COMP9444
25T2 Jinzhou Peng (z5570973)

Part 1: Japanese Character Recognition

1. NetLin Implementation

Model Description:
NetLin implements a linear function of the pixels in the image, followed by log softmax. This is the simplest possible neural network for classification.

Implementation:

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

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

Results:

- **Final Accuracy:** 70%
- **Final Loss:** 1.0084
- **Number of Parameters:** 7,850 (784×10 weights + 10 biases)

Confusion Matrix:

```
[[765.   5.   8.  13.  30.  66.   2.  61.  31.  19.]
 [  7. 670. 109.  18.  27.  24.  58.  12.  25.  50.]
 [  7.  61. 692.  27.  28.  21.  45.  37.  45.  37.]
 [  5.  34.  63. 757.  14.  54.  14.  18.  29.  12.]
 [ 61.  54.  79.  21. 623.  20.  31.  34.  19.  58.]
 [  8.  27. 124.  16.  20. 724.  28.   9.  34.  10.]
 [  5.  23. 146.   9.  24.  25. 725.  19.  10.  14.]
 [ 18.  29.  26.  12.  85.  21.  53. 620.  89.  47.]
 [ 11.  37.  89.  39.   7.  29.  47.   7. 712.  22.]
 [  8.  49.  89.   3.  54.  32.  20.  27.  38. 680.]]
```

2. NetFull Implementation

Model Description:
NetFull implements a fully connected 2-layer network with one hidden layer using tanh activation and log softmax at the output.

Implementation:

```python
class NetFull(nn.Module):
    def __init__(self):
        super(NetFull, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 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)
```

Results:

- **Final Accuracy:** 84%
- **Final Loss:** 0.5264
- **Number of Parameters:** 101,770 (784×128 + 128 + 128×10 + 10)

Parameter Calculation:

- First layer: 784 × 128 = 100,352 weights + 128 biases = 100,480
- Second layer: 128 × 10 = 1,280 weights + 10 biases = 1,290
- Total: 100,480 + 1,290 = 101,770 parameters

Confusion Matrix:

```
[[855.   3.   1.   6.  29.  33.   4.  31.  31.   7.]
 [  4. 803.  33.   6.  18.  15.  65.   7.  22.  27.]
 [  7.  12. 840.  42.  13.  12.  25.  15.  21.  13.]
 [  4.  11.  30. 916.   2.  15.   3.   4.   8.   7.]
 [ 43.  31.  24.  12. 802.   4.  37.  14.  21.  12.]
 [  9.  11.  83.  12.  14. 821.  23.   2.  15.  10.]
 [  3.   8.  61.  11.  18.   3. 878.  10.   1.   7.]
 [ 23.  17.  25.   8.  23.   8.  33. 810.  24.  29.]
 [ 13.  30.  28.  54.   4.   8.  30.   2. 821.  10.]
 [  4.  19.  56.   4.  26.   4.  21.  20.  11. 835.]]
```

3. NetConv Implementation

Model Description:
NetConv implements a convolutional network with two convolutional layers plus one fully connected layer, all using ReLU activation function, followed by log softmax.

Implementation:

```
class NetConv(nn.Module):
    def __init__(self):
        super(NetConv, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, 5, 1)
        self.conv2 = nn.Conv2d(64, 128, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(128 * 5 * 5, 256)
        self.fc2 = nn.Linear(256, 10)

```

```
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.dropout2(x)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Results:

- **Final Accuracy:** 94%
- **Final Loss:** 0.2158
- **Number of Parameters:** 838,922

Parameter Calculation:

- Conv1: (5×5×1 + 1) × 64 = 1,664 parameters
- Conv2: (3×3×64 + 1) × 128 = 73,856 parameters
- FC1: (128×5×5 + 1) × 256 = 819,456 parameters
- FC2: (256 + 1) × 10 = 2,570 parameters
- Total: 1664 + 73856 + 819456 + 2570 = 897546 parameters

Confusion Matrix:

```
[[956.   3.   2.   1.  29.   2.   0.   5.   1.   1.]
 [  1. 918.  11.   0.  14.   2.  39.   5.   2.   8.]
 [ 12.   1. 879.  46.   7.   8.  26.   3.   8.  10.]
 [  1.   1.  10. 975.   0.   5.   3.   1.   2.   2.]
 [ 25.   5.   0.   9. 924.   3.  17.   1.  13.   3.]
 [  2.   5.  28.   7.   3. 934.  15.   0.   3.   3.]
 [  4.   1.  15.   2.   2.   2. 971.   2.   0.   1.]
 [  5.   2.   6.   1.   8.   3.   8. 947.   8.  12.]
 [  3.   9.   5.   8.   7.   5.   5.   1. 957.   0.]
 [  9.   4.   9.   2.  19.   2.   8.   1.   9. 937.]]
```

4. Discussion of Results

Relative Accuracy of the Three Models

The experimental results show a clear hierarchy in model performance:

- **NetConv: 94%** - Highest accuracy, exceeding the 93% requirement
- **NetFull: 84%** - Good performance, meeting the 84% requirement
- **NetLin: 70%** - Baseline performance, meeting the ~70% expectation

The convolutional network significantly outperforms both fully connected networks, demonstrating the effectiveness of convolutional layers for image recognition tasks. The improvement from NetLin to NetFull (14% increase) shows the importance of non-linear transformations, while the improvement from NetFull to NetConv (10% increase) demonstrates the power of spatial feature extraction.

Number of Independent Parameters

1. **NetLin: 7,850 parameters** - Minimal parameter count with direct pixel-to-class mapping
2. **NetFull: 101,770 parameters** - Moderate parameter count with hidden layer representation

3. **NetConv: 897,546 parameters** - Highest parameter count with hierarchical feature learning

Interestingly, while NetConv has the most parameters, it achieves the best generalization performance, indicating that the convolutional architecture provides effective regularization through weight sharing and spatial locality.

Confusion Matrix Analysis

Character Mapping: 0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo"

NetLin Confusion Patterns:

- High confusion between characters 2 (su) and 6 (ma): 146 misclassifications
- Significant confusion between characters 1 (ki) and 2 (su): 109 misclassifications
- Character 4 (na) frequently misclassified as 0 (o): 61 instances

NetFull Improvements:

- Reduced confusion overall with higher diagonal values
- Character 2 (su) still confused with 6 (ma) but reduced to 61 instances
- Better distinction between most character pairs

NetConv Excellence:

- Minimal confusion with very high diagonal values (>90% for most characters)
- Character 3 (tsu) achieves 97.5% accuracy (975/1000)
- Remaining confusions are mostly between structurally similar characters:
    - Characters 2 (su) and 3 (tsu): 46 misclassifications
    - Characters 0 (o) and 4 (na): 29 misclassifications

Why These Confusions Occur:

- **Structural Similarity:** Characters like "su" and "tsu" share similar stroke patterns
- **Spatial Features:** Linear models cannot capture spatial relationships, leading to confusion between characters with similar pixel distributions
- **Local Patterns:** Convolutional networks excel at detecting local features (strokes, curves) that distinguish similar characters

The progression from NetLin → NetFull → NetConv shows increasingly sophisticated feature extraction capabilities, with convolutional layers providing the best spatial understanding for handwritten character recognition.

Part 2: 2-Layer Neural Networks

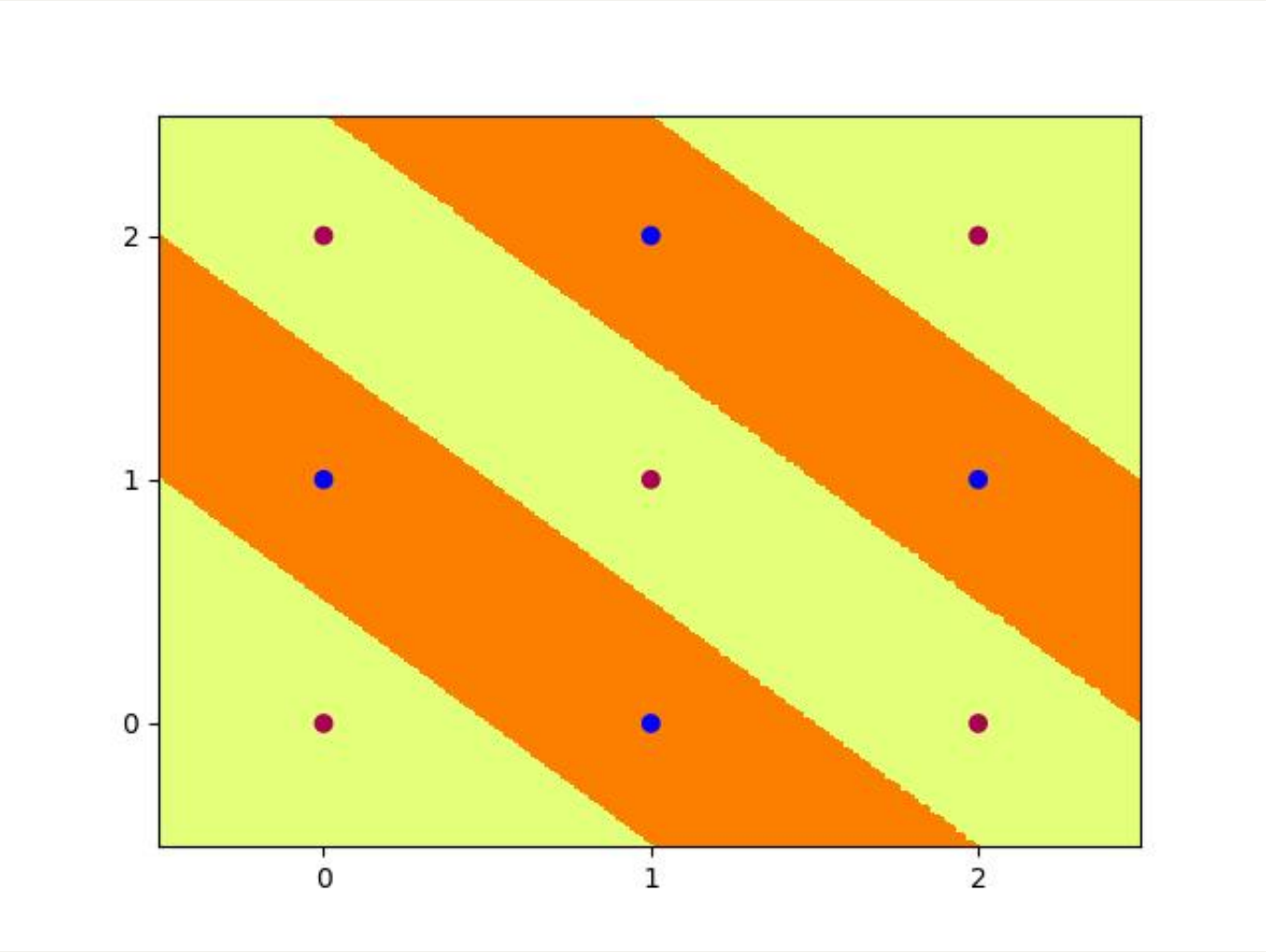
Data Classification Problem

The task involves classifying the following 9 data points:

X	Y	TARGET
0	0	0
0	1	1
0	2	0
1	0	1
1	1	0
1	2	1
2	0	0
2	1	1
2	2	0

The pattern shows that points where (x+y) is odd should be classified as 1, while points where (x+y) is even should be classified as 0.

Data Visualization:



Initial data distribution showing the checkerboard pattern

Part 2.1: Sigmoid Network with 5 Hidden Nodes

A 2-layer neural network with 5 hidden nodes using sigmoid activation was trained successfully to achieve 100% accuracy.

Command used:

```
python3 check_main.py --act sig --hid 5
```

Final Results:

- **Accuracy:** 100.0%
- **Training epochs:** 6900

- **Final loss:** 0.1572

Final Weights:

- **Input to Hidden weights:**

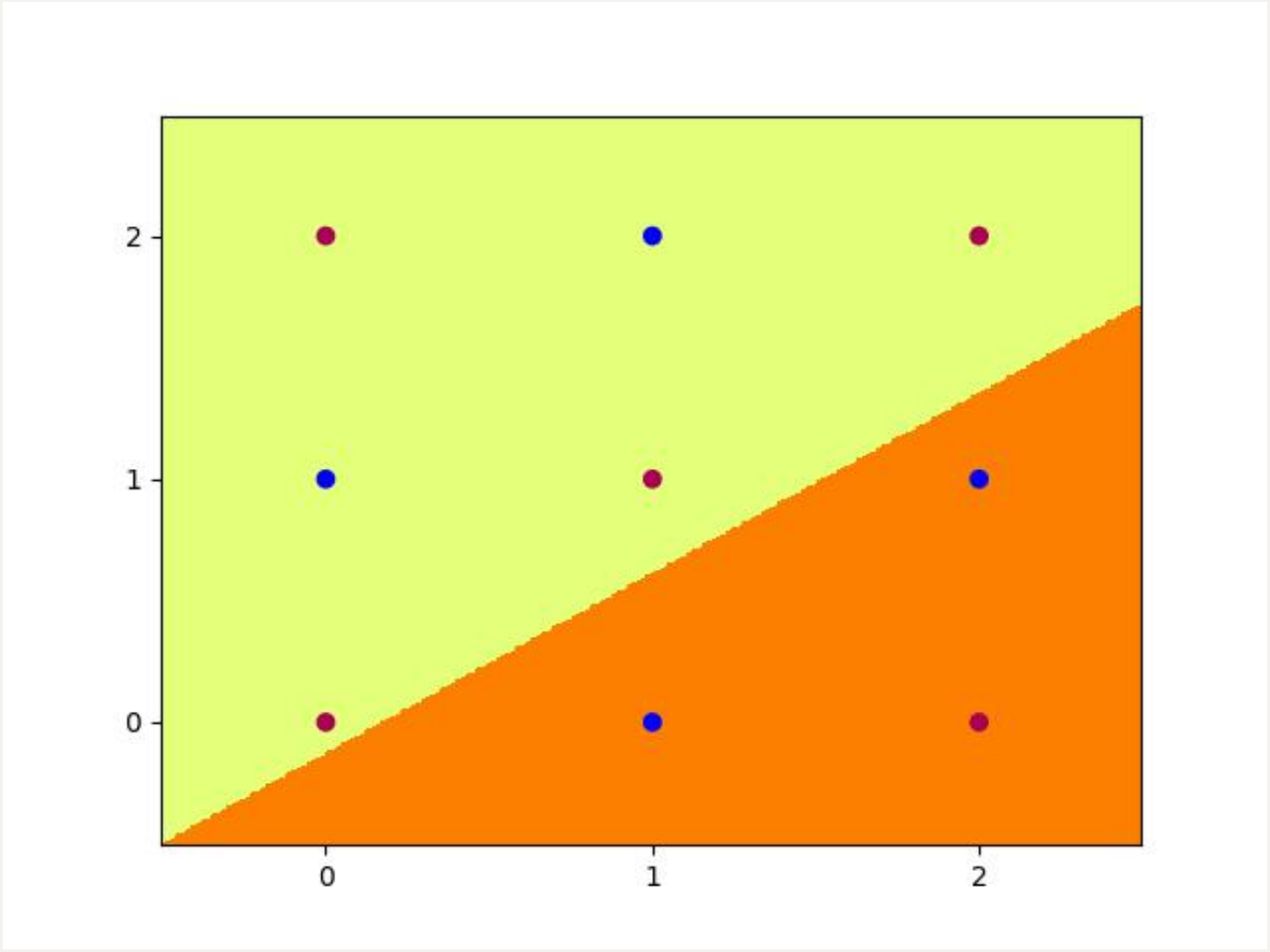
```
[[ 5.9918, -8.0904],
 [-6.3039, -6.2848],
 [-6.5462,  5.1155],
 [ 5.6882, -0.3589],
 [-0.1807,  4.2494]]
```

- **Hidden biases:** `[-1.0168, 1.1771, -7.9317, -5.6044, -3.4962]`
- **Hidden to Output weights:** `[7.3793, -5.9655, -7.2185, -6.9491, 6.2850]`
- **Output bias:** `[-2.1268]`

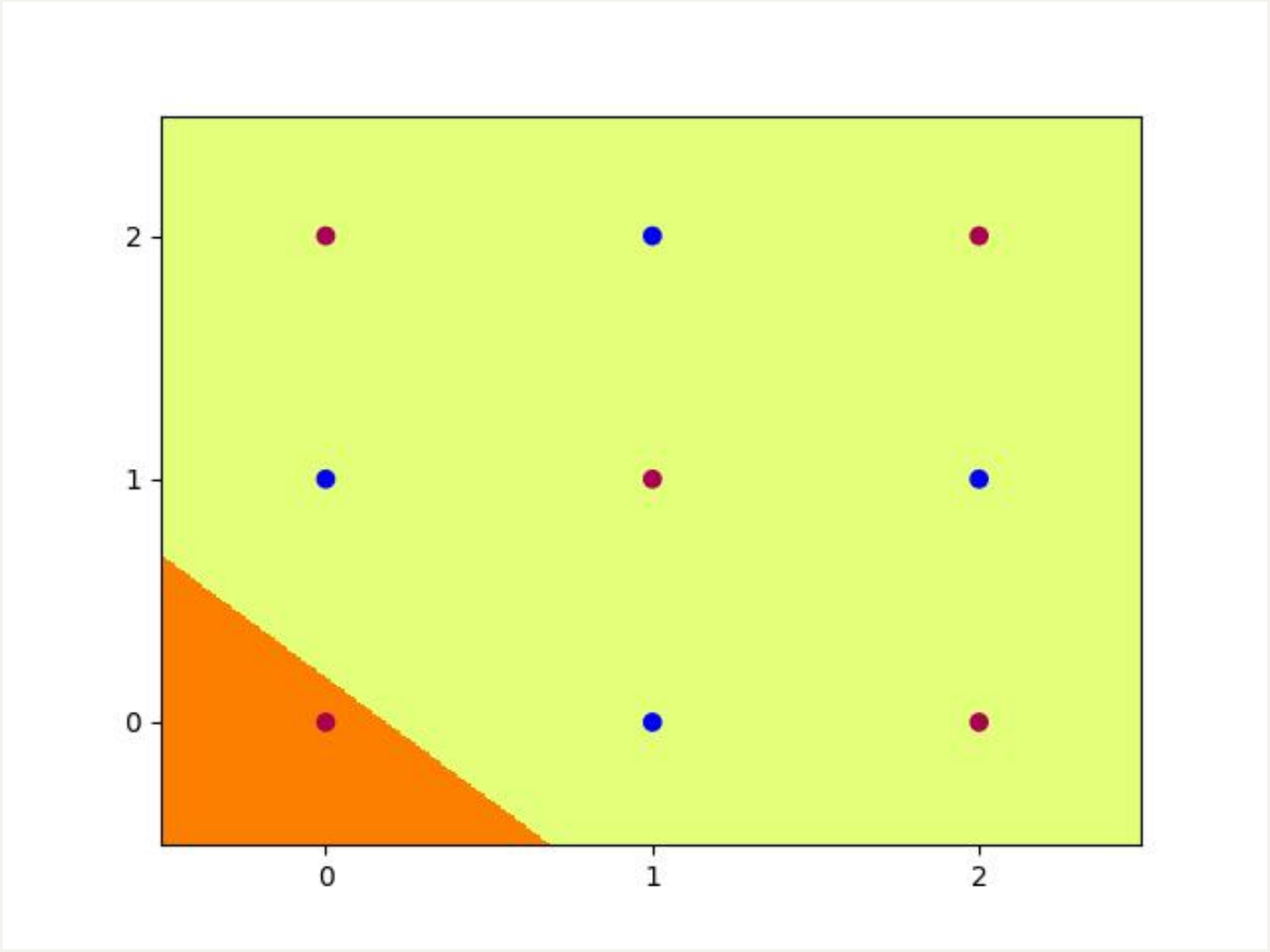
Generated Images:

- `hid_5_0.jpg` to `hid_5_4.jpg`: Functions computed by each hidden node
- `out_5.jpg`: Overall network function

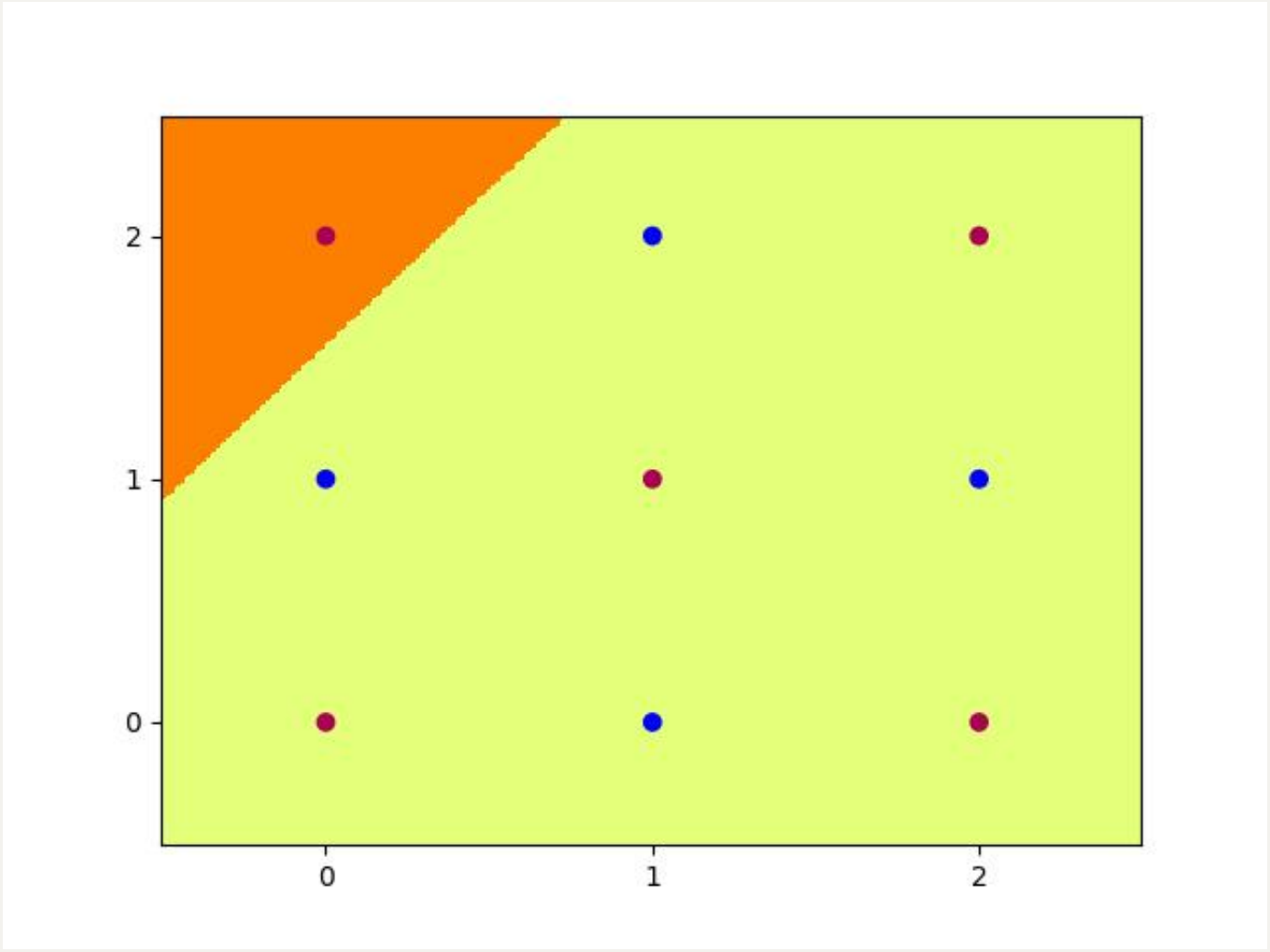
Network Visualization:



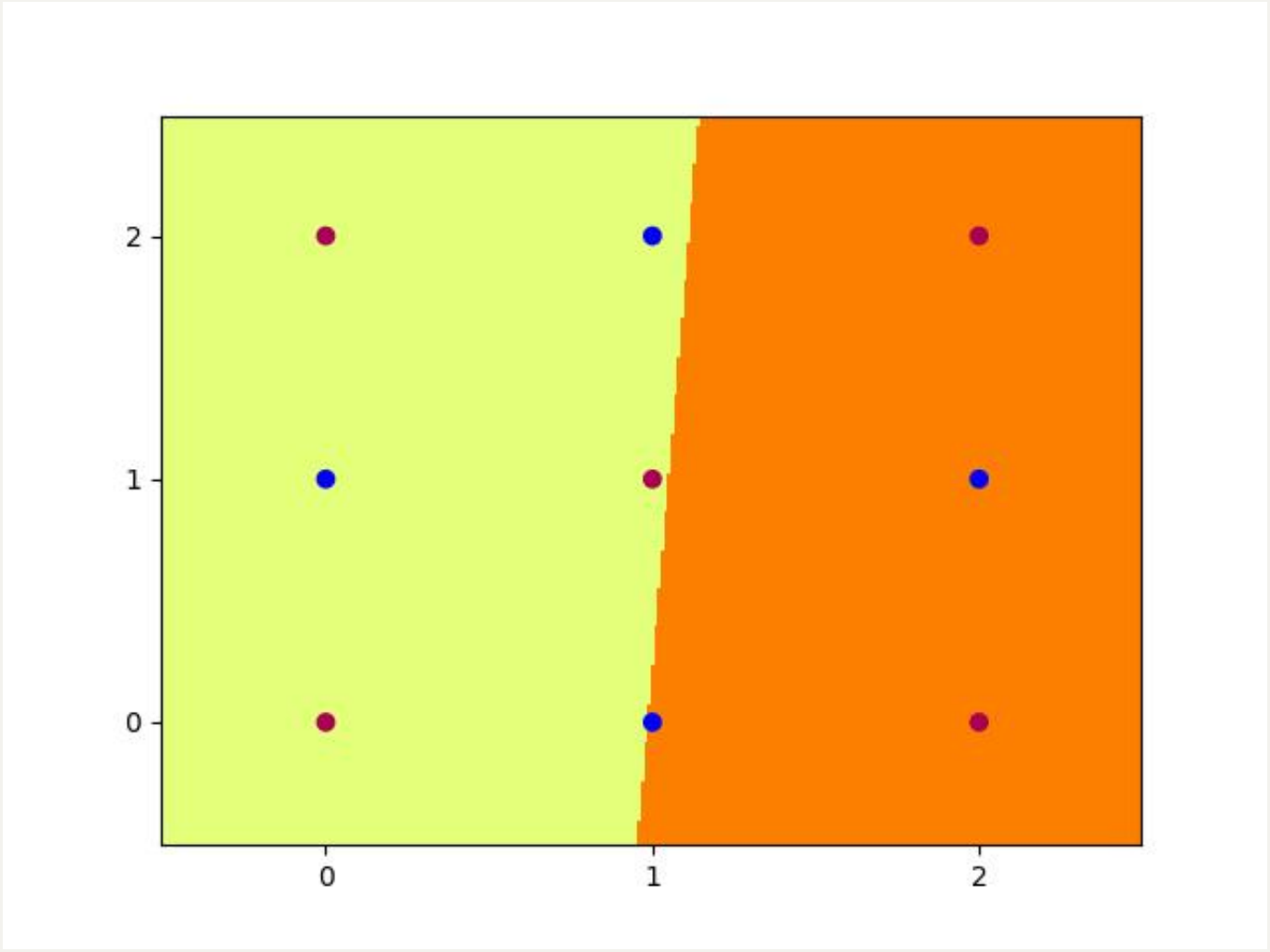
Hidden Node 0 Function



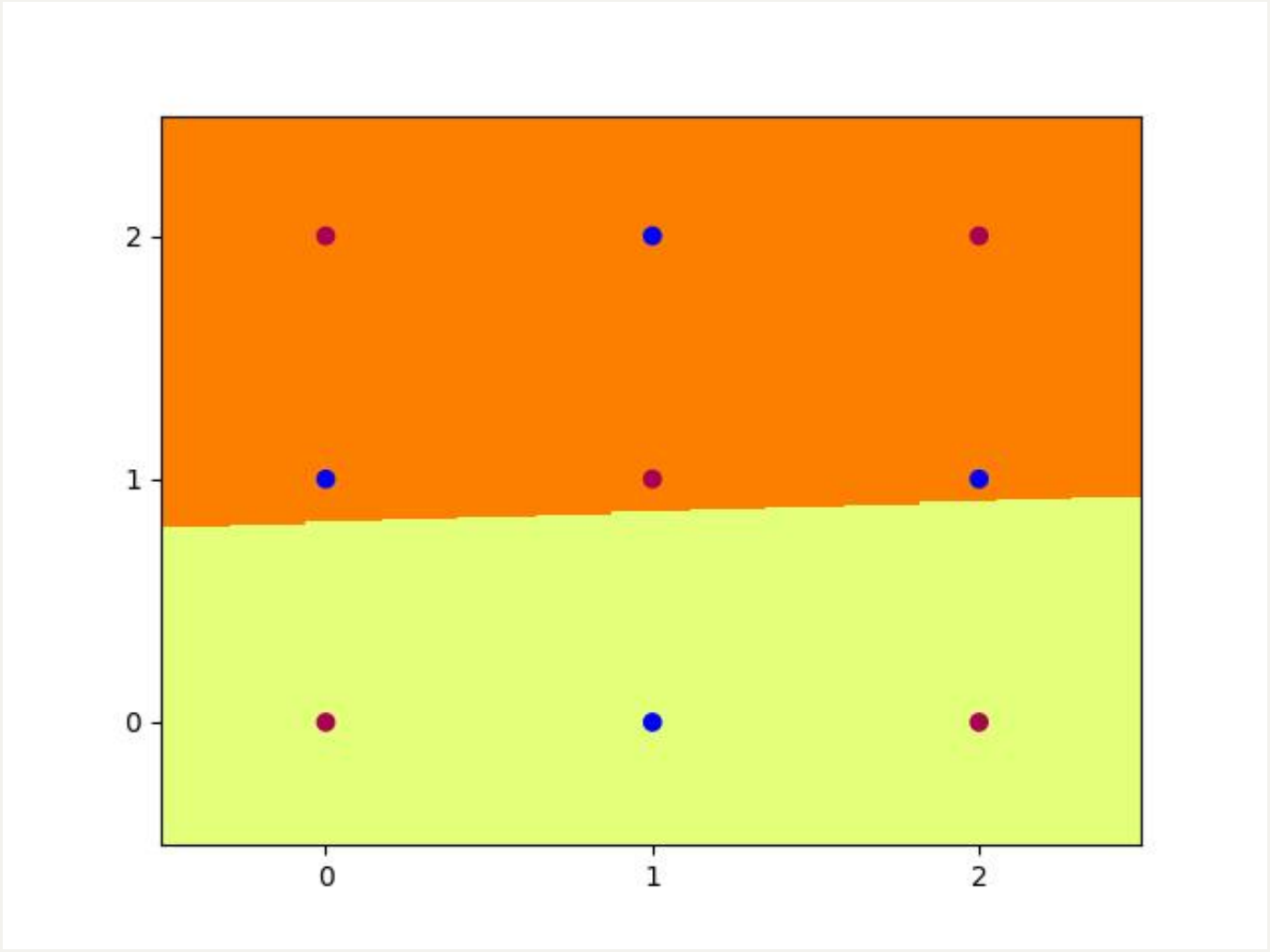
Hidden Node 1 Function



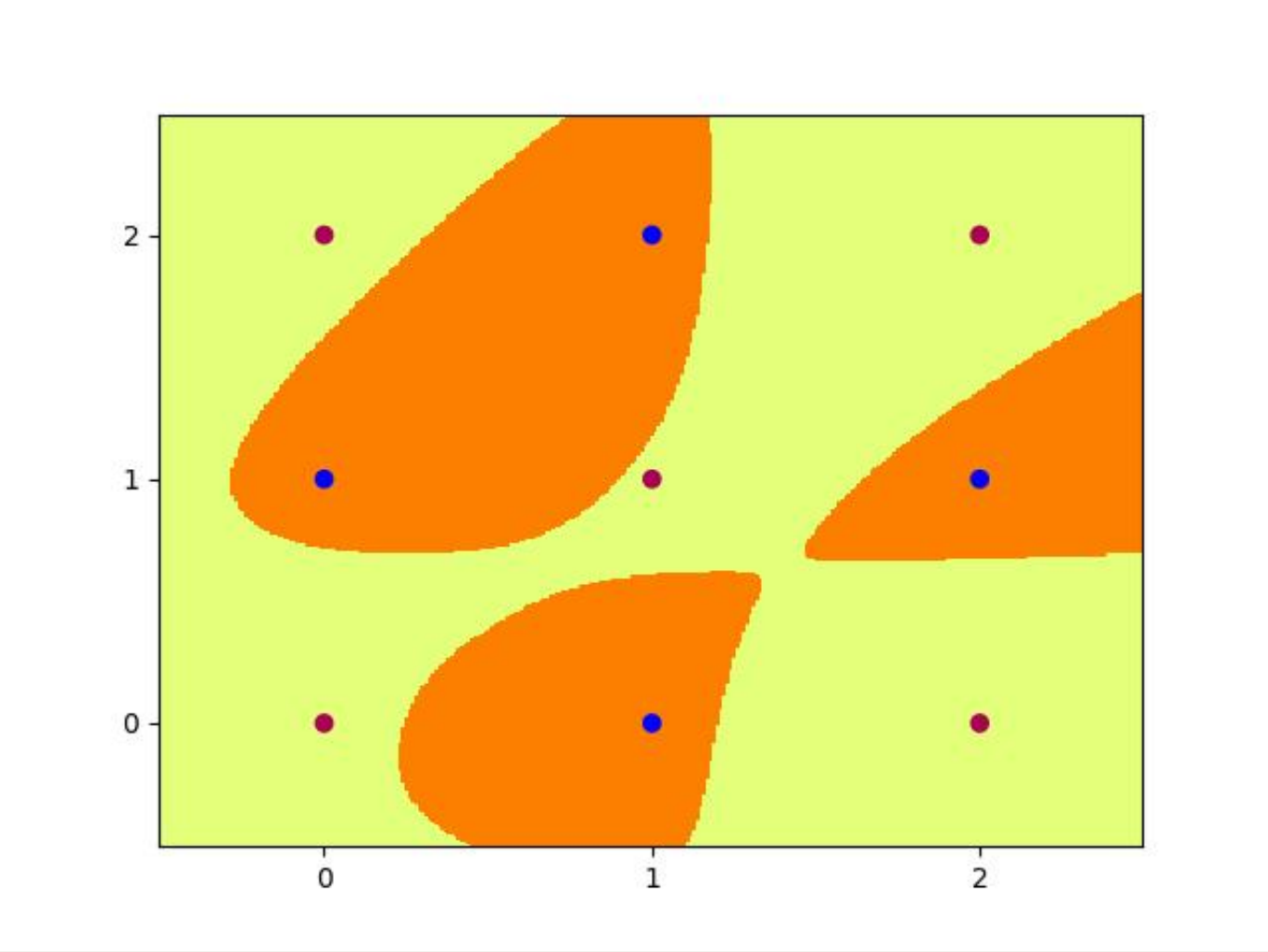
Hidden Node 2 Function



Hidden Node 3 Function



Hidden Node 4 Function



Overall Network Function (5 Hidden Nodes)

Part 2.2: Hand-Designed Heaviside Network with 4 Hidden Nodes

Network Architecture

A 2-layer neural network with 4 hidden nodes using Heaviside (step) activation was designed by hand to correctly classify the data.

Network Diagram

```
Input Layer (2 nodes: x, y)
     ↓
Hidden Layer (4 nodes with Heaviside activation)
     ↓
Output Layer (1 node with Heaviside activation)
```

Weights and Biases

Input to Hidden Layer:

- **Node 0:** $w_0 = [1, 1]$, $b_0 = -0.5$ (computes $x + y \geqslant 1$)
- **Node 1:** $w_1 = [-1, -1]$, $b_1 = 3.5$ (computes $-(x + y) \geqslant -3$, i.e., $x + y \leqslant 3$)
- **Node 2:** $w_2 = [-1, -1]$, $b_2 = 1.5$ (computes $-(x + y) \geqslant -1$, i.e., $x + y \leqslant 1$)
- **Node 3:** $w_3 = [1, 1]$, $b_3 = -2.5$ (computes $x + y \geqslant 2.5$)

Hidden to Output Layer:

- **Weights:** w_out = [1, 1, 1, 1]
- **Bias:** b_out = -2.5

Dividing Lines

Each hidden node determines a dividing line:

1. **Hidden Node 0:** $x + y = 1$ (activates when $x + y \geqslant 1$)
2. **Hidden Node 1:** $x + y = 3$ (activates when $x + y \leqslant 3$)
3. **Hidden Node 2:** $x + y = 1$ (activates when $x + y \leqslant 1$)
4. **Hidden Node 3:** $x + y = 2.5$ (activates when $x + y \geqslant 2.5$)

Activation Table

INPUT (X,Y)	X+Y	H_0	H_1	H_2	H_3	OUTPUT	TARGET
(0,0)	0	0	1	1	0	0	0
(0,1)	1	1	1	1	0	1	1
(0,2)	2	1	1	0	0	0	0
(1,0)	1	1	1	1	0	1	1
(1,1)	2	1	1	0	0	0	0
(1,2)	3	1	1	0	1	1	1
(2,0)	2	1	1	0	0	0	0
(2,1)	3	1	1	0	1	1	1
(2,2)	4	1	0	0	1	0	0

Verification Command:

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

Result: 100% accuracy achieved

Part 2.3: Rescaled Weights for Sigmoid Network

The hand-crafted weights from Part 2.2 were rescaled by multiplying all weights and biases by a factor of 10 to make the sigmoid activation mimic the step function behavior.

Rescaled Weights and Biases

Input to Hidden Layer:

- **Node 0:** $w_0 = [10, 10]$, $b_0 = -5$
- **Node 1:** $w_1 = [-10, -10]$, $b_1 = 35$
- **Node 2:** $w_2 = [-10, -10]$, $b_2 = 15$
- **Node 3:** $w_3 = [10, 10]$, $b_3 = -25$

Hidden to Output Layer:

- **Weights:** w_out = [10, 10, 10, 10]
- **Bias:** b_out = -25

Verification Command:

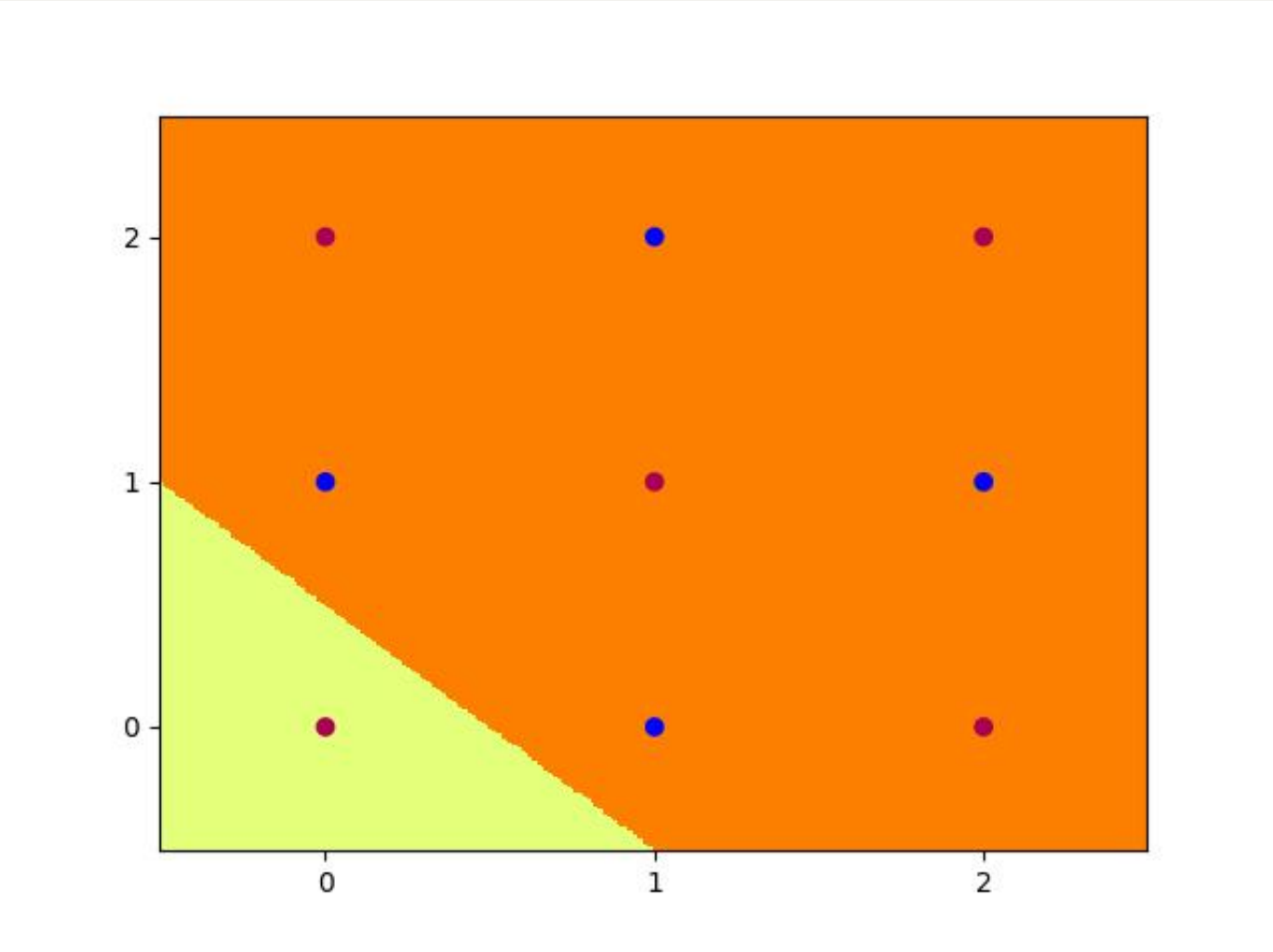
```
python3 check_main.py --act sig --hid 4 --set_weights
```

Result: 100% accuracy achieved

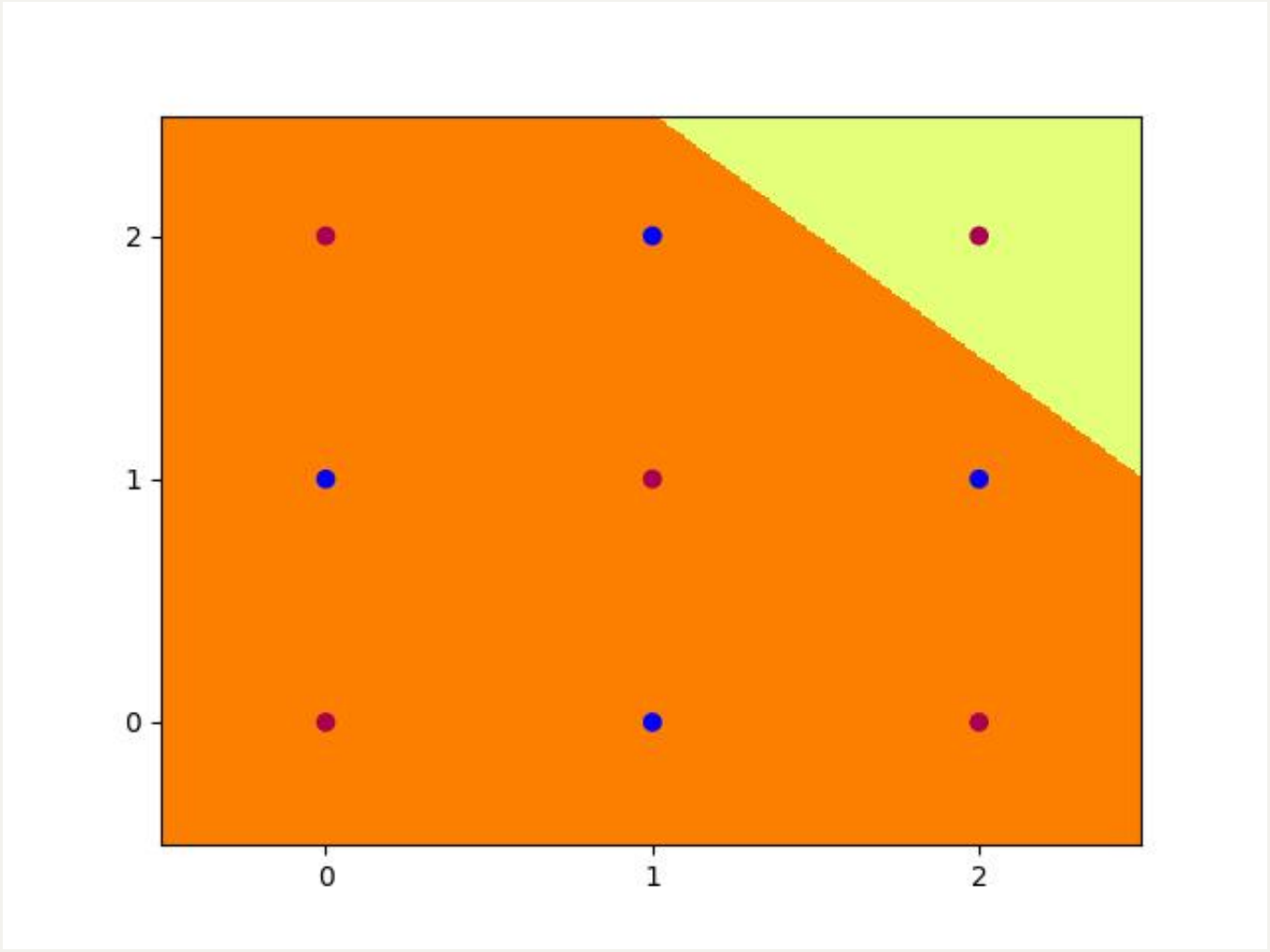
Generated Images:

- `hid_4_0.jpg` to `hid_4_3.jpg`: Functions computed by each hidden node
- `out_4.jpg`: Overall network function

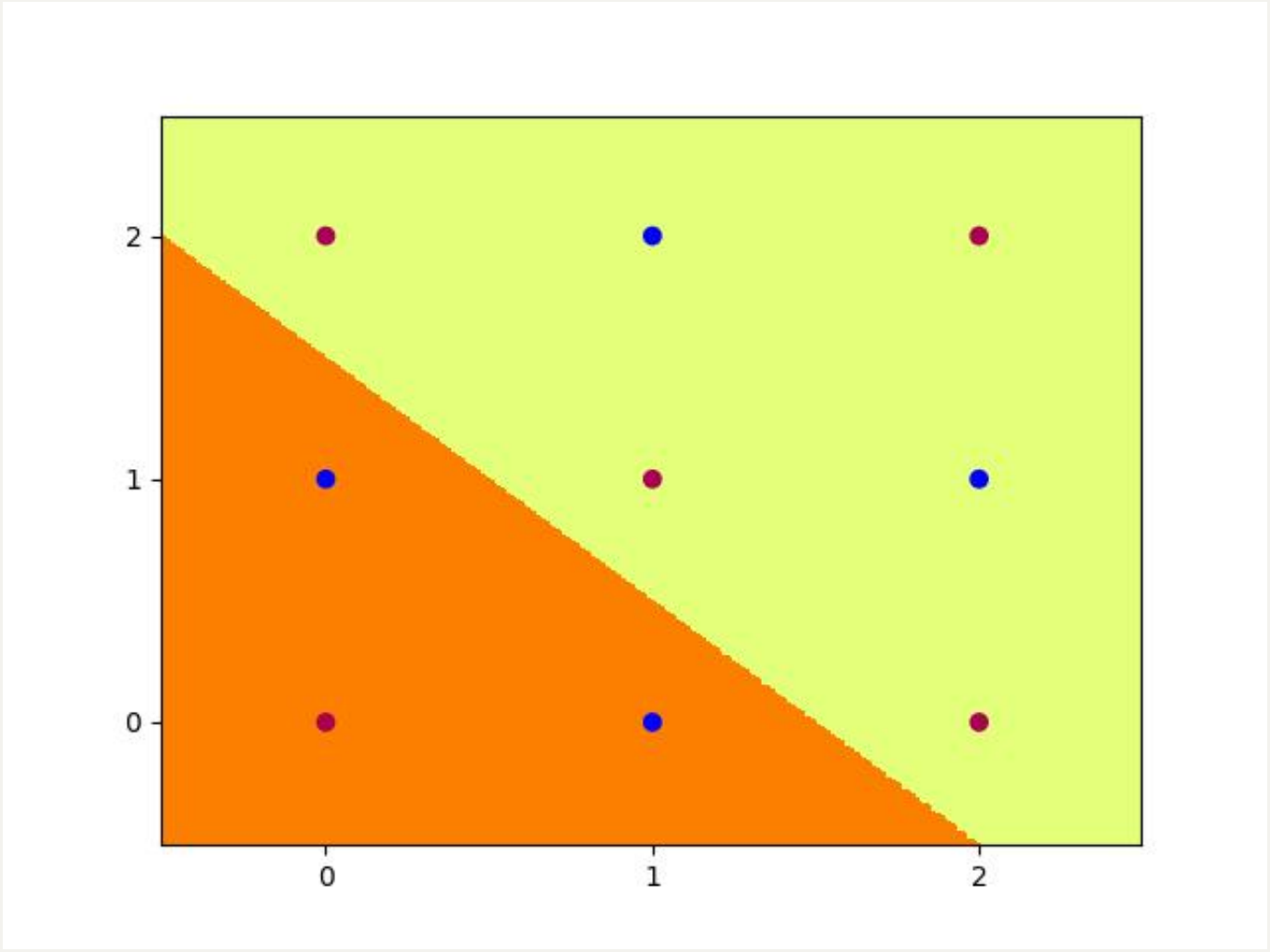
Network Visualization:



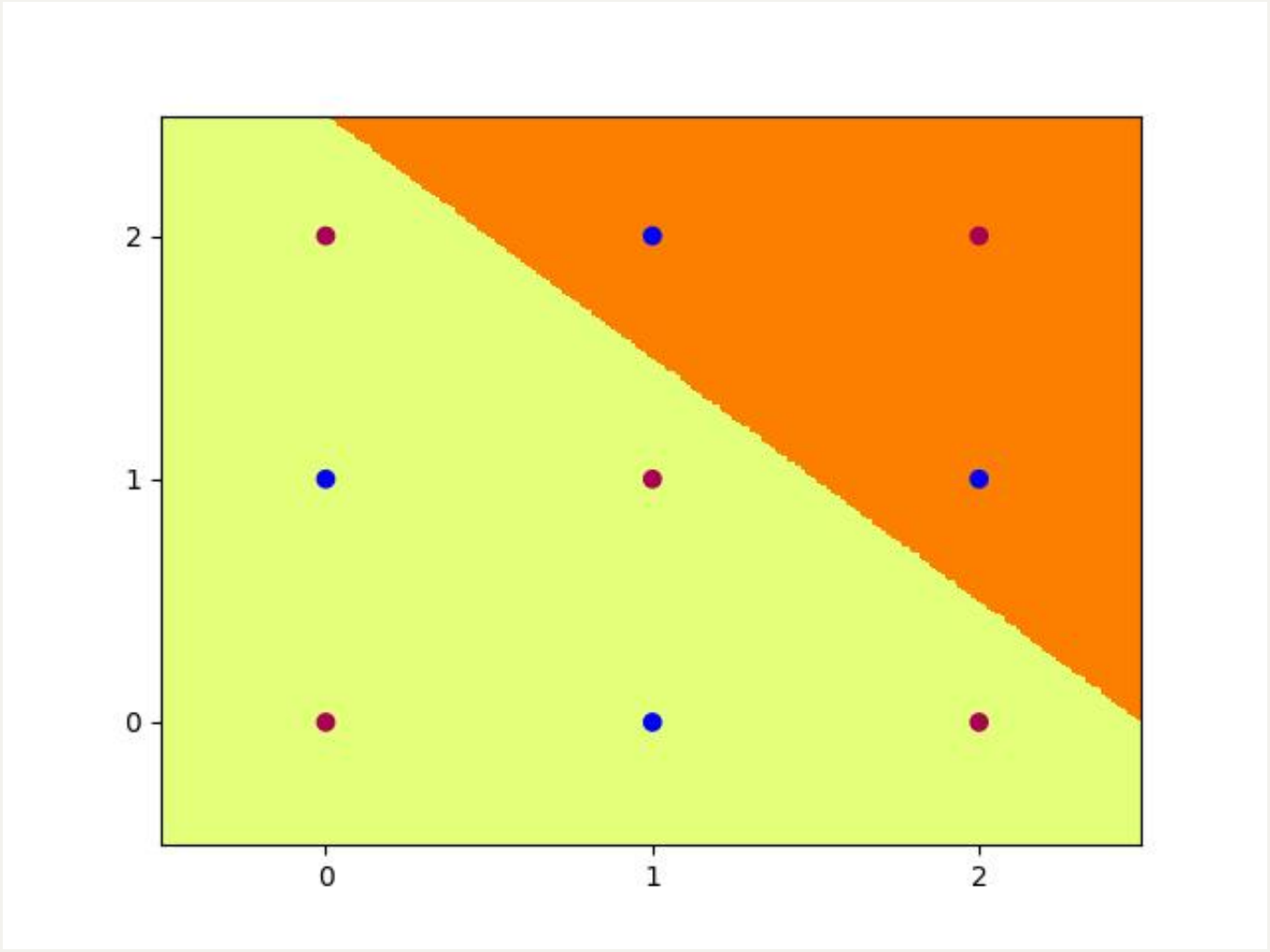
Hidden Node 0: $x + y \geq 1$



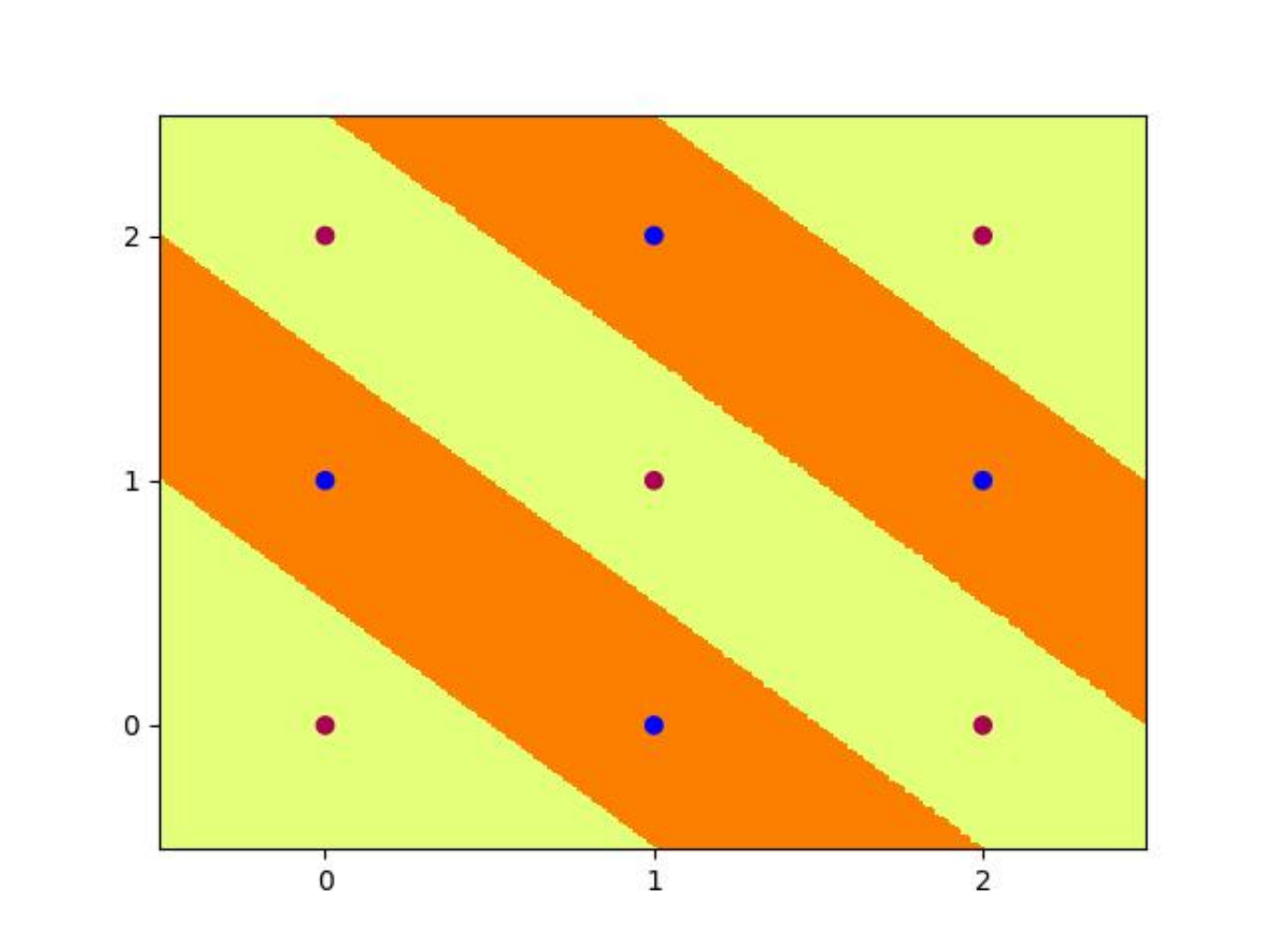
Hidden Node 1: $x + y \leqslant 3$



Hidden Node 2: $x + y \leqslant 1$



Hidden Node 3: $x + y \geqslant 2.5$



Overall Network Function (4 Hidden Nodes with Rescaled Weights)

Analysis and Discussion

Pattern Recognition

The data exhibits a checkerboard pattern where points with odd coordinate sums (x+y = 1 or 3) are classified as 1, while points with even coordinate sums (x+y = 0, 2, or 4) are classified as 0.

Network Design Strategy

The hand-designed network uses geometric constraints to identify the target pattern:

1. Nodes 0 and 1 ensure $1 \leqslant x+y \leqslant 3$
2. Nodes 2 and 3 help distinguish between odd and even sums within this range

3. The output layer combines these constraints to achieve the desired classification

Sigmoid vs. Heaviside

By scaling the weights by a factor of 10, the sigmoid function becomes very steep, effectively approximating the step function behavior. This demonstrates how continuous activation functions can approximate discrete ones when weights are sufficiently large.

Conclusion

Both approaches successfully achieved 100% classification accuracy:

- The trained sigmoid network learned the pattern through gradient descent
- The hand-designed network used geometric intuition to construct explicit decision boundaries
- Weight scaling successfully bridged the gap between step and sigmoid activations

Part 3: Hidden Unit Dynamics for Recurrent Networks

Introduction

This section analyzes the hidden unit dynamics of recurrent networks trained on two language prediction tasks: a^n b^{2n} and a^n b^{2n} c^{3n}. We examine how Simple Recurrent Networks (SRN) and Long Short-Term Memory (LSTM) networks learn to process these context-free languages.

Step 1: SRN Training on a^n b^{2n} Task

Training Results:

- Model: Simple Recurrent Network with 2 hidden nodes
- Task: a^n b^{2n} language prediction ($n \in \{1,2,3,4\}$)
- Final Loss: 0.0495 (below required threshold of 0.05)

- Architecture: 2 inputs, 2 hidden nodes, 2 outputs

Training Success:
The network successfully learned to predict:

- All B's after the first B with high confidence (probability ≈ 1.0)
- The transition from the last B to the next sequence's first A
- Appropriate probabilistic predictions for non-deterministic symbols

Hidden Unit Activation Plot:
The generated image `anb2n_srn2_01.jpg` shows the hidden unit activations in 2D space, with different colors representing different states in the finite state machine.

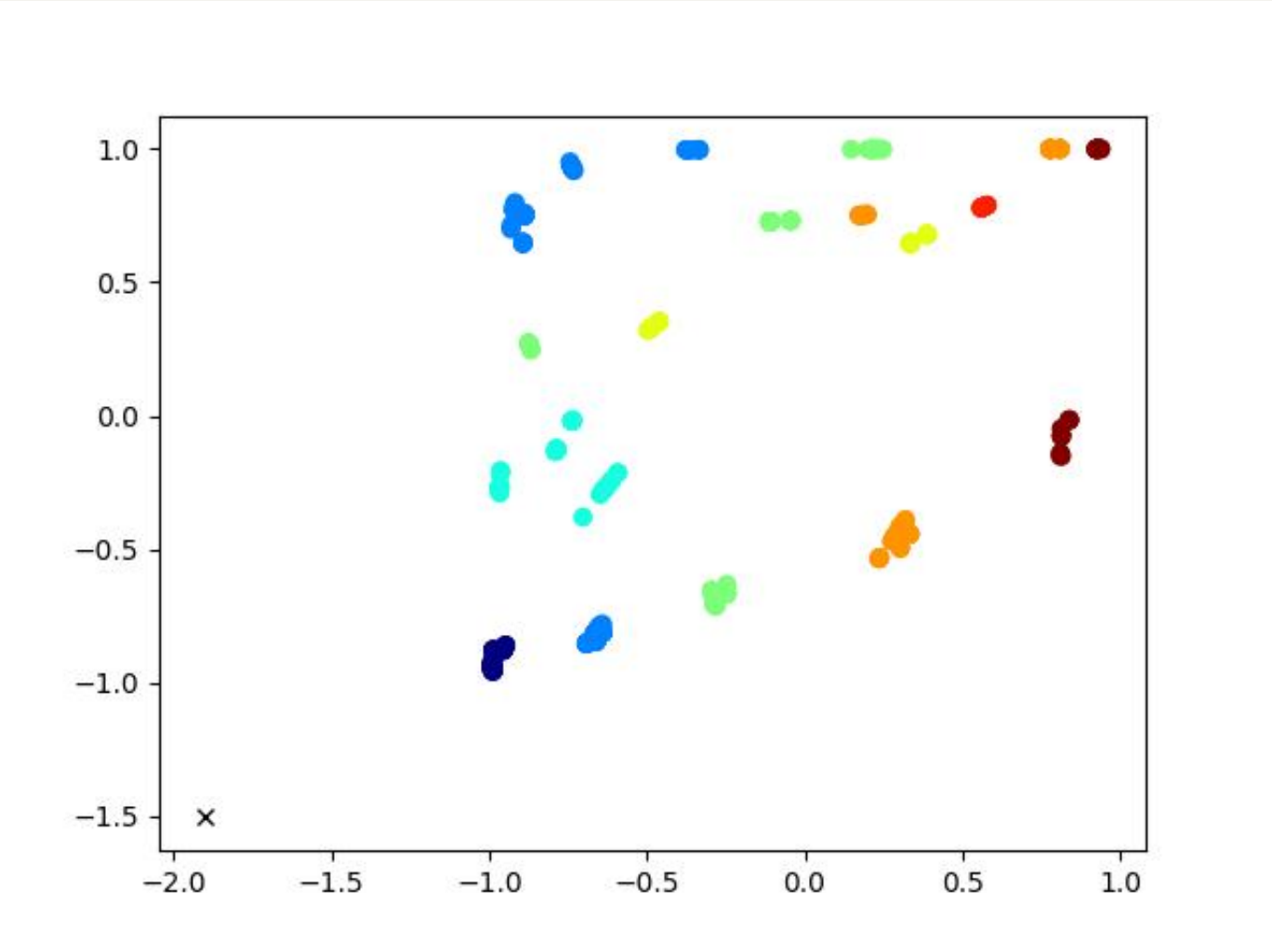
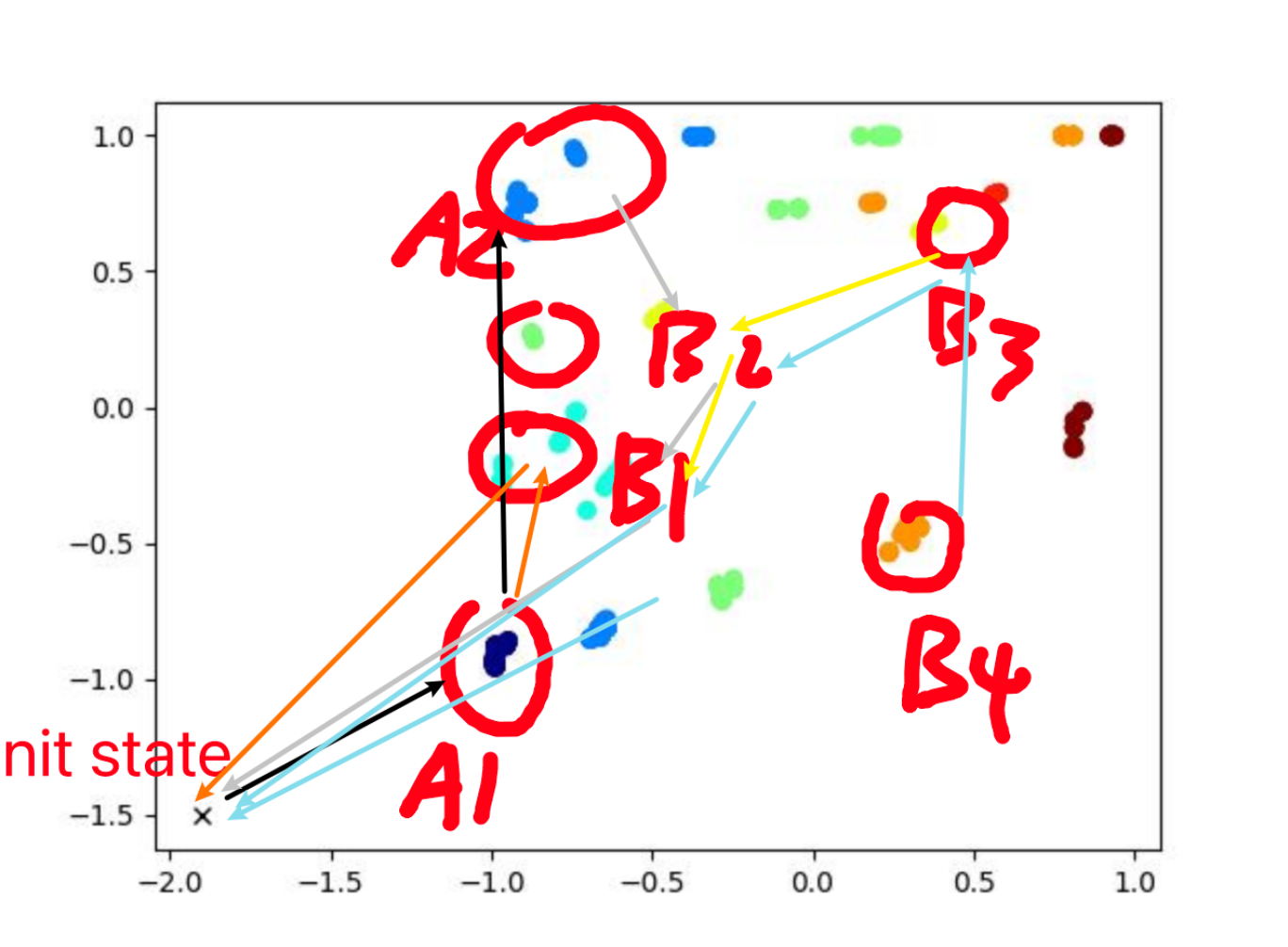


Figure 1: Hidden unit activations for SRN on a^n b^{2n} task. The cross (×) marks the initial state, and different colors represent different states in the learned finite state machine.

Step 2: Finite State Machine Analysis

Annotated State Machine:
Based on the hidden unit activations, the network implements a finite state machine with the following structure:



State Identification:

1. **Initial State (×)**: Starting position, ready to process A's
2. **A-counting States**: Progressive states that count the number of A's seen
3. **B-processing States**: States that track progress through the 2n B's
4. **Transition State**: State that predicts the next sequence's first A

State Transitions:

- A → A: Within A-counting phase (probabilistic)
- A → B: Transition from A's to B's (deterministic after first A)

- B → B: Within B-processing phase (deterministic)
- B → A: Transition to next sequence (deterministic)

Step 3: Network Mechanism Explanation

How the SRN Accomplishes the a^n b^{2n} Task:

1. **A-counting Phase**:
    - The network uses its hidden state to implicitly count the number of A's
    - Hidden activations gradually change as more A's are processed
    - The network maintains information about how many A's have been seen
2. **Transition to B's**:
    - When the first B appears, the network transitions to a B-processing mode
    - The hidden state encodes both that we're now in the B phase and how many B's are expected
3. **B-processing Phase**:
    - The network deterministically predicts B's for the first 2n-1 positions
    - Hidden activations track progress through the expected number of B's
    - The network "counts down" the remaining B's needed
4. **Sequence Completion**:
    - After processing exactly 2n B's, the network predicts the next A with high confidence
    - This demonstrates the network has learned the 2:1 ratio between B's and A's
5. **Multiplicative Counting**: The network implicitly learns the relationship n → 2n through its hidden state dynamics, allowing it to predict the correct sequence lengths.

Key Insight: The 2-dimensional hidden space allows the network to encode both the current phase (A's vs B's) and the count information necessary to maintain the 2:1 ratio.

Step 4: LSTM Training on a^n b^{2n} c^{3n} Task

Training Results:

- Model: LSTM with 3 hidden nodes
- Task: a^n b^{2n} c^{3n} language prediction (n ∈ {1,2,3,4})
- Final Loss: 0.0153 (well below required threshold of 0.02)
- Architecture: 3 inputs, 3 hidden nodes, 3 outputs

Training Success:
The LSTM successfully learned to predict:

- All B's after the first B (probability ≈ 1.0)
- All C's with high confidence (probability ≈ 1.0)
- The transition from the last C to the next sequence's first A
- Appropriate handling of the more complex 1:2:3 ratio

Generated Visualizations:
Three 2D projections were generated:

- `anb2nc3n_lstm3_01.jpg`: Hidden units 0 vs 1
- `anb2nc3n_lstm3_02.jpg`: Hidden units 0 vs 2
- `anb2nc3n_lstm3_12.jpg`: Hidden units 1 vs 2

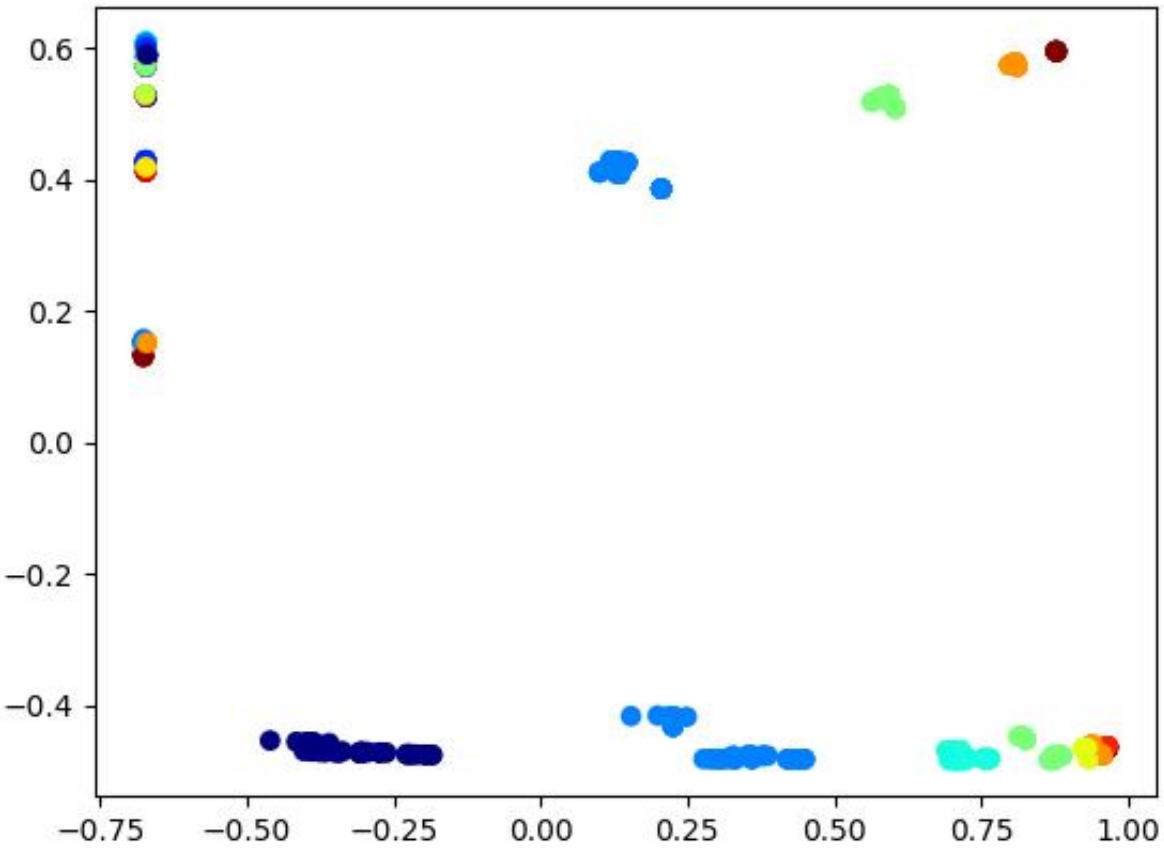


Figure 2: LSTM hidden unit activations (dimensions 0 vs 1) for a^n b^{2n} c^{3n} task

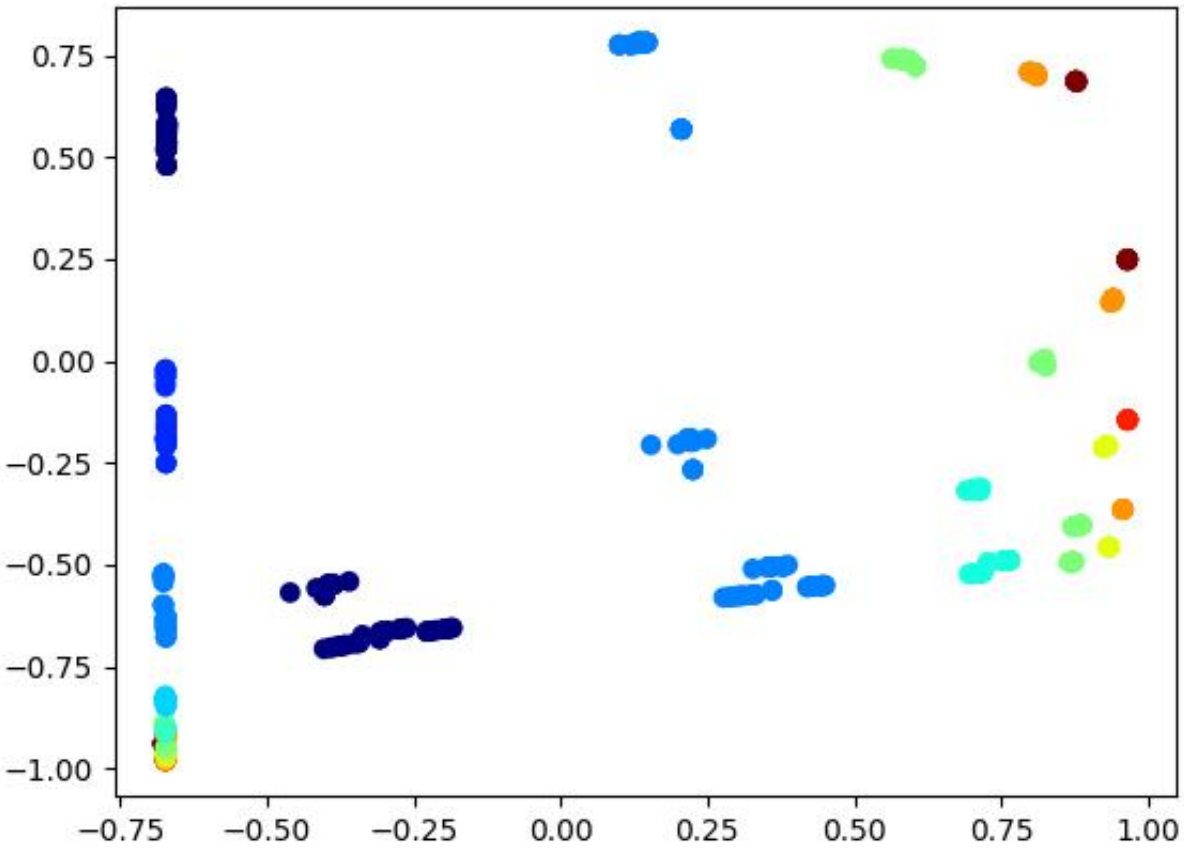


Figure 3: LSTM hidden unit activations (dimensions 0 vs 2) for a^n b^{2n} c^{3n} task

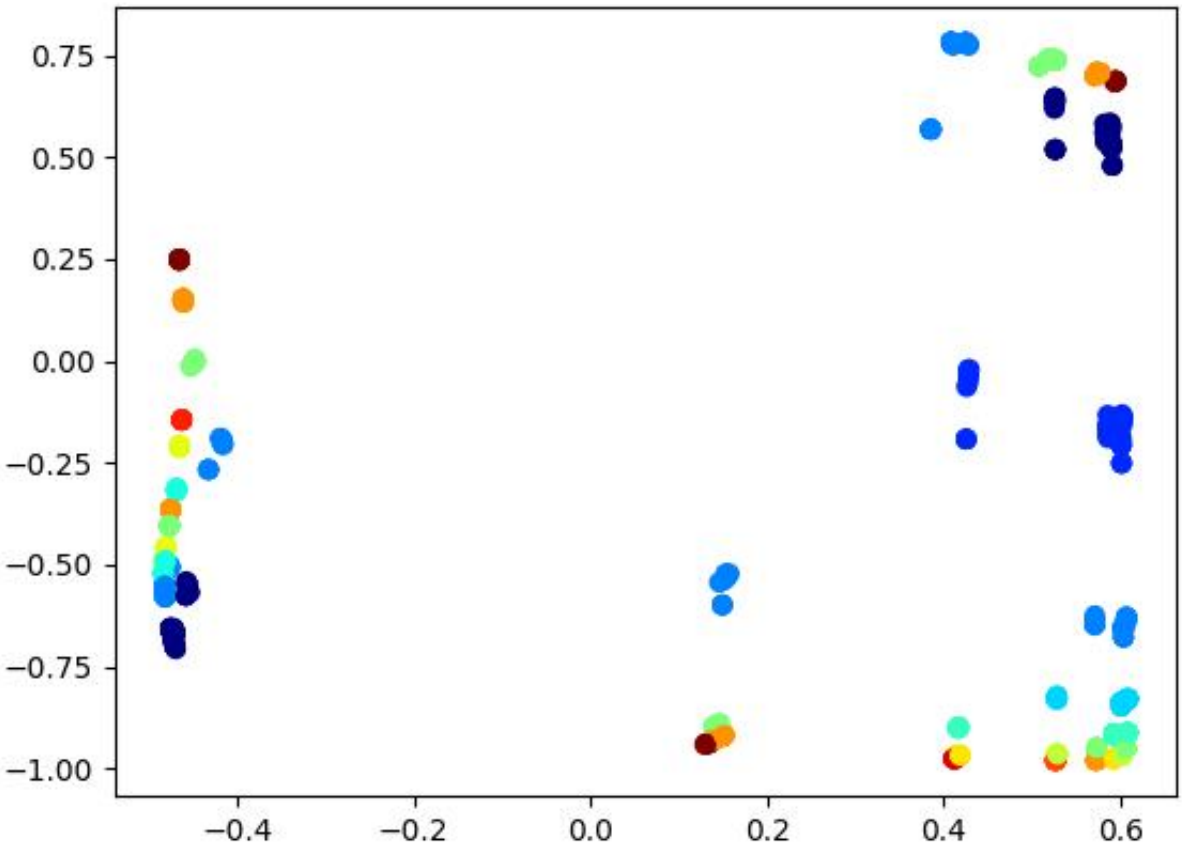


Figure 4: LSTM hidden unit activations (dimensions 1 vs 2) for a^n b^{2n} c^{3n} task

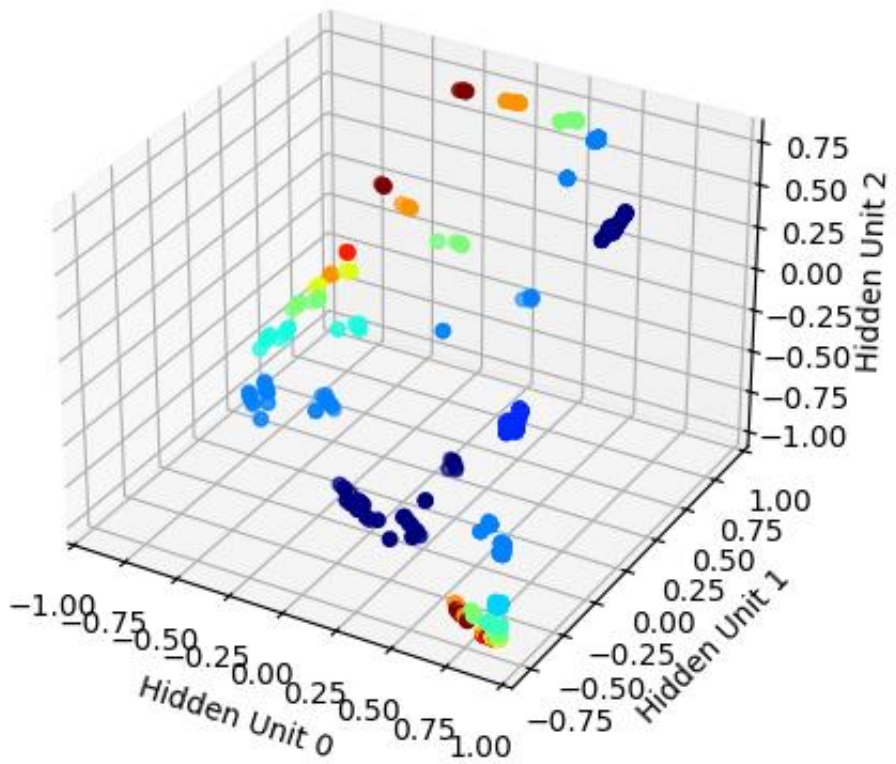


Figure 5: LSTM hidden unit activations in 3D space for a^n b^{2n} c^{3n} task. This 3D visualization shows the complete hidden state trajectory through all three dimensions, revealing the complex dynamics of the LSTM as it processes the A→B→C→A transitions.

Step 5: LSTM Dynamics Analysis

How the LSTM Accomplishes the a^n b^{2n} c^{3n} Task:

Hierarchical State Representation:
The LSTM uses its 3-dimensional hidden space with each unit specializing in different aspects:

- **Unit 1**: Tracks the base count (n)
- **Unit 2**: Encodes current phase and phase progress
- **Unit 3**: Maintains overall sequence state

Three-Dimensional State Space:
The LSTM uses its 3-dimensional hidden space to encode:

1. **Current phase** (A's, B's, or C's)
2. **Count information** for the current phase
3. **Global sequence context** (the value of n)

Phase-Specific Dynamics:

1. **A-processing Phase**:
    - Hidden activations: Gradual progression as A's accumulate
    - The network builds up a representation of n
    - Trajectory moves through distinct regions of the 3D space
2. **A→B Transition**:
    - Sharp transition in hidden space when first B appears
    - Network switches to B-processing mode while retaining count information
    - Hidden state encodes "expect 2n B's total"
3. **B-processing Phase**:
    - Systematic progression through B-states
    - Network tracks progress: "seen k B's, expect 2n-k more"
    - Hidden trajectory shows countdown behavior
4. **B→C Transition**:
    - Another sharp transition when first C appears
    - Network switches to C-processing while maintaining global context
    - Hidden state now encodes "expect 3n C's total"
5. **C-processing Phase**:
    - Most complex phase due to 3n requirement
    - Network must track progress through the longest subsequence
    - Hidden dynamics show systematic countdown from 3n to 0
6. **C→A Transition**:

- Final transition back to initial state
- Network resets for next sequence while maintaining learned structure

7. **Multiplicative Counting**: The network implicitly learns the relationships n → 2n → 3n through its hidden state dynamics, allowing it to predict the correct sequence lengths for each phase.

Key LSTM Advantages:

1. **Memory Capacity**: The LSTM's cell state allows long-term retention of the count n across all three phases
2. **Gating Mechanisms**:
    - **Forget gate**: Selectively removes irrelevant information at phase transitions
    - **Input gate**: Controls when to update count information
    - **Output gate**: Determines what information to use for predictions
3. **Gradient Flow**: LSTM architecture prevents vanishing gradients, enabling learning of long-range dependencies (especially important for the 3n C's)

Visualization Insights:
The 2D projections reveal:

- **Distinct clusters** for each phase (A's, B's, C's)
- **Smooth trajectories** within each phase showing count progression
- **Sharp transitions** between phases
- **Consistent patterns** across different values of n

Conclusion

Both the SRN and LSTM successfully learn their respective context-free languages, but through different mechanisms:

- **SRN (2D)**: Uses a compact 2D representation to encode the simpler 1:2 ratio in a^n b^{2n}

- **LSTM (3D)**: Leverages its more sophisticated architecture and 3D space to handle the complex 1:2:3 ratio in a^n b^{2n} c^{3n}

The LSTM's superior performance on the more complex task demonstrates the importance of:

1. **Adequate representational capacity** (3D vs 2D)
2. **Sophisticated memory mechanisms** (cell state and gates)
3. **Gradient stability** for learning long-range dependencies