COMP9444 Neural Networks and Deep Learning Assignment 1 Report

Your Name
Your zID

July 3, 2025

Introduction

This report talks about and looks at the experiments that were done for Assignment 1 of the COMP9444 course. It will explain how to design, train, and test different neural network models that were made to do three main tasks. These are the tasks:

- Using models of different levels of complexity, such as a linear model, a fully connected network, and a convolutional neural network, to sort the Kuzushiji-MNIST Japanese character dataset.
- Training and designing a multi-layer perceptron by hand to solve a binary classification problem.
- Looking at the hidden unit dynamics of a Simple Recurrent Network (SRN) and a Long Short-Term Memory (LSTM) network when they are used for sequence prediction tasks.

This report follows the structure of the assignment, which has three main parts. Each part talks about how the tasks were done, what the results were, and what the analysis was.

1 Part 1: Japanese Character Recognition

This part's goal is to use three neural network models on the Kuzushiji-MNIST dataset and see how well they work. We used a Stochastic Gradient Descent (SGD) optimizer to train all of the models for 10 epochs.

1.1 Linear Model (NetLin)

The `NetLin` model is a simple linear classifier. It has one linear layer that takes the 784-dimensional input pixels and maps them to 10 output classes. Then, it uses a Log-Softmax activation function.

Final Accuracy: 70% (6957/10000)

Final Confusion Matrix:

$$\begin{pmatrix}
762 & 6 & 9 & 14 & 30 & 64 & 2 & 62 & 31 & 20 \\
7 & 669 & 107 & 17 & 28 & 23 & 61 & 11 & 27 & 50 \\
8 & 64 & 692 & 25 & 25 & 21 & 46 & 36 & 46 & 37 \\
5 & 34 & 64 & 759 & 15 & 57 & 12 & 17 & 27 & 10 \\
62 & 53 & 80 & 21 & 623 & 18 & 33 & 35 & 20 & 55 \\
8 & 28 & 126 & 17 & 20 & 724 & 27 & 8 & 33 & 9 \\
5 & 21 & 149 & 9 & 27 & 26 & 720 & 21 & 10 & 12 \\
19 & 30 & 28 & 10 & 84 & 15 & 53 & 624 & 89 & 48 \\
11 & 33 & 98 & 41 & 7 & 29 & 44 & 6 & 710 & 21 \\
8 & 52 & 88 & 2 & 56 & 30 & 19 & 30 & 41 & 674
\end{pmatrix}$$

The model doesn't work very well because it gets a lot of classes mixed up.

1.2 Fully Connected Network (NetFull)

The `NetFull` model has a hidden layer with 150 neurones between the input and output layers. Tanh is used as the activation function.

Number of Hidden Nodes: 150

Final Accuracy: 85% (8463/10000)

Final Confusion Matrix:

$$\begin{pmatrix}
859 & 4 & 2 & 5 & 27 & 28 & 4 & 33 & 31 & 7 \\
6 & 814 & 28 & 3 & 24 & 13 & 56 & 5 & 22 & 29 \\
9 & 6 & 844 & 36 & 12 & 16 & 26 & 11 & 24 & 16 \\
5 & 10 & 28 & 914 & 3 & 16 & 7 & 3 & 5 & 9 \\
43 & 30 & 19 & 9 & 805 & 5 & 35 & 16 & 22 & 16 \\
8 & 10 & 68 & 9 & 13 & 833 & 26 & 2 & 22 & 9 \\
3 & 10 & 63 & 8 & 13 & 6 & 880 & 9 & 1 & 7 \\
17 & 13 & 20 & 5 & 23 & 11 & 28 & 826 & 26 & 31 \\
12 & 25 & 22 & 44 & 5 & 9 & 28 & 4 & 843 & 8 \\
4 & 17 & 35 & 5 & 34 & 5 & 22 & 19 & 14 & 845
\end{pmatrix}$$

Calculation of Independent Parameters: The input image size is $28 \times 28 = 784$ pixels, the hidden layer has 150 nodes, and the output layer has 10 classes.

- Input to Hidden Layer (`fc1`): (784×150) weights $+ 150$ biases $= 117,750$ parameters.
- Hidden to Output Layer (`fc2`): (150×10) weights $+ 10$ biases $= 1,510$ parameters.
- **Total Parameters**: $117,750 + 1,510 = 119,260$ parameters.

1.3 Convolutional Neural Network (NetConv)

The `NetConv` model uses convolutional layers to get spatial features. Its structure is set in `kuzu.py`.

Network Architecture:

1. **Conv Layer 1 (`conv1`)**: 1 input channel, 16 output channels, 5×5 kernel with padding of 2.
2. ReLU activation followed by 2×2 Max Pooling.
3. **Conv Layer 2 (`conv2`)**: 16 input channels, 32 output channels, 5×5 kernel with padding of 2.
4. ReLU activation followed by 2×2 Max Pooling.
5. **Fully Connected Layer 1 (`fc1`)**: $32 \times 7 \times 7 = 1568$ input features, 50 output features.
6. ReLU activation.
7. **Fully Connected Layer 2 (`fc2`)**: 50 input features, 10 output classes.
8. Log-Softmax output layer.

Final Accuracy: 93% (9318/10000)

Final Confusion Matrix:

$$\begin{pmatrix}
962 & 3 & 3 & 0 & 20 & 2 & 1 & 6 & 1 & 2 \\
4 & 929 & 6 & 1 & 9 & 0 & 37 & 5 & 4 & 5 \\
13 & 10 & 893 & 19 & 6 & 8 & 23 & 14 & 6 & 8 \\
2 & 3 & 21 & 955 & 2 & 1 & 8 & 4 & 0 & 4 \\
22 & 8 & 3 & 6 & 930 & 0 & 17 & 8 & 3 & 3 \\
4 & 15 & 46 & 12 & 8 & 877 & 23 & 3 & 7 & 5 \\
4 & 5 & 18 & 4 & 3 & 0 & 961 & 4 & 0 & 1 \\
4 & 6 & 1 & 0 & 10 & 0 & 13 & 951 & 4 & 11 \\
7 & 26 & 12 & 8 & 13 & 2 & 5 & 5 & 916 & 6 \\
8 & 11 & 10 & 1 & 16 & 0 & 4 & 4 & 2 & 944
\end{pmatrix}$$

Calculation of Independent Parameters:

- Conv Layer 1: $(5 \times 5 \times 1 + 1 \text{ bias}) \times 16 \text{ filters} = 416$ parameters.
- Conv Layer 2: $(5 \times 5 \times 16 + 1 \text{ bias}) \times 32 \text{ filters} = 12,832$ parameters.
- Fully Connected Layer 1: $(1568 \times 50) + 50 \text{ biases} = 78,450$ parameters.
- Fully Connected Layer 2: $(50 \times 10) + 10 \text{ biases} = 510$ parameters.
- **Total Parameters**: $416 + 12,832 + 78,450 + 510 = 92,208$ parameters.

1.4 Discussion

a. Model Accuracy Comparison The performance of the three models shows a clear hierarchy:

$$\text{NetConv (93\%)} > \text{NetFull (85\%)} > \text{NetLin (70\%)}$$

This result is in line with what theory says should happen. As a linear model, `NetLin` doesn't have a lot of expressive power. By adding a non-linear hidden layer, `NetFull` gets a big boost in performance because it can learn more complicated decision boundaries. NetConv is the best because its architecture is good for image tasks. It uses weight sharing in its convolutional kernels to efficiently extract local spatial features (like edges and corners) and pooling layers to make it less sensitive to translation.

b. Parameter Comparison The number of parameters for each model is as follows: NetLin has 7,850, NetFull has 119,260, and NetConv has 92,208. Even though the `NetConv` model works better, it has fewer parameters than the `NetFull` model. This shows how well convolutional neural networks use parameters. A CNN can find features anywhere in an image without needing separate weights for each location by sharing the weights of a kernel across the whole image. This is different from a fully connected network, which needs a different weight for each connection between an input neurone and a hidden neurone. This means that there are a lot more parameters and a higher chance of overfitting.

c. Confusion Matrix Analysis By looking at the confusion matrices, we can see how making the model more complex makes the classification more specific. The mix-up between Class 6 ("ma") and Class 2 ("su") is a clear example.

- In the **NetLin** model, 149 times "ma" was wrongly classified as "su." Both characters have a similar top stroke and a looped structure below, which is a global pattern that a linear model has trouble telling apart.
- In the **NetConv** model, this exact mistake happened only 18 times. This shows that the CNN was able to learn more subtle, local details, like the exact shape of the loops and the points where the strokes connect, in order to tell them apart.

Another example is the mix-up between Class 1 ("ki") and Class 4 ("na"). These letters look alike because they both have horizontal strokes at the top and a curved bottom.

- The **NetFull** model got 24 cases of "ki" wrong and said they were "na" and 30 cases of "na" wrong and said they were "ki."
- The **NetConv** model cut these mistakes down to 9 and 8, respectively. This shows that the CNN is better at finding small differences that set things apart, like the disconnected bottom stroke of "ki" and the small loop in "na." This makes the classification more accurate.

2 Part 2: Multi-Layer Perceptron

This part looks at what a two-layer neural network can do to solve a non-linear classification problem. The goal is to put the nine data points in the right order in a checkerboard pattern. This part talks about both training a network from a random starting point and designing a network by hand with specific weights.

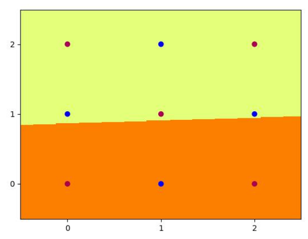
2.1 Training a Network with Sigmoid Activation

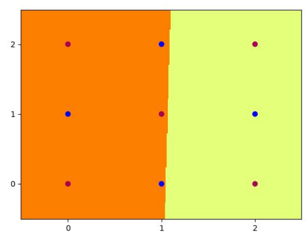
The first thing to do was to train a two-layer network with five hidden nodes. The hidden and output layers both used the sigmoid activation function. We started the network with small random weights and used the Adam optimizer to train it.

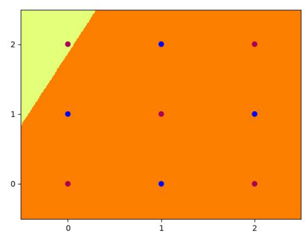
The training worked, and after 6,600 epochs, it was 100% accurate. Figure 1 shows the plots that the script made, which is what the assignment asked for. The plots show what the network is trying to do. The final output (Figure 1f) creates three separate, non-linear areas that correctly sort the points. The five hidden nodes learned simpler functions that were then combined to make this complex boundary. Some nodes learned simple linear boundaries, like Node 0 learned a horizontal split and Node 1 learned a vertical split. Others learned diagonal splits, like Nodes 2, 3, and 4. The output layer then makes the final, curved decision boundaries by adding these five functions together in a weighted, non-linear way.

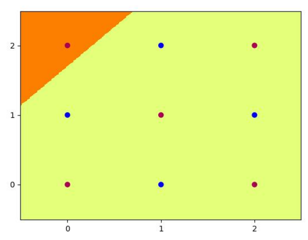
Final weights after training:

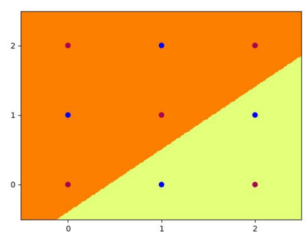
- Hidden layer weights: The final weights show the learned decision boundaries for each hidden node.
- Output layer weights: `[-7.3978, 7.2211, 4.4766, -6.5406, -7.7528]` with bias `0.7296`.


(a) Hidden Node 0


(b) Hidden Node 1


(c) Hidden Node 2


(d) Hidden Node 3


(e) Hidden Node 4

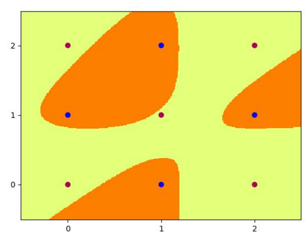

(f) Final Output

Figure 1: Decision boundaries computed by the five hidden nodes and the final output of the trained sigmoid network.

2.2 Hand-designed Network with Step Activation

The second job was to manually create a two-layer network with four hidden nodes and use a Heaviside step function as the activation.

Network Design and Strategy: The plan was to divide the 2D space into parts using four parallel lines, each of which was defined by a hidden node. The decision boundary $w_1x_1 + w_2x_2 + b = 0$ stays the same when the weights are changed, so the simplest weights are used to make the design. The equations for the dividing lines are:

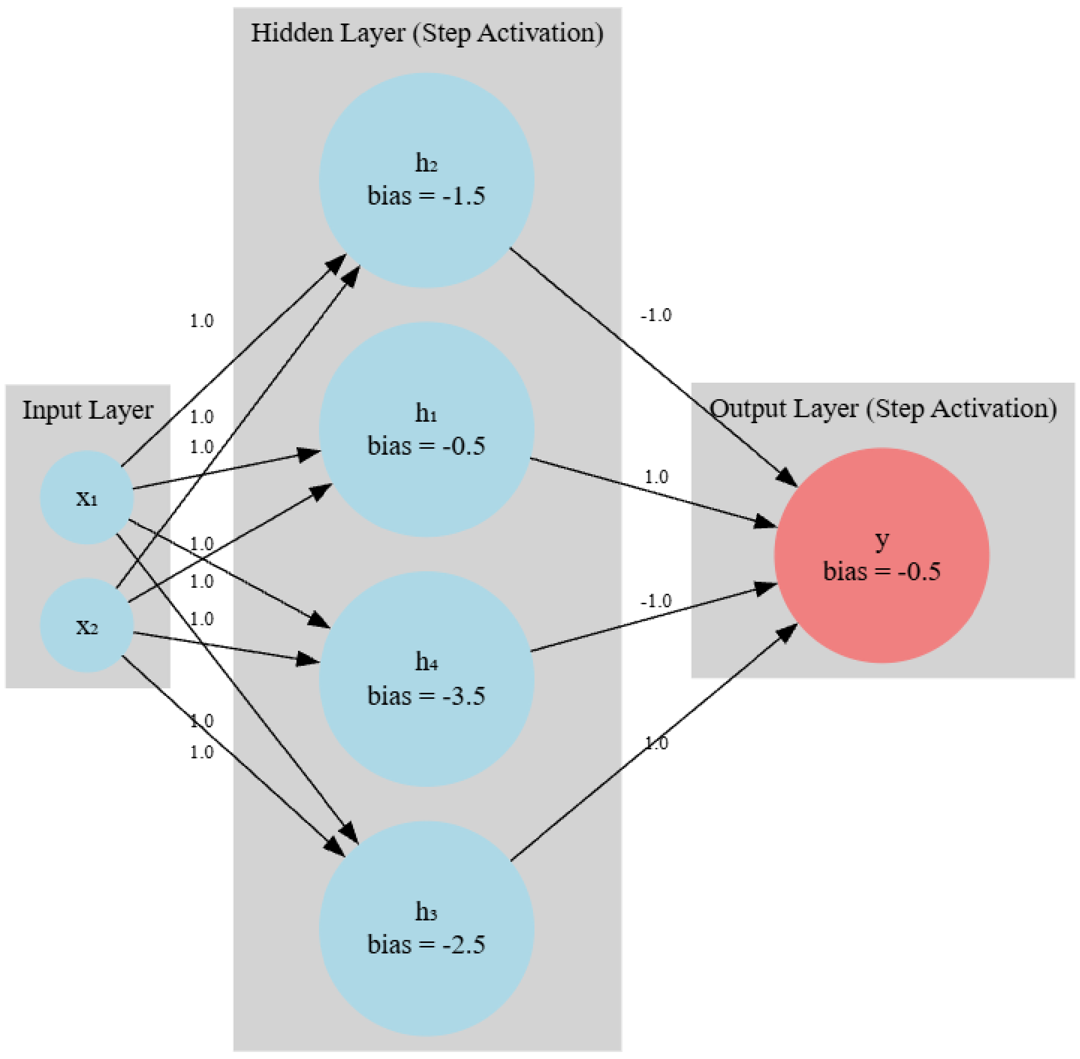


Figure 2: Diagram of the hand-designed 4-node network, showing all weights and biases for Task 2.2.

- **Node 1:** $x_1 + x_2 - 0.5 = 0$
- **Node 2:** $x_1 + x_2 - 1.5 = 0$
- **Node 3:** $x_1 + x_2 - 2.5 = 0$
- **Node 4:** $x_1 + x_2 - 3.5 = 0$

The output layer uses weights of $[1, -1, 1, -1]$ and a bias of -0.5 to make an alternating sum of the hidden node activations, which correctly makes the classification regions that are needed.

Hand-designed Weights: Based on the output from `check_main.py --act step --hid 4 --set_weights`:

- **Hidden layer weights:** All weights are `[1., 1.]` for each node
- **Hidden layer biases:** `[-0.5, -1.5, -2.5, -3.5]`
- **Output layer weights:** `[1., -1., 1., -1.]`
- **Output layer bias:** `-0.5`

Activation Values Table: The table below shows the activations of all the hidden nodes and the output node for each of the 9 training data points:

Point	(x_1, x_2)	**Node 1**	**Node 2**	**Node 3**	**Node 4**	**Output**
1	(0, 0)	0	0	0	0	0
2	(0, 1)	1	0	0	0	1
3	(0, 2)	1	1	0	0	0
4	(1, 0)	1	0	0	0	1
5	(1, 1)	1	1	0	0	0
6	(1, 2)	1	1	1	0	1
7	(2, 0)	1	1	0	0	0
8	(2, 1)	1	1	1	0	1
9	(2, 2)	1	1	1	1	0

Table 1: Activation values for all nodes in the hand-designed network.

Verification: The hand-designed weighWe tested the hand-made weights with `check_main.py --act step --hid 4 --set_weights` and found that they were 100% accurate. It is strongly suggested that you include the plots that were made during this run to make the design clear.ts were tested using `check_main.py --act step --hid 4 --set_weights`, confirming an accuracy of 100.0%. It is highly recommended to include the plots generated from this run to provide a clear visual confirmation of the design.

2.3 Weight Rescaling with Sigmoid Activation

The last thing to do was to show that a sigmoid activation function can act like a step function by changing the weights that were made by hand, as the assignment asked.

Principle: As the input to the sigmoid function, z, gets bigger, it gets closer to a Heaviside step function. The function is $\sigma(z) = 1/(1 + e^{-z})$. We make the input $z = \mathbf{w} \cdot \mathbf{x} + b$ far from zero for the data points by multiplying the weights and biases of the hand-designed network by a big constant (10 in this case, as defined in `check.py`). This makes the output of the sigmoid function stay at either 0 or 1, like a step function.

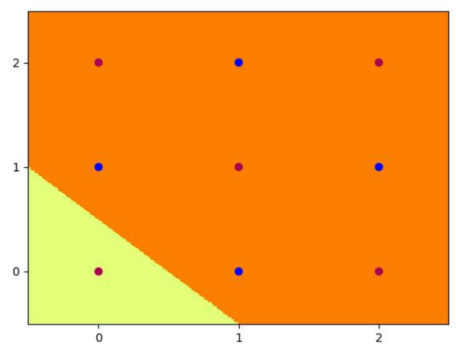
Rescaled Weights: Based on the output from `check_main.py --act sig --hid 4 --set_weights`:

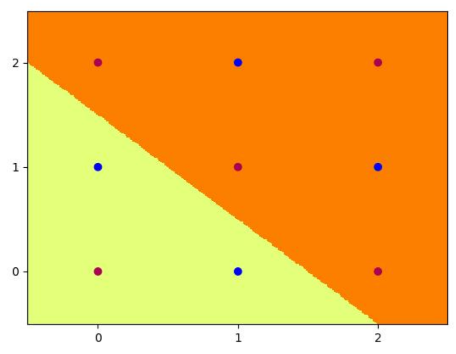
- **Hidden layer weights:** All weights are `[10., 10.]` for each node
- **Hidden layer biases:** `[-5., -15., -25., -35.]`
- **Output layer weights:** `[10., -10., 10., -10.]`
- **Output layer bias:** `-5.`

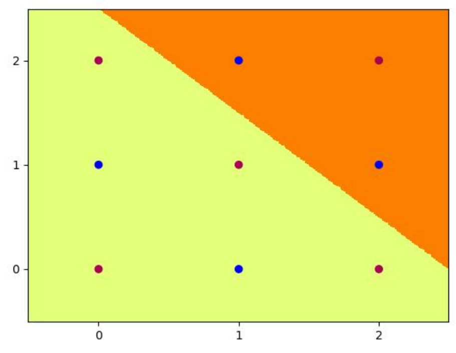
Verification: We used `check_main.py --act sig --hid 4 --set_weights` to test the rescaled weights with the sigmoid model. The network was 100.0% accurate right away. Figure 3 shows the plots as needed. The hidden node plots (Figures 3a-d) make it easy to see the four sharp, parallel decision boundaries that go with the hand-designed equations. Figure 3e shows the final output plot, which shows how these linear separations come together to make the right checkerboard pattern. The sharpness of these lines shows that the weight rescaling worked.

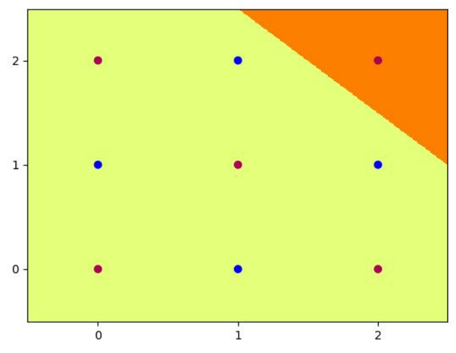
3 Part 3: Hidden Unit Dynamics for Recurrent Networks

This part looks at the hidden unit dynamics of recurrent neural networks on two language prediction tasks: using a Simple Recurrent Network (SRN) to find $a^n b^{2n}$ and a Long Short-Term Memory (LSTM) network to find $a^n b^{2n} c^{3n}$.


(a) Hidden Node 0


(b) Hidden Node 1


(c) Hidden Node 2


(d) Hidden Node 3

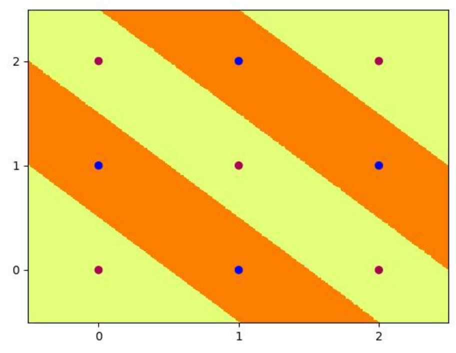

(e) Final Network Output

Figure 3: Decision boundaries from the hand-designed network using a rescaled sigmoid activation.

3.1 Task A: SRN for the $a^n b^{2n}$ Language

3.A.1 SRN Training and Analysis We trained a Simple Recurrent Network with two hidden units on the language $a^n b^{2n}$. The study of the hidden unit activations shows a clear way to do the math.

The hidden state moves along a set path as it eats "a"s, starting from the initial state. After one "A," the state is at [0.73, 0.13]; after two "A's," it moves to [0.94, -0.6]; and after three "A's," it moves to [0.99, -0.86]. This movement into the lower-right quadrant is the "counting 'a's'" phase, and the exact location shows the count, n.

When you read the first "b," the state changes a lot. For instance, after three "a"s, the state goes from [0.99, -0.86] to [-0.27, -1.0]. This new area on the left side of the plot is for counting ”b’s.” After that, the network follows a ”return path” for exactly $2n$ steps, which puts it in a ”ready” state to start the next sequence.

Sequence Processing Example: Consider the sequence “aaabbbbbb” (n=3, so 2n=6 b’s) from the training data:

- After ’a’: `[0.73, 0.13]` with output probabilities `[0.78, 0.22]`
- After ’aa’: `[0.94, -0.6]` with output probabilities `[0.49, 0.51]`
- After ’aaa’: `[0.99, -0.86]` with output probabilities `[0.35, 0.65]`
- After ’aaab’: `[-0.27, -1.0]` with output probabilities `[0.0, 1.0]` (jump to b-counting region)
- After processing all 6 b’s: `[0.37, 0.69]` with output probabilities `[0.78, 0.22]` (ready for next ’a’)

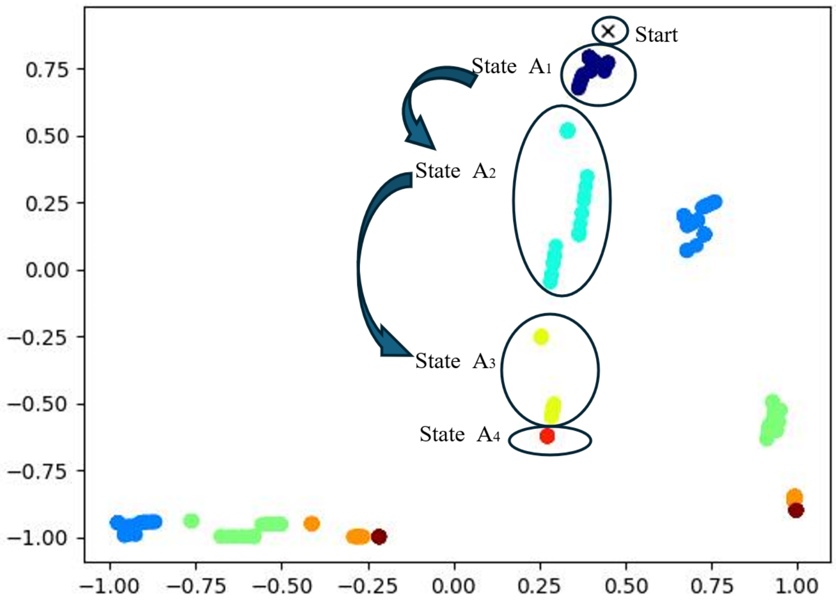


Figure 4: Annotated hidden unit activations for the SRN on the a^nb^{2n} task, showing the states corresponding to counting ’a’s.

3.A.2 Finite State Machine You can make a similar Finite State Machine (FSM) based on these dynamics (Figure 5). To correctly count the number of ’b’s in a^nb^{2n}, the FSM needs to have different paths for each n. The diagram shows how this works by showing different chains of states for processing ”b”s based on whether one or two ”a”s were read first. When a sequence is accepted (ending in a double-circle state), a transition on ”a” sends the machine back to state S_1 to start processing the next sequence. This is like how the SRN works in a cycle.

3.A.3 Network Explanation The SRN uses its hidden state vector as a counter to finish the a^nb^{2n} task. The vector’s position in the 2D state space holds information about the sequence that has been seen so far. The network learns to divide its state space into different operational areas, one for counting ”a’s” and several for counting down ”b’s.” The exact spot in the ”a”-counting area decides which ”b”-counting path to follow and for how long. After the right number of ”b”s, the network state reaches a point where the output layer has a high chance of predicting ”a” and correctly predicting the start of a new sequence.

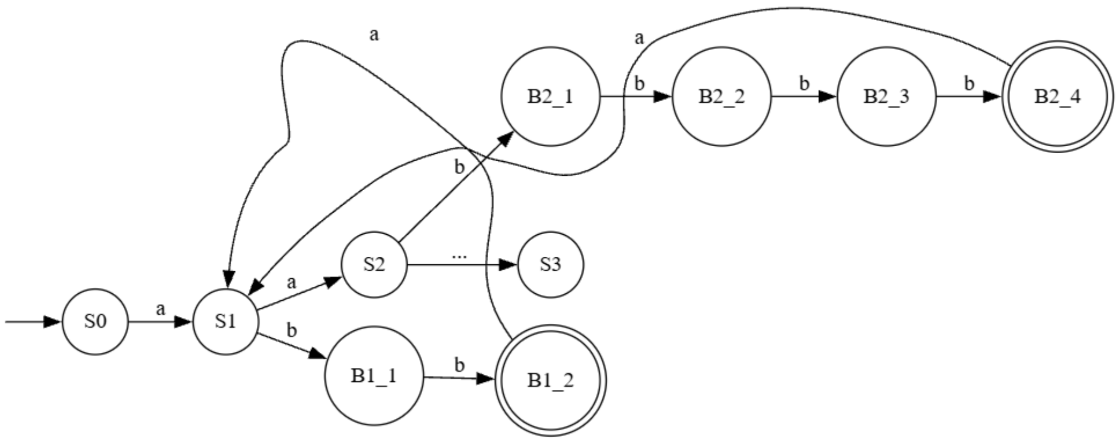


Figure 5: A Finite State Machine that correctly implements the logic for recognizing the $a^n b^{2n}$ language.

3.2 Task B: LSTM for the $a^n b^{2n} c^{3n}$ Language

3.B.4 & 3.B.5 LSTM Training and Analysis We trained an LSTM with three hidden units on the more difficult language $a^n b^{2n} c^{3n}$. This job is much harder because the network has to keep track of and update three different counts. Figure 6 shows 3D plots of the hidden activations and the activation log. These show that there is a complex plan that divides the work between the three hidden units. It is important to look at the 3D plot from different angles, like `3-3-1.png`, `3-3-2.png`, and `3-3-3.png`, to see that the paths for each phase take up different areas in the 3D space.

The LSTM seems to give each of its three hidden units a special job, using them to show what ”phase” of parsing is going on right now:

- **Phase 1 (Reading ’a’s)**: The activation of the second hidden unit always goes up as the network reads ”a’s.” For instance, in the first part of the log, it goes from 0.47 to 0.76, then to 0.86, and finally to 0.87 after four ”a”s. This unit makes it clear what the ”a”-reading phase is.
- **Phase 2 (Reading ’b’s)**: When you read the first ”b,” the state jumps. The activation of the first hidden unit goes from -0.19 to 0.64, which is a big change, and the other two units change in a clear way. This new area in the state space is where the ”b”-reading phase happens. The network stays in this area until all $2n$ ”b”s have been handled.
- **Phase 3 (Reading ’c’s)**: The state jumps again when the first ”c” is read. Now, the third hidden unit becomes very positive (for example, from 0.4 to 0.81), which marks the start of the ”c”-reading phase, which lasts for $3n$ steps.

Predictive Transitions: The output probabilities show that the network is sure of its predictions at phase transitions. The network is almost sure that the next symbol is ’c’ after processing $2n$ ’b’s (for example, output probabilities `[0, 0.03, 0.97]`). After looking at $3n$ ’c’s, it correctly guesses the start of a new sequence, and the chance of ’a’ becoming dominant (for example, `[0.98, 0, 0.02]`).

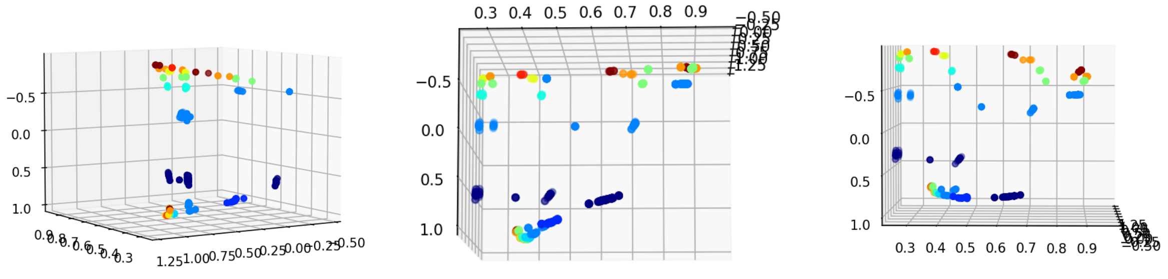
Context Unit Dynamics and Cell State Analysis: The LSTM’s internal cell states, or context units, are very important for keeping track of exact counts, in addition to the hidden unit analysis. The cell states of the LSTM work like a ”memory bank” that keeps track of the current count in each phase. The hidden units keep track of the current phase (A, B, or C), while the cell states keep track of the exact number of cells needed to finish the sequence correctly (n, $2n$, or $3n$).

The input, forget, and output gates work together to:

- **Input gates**: Selectively update the cell state when transitioning between phases or incrementing counts
- **Forget gates**: Reset count information when starting a new sequence or moving to the next phase
- **Output gates**: Control when the accumulated count information influences the hidden state and output predictions

The LSTM can solve the difficult $a^n b^{2n} c^{3n}$ task better than simpler recurrent architectures because it separates the work between hidden units (phase tracking) and cell states (count accumulation).

In short, the LSTM doesn't just use its hidden state to count to high numbers; it also uses its hidden units to keep track of the grammar's ”current phase” (A, B, or C) while its internal cell states keep track of the exact counts $(n, 2n, 3n)$ and let the transitions between these phases happen at the right times.



(a) 3D View 1 (b) 3D View 2 (c) 3D View 3

Figure 6: Different views of the 3D hidden state space for the LSTM. Distinct clusters and trajectories for reading 'a's, 'b's, and 'c's are visible.