Part 1: Japanese Character Recognition

1. Answer question 1

```
[[769.   5.   7.  15.  30.  64.   1.  61.  29.  19.]
 [  7. 673. 108.  17.  29.  23.  58.  13.  24.  48.]
 [  7.  62. 692.  26.  27.  19.  46.  36.  46.  39.]
 [  4.  39.  58. 755.  16.  57.  14.  18.  28.  11.]
 [ 60.  52.  81.  21. 622.  19.  32.  36.  20.  57.]
 [  8.  27. 122.  17.  20. 727.  27.   7.  34.  11.]
 [  5.  22. 148.  10.  26.  25. 720.  20.  10.  14.]
 [ 16.  28.  28.  11.  86.  17.  54. 622.  90.  48.]
 [ 11.  36.  92.  40.   7.  31.  45.   7. 708.  23.]
 [  9.  53.  84.   3.  52.  30.  19.  31.  41. 678.]]
Test set: Average loss: 1.0096, Accuracy: 6966/10000 (70%)
```

NetLin
Input: Flattened 28×28 image = 784 pixels
Output: 10 classes
Only one linear layer:
Weights: 784 × 10 = 7840
Biases: 10
Total Parameters = **7840 + 10 = 7850**

2. Answer question 2

```
[[859.   4.   2.   5.  29.  29.   2.  40.  24.   6.]
 [  6. 819.  27.   2.  15.  11.  64.   6.  18.  32.]
 [  8.  10. 841.  47.  10.  17.  25.  10.  17.  15.]
 [  4.   9.  30. 921.   1.  14.   4.   2.   8.   7.]
 [ 35.  28.  21.   6. 820.   6.  33.  17.  17.  17.]
 [ 10.  17.  81.   6.  11. 829.  21.   2.  18.   5.]
 [  3.  16.  56.   9.  14.   8. 875.  10.   2.   7.]
 [ 19.  10.  23.   6.  24.  12.  24. 833.  21.  28.]
 [  7.  23.  31.  56.   3.   7.  30.   3. 828.  12.]
 [  3.  14.  50.   6.  28.   2.  19.  13.  12. 853.]]

Test set: Average loss: 0.4946, Accuracy: 8478/10000 (85%)
```

NetFull
Input layer: 784 units
Hidden layer: 384 units (using tanh)
Output layer: 10 classes
Layers:
fc1: 784 → 384
- Weights: 784 × 384 = 300,096
- Biases: 384

fc2: 384 → 10

- Weights: 384 × 10 = 3840
- Biases: 10

Total Parameters = 300,096 + 384 + 3840 + 10 = **304,330**

3. Answer question 3

```
[[968.   3.   2.   0.  18.   2.   0.   3.   0.   4.]
 [  0. 931.   3.   0.  10.   1.  31.   3.   3.  18.]
 [ 11.  10. 902.  15.   7.  13.  17.   8.   4.  13.]
 [  2.   1.  16. 957.   3.   5.   5.   2.   2.   7.]
 [ 14.   8.   1.   4. 952.   0.   8.   2.   9.   2.]
 [  4.   9.  29.   5.   4. 915.  21.   1.   2.  10.]
 [  4.   3.  12.   2.   9.   3. 965.   2.   0.   0.]
 [  5.   5.   2.   0.   6.   0.   6. 951.   4.  21.]
 [  3.  20.   5.   0.   7.   1.   4.   1. 953.   6.]
 [  5.   5.   7.   1.   7.   0.   5.   4.   6. 960.]]
```

Test set: Average loss: 0.2241, Accuracy: 9454/10000 (95%)

NetConv
Layer Config:
conv1: in_channels=1, out_channels=32, kernel_size=3×3, padding=1
- Weights: 1 × 32 × 3 × 3 = 288
- Biases: 32

conv2: in_channels=32, out_channels=64, kernel_size=3×3, padding=1
- Weights: 32 × 64 × 3 × 3 = 18,432
- Biases: 64

After two 2×2 poolings (28×28 → 14×14 → 7×7), final conv feature map size = 64 × 7 × 7 = 3136
fc1: 3136 → 384
- Weights: 3136 × 384 = 1,204,224
- Biases: 384

fc2: 384 → 10
- Weights: 384 × 10 = 3840
- Biases: 10

Total Parameters = 288 + 32 + 18,432 + 64 + 1,204,224 + 384 + 3840 + 10 = **1,227,274**

4. Answer question 4

a. Relative Accuracy

Model	Accuracy	Avg. Loss
NetLin	70.0%	1.0096
NetFull	84.8%	0.4946
NetConv	94.5%	0.2241

Accuracy increases significantly with model complexity:

- **NetLin** (simple linear classifier) performs worst.
- **NetFull** improves by capturing nonlinearities via hidden tanh layer.
- **NetConv** performs best by extracting spatial patterns using convolution, making it well-suited for image tasks.

b. Parameter Counts

Model	# Parameters
NetLin	7,850
NetFull	304,330
NetConv	1,227,274

Observation:

The **accuracy improvement correlates with increased model capacity**, though this also means greater computational cost and risk of overfitting if not regularized.

NetConv has **~156× more parameters than NetLin**, and ~4× more than NetFull, but achieves **~25% better accuracy than NetLin**.

c. Confusion Matrix Analysis

NetLin *(70% accuracy):*

Significant confusion between:

- Class 6 (ma) ↔ Class 2 (su) → 148 errors
- Class 1 (ki) ↔ Class 2 (su) → 108 errors
- Class 7 (ya) misclassified often as 0, 4, 6, 8

Weak spatial awareness — fails to differentiate similar-looking characters with minor shape differences.

NetFull *(85% accuracy):*

Still some confusion:

- Class 1 (ki) ↔ 6 (ma) → 64 errors
- Class 2 (su) ↔ 3 (tsu) → 47 errors
- Class 8 (re) ↔ 3 (tsu) → 56 errors

Non-linear activation (tanh) helps resolve some ambiguity, but visually similar characters still get confused.

NetConv *(95% accuracy):*

Vastly improved performance; most confusion eliminated.

Minor confusion:

- Class 1 (ki) ↔ 6 (ma) → 31 errors
- Class 2 (su) ↔ 3 (tsu) → 15 errors
- Class 7 (ya) ↔ 9 (wo) → 21 errors

These reflect **subtle visual similarity in cursive shapes**, such as:

- su and tsu having similar swirls
- ya and wo both having complex loops

Conclusion:

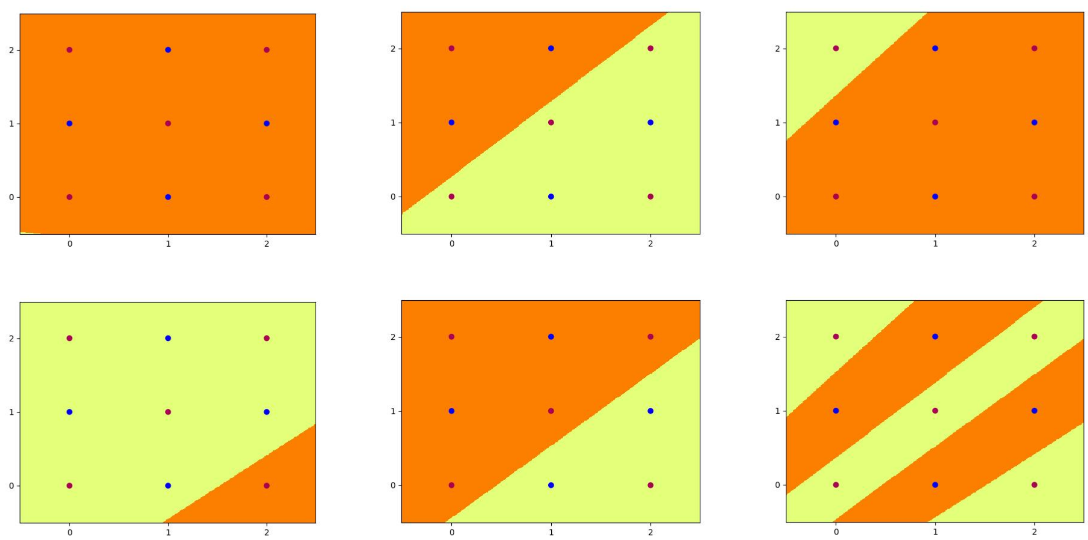
Character pairs confused across all models tend to be visually similar in stroke pattern.

NetConv can distinguish them best due to its spatial feature extraction.

Part 2: Multi-Layer Perceptron

1. Answer question 1

Final Weights:
tensor([[0.5287, 3.9119],
[-10.0896, 9.9201],
[3.5536, -2.9281],
[5.2396, -6.0887],
[-6.8512, 7.0664]])
tensor([2.1105, -2.6631, 3.9983, -7.9764, 3.1028])
tensor([[0.7431, 11.2113, 5.3990, -14.9010, -15.3084]])
tensor([1.2663])
Final Accuracy: 100.0



2. Answer question 2

Initial Weights:
tensor([[-4., 4.],
[-7., 7.],
[-7., 7.],
[-4., 4.]])
tensor([6., 1., -1., -6.])
tensor([[3., -2., 3., -2.]])
tensor([-3.])
Initial Accuracy: 100.0

Here's the structure:
Input Layer (2 nodes: x, y)
|
v
Hidden Layer (4 nodes, step activation)
|
v
Output Layer (1 node, step activation)
Weights and Biases:
Hidden Layer (in_hid):

Node	Weights [w_1, w_2]	Bias
H_1	[-4, 4]	6
H_2	[-7, 7]	1
H_3	[-7, 7]	-1
H_4	[-4, 4]	-6

Output Layer (hid_out):

Weight Vector (H_1–H_4)	Bias
[3, -2, 3, -2]	-2

2. Dividing Line Equations for Each Hidden Node
Each hidden node computes: **step(w_1·x + w_2·y + b ≥ 0)** This is equivalent to dividing the plane with a line: **w_1·x + w_2·y + b = 0**
So, the **dividing lines** (where the node switches from 0 to 1) are:
1. **Node H_1:** $-4x + 4y + 6 = 0 \rightarrow y = x - 1.5$
2. **Node H_2:** $-7x + 7y + 1 = 0 \rightarrow y = x - 1/7 \approx x - 0.14$
3. **Node H_3:** $-7x + 7y - 1 = 0 \rightarrow y = x + 1/7 \approx x + 0.14$
4. **Node H_4:** $-4x + 4y - 6 = 0 \rightarrow y = x + 1.5$

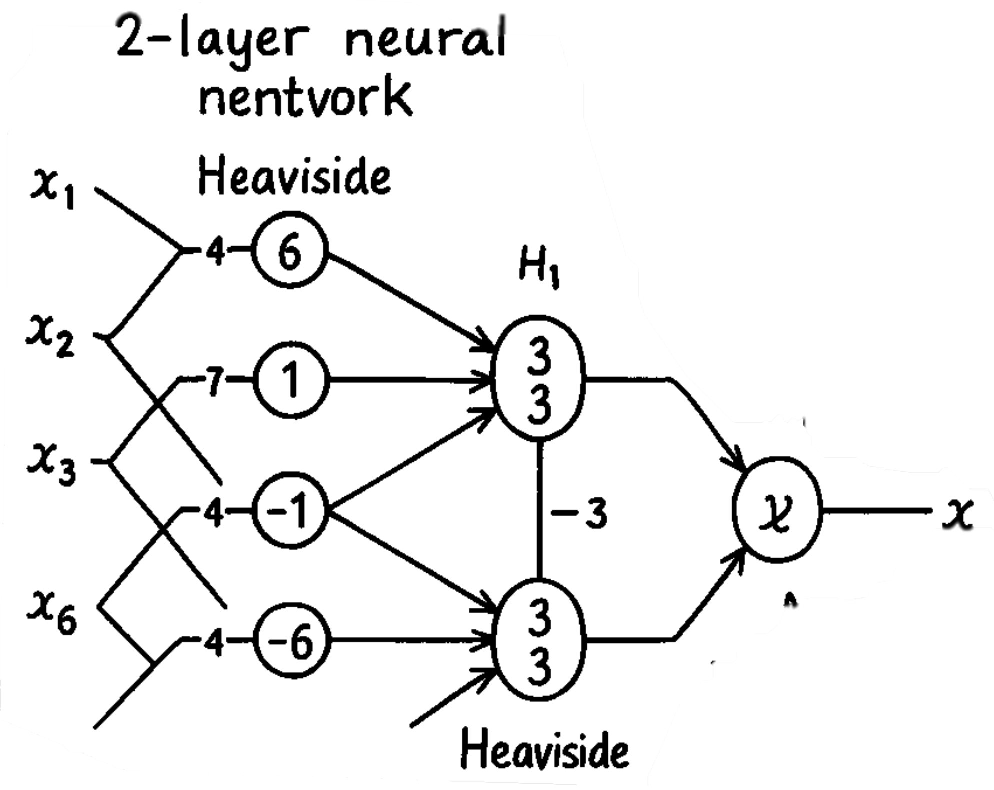
3. Activation Table
We compute all hidden and output node activations for each input.

#	x	y	H_1	H_2	H_3	H_4	Output	Target

1	0	0	1	1	0	0	0	0
2	0	1	1	1	1	0	1	1
3	0	2	1	1	1	1	0	0
4	1	0	0	0	0	0	1	1
5	1	1	1	1	1	0	0	0
6	1	2	1	1	1	1	1	1
7	2	0	0	0	0	0	0	0
8	2	1	0	0	0	1	1	1
9	2	2	1	1	1	1	0	0

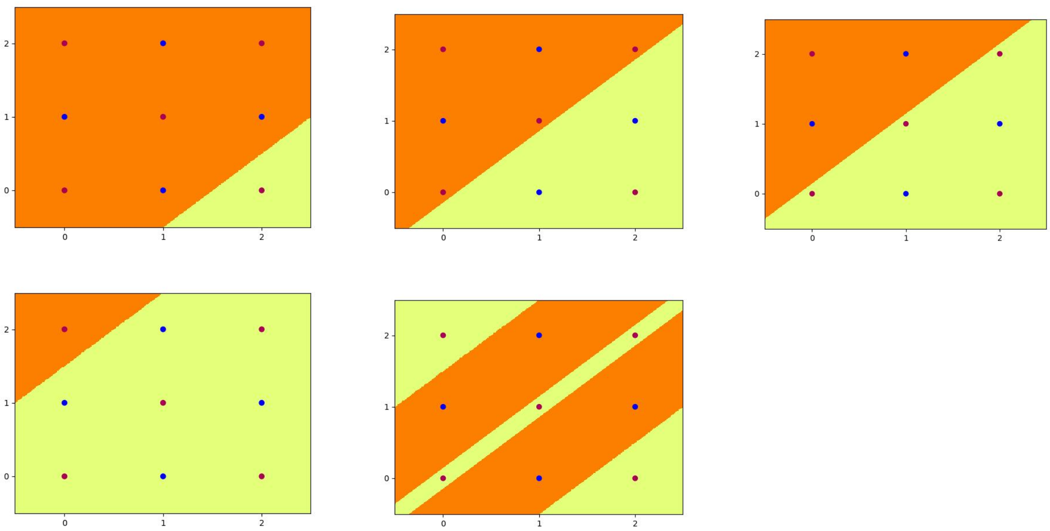
Output Node Logic:
The output node uses:

$$\text{Output} = \text{step}(3H_1 - 2H_2 + 3H_3 - 2H_4 - 2)$$



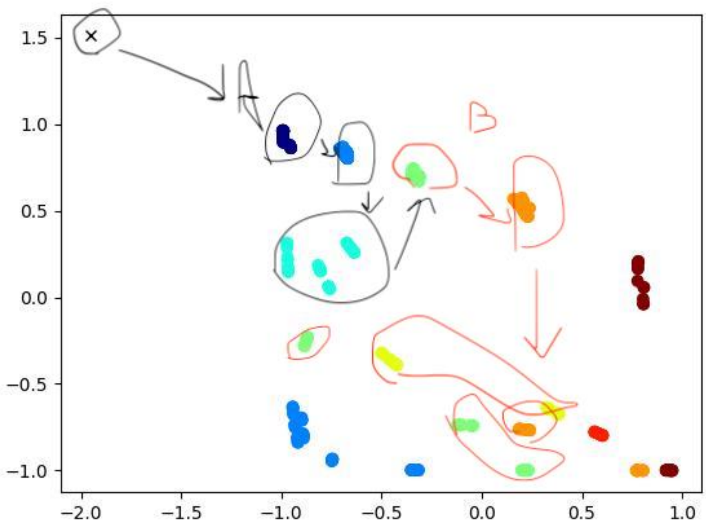
3. Answer question 3

Initial Weights:
tensor([[-40.,   40.],
        [-70.,   70.],
        [-70.,   70.],
        [-40.,   40.]])
tensor([60.,   10., -10., -60.])
tensor([[30., -20.,   30., -20.]])
tensor([-30.])
Initial Accuracy: 100.0

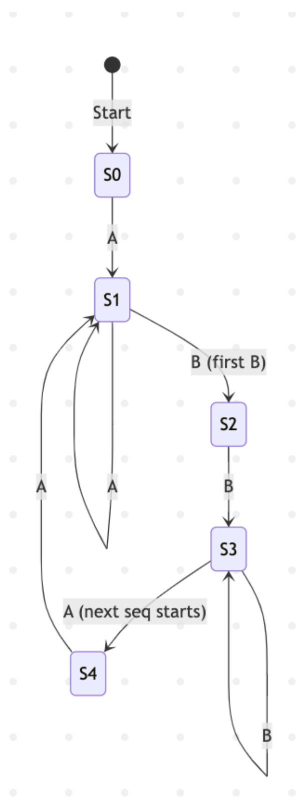


Part 3: Hidden Unit Dynamics for Recurrent Networks

1. Answer question 1

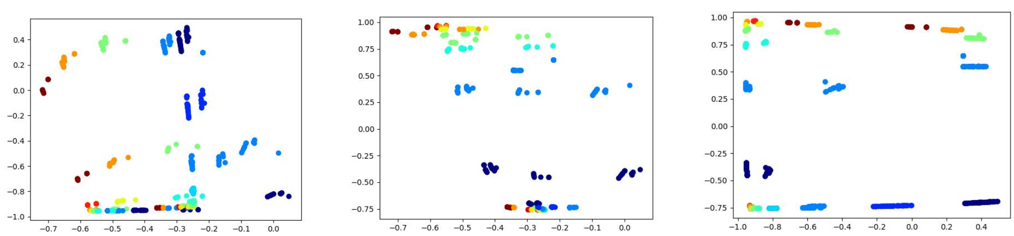


2. Answer question 2



3. Answer question 3
   The network accomplishes the *anb2n* task by encoding the count and phase of the input sequence within its hidden unit activations. As it processes a series of A's, the hidden state accumulates a representation of how many A's have been seen. When the first B appears, the network transitions into a new hidden state, marking the start of the B phase. During the sequence of B's, the hidden activations follow a predictable trajectory, allowing the network to count out exactly twice the number of B's. Because the B sequence length is deterministically linked to the number of preceding A's, the network can reliably predict every B after the first one with high confidence. Once the final B is seen, the hidden state resets into a distinct configuration corresponding to the end of one *anb2n* sequence, enabling the network to correctly predict that the next symbol is an A, thus initiating the next cycle.

4. Answer question 4



5. Answer question 5

In the *anb2nc3n* task, the LSTM must not only count the number of A’s but also distinguish between multiple phases: A’s, 2n B’s, and 3n C’s. The LSTM accomplishes this by leveraging its **cell state** (context units) to store long-term information such as the encoded value of *n*, while the **hidden state** reflects the current phase and guides prediction. During the sequence of A’s, the cell state accumulates a representation proportional to the number of A’s. When the transition to B’s begins, the LSTM switches to a different activation regime while maintaining the count in the context units, allowing it to output the correct number of B’s. This continues into the C phase, where a further internal state transition allows the network to produce 3n C’s. The separation between hidden and cell states enables the LSTM to **decouple memory from immediate output**, thus maintaining accurate long-term counting while adapting short-term behavior for each phase. By analyzing annotated plots of the hidden and context activations, one typically observes three distinct trajectories (or state clusters) corresponding to each symbol class (A, B, C), with gradual transitions marking the phase changes, and regular spacing reflecting the underlying count.