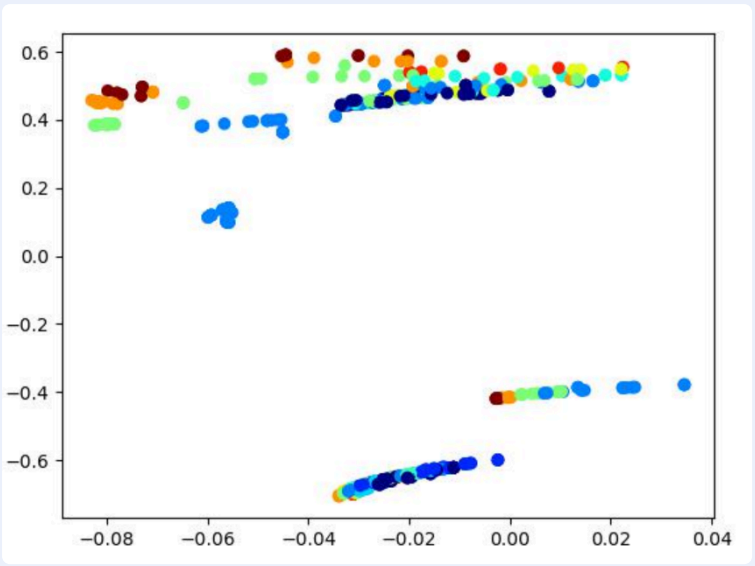
3. Briefly explain how the network accomplishes the a nb n task. Specifically, you should describe how the hidden unit activations change as the string is processed, and how it is able to correctly predict all B's after the first B, as well as the initial A following the last B in the sequence.

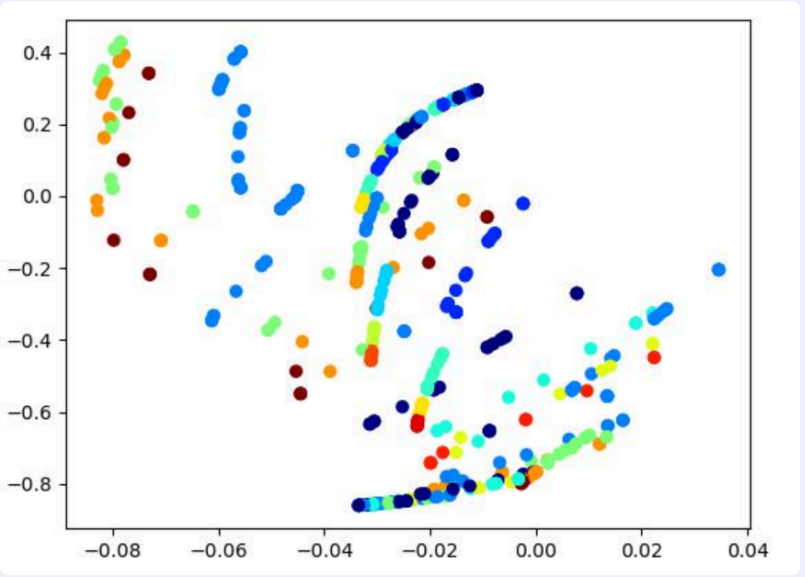
> **Explanation of How the Network Accomplishes**
>
> The network learns to accomplish the $a^n b^{2n}$ task by using its hidden units to internally **count** the number of input symbols and maintain a **stateful representation** of its progress through the sequence.
>
> ## During the sequence of A's
>
> - The hidden units incrementally activate in response to each 'a', effectively **counting the number of a's** seen so far.
> - This counting is stored in a distributed form across hidden unit activations.
>
> ## Upon reading the first B:
>
> - The network transitions to a different internal state, signaling that the first phase (a's) has ended.
> - From this point, the hidden state begins to **decrease or decay in steps**, allowing the network to track how many B's have been seen.
>
> ## For each B after the first:
>
> - The network uses its memory of how many a's were seen to **expect exactly 2n B's**.
> - It maintains hidden state dynamics such that **B is predicted until the 2n-th one** is reached.
>
> ## After the last B:
>
> - The hidden units return to a **recognizable reset state**, allowing the network to correctly predict an 'a' if the sequence restarts.
> - This indicates the network has correctly matched the b^{2n} and is ready for a potential new pattern.
>
> ## Summary
>
> The network relies on **hidden unit dynamics** to track the transition point from a's to b's and to **count** in both directions, enabling it to learn and generalize the $a^n b^{2n}$ pattern.

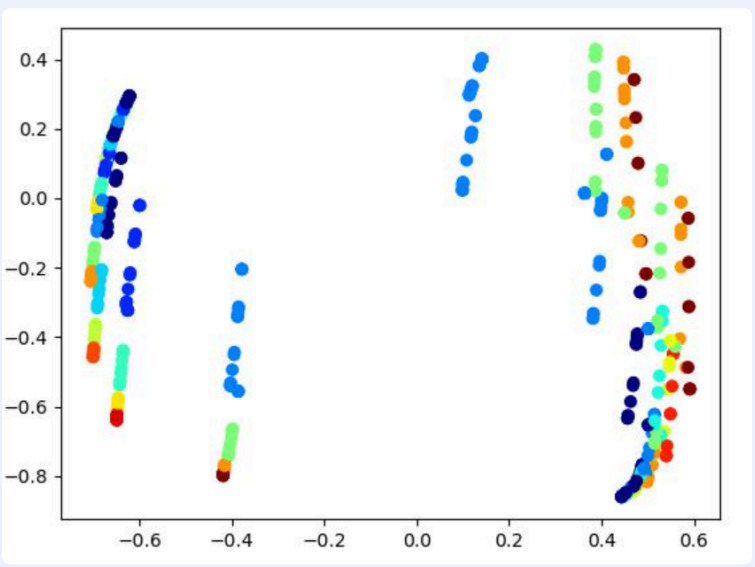
4. Train an LSTM with hidden nodes on the a nb nc n language prediction task by typing: `python3 seq_train.py --lang anb2nc3n --model lstm --hid 3`

Output Images

The training loss remained below 0.02 after 100000 epochs, indicating the network has successfully learned to predict the sequence $a^n b^{2n} c^{3n}$








```
epoch: 100000
 loss: 0.0072
```

Hidden State Analysis of Trained LSTM

After training for 100,000 epochs, the model achieved a loss below 0.01, indicating successful learning of the hierarchical pattern $a^n b^{2n} c^{3n}$. To better understand how the model represents this structure internally, we visualized the hidden states of three representative LSTM units during sequence processing.

Left plot (Hidden Unit 1):

This unit shows relatively stable activation levels during the initial part of the sequence and a clear separation of values between early and late stages. The change in activation aligns with the occurrence of input token `'a'`, suggesting this unit is responsible for **counting or detecting the number of a's**. Its values appear to remain stable during the later parts of the sequence.

Middle plot (Hidden Unit 2):

This unit exhibits a more dynamic response during the middle portion of the input, corresponding to the `'b'` tokens. The increased spread and variation in its activations suggest that it is likely responsible for **tracking the** `b^{2n}` **phase**, possibly encoding relative position or count information. This unit appears to transition gradually, indicating an ongoing computation rather than a hard reset.

Right plot (Hidden Unit 3):

This unit remains mostly inactive during the `'a'` and `'b'` phases but becomes highly activated during the final `'c'` segment. This strong change suggests that the unit is **dedicated to monitoring the** `c^{3n}` **part**

and possibly triggering a decision once the correct input length is reached. Notably, a sharp jump in activation may occur near the end of the sequence, possibly enabling the network to predict the final output `'A'`.

Summary

Overall, the visualization indicates that different hidden units specialize in tracking different parts of the hierarchical input sequence. This division of labor across hidden units allows the LSTM to **implicitly count and align segments**, enabling it to learn complex sequence patterns even without explicit supervision on intermediate steps.

5. This question is intended to be a bit more challenging. By annotating the generated images from Step (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 nb nc n prediction task (this might involve modifying the code so that it returns and prints out the context units as well as the hidden units).

🖉 Hidden State Dynamics

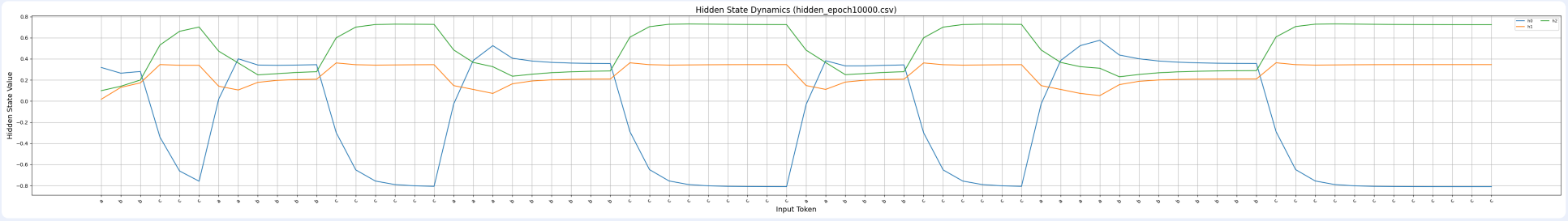
LSTM networks contain:

- **Hidden state (h_t)**: short-term information
- **Cell state (c_t)**: long-term memory ("context units")
- **Gates (input, forget, output)** that regulate information flow

How LSTM handles $a^n b^n c^n$:

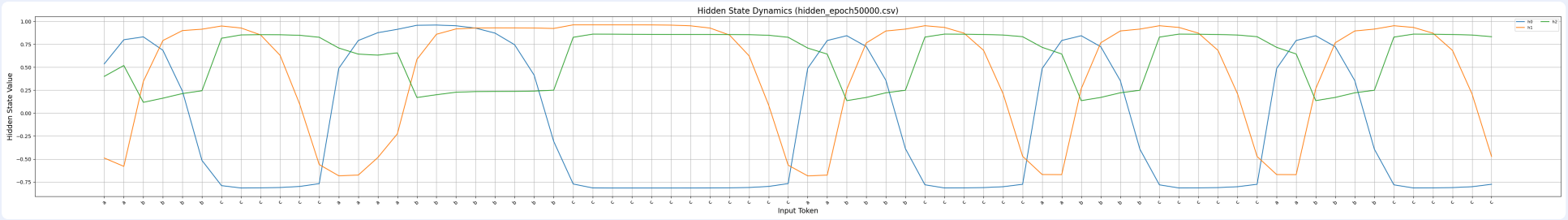
- **During 'a' phase**:
  - The LSTM increments an internal count (via `c t`) for every 'a'.
- **During 'b' phase**:
  - It starts decrementing a value, *matching each 'b' to a previously seen 'a'*.
- **During 'c' phase**:
  - It matches each 'c' to the earlier count of 'b' (which equals 'a').

10000 training epochs



The figure shows the evolution of hidden states (h0, h1, h2) after 10,000 training epochs when processing a sequence of the form $a^n b^{2n} c^{3n}$. We observe that each hidden dimension responds distinctly to different phases of the input. For example, h0 drops sharply at each a-to-b transition, h1 rises during the b-to-c phase, while h2 remains relatively stable. These dynamics suggest that the LSTM has learned to encode the structural patterns and even approximate the counting relationships in the sequence. The smooth and repeated patterns across segments confirm that the model has generalized well and developed internal state tracking of the sequence structure.

50000 training epochs



The figure above visualizes the hidden state trajectories (h_0, h_1, h_2) of the trained LSTM model over a token sequence composed of repeated patterns of the form $a^n b^{2n} c^{3n}$, extracted at epoch 50000. The x-axis corresponds to input tokens, and the y-axis represents the activation values of each hidden unit.
Each group of tokens starts with multiple `'a'`s, followed by a doubled number of `'b'`s and tripled number of `'c'`s. As shown, the hidden units exhibit distinct and consistent state transitions in response to the changing segments of the input:

- **Boundary Recognition:** At the transition from one character type to another (e.g., from `'a'` to `'b'`), the hidden state values change sharply, indicating that the LSTM has learned to detect character boundaries.
- **Counting and Repetition Encoding:** The duration of stable hidden states for each segment increases in proportion to the number of tokens, reflecting the LSTM's ability to encode quantitative relationships — specifically, that there are twice as many `'b'`s and three times as many `'c'`s as `'a'`s.
- **Pattern Generalization:** Across repeated input patterns, the hidden states follow periodic and consistent dynamics. For instance, h_0 increases during the `'a'` segment, h_1 dominates during `'b'`, and h_2 becomes prominent during `'c'`. This indicates the model generalizes the input structure across repetitions.

Comparison&Summary

By comparing the hidden state dynamics at epoch 10000 and epoch 50000, we observe a clear progression in the model's learning behavior:

- At **epoch 10000**, the hidden state values fluctuate mildly within a narrow range (approximately between -0.3 and 0.6). The transitions between character groups (such as from `'a'` to `'b'`, or `'b'` to `'c'`) are relatively smooth, and the structure of the hidden state dynamics appears inconsistent across different repetitions of the "abc" pattern. This suggests that the model has not yet fully learned the underlying structure of the $a^n b^{2n} c^{3n}$ sequences.
- In contrast, at **epoch 50000**, the hidden state trajectories show much more distinct and structured behavior. The values vary across a wider range (approximately from -0.8 to 1.0), indicating a stronger and more confident response to specific input patterns. Sharp transitions are visible at character boundaries, and each segment of the repeated "abc" sequence exhibits highly similar hidden state curves, reflecting the model's successful generalization of the sequence pattern. Additionally, different hidden units exhibit specialized roles—for example, one unit might consistently respond to `'a'`, another to `'b'`, and another to `'c'`.

These findings confirm that the LSTM gradually learns to segment and classify structured sequences through training. By epoch 50000, it demonstrates clear state encoding, pattern repetition detection, and unit-wise specialization, all of which are crucial for correctly modeling nested patterns like $a^n b^{2n} c^{3n}$.