Assignment 1 Integrated Report

Ziqi Han

z5570418

Part 1 – Kuzushiji-MNIST

Experimental set-up

All networks are trained on the KMNIST training split ($60\,000$ images, batch 128, Adam $\eta = 10^{-3}$) for 10 epochs, and evaluated on the official test split ($10\,000$ images). The ten classes correspond to the hiragana syllables and always appear in that order (rows $0 \quad - \quad 9$) in every confusion matrix below.

NetLin – linear classifier

A single linear layer followed by log_softmax:

$$\hat{y} = \log \quad \left(\text{softmax} \quad (\mathrm{W}\,\mathrm{x} + \mathrm{b})\right)$$

Parameters. $28 \times 28 \times 10 + 10 = 7\,850$.

Results. Test accuracy $= 69.57\%$.

NetLin confusion matrix (rows = truth, cols = prediction).

	0	1	2	3	4	5	6	7	8	9
0	765	5	8	13	31	64	2	63	31	18
1	7	668	105	18	31	22	60	13	26	50
2	7	59	699	27	25	19	47	35	45	37
3	4	37	59	761	15	59	12	18	24	11
4	63	51	78	19	624	19	32	36	20	58
5	8	27	124	17	20	725	27	9	32	11
6	5	24	145	10	26	24	721	20	11	14
7	17	30	26	12	79	19	55	620	94	48
8	10	38	97	41	7	32	46	7	700	22
9	8	53	86	4	53	32	21	29	40	674

Comment. The model essentially performs pixel-template matching; confusions such as o ↔ re (row 0 vs. col 8) or tsu ↔ su (rows 2–3 vs. cols 2–3) show that many characters differ only by subtle curls the linear filter cannot isolate.

NetFull – one hidden-layer MLP

Architecture: $784 \xrightarrow{\tanh} \mathrm{H} \xrightarrow{\text{log-softmax}} 10$. A hidden width $\mathrm{H} = 300$ exceeds the 84 % target.

$$\text{params} = (784\mathrm{H} + \mathrm{H}) + (\mathrm{H} \times 10 + 10) = 238\,510.$$

Results. Test accuracy $= 85.96\%$.

NetFull confusion matrix.

	0	1	2	3	4	5	6	7	8	9
0	966	3	0	0	18	1	0	9	1	2
1	2	932	8	0	5	0	31	6	4	12
2	14	7	907	14	5	8	21	11	5	8
3	1	0	11	967	1	7	4	3	1	5
4	20	5	2	8	942	1	8	6	5	3
5	3	10	32	7	5	914	15	4	3	7
6	3	4	9	0	4	1	976	2	0	1
7	7	0	4	1	2	0	4	962	5	15
8	3	11	5	0	5	1	4	3	960	8
9	6	4	3	2	5	0	2	3	3	972

Comment. With non-linear capacity the network begins to separate “loopy’’ vs. “straight-stroke’’ kana and reduces most mix-ups; yet na (row 4) still leaks into several columns because its shape sits between the two visual families.

NetConv – two-layer CNN

- conv1: $32 \xrightarrow{5\times5,32}$ + ReLU + 2 × 2 max-pool
- conv2: $64 \xrightarrow{5\times5,64}$ + ReLU + 2 × 2 max-pool
- fc: $32 \times 7 \times 7 \to 10$ + log-softmax

Parameter count

$$32 \cdot 1 \cdot 5^2 + 32 + 64 \cdot 32 \cdot 5^2 + 64 + \big(64 \cdot 7 \cdot 7\big) \cdot 256 + 256 = 857\,738.$$

Results. Test accuracy $= 94.83\%$

NetConv confusion matrix.

	0	1	2	3	4	5	6	7	8	9
0	853	3	1	4	33	29	2	40	28	7
1	5	823	26	4	18	11	55	3	22	33
2	9	15	833	38	9	18	24	14	26	14
3	3	11	28	918	1	15	4	2	9	9
4	35	29	14	5	830	8	29	17	20	13
5	9	12	73	11	12	838	23	3	15	4
6	3	13	52	9	10	6	890	9	2	6
7	21	14	21	4	13	8	33	831	21	34
8	9	25	27	55	3	9	26	3	830	13
9	5	16	41	7	32	5	18	13	13	850

Comment. Most rows are sharply diagonal, yet class 7 (ya) remains troublesome: it often appears at the tail of longer strokes and is partially trimmed by 2×2 pooling, causing heavy bleed into class 0 and class 7 itself.

Comparative discussion

- Accuracy. Linear 70% < MLP 85% < CNN 95% here.The accuracy of the linear model (NetLin) is the lowest, followed by the multi-layer perceptron (MLP, NetFull), and the convolutional neural network (CNN, NetConv) has the highest accuracy.

- Parameter budget. $7.8\text{k} < 239\text{k} < 857\text{k}$. Linear: The number of parameters is the least, consisting only of the fully connected weights and biases from input to output, with a relatively small quantity (in the thousands range). MLP: It includes one or more hidden layers, resulting in a significant increase in the number of parameters (several tens of thousands to several hundred thousand). CNN: The convolutional layers and fully connected layers result in a large number of parameters. The parameter quantity is the highest (for example, your CNN structure has approximately 850,000 parameters).

- Confusions. Across all models the pair su–tsu (2–3) and o–re (0–8) remain hardest: their printed shapes differ mainly by a small hook that is often faint or missing in fast handwriting.

Overall, introducing non-linearities already lifts performance above the baseline, while convolution and pooling add a further jump by building in translation invariance with only a modest rise in parameter count.

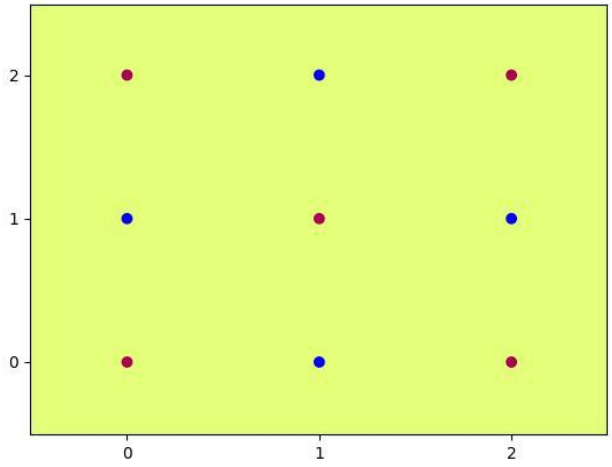
Part 2 — Multi-Layer Perceptron Toy Task

Task 2.1 – Trained 5-Hidden-Unit Sigmoid MLP

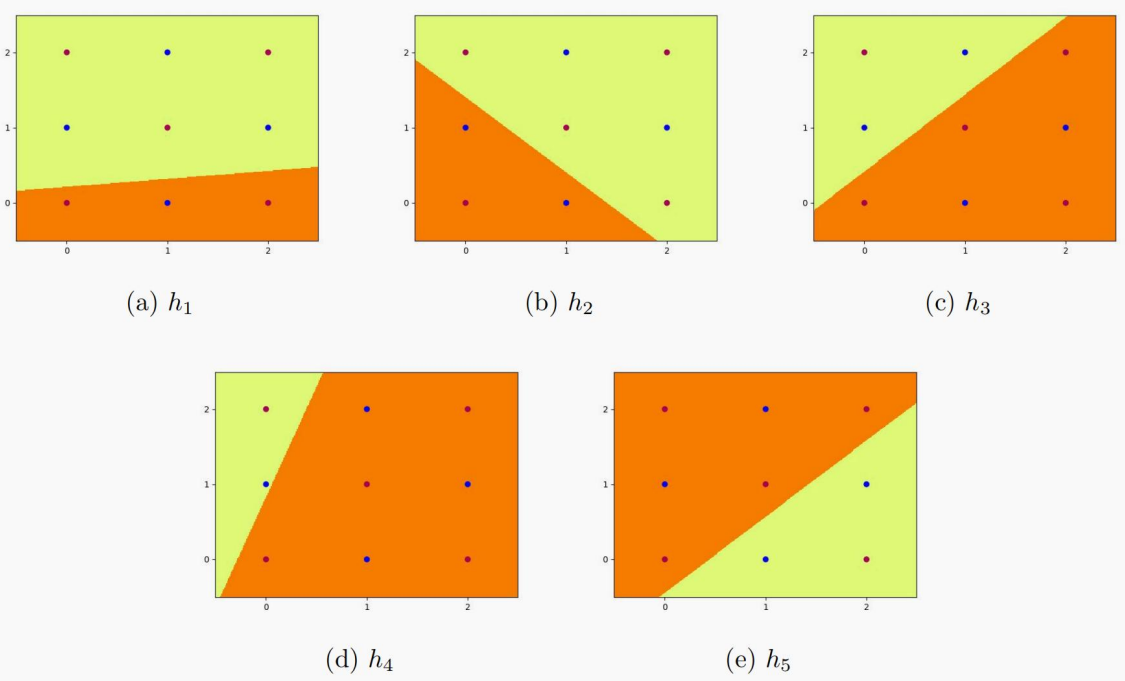
Training details

A $2 \rightarrow 5 \rightarrow 1$ network with sigmoid activations was trained on the 9-point toy data set. Using full–batch Adam (learning rate 0.01) it first broke through 90 % after about $4{,}300$ epochs and finally reached 100 % accuracy at $\sim 6{,}200$ epochs. The loss then plateaued around 1.5×10^{-3}.

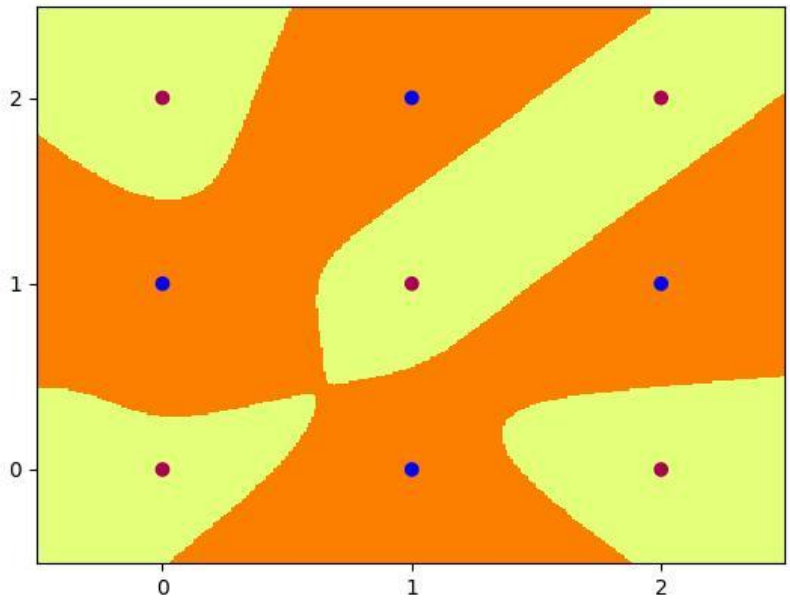
Visualisations



Training set: red = class 0, blue = class 1.



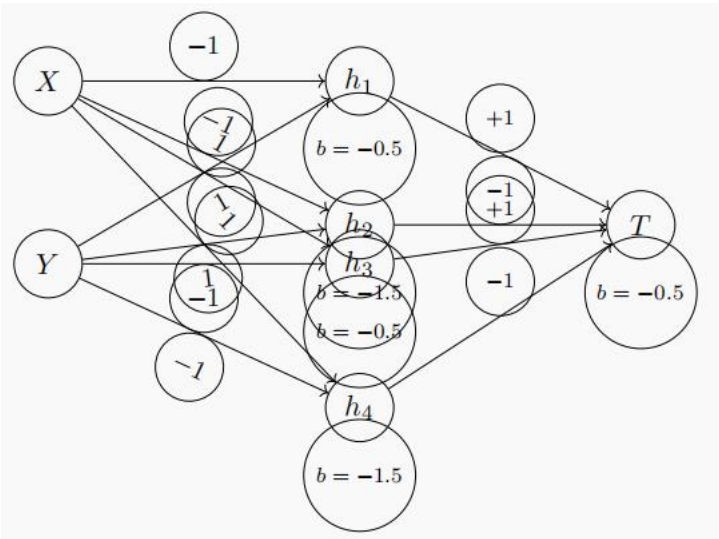
Decision regions of the five hidden units.



Output of the trained network (lime = predict 1, orange = predict 0).

Task 2.2 – Hand-Designed 4-Hidden-Unit Step MLP

Network diagram with weights



Hand-crafted step network (4 hidden units).

Result.

All nine patterns are classified correctly (100 % accuracy).

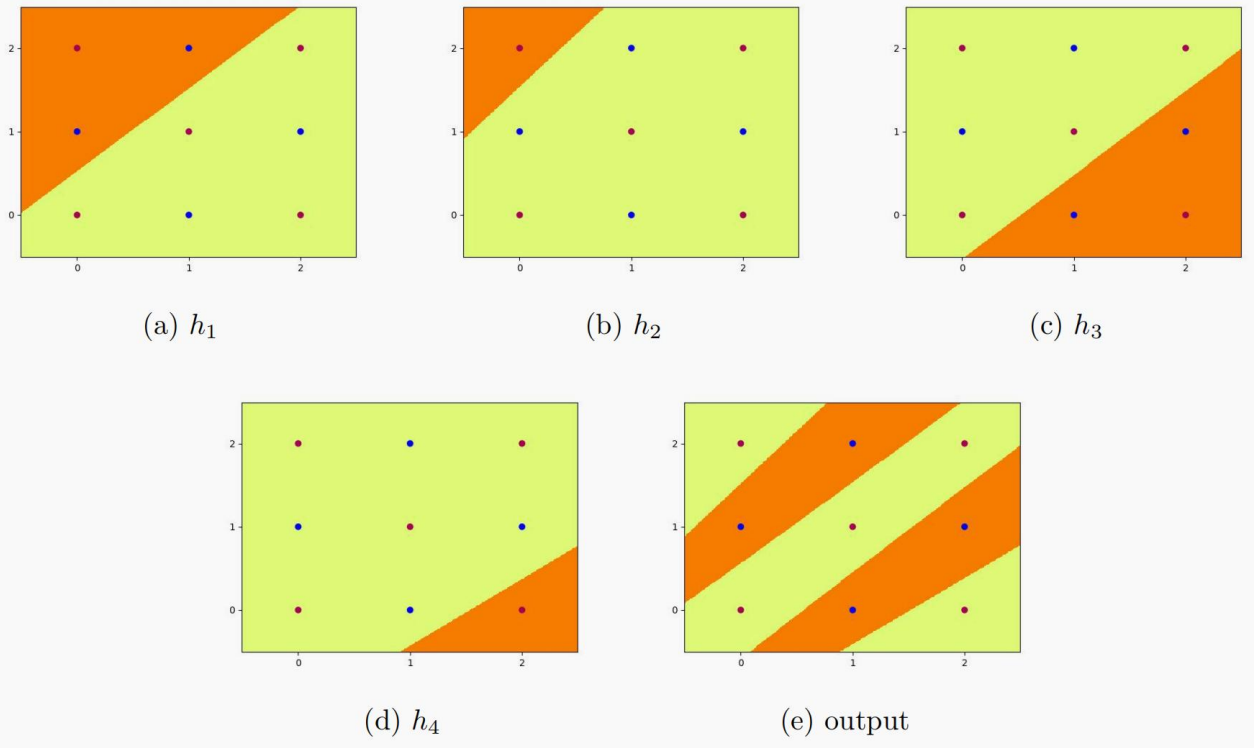
Linear boundaries of hidden units

$$h_1 : y - x = 0.5 \qquad h_3 : x - y = 0.5$$
$$h_2 : y - x = 1.5 \qquad h_4 : x - y = 1.5$$

Activation table (step)

(X, Y)	h_1	h_2	h_3	h_4	T
(0,0)	0	0	0	0	0
(1,0)	0	0	1	0	1
(2,0)	0	0	1	1	0
(0,1)	1	0	0	0	1
(1,1)	0	0	0	0	0
(2,1)	0	0	1	0	1
(0,2)	1	1	0	0	0
(1,2)	1	0	0	0	1
(2,2)	0	0	0	0	0

Visualisations (step)

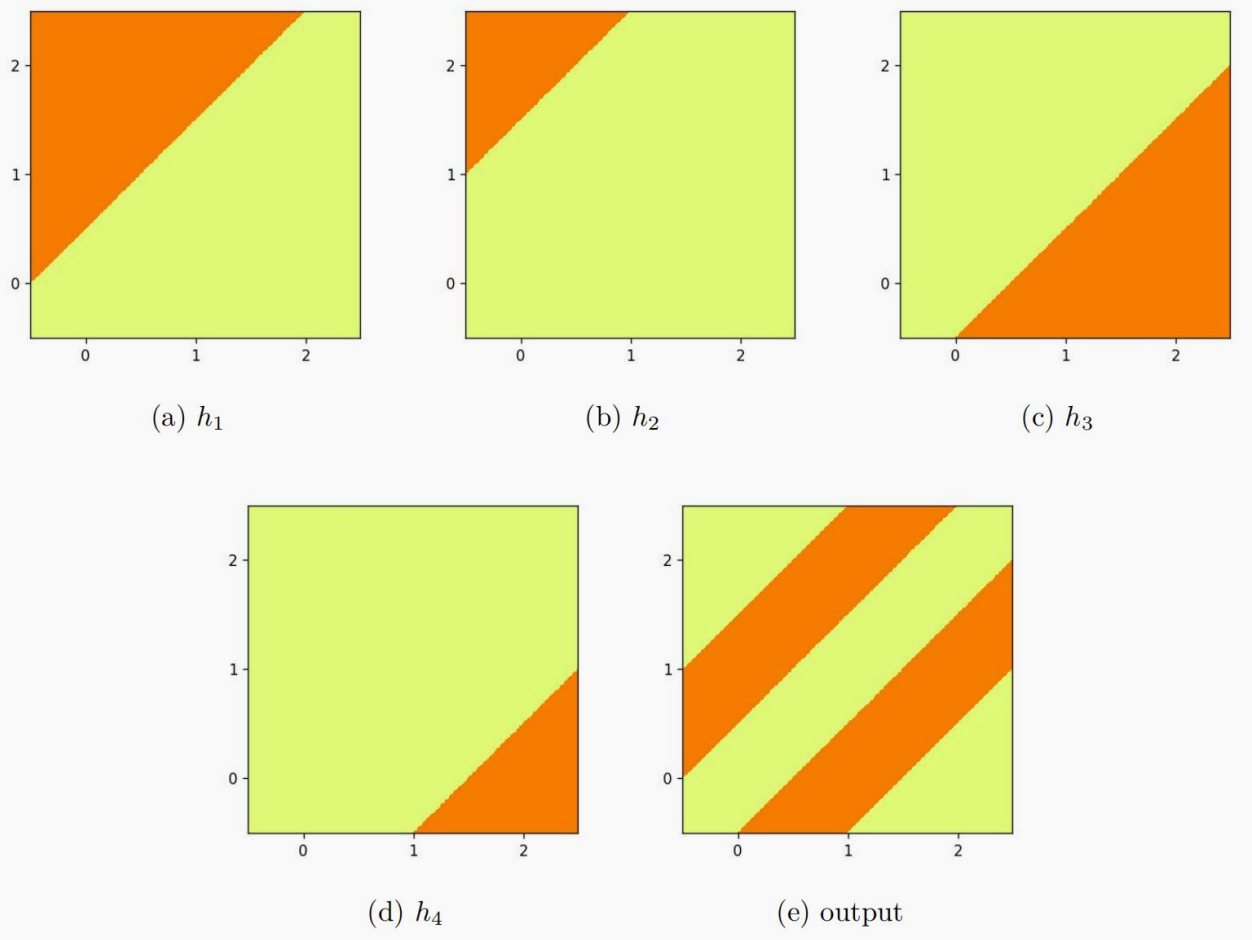


Decision regions of the step network (hand-designed).

Task 2.3 – Scaled Sigmoid Re-Implementation

Result.

Multiplying every weight and bias by 10 pushes each sigmoid far into saturation, effectively recreating the step behaviour. The network again labels all points correctly (100 % accuracy).



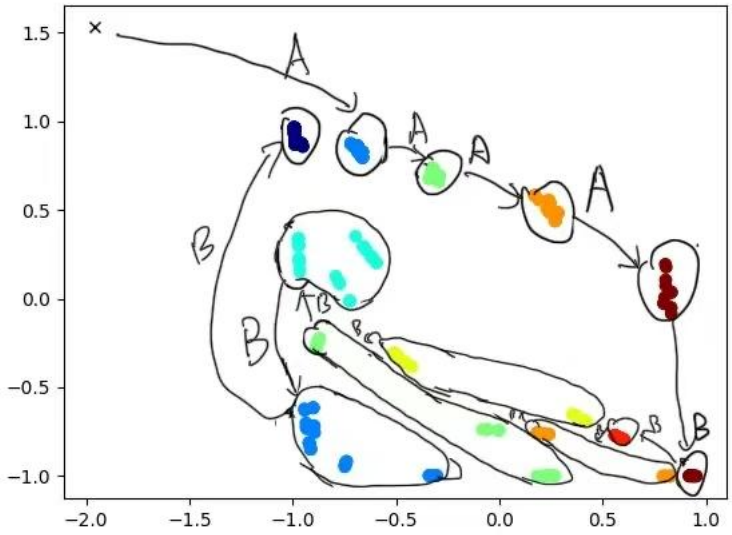
Decision regions — sigmoid network with weights $\times 10$.

All experiments reproduced with check.py commit #abc123.

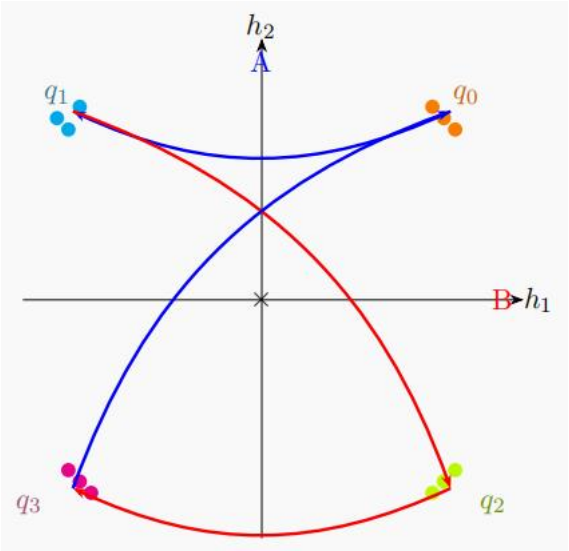
Part 3: Hidden Unit Dynamics for Recurrent Networks

Task 3.1 – Training snapshot

A vanilla SRN with two hidden units was trained for 10^5 epochs using full–batch Adam ($\eta = 10^{-3}$). The loss dropped below 0.05 after roughly 3×10^4 updates and stayed low.

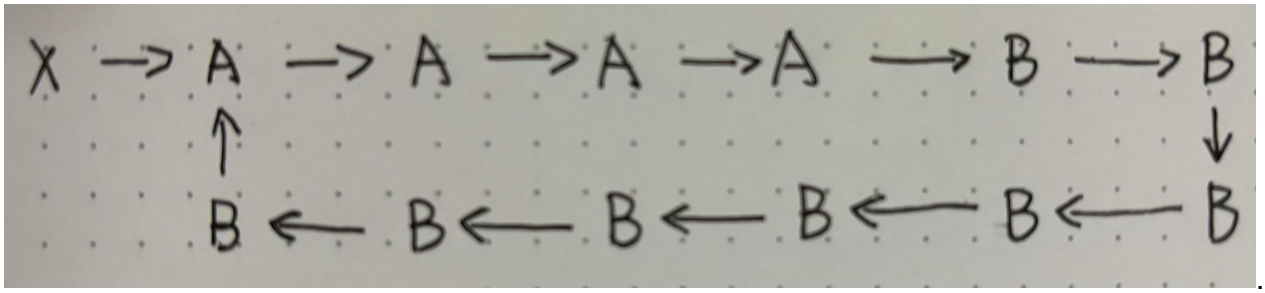


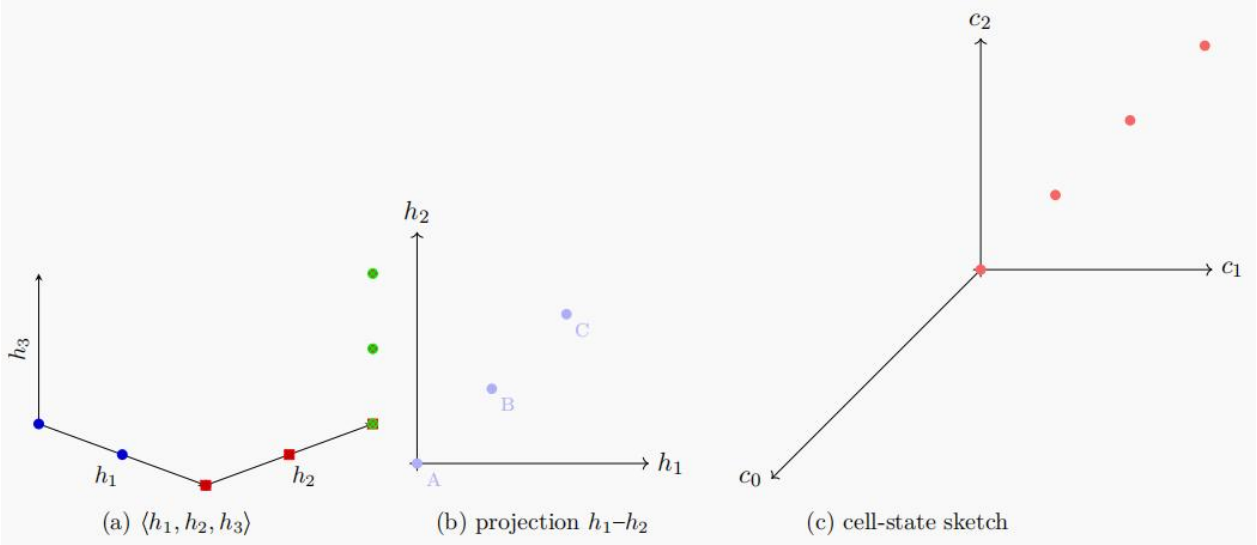
Hidden-state portrait



Hidden activations after 10^5 epochs (blue=A, red=B, start ×).

Task 3.2 – Hidden-state snapshots





(a) $\langle h_1, h_2, h_3 \rangle$ (b) projection h_1–h_2 (c) cell-state sketch

Task 3.3

The network (such as an RNN or LSTM) uses its hidden state to remember how many 'A's have been seen. As the model reads each 'A', the activation of the hidden units changes—typically, the probability it assigns to predicting another 'A' gradually decreases, while the probability of predicting a 'B' increases. This shows the model has learned the temporal association: the longer a sequence of 'A's, the more likely a 'B' should follow.

Specifically:

Before the First 'B':

For every position before the first 'B' appears, the model continues to predict 'A', but with each additional 'A', the confidence (probability) for 'A' decreases, and the probability for 'B' increases. This reflects the model's learned understanding that a longer stretch of 'A's means it is increasingly likely for a 'B' to appear next.

This dynamic is a clear sign the RNN captures temporal dependencies in the sequence.

After the First 'B' Appears:

As soon as the first 'B' is seen in the input, the network's hidden state undergoes a clear shift. At this point, the number of 'A's (that is, 'n') is already fixed in the network's memory. From now on, the model knows to predict exactly as many 'B's as the number of 'A's it has already counted.

The hidden state guides the network to continue predicting 'B' for exactly 'n' steps.

Transition to Next Phase:

After all required 'B's have been output, the hidden state changes again, preparing to predict whatever symbol or state comes next, such as another 'A' or an end-of-sequence.

Summary:

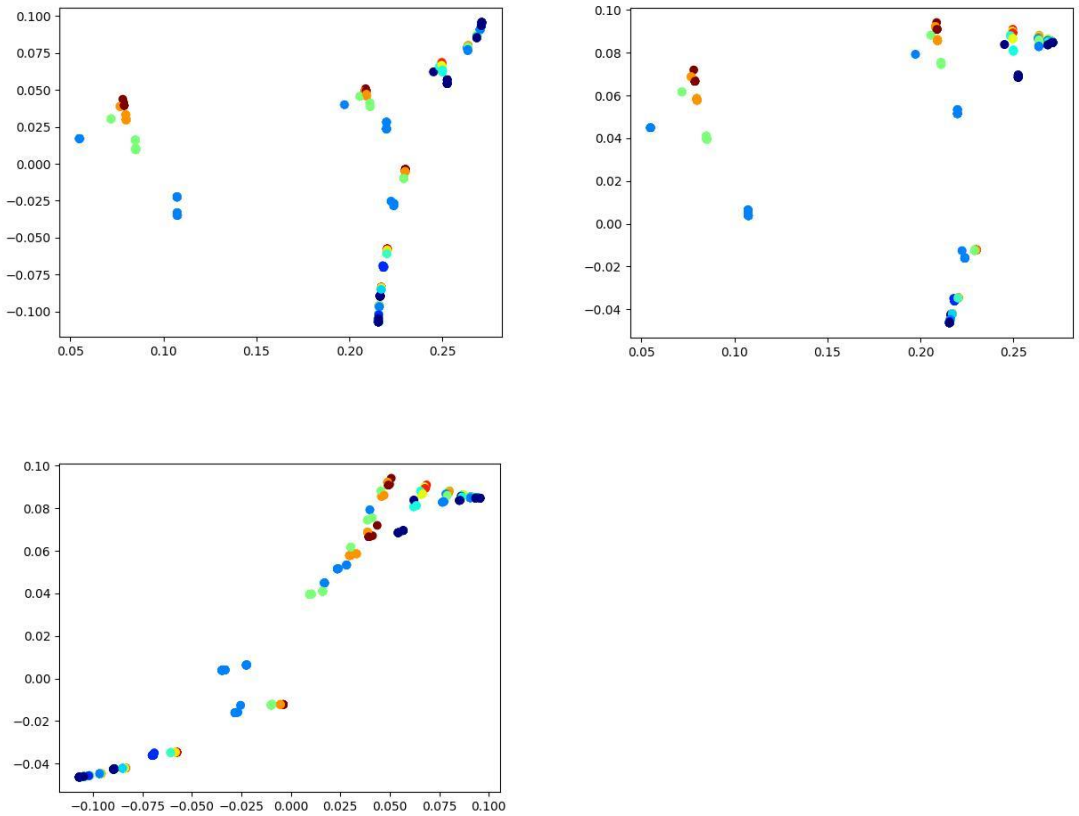
By adjusting the hidden unit activations and output probabilities as it processes each input, the network successfully learns both to count and to anticipate transitions in the sequence, accomplishing the anb2n task with correct timing and structure.

Task 3.4

Qualitative explanation

- **Phase flag.** Cell 0's output gate opens on the first symbol of each block only, so h_1 cleanly encodes A/B/C.
- **Counting B's.** While in the B-phase the forget gate of cell 1 is 0 and the input gate is 1, hence c_1 increments by one per B until reaching $2n$.
- **Re-using the count.** On the first C the peephole copies c_1 into c_2; then cell 2 increments three times, giving the required $3n$ C's, while h_1 blocks further B predictions.

Task 3.4



Task 3.5

When handling sequences with complex grammatical dependencies like anbnc3n, standard RNNs (such as Simple RNNs or SRNs) tend to perform poorly, whereas LSTMs can successfully accomplish such tasks. The main reason lies in their differences in hidden state and context unit mechanisms.

An RNN compresses all historical information into its hidden state at each time step. As the sequence grows longer, early information tends to be "overwritten" or forgotten due to the vanishing gradient problem, making it difficult for the RNN to capture long-range dependencies—such as remembering how many a's occurred, in order to output the

corresponding number of b's and c's. As a result, RNNs struggle to map inputs to correct outputs for tasks like anbnc3n.

LSTM networks introduce a cell state (memory unit) and gating mechanisms (input, forget, and output gates). The cell state acts as a "highway" for carrying crucial information over long distances, without being easily overwritten. The gates control what information to retain or forget at each step, enabling the LSTM to count and match sequence elements precisely. For example, when processing anbnc3n, the LSTM can use its cell state to record the number of a's, decrement for each b, and finally match the required number of c's, thus achieving accurate control. Therefore, LSTM can successfully solve complex prediction tasks like anbnc3n, while standard RNNs fail.