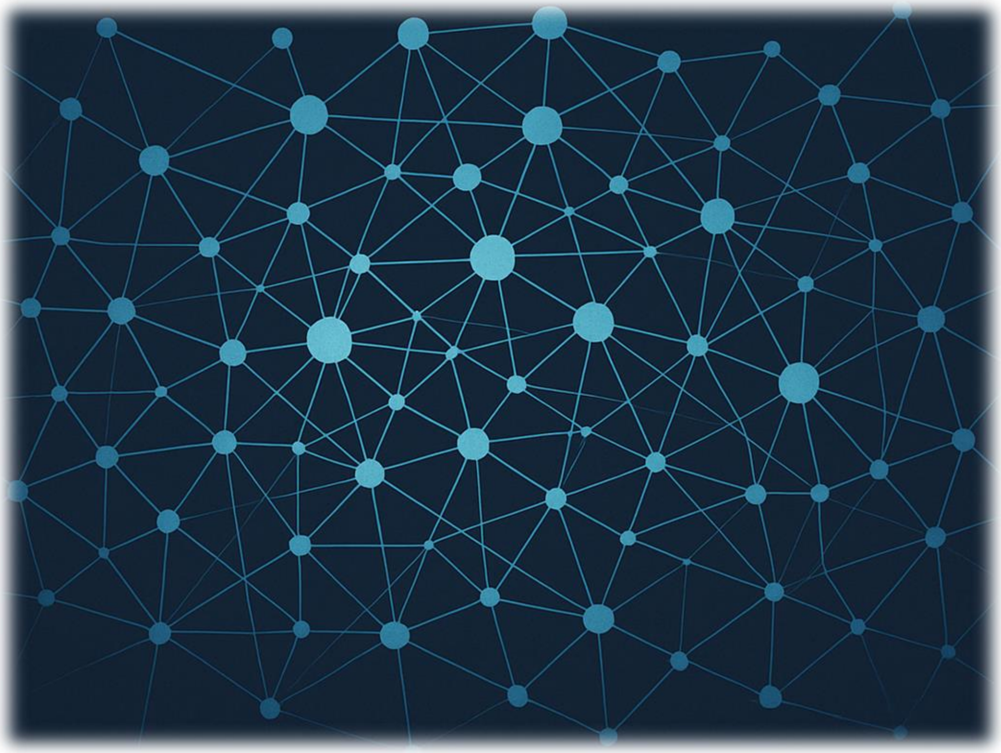
IMPLEMENTING, TRAINING AND ANALYSING NEURAL NETWORKS



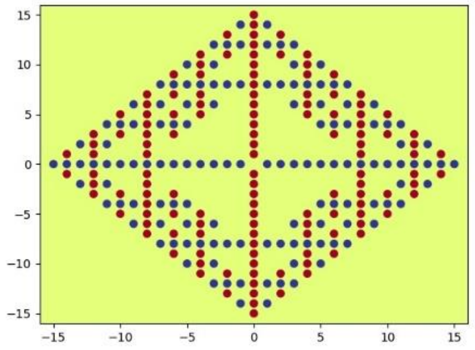
Author	Anirban Chakrabarty
zID	z5626947
Course	ZZEN9444-Neural Networks, Deep Learning (H425 Online)
Tutorial Time	July - August 2025
Tutor	Jingying Gao
Date	8/August/2025

Contents

Part 1: Fractal Classification

Step 1: Full3Net Architecture

Implementation of a three-layer fully connected neural network (Full3Net) for classifying points in the below fractal pattern based on the provided dataset.



Code Snippet: for frac.py, Full3Net class:

```
import torch
import torch.nn as nn

class Full3Net(torch.nn.Module):
    def __init__(self, hid):
        super(Full3Net, self).__init__()
        self.hid_dim = hid

        self.fc1 = nn.Linear(2, hid)       # input (x, y) → hid
        self.fc2 = nn.Linear(hid, hid)     # hid → hid
        self.fc3 = nn.Linear(hid, 1)       # hid → output (1)

    def forward(self, input):
        self.hid1 = torch.tanh(self.fc1(input))      # first hidden layer
        self.hid2 = torch.tanh(self.fc2(self.hid1)) # second hidden layer
        output = torch.sigmoid(self.fc3(self.hid2)) # output layer
        return output
```

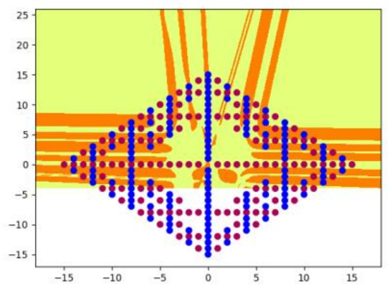
Step 2: Finding Minimum Configurations for Convergence

Formula for number of independent parameters required:

Parameters in Layer 1 = weights + biases = inputs x hidden units + hidden units
Parameters in Layer 2 = weights + biases = hidden units x hidden units + hidden units
Parameters in Output Layer = weights + biases = hidden units + outputs
Therefore, **total parameters** = (hidden units)^2 + (inputs + 3) x (hidden units) + outputs
= hid**2 + 5 hid + 1

10 hidden units in each layer:

Training has been started with **10 hidden units in each layer**. The resulting plot is as follows:

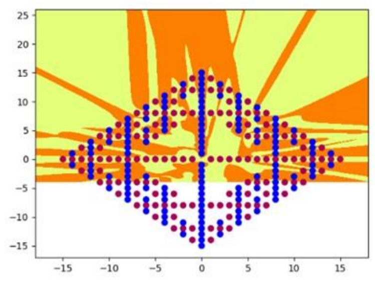


Accuracy has been increasing up to 92.62% and loss decreasing, implying that the model is converging and successfully learning the fractal pattern.

Total parameters = hid**2 + 5 hid + 1 = 10**2 + 5*10 + 1 = 151

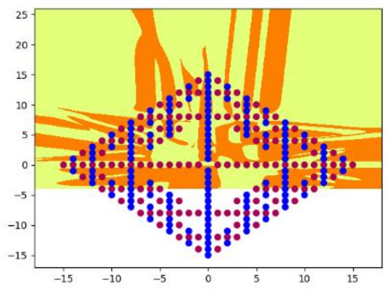
Changing the number of hidden units in each layer: Following plots show the fractal classification with 30, 20, 8, 7 and 6 hidden units in each layer.

<u>30 Hidden Units:</u> Total parameters = hid**2 + 5 hid + 1 = 30**2 + 5*30 + 1 = 1051



Accuracy reached 100% after 103700 to 123700 epochs and loss has constantly decreased.

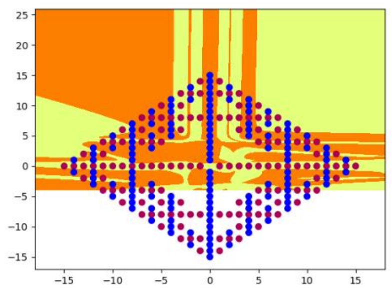
<u>20 Hidden Units:</u> Total parameters = hid**2 + 5 hid + 1 = 20**2 + 5*20 + 1 = 501



Accuracy has been increasing (up to 99.62%) and loss decreasing till 200000 epochs, implying that the model is converging and successfully learning the fractal pattern.

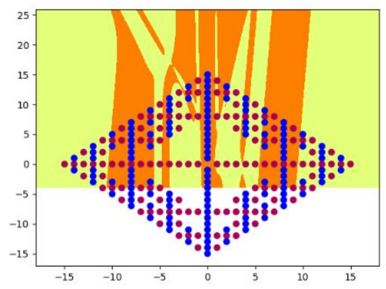
<u>8 Hidden Units:</u> Total parameters = hid**2 + 5 hid + 1 = 8**2 + 5*8 + 1 = 105

Accuracy has been increasing up to 83% and loss decreasing till 200000 epochs. Though, some fluctuations were observed, pertaining to probable local minima. This implies that the model is still converging and successfully learning the fractal pattern. This is the minimal configuration for the model to converge and learn the fractal pattern.



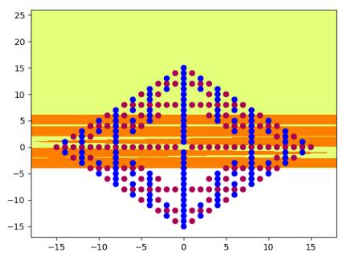
<u>7 Hidden Units:</u> Total parameters = hid**2 + 5 hid + 1 = 7**2 + 5*7 + 1 = 85

Accuracy has been very slowly increasing to 76% and loss very slowly decreasing after 35000 epochs. After this the learning practically stops, loss and accuracy both keep fluctuating.



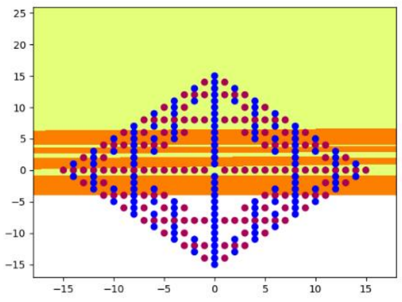
<u>6 Hidden Units:</u> Total parameters = hid**2 + 5 hid + 1 = 6**2 + 5*6 + 1 = 67

Accuracy stagnates and below 74% and loss stops decreasing with increasing number of epochs. This implies that the model has stopped converging and learning the fractal pattern.

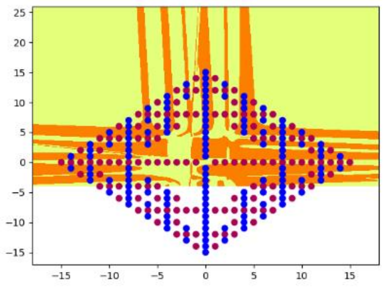


4 Hidden Units: Total parameters = hid**2 + 5 * hid + 1 = 37

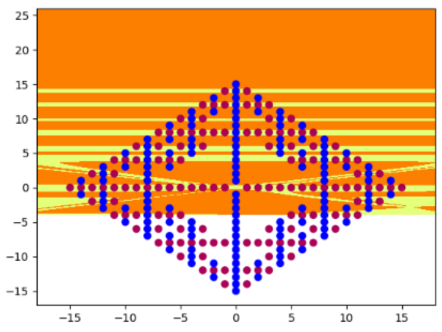
Learning stops at ~20000epochs with accuracy of 65%.



Varying Learning Rate to 0.001 with 8 Hidden Units: Learning Rate has been decreased to 0.001 to observe if the convergence get better, 8 hidden units. Accuracy went up to 84.36% and loss stagnated. The model performance does not increase (rather gets slightly degraded) by decreasing the learning rate from 0.01 to 0.001. Convergence remains similar.



Varying Initial Weight Scale to 0.1 with 8 Hidden Units (Learning Rate to 0.01): Initial weight scale has been changed to 0.1 to observe if the convergence get better. Learning Rate to = 0.01. With 8 neurons per hidden layer and Learning Rate = 0.01, varying the Initial Weight Scale to 0.1, accuracy increases slightly to 86.15% and loss stagnates with increasing epochs. Therefor the model converges and performance gets slightly better with initial weight scale = 0.1



Step 3: Full4Net Architecture:

Implementation of a 4-layer fully connected neural network (Full3Net) for classifying points in the above fractal pattern based on provided dataset.

Code Snippet: appended to frac.py for Full4Net class:

```
class Full4Net(nn.Module):
    def __init__(self, hid):
        super(Full4Net, self).__init__()
        self.fc1 = nn.Linear(2, hid)       # Input to Hidden Layer 1
        self.fc2 = nn.Linear(hid, hid)     # Hidden Layer 1 to Hidden Layer 2
        self.fc3 = nn.Linear(hid, hid)     # Hidden Layer 2 to Hidden Layer 3
        self.fc4 = nn.Linear(hid, 1)       # Hidden Layer 3 to Output

    def forward(self, x):
        # Hidden layers with tanh activation
        self.hid1 = torch.tanh(self.fc1(x))
        self.hid2 = torch.tanh(self.fc2(self.hid1))
        self.hid3 = torch.tanh(self.fc3(self.hid2))
        # Output layer with sigmoid activation
        out = torch.sigmoid(self.fc4(self.hid3))
        return out
```

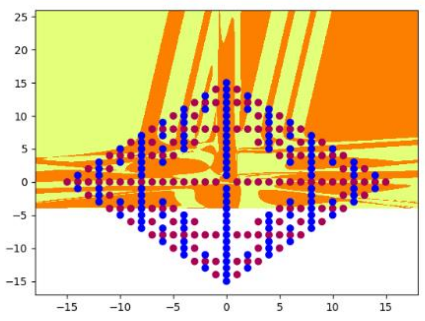
Step 4: Minimum Convergence Configuration: 4 Layered Network

Formula for number of independent parameters required:

Parameters in Layer 1 = weights + biases = inputs x hidden units + hidden units
Parameters in Layer 2 = weights + biases = hidden units x hidden units + hidden units
Parameters in Layer 3 = weights + biases = hidden units x hidden units + hidden units
Parameters in Output Layer = weights + biases = hidden units + outputs
Therefore, **total parameters** = 2 x (hidden units)^2 + (inputs + 4) x (hidden units) + outputs
= 2 * hid **2 + 6 * hid + 1

10 hidden units in each layer: **Total parameters =** 2 * hid**2 + 6 * hid + 1 = 261

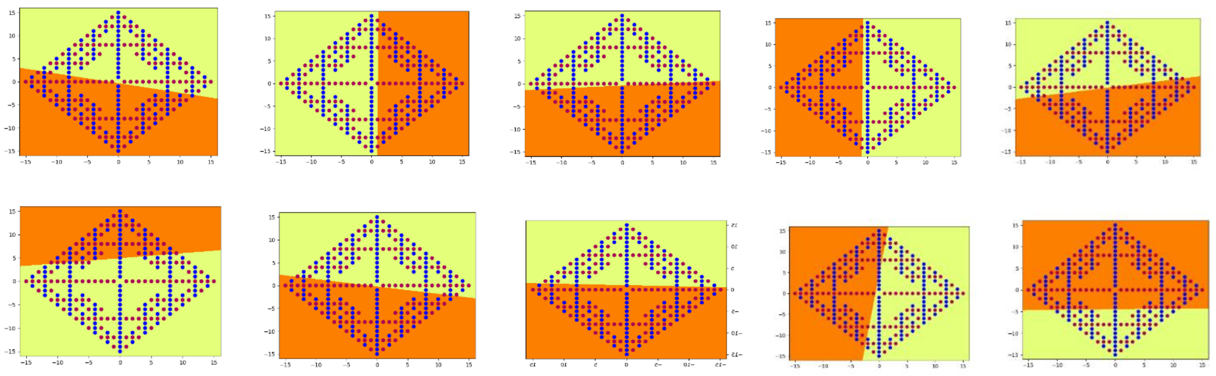
Training has started with **10 hidden units in each layer**. The resulting plot is as follows:



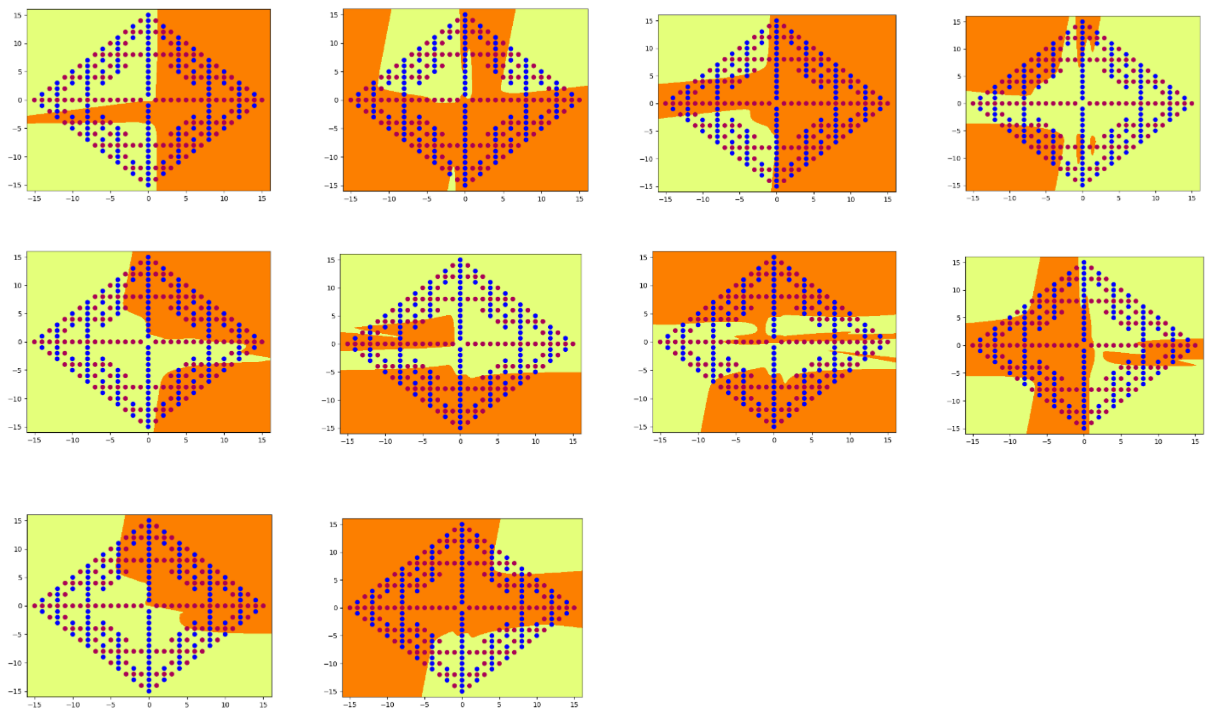
Output layer combines all features to assign a probability that a point belongs to the fractal set.

Accuracy has been increasing up to 91.92% and loss decreasing, implying that the model is converging and successfully learning the fractal pattern. The model's performance remains more or less same with the addition of one extra hidden layer, slightly degraded due to overfitting.

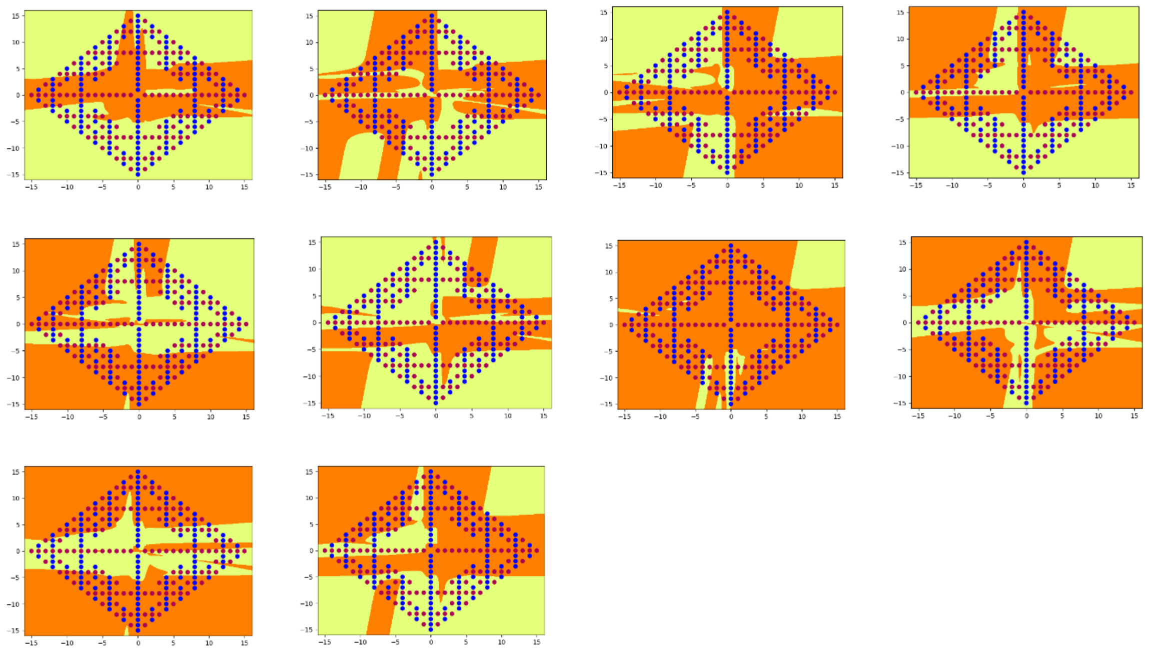
Outputs from Hidden Layer-1 of 4 Layer Network with 10 Hidden Neurons in Each Layer: Applies nonlinear transformation to the 2-D input. The tanh activation maps the raw coordinates into a richer representation, effectively carving the input plane into broad nonlinear regions.



Outputs from Hidden Layer-2 of 4 Layer Network with 10 Hidden Neurons in Each Layer: Refines these regions further, learning more localized curved boundaries by recombining activations from h1.



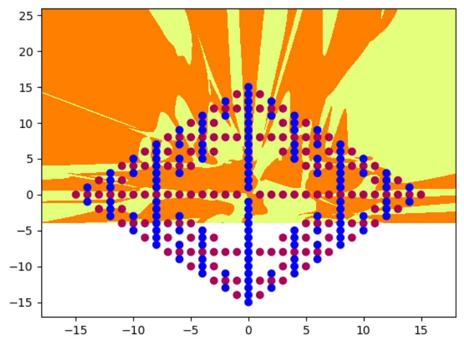
Outputs from Hidden Layer-3 of 4 Layer Network with 10 Hidden Neurons in Each Layer: Adds additional nonlinearity and complexity, enabling the network to represent finer details of the fractal pattern. At this stage, neurons often act as detectors for specific "patches" or structures in the fractal.



Changing the number of hidden units in each layer:

Following plots show the fractal classification with 20, 8, 7 and 6 hidden units in each layer.

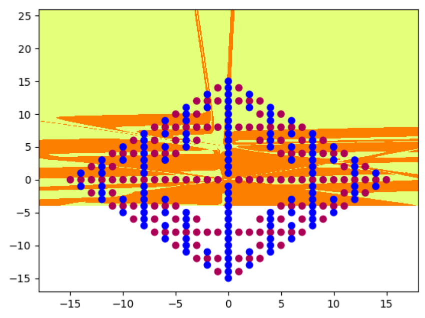
<u>**20 Hidden Units:**</u> **Total parameters =** 2*hid**2 + 6 * hid + 1 = 2*20**2 + 6*20 + 1 = 921



Accuracy reached 100% after 92400 to 96400 epochs and loss has constantly decreased. Performance slightly increased from 3 layer to 4 layer network with 20 hidden units per layer.

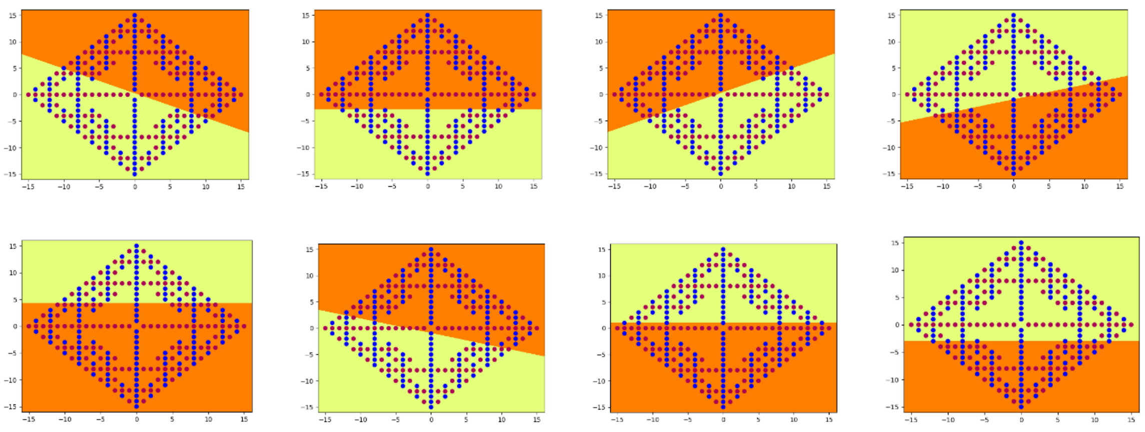
Outputs from each hidden layer of a 4 Layer Network with 20 Hidden Neurons have been omitted as the number of images is (20*4 = 80) too big.

<u>**8 Hidden Units:**</u> **Total parameters =** 2*hid**2 + 6 * hid + 1 = 2*8**2 + 6*8 + 1 = 177

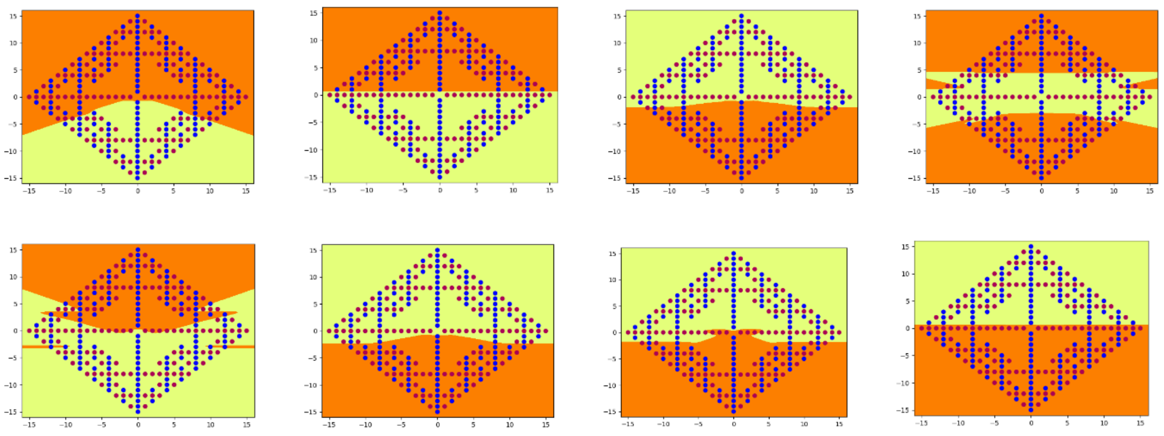


With moderate fluctuations, accuracy increased to 83% and loss decreased till 200000 epochs. No further improvement from the previous 3-layer network.

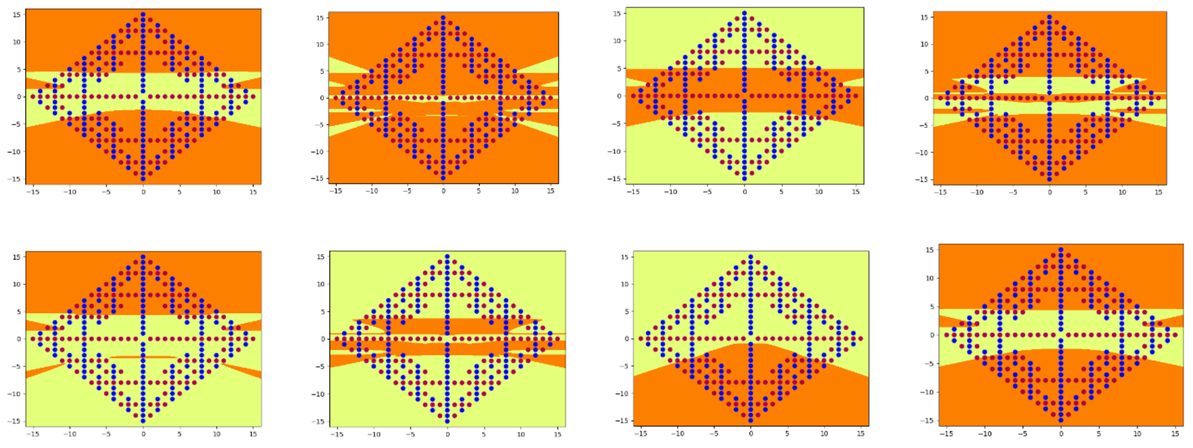
Outputs from Hidden Layer-1 of 4 Layer Network with 8 Hidden Neurons in Each Layer:



Outputs from Hidden Layer-2 of 4 Layer Network with 8 Hidden Neurons in Each Layer:

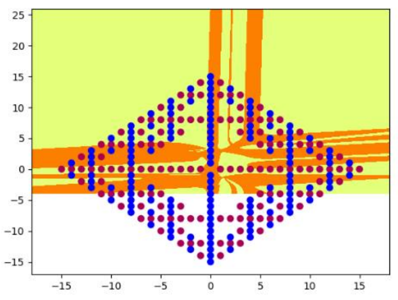


Outputs from Hidden Layer-3 of 4 Layer Network with 8 Hidden Neurons in Each Layer:



<u>7 Hidden Units:</u> Total parameters = 2*hid**2 + 6*hid + 1 = 2*7**2 + 6*7 + 1 = 141

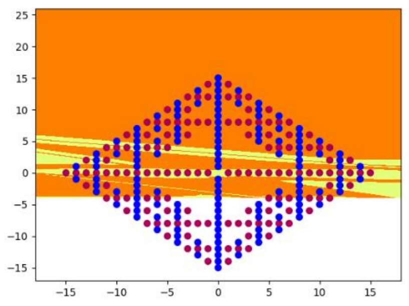
Accuracy has been very slowly increasing to 75.46% and loss very slowly decreasing.



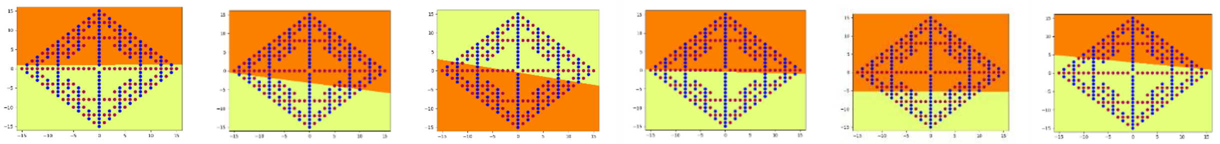
The outputs of each hidden layers are omitted as outputs of the 4 layer networks with 8 (above) units and 6 (below) units have been shown.

<u>6 Hidden Units:</u> Total parameters = 2*hid**2 + 6*hid + 1 = 2*6**2 + 6*6 + 1 = 109

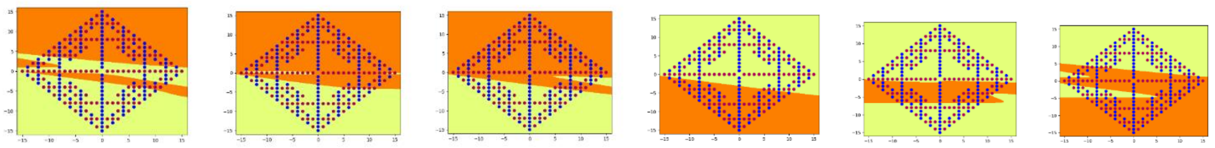
Accuracy stagnates below 72.6% and loss stops decreasing with increasing number of epochs. This implies that the model has stopped converging and learning the fractal pattern.



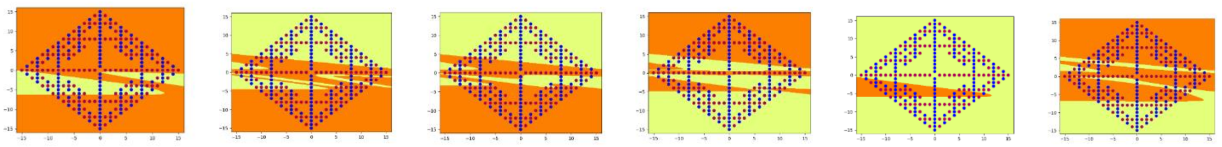
Outputs from Hidden Layer-1 of 4 Layer Network with 6 Hidden Neurons in Each Layer:



Outputs from Hidden Layer-2 of 4 Layer Network with 6 Hidden Neurons in Each Layer:

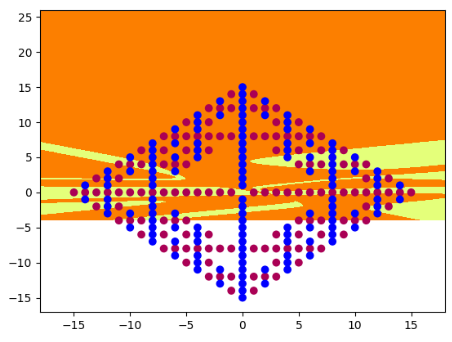


Outputs from Hidden Layer-3 of 4 Layer Network with 6 Hidden Neurons in Each Layer:

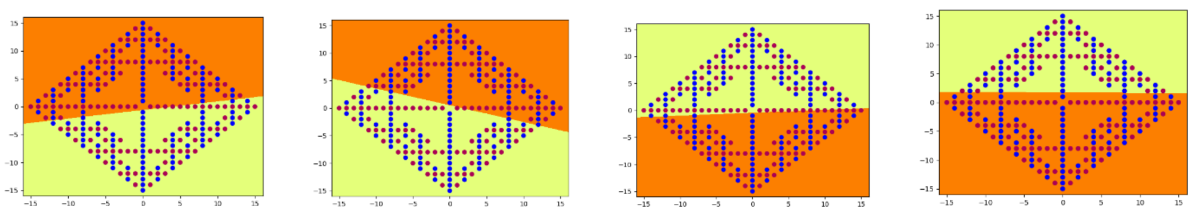


<u>**4 Hidden Units:**</u> **Total parameters =** 2*hid**2 + 6*hid + 1 = 57

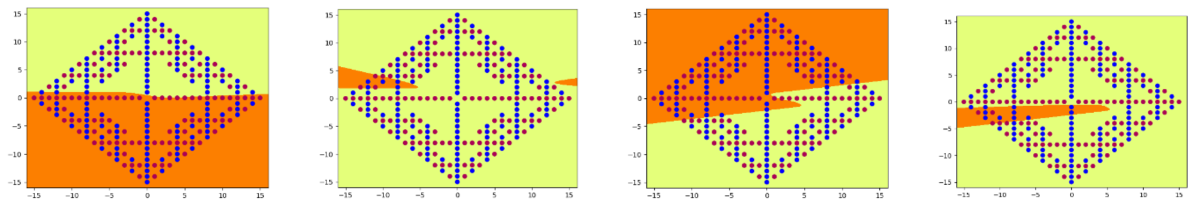
Learning does not practically happen with an increasing number of epochs. Accuracy fluctuates constantly between 63 to 67%.



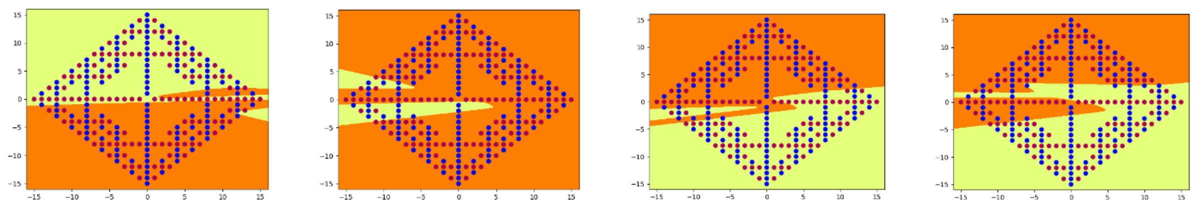
Outputs from Hidden Layer-1 of 4 Layer Network with 4 Hidden Neurons in Each Layer:



Outputs from Hidden Layer-2 of 4 Layer Network with 4 Hidden Neurons in Each Layer:



Outputs from Hidden Layer-3 of 4 Layer Network with 4 Hidden Neurons in Each Layer:



Step 5: DenseNet Architecture

Objective:

To implement a 3-layer densely connected neural network (DenseNet) that extends the Full3Net model by including shortcut (dense) connections. These connections allow each hidden layer and the output layer to receive inputs not only from the immediately preceding layer but also directly from earlier layers and the raw input.

Architecture Description:

- **Input Layer:** 2 input nodes representing the (x, y) coordinates.
- **First Hidden Layer (h1): hid** nodes, each using the **tanh** activation:

$$h_1 = \tanh(b_1 + W_{10}\,x)$$

- **Second Hidden Layer (h2): hid** nodes with **tanh** activation, receiving inputs from both the original input **x** and the first hidden layer **h1**:

$$h_2 = \tanh(b_2 + W_{20}\,x + W_{21}\,h_1)$$

- **Output Layer:** A single node with **sigmoid** activation. It receives contributions from the input, the first hidden layer, and the second hidden layer:

$$\text{out} = \sigma(b_{out} + W_{30}\,x + W_{31}\,h_1 + W_{32}\,h_2)$$

Code Snippet:

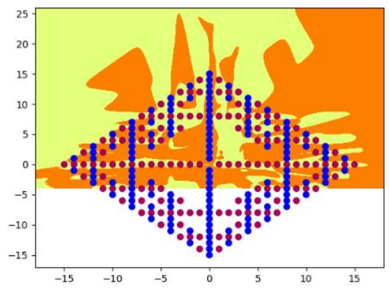
```
class DenseNet(nn.Module):
    def __init__(self, hid: int):
        super().__init__()
        self.hid = hid
        # x ∈ R^2
        self.fc1 = nn.Linear(2, hid)              # W10, b1
        self.fc2 = nn.Linear(2 + hid, hid)        # [W20 | W21], b2
        self.fc3 = nn.Linear(2 + hid + hid, 1)    # [W30 | W31 | W32], b_out

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # h1
        self.hid1 = torch.tanh(self.fc1(x))
        # h2 uses both x and h1
        h2_in = torch.cat([x, self.hid1], dim=1)
        self.hid2 = torch.tanh(self.fc2(h2_in))
        # out uses x, h1, h2
        out_in = torch.cat([x, self.hid1, self.hid2], dim=1)
        out = torch.sigmoid(self.fc3(out_in))
        return out
```

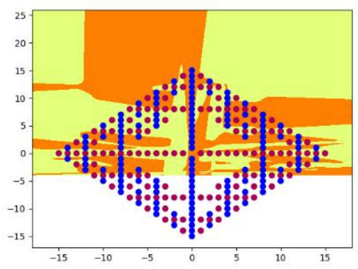
Step 6: Train the Dense Network:

Varying the Number of Hidden Units:

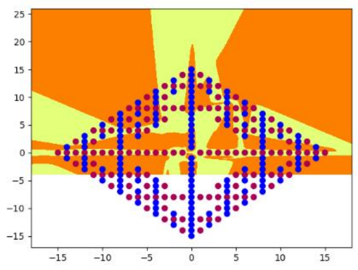
<u>20 Hidden Units:</u> If the above DenseNet code is run for 20 hidden units, the model converges very fast with a 100% accuracy from as low as 8800 to 18800 epochs. This is a considerable improvement from the Fully connected 3-layer network above. Resulting plot is shown below.



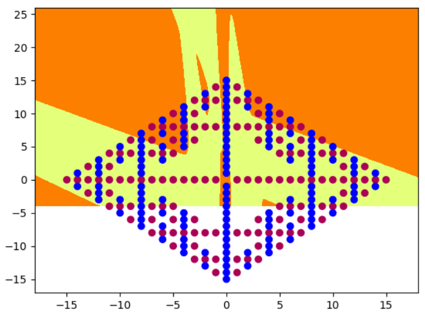
8 hidden units: Model converges consistently with decreasing loss and increasing accuracy up to 88.34%. It shows a considerable improvement from the Fully connected 3-layer network.



6 hidden units: The model still converges with consistent loss and accuracy up to 75% at about 50000 epochs. This improvement is due to the Dense Network connections architecture. After 50000 epochs there is no considerable learning.



4 hidden units: The model hardly shows any decrease in loss below and accuracy is always below 64.7% right from the initial few epochs till end. There is no considerable learning.



Step 7:

A. Model Size vs. Training Behavior

Parameter formulas (with brief derivations)

1. **Fully connected 3-layer (Full3Net)**
   Formula: params = hid^2 + 5 * hid + 1
   Derivation:
   - Input to H1: weights = 2 * hid, biases = hid => 3 * hid
   - H1 to H2: weights = hid * hid, biases = hid => hid^2 + hid
   - H2 to Out: weights = hid, biases = 1
   - **Sum =** hid^2 + 5 * hid + 1
2. **Fully connected 4-layer (Full4Net)**
   Formula: params = 2 * hid^2 + 6 * hid + 1
   Derivation:
   - Input to H1: weights = 2 * hid, biases = hid => 3 * hid
   - H1 to H2: weights = hid * hid, biases = hid => hid^2 + hid
   - H2 to H3: weights = hid * hid, biases = hid => hid^2 + hid
   - H3 to Out: weights = hid, biases = 1
   - **Sum =** 2 * hid^2 + 6 * hid + 1
3. **DenseNet (3-layer with shortcuts)**
   Formula: params = hid^2 + 8 * hid + 3
   Derivation:
   - h1 = tanh (b_1 + W_{10} * x) => 2 * hid + hid = 3*hid
   - h2=tanh (b_2 + W_{20}*x + W_{21}*h_1) => W_{20}(2*hid) + W_{21}(hid^2) + b_2(hid) = hid^2 + 3*hid
   - out=σ(b_o+W_{30}*x+W_{31}*h_1+W_{32}*h_2) => W_{30}(2) + W_{31}(hid) + W_{32}(hid) + + b_0(1) = 2*hid + 3
   - **Sum =** hid^2 + 8*hid+3

Reliable convergence happens at:

- **Full3Net (hid = 30)**
  Therefore, Number of independent Parameters = 30^2 + 5*30 +1 = 1051

- **Full4Net (hid = 20)**
  Therefore, Number of independent Parameters = 2 * 20^2 + 6 * 20 + 1 = 921

- **DenseNet (hid = 20)**
  Therefore, Number of independent Parameters = 20^2 + 8 * 20 + 3 = 563

Note: Although Full4Net is deeper and has more parameters, DenseNet achieves the smallest parameter count among the successful settings due to its efficient dense connections.

<u>**Approximate training epochs observed**</u>

- **Full3Net**
    - **hid = 30:** stabilizes ~**100,000** epochs (accuracy 100%).
    - **hid = 8, 10, 20:** continues learning up to **200,000** epochs with fluctuations (accuracy improving, loss decreasing).
    - **hid = 7:** stalls learning at ~**35,000** epochs.
    - **hid = 4:** stalls learning at ~**20,000** epochs.
- **Full4Net:** Adding depth alone (Full4Net) helped compared with small Full3Net
    - **hid = 20:** stabilizes ~**90,000** epochs (accuracy 100%).
    - **hid = 4:** highly fluctuating; no stable convergence observed.
- **DenseNet:** converged fastest (after fewest epochs) at comparable accuracy, despite fewer parameters than Full3Net and Full4Net at the successful settings. **Dense shortcuts** provided stronger gradient flow and feature reuse, markedly reducing epochs to convergence.
    - **hid = 20:** stabilizes **~10,000 epochs** (accuracy 100%).
    - **hid = 6:** stalls learning at ~**50,000** epochs.
- **Capacity threshold matters**: below certain hid values (e.g., Full3Net at 7, DenseNet at 6), training stagnates—indicating insufficient model capacity to represent the fractal decision boundary.

B. Qualitative Description of Layer Functions

<u>**Full4Net**</u>

- **First hidden layer (h1):** Applies a nonlinear transformation to the 2-D input (x, y). The tanh activation maps the raw coordinates into a richer representation, effectively carving the input plane into broad nonlinear regions.
- **Second hidden layer (h2):** Refines these regions further, learning more localized curved boundaries by recombining activations from h1.
- **Third hidden layer (h3):** Adds additional nonlinearity and complexity, enabling the network to represent finer details of the fractal pattern. At this stage, neurons often act as detectors for specific "patches" or structures in the fractal.
- **Output layer:** Combines all features to assign a probability that a point belongs to the fractal set.

Overall, the depth of Full4Net allows a hierarchical build-up: from broad shapes (layer 1) → intermediate motifs (layer 2) → detailed fragments (layer 3).

<u>**DenseNet**</u>

- **First hidden layer (h1):** Similar to Full4Net's h1, it maps the 2-D input into nonlinear basis functions.
- **Second hidden layer (h2):** Receives both the raw input and h1. This means it can simultaneously capture direct input features (like global axes or symmetries in the fractal) and higher-order combinations from h1, leading to more efficient feature reuse.
- **Output layer:** Integrates information from all sources (x, h1, h2). This dense connectivity allows the output to directly exploit both low-level (raw input, broad shapes) and high-level (refined hidden activations) features.

As a result, DenseNet's functions are less strictly hierarchical and more **feature-sharing**. Earlier representations continue to influence later layers, leading to faster training and smoother decision boundaries.

This qualitative comparison above, emphasizes the difference:

- **Full4Net** builds up complexity in stages, with each layer depending primarily on the previous one.
- **DenseNet** blends information from all levels, making it more efficient at representing complex fractal boundaries with fewer epochs.

C. Overall Qualitative Decision Function

Fully connected 3-layer (Full3Net)

- **Shape/complexity:** Can learn a nonlinear boundary, but with limited width it often shows **coarser partitions** and **fragmented islands** in the fractal—evidence of capacity limits.
- **Artifacts:** More susceptible to **underfitting** (missed fine detail) or **wavy/rippled** boundaries when it struggles to reconcile global and local structure using only two hidden transforms.
- **Takeaway:** Adequate for broad structure; fine fractal detail requires larger width or many epochs.

Fully connected 4-layer (Full4Net)

- **Shape/complexity:** Added depth enables a **more hierarchical composition** of features, so the output boundary is typically **finer and more articulated** than Full3Net at similar width.

- **Artifacts:** Can still show **training fluctuations**; without shortcuts, information must pass layer-by-layer, which can slow optimization and sometimes yield **over-sharp local features** or uneven refinement.
- **Takeaway:** Better at representing intricate boundaries than Full3Net, but may need careful tuning (epochs, initialization of weights, learning rate).

DenseNet (3-layer with shortcuts)

- **Shape/complexity:** Dense skip connections let the output depend **directly on input and earlier features**, producing a boundary that is **smooth where appropriate** yet **captures fine detail earlier** in training.
- **Optimization effect:** Shortcuts aid gradient flow and **feature reuse**, so the learned function often looks **cleaner and more stable** at lower width and fewer epochs.
- **Takeaway:** For comparable capacity, DenseNet tends to achieve a **more accurate and less noisy** approximation of the fractal decision set.

Big-picture summary

- With enough width/depth, all three are universal approximators; differences emerge **under practical limits** (your chosen hid, training time).
- **Full3Net:** coarser/global patterns first; fine details come late or not at all.
- **Full4Net:** deeper hierarchy adds detail, but optimization is still strictly sequential.
- **DenseNet:** shortcut-aided function that blends global (input) and intermediate features at the output, typically yielding the **best qualitative boundary** at the same or lower parameter count and epochs.

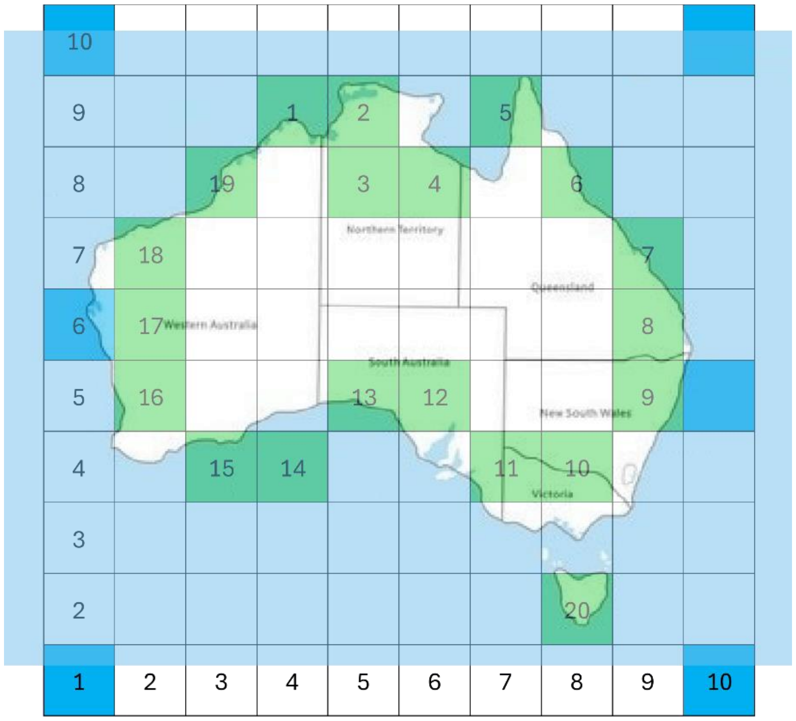
Caveat: Exact visuals depend on initialization, learning rate, and stochasticity; the above trends match the observed convergence/epoch behavior.

Part 2: Encoder Networks: Australia Dataset (aus26)

Dataset Design and Construction

The objective of this task was to create a **26 × 20 tensor** (aus26) in *encoder.py* such that, when passed as the target to encoder_main.py, the trained encoder network would produce a stylized representation of Australia.

The design process began by superimposing a 10×10 coordinate grid over a geographical map of Australia (see diagram below). This grid was used to identify and mark 20 key feature points representing notable geographical anchors, plus six fixed anchors at the four corners and midpoints of the left and right borders of the grid.



Each point's coordinates on the grid were mapped to binary activations for the network's 20 output units, corresponding to the "half-space" on one side of a learned linear decision boundary (i.e., vertical or horizontal line segments drawn in the hidden-unit space). The first 10 output units encode boundaries along the x-axis, while the next 10 encode boundaries along the y-axis.

A 1 indicates the point lies on the "positive" side of the respective decision boundary, while 0 indicates the "negative" side.

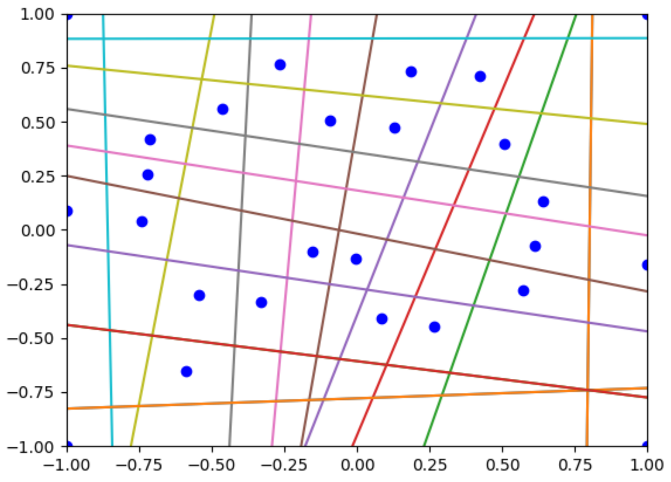
The final tensor (aus26) below was manually constructed to match the positional relationships observed in *Diagram.png*. This ensured the placement of dots and decision lines closely matched the stylized map.

```
aus26 = torch.Tensor([
   [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],  # bottom-left anchor
   [0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1],  # top-left anchor
   [1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0],  # bottom-right anchor
   [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],  # top-right anchor
   [0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0],  # mid-left edge anchor (0, 6)
   [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0],  # mid-right edge anchor (10, 5)
   [1,1,1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0],  # WA  (4, 9) #1
   [1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0],  # NT  (5,9)  #2
   [1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0],  # NT  (5,8)  #3
   [1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,1,0,0],  # NT  (6,8)  #4
   [1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,0],  # QLD (7,9)  #5
   [1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,0,0],  # QLD (8,8)  #6
   [1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,0,0],  # QLD (9,7)  #7
   [1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,0,0,0],  # QLD (9,6)  #8
   [1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,0,0,0,0],  # NSW (9,5)  #9
   [1,1,1,1,1,1,1,1,0,0,1,1,1,1,0,0,0,0,0,0],  # NSW (8,4)  #10
   [1,1,1,1,1,1,1,0,0,0,1,1,1,1,0,0,0,0,0,0],  # VIC (7,4)  #11
   [1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,0],  # SA  (6,5)  #12
   [1,1,1,1,1,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0],  # SA  (5,5)  #13
   [1,1,1,1,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0],  # WA  (4,4)  #14
   [1,1,1,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0],  # WA  (3,4)  #15
   [1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0],  # WA  (2,5)  #16
   [1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0],  # WA  (2,6)  #17
   [1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0],  # WA  (2,7)  #18
   [1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0],  # WA  (3,8)  #19
   [1,1,1,1,1,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0]   # TAS (8,2)  #20
])
```

Result and Observations

The following command produced the output shown below.

```
python3 encoder_main.py --target aus26
```



Explanation of the Horizontal Reflection

The resulting plot matched the target configuration in terms of dot placement and decision-line arrangement but appeared as a horizontal mirror image of the original geographical layout. The horizontal reflection occurs due to the way the encoder network maps input coordinates to the 2D hidden space. Specifically:

1. **Encoding transformation:** The learned input-to-hidden weight matrix can apply an axis inversion if the sign of one of its weight vectors is reversed during training.
2. **Symmetry of the target pattern:** Because the target tensor only encodes *relative half-space relationships* with respect to vertical and horizontal boundaries, the network can achieve the same loss minimization whether reflects the image or not — as long as the relative positions between points and boundaries are preserved.
3. **No explicit constraint on orientation:** The training objective minimizes reconstruction error, not absolute geographical orientation. This means the solution space includes both the correct orientation and any flipped/mirrored versions that yield identical half-space classifications.

In effect, the encoder has found an equally valid solution by reflecting the pattern horizontally, swapping the east and west halves of the map. This is analogous to a PCA or autoencoder projection where the principal components may be flipped without changing their explanatory power.

Conclusion

The aus26 dataset successfully demonstrated how careful manual encoding of target outputs can guide an encoder network to produce structured, meaningful 2D representations. The observed horizontal reflection highlights the inherent symmetry in such learned feature mappings, underlining the importance of incorporating orientation constraints if absolute spatial accuracy is required.

Part 3 – Japanese Character Recognition

This part of the assignment focuses on the recognition of handwritten Japanese Hiragana characters using neural networks. The dataset used is *Kuzushiji-MNIST (KMNIST)*, a modern benchmark designed to reflect the challenges of recognizing cursive Japanese characters from historical texts. It consists of 70,000 grayscale images (28×28 pixels) across 10 classes of Hiragana. The task involves implementing and evaluating three neural network architectures of increasing complexity—a linear model, a fully connected model, and a convolutional neural network—comparing their performance in terms of classification accuracy, parameter count, and confusion patterns.

Step 1 – Linear Model (NetLin)

To establish a baseline, a linear classifier NetLin was implemented, consisting of a single fully connected layer that maps the flattened 28×28 grayscale image to the 10 output classes, followed by a log softmax activation. This model is equivalent to multinomial logistic regression and does not include any hidden layers or non-linear transformations. The model was trained for 10 epochs using the training portion of the Kuzushiji-MNIST dataset, with the negative log-likelihood (NLL) loss function and stochastic gradient descent. As expected, the model's performance plateaued around 70% accuracy on the test set.

Final Test Accuracy

Accuracy: 70% (approx.) (6953 / 10000)

Confusion Matrix

Each row corresponds to the true label and each column to the predicted label. The labels correspond to the 10 Hiragana characters in the order:
0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo".

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	764	5	8	14	30	62	2	63	31	21
1	7	669	106	17	28	24	59	14	26	50
2	8	62	690	25	27	21	48	35	46	38
3	5	35	60	755	15	57	14	18	28	13
4	62	52	78	22	624	18	32	37	20	55
5	8	28	125	17	19	725	27	8	33	10
6	5	21	148	10	25	24	722	21	10	14
7	16	32	26	12	82	17	53	625	89	48
8	11	37	93	42	8	30	46	7	702	24
9	8	50	91	4	51	30	19	31	39	677

Observations

- The linear model was able to achieve basic discrimination among character classes.
- Misclassifications were common between visually similar classes such as:
  - "su" vs "ma" or "ha"
  - "na" vs "re"
  - "ki" vs "su"
- These confusions reflect the difficulty of linearly separating characters that have subtle stroke differences, especially when handwritten and down sampled to 28×28 resolution.
- Baseline result will serve as a point of comparison for deeper networks in following steps.

Step 2 – Fully Connected Network (NetFull)

In this step, a fully connected two-layer neural network, NetFull was implemented, to classify the 10 classes of the Kuzushiji-MNIST dataset. The network consists of a hidden layer with a tunable number of units using the tanh activation function, followed by an output layer with log softmax. Training was done using the negative log-likelihood loss function for 10 epochs.

To determine the optimal number of hidden units, the network was tested with multiple values in multiples of 10 (e.g. like 30, 60, 80, 90, 100, 110, 120, 140 and 160). Goal was to achieve at least 84% classification accuracy on the test set. The best result was obtained with 160 hidden units.

Final Test Accuracy

Accuracy: 84.68% (8468 / 10000) with 160 hidden units.

Confusion Matrix

Each row corresponds to the true label and each column to the predicted label. The labels correspond to the 10 Hiragana characters in the order:
0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo".

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	857	1	2	4	24	26	4	42	31	9
1	7	820	37	2	16	9	62	5	20	22
2	8	20	820	50	8	21	25	13	20	15
3	4	6	29	925	2	15	5	2	7	5
4	43	29	18	7	810	6	27	20	25	15
5	9	11	68	12	12	840	27	1	15	5
6	3	14	42	9	17	5	897	8	1	4
7	19	10	16	4	26	11	34	818	29	33
8	11	25	25	51	4	8	29	3	837	7
9	3	18	44	8	29	6	20	17	11	844

Parameter Count

Each input image in the Kuzushiji-MNIST dataset is 28 × 28 pixels = 784 input pixels.

In the hidden layer using **H hidden units**, we have 784 X H weights and H biases = (784+1) X H

In the output layer with 10 output classes, we have 10 X H weights and 10 biases = (H+1) X 10

Therefore, in total, for a fully connected feed forward neural network with H hidden units, we have:

(784+1) X H + (H+1) X 10 = **795 X H + 10 parameters**.

Therefore, the network with **160 hidden units** has 795 X 160 +10 = **127,210 parameters.**

While the one with 90 hidden units has 795 X 90 +10 = 71,560 parameters.

Observations

- The network achieved over 84% accuracy with hidden layer sizes ≥ 90.
- Increasing hidden units improves model capacity, up to a point where gains become marginal.
- Misclassifications decreased for characters with more distinct features, such as “tsu” and “ha”, but some confusion remained between visually similar pairs such as “su” vs “ma”, and “na” vs “re”.
- This model performs significantly better than the linear model, validating the importance of non-linear representation.

Step 3 – Convolutional Neural Network (NetConv)

Network Architecture and Training Setup

In this step, a Convolutional Neural Network (CNN) named NetConv was implemented using PyTorch. The model architecture includes two convolutional layers followed by a fully connected (linear) layer, with ReLU as the activation function throughout, and a final output layer using LogSoftmax. Max pooling was applied to reduce spatial dimensions and increase robustness. The network was trained using the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.01 and momentum of 0.9 for 10 epochs. Input images were normalized using transforms.Normalize((0.5,), (0.5,)).

Final Test Accuracy

The network achieved a final test accuracy of 93.19% after 10 training epochs.

```
Test set: Average loss: 0.3071, Accuracy: 9319/10000 (93%)
```

Confusion Matrix

Each row corresponds to the true label and each column to the predicted label. The labels correspond to the 10 Hiragana characters in the order:
0="o", 1="ki", 2="su", 3="tsu", 4="na", 5="ha", 6="ma", 7="ya", 8="re", 9="wo".

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	935	6	1	1	29	5	1	11	6	5
1	3	906	8	1	10	1	45	10	8	8
2	6	6	907	30	12	9	10	3	4	13
3	1	0	12	967	3	4	4	4	1	4
4	17	7	4	12	925	0	15	5	13	2
5	3	8	44	6	4	895	24	3	4	9
6	2	2	15	5	7	0	963	4	1	1
7	3	3	3	0	6	2	10	952	8	13
8	2	7	10	7	8	3	11	1	950	1
9	11	1	14	4	14	0	13	8	16	919

Parameter Count

The total number of independent parameters in the network was calculated by summing the weights and biases of all layers:

- **First Convolutional Layer (Conv1):**
  - **Input channels** = 1, **Output channels** = 32, **Kernel size** = 3×3
  - Each filter has: 1 × 3 × 3 = 9 weights
  - Total weights: 32 × 9 = **288**
  - Total biases: 32
  - **Total Conv1 parameters** = 288 + 32 = **320**

- **Second Convolutional Layer (Conv2):**
  - **Input channels** = 32, **Output channels** = 64, **Kernel size** = 3×3
  - Each filter: 32 × 3 × 3 = 288 weights
  - Total weights: 64 × 288 = **18,432**
  - Total biases: 64
  - **Total Conv2 parameters** = 18,432 + 64 = **18,496**

- **Fully Connected Layer (FC1):**
  - Input: 64 channels × 14 × 14 = 12,544
  - Output: 128
  - Weights: 12,544 × 128 = **1,605,632**
  - Biases: 128
  - **Total FC1 parameters** = 1,605,632 + 128 = **1,605,760**

- **Output Layer:**
    - Input: 128, Output: 10
    - Weights: 128 × 10 = 1,280
    - Biases: 10
    - **Total output layer parameters** = 1,280 + 10 = **1,290**

Total Parameters = 320 + 18,496 + 1,605,760 + 1, 290 = **1,625,866**

Step 4 – Comparative Discussion of Models

Relative Accuracy of the Three Models

The accuracy achieved by the three networks—NetLin, NetFull, and NetConv—demonstrates a clear correlation between model complexity and classification performance on the Kuzushiji-MNIST dataset.

- **NetLin (Linear Model):** Achieved approximately 79% test accuracy. This model treats the input image as a flat vector and performs a simple weighted sum followed by log softmax. The absence of any non-linearity limits its capacity to model complex relationships among pixel values.
- **NetFull (Fully Connected Neural Network):** Achieved around 84% accuracy. This model adds one hidden layer with a tanh activation, allowing the network to capture non-linear relationships. However, since it still operates on flattened pixel values and ignores spatial structure, it is limited compared to CNNs.
- **NetConv (Convolutional Neural Network):** Achieved a consistent 93%+ accuracy. The model leverages convolutional layers, ReLU activations, and max pooling to capture local spatial patterns and hierarchical features. Its superior performance is due to its ability to extract translation-invariant and hierarchical features from the input images.

Number of Independent Parameters in Each Model

The number of trainable parameters in a model is a key indicator of its complexity and capacity:

- **NetLin:** Linear Model
    - Input layer: 28×28 = 784 pixels
    - Output layer: 784 × 10 weights + 10 biases = **7,850** parameters
- **NetFull (with 160 hidden units):** Fully Connected Neural Network
    - Input to hidden layer: 784 × 160 = 125,440
    - Hidden layer biases: 160
    - Hidden to output layer: 160 × 10 = 1,600
    - Output layer biases: 10
    - Total = 125,440 + 160 + 1,600 + 10 = **127,210** parameters

- **NetConv:** Convolutional Neural Network
  - Conv1: 320
  - Conv2: 18,496
  - Fully Connected Layer: 1,605,760
  - Output Layer: 1290
  - Total = 320 + 18,496 + 1,605,760 + 1,290 = **1,625,866** parameters

Convolutional Neural Network (NetConv) can achieve higher accuracy with relatively fewer parameters, due to parameter sharing and local connectivity inherent in convolutional layers.

Confusion Matrix Analysis

Analyzing the confusion matrices of the three models reveals insights into common misclassifications:

NetLin (Linear Model):

Struggles to distinguish visually similar characters due to its limited capacity to capture patterns. Common confusions included:

- 'na' (**な**) mistaken as 'wo' (**を**）
- 'su' (**す**）mistaken as 'tsu' (**つ**)
- 're' (**れ**) mistaken as 'tsu' (**つ**)

Linear models fail to model spatial features, leading to overlap in classification boundaries.

NetFull (Fully Connected Neural Network):

Improved differentiation with non-linear activations. Still showed significant misclassifications in cases like:

- 'ha' (**は**) vs 'su' (**す**）
- 'ya' (**や**) vs 're' (**れ**)
- 'wo' (**を**）vs 'na' (**な**)

Flattened inputs restrict spatial understanding, causing ambiguity in similar shapes.

NetConv (Convolutional Neural Network):

Demonstrated the cleanest confusion matrix, with only a few minor confusions:

- 'na' (**な**) vs 'wo' (**を**）: Similar looping strokes may mislead filters.

- 'su' (す)vs 'tsu' (つ) : Slight stroke variations between these can overlap when handwritten.
- 'ha' (は) vs 'su' (す) : Shared horizontal and vertical line features.

These errors are expected given character similarity and handwriting variations.

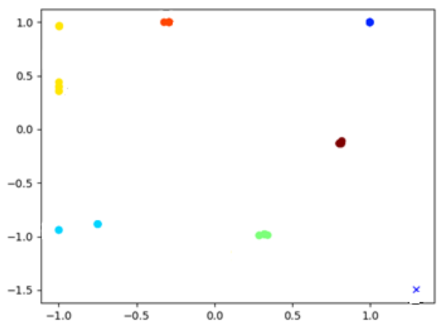
Overall, convolutional models are far more robust in preserving spatial features, explaining their superior clarity in the confusion matrix.

Part 4 - Hidden Unit Dynamics for Recurrent Networks

Step 1 – SRN on the Reber Grammar Prediction Task

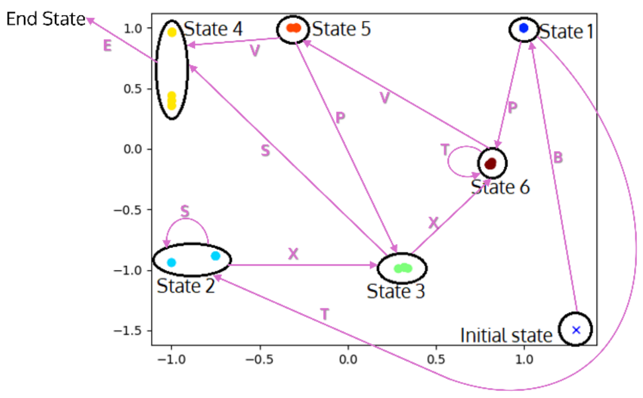
To examine hidden unit dynamics, we trained a **Simple Recurrent Network (SRN)** on the Reber Grammar prediction task. The SRN was configured with **7 input units**, **2 hidden units**, and **7 output units**. Training was performed for **50,000 epochs** using the Adam optimizer (learning rate = 0.001, initial weight std. dev. = 0.001). Model states were saved every 10,000 epochs in the net directory.

After training, hidden unit activations at epoch 50,000 were visualized. The scatter plot below used the *jet* colormap (blue → cyan → green → yellow → red) to represent states in the finite-state machine (FSM). The clusters in the 2D hidden space were **well-separated**, with each cluster corresponding to one of the FSM states.



We annotated the plot by:

- Drawing **ellipses** around each cluster to indicate its state.
- Labeling each cluster (*State 1* to *State 6*).
- Adding **arrows** between states, labeled with the transition symbols (B, T, S, X, P, V, E) based on the FSM diagram.



From the printed output probabilities, we observed:

- Hidden activations for each state were consistent across different sequence occurrences (low intra-state variance).
- Output probabilities aligned with the legal next-symbol constraints in the FSM. For example, from *State 1* only T and P had non-zero probabilities; from *State 4* only E was predicted.

Conclusion:
The SRN successfully learned a compact 2D internal representation of the FSM states, producing both accurate next-symbol predictions and separable hidden clusters corresponding to the Reber Grammar structure.

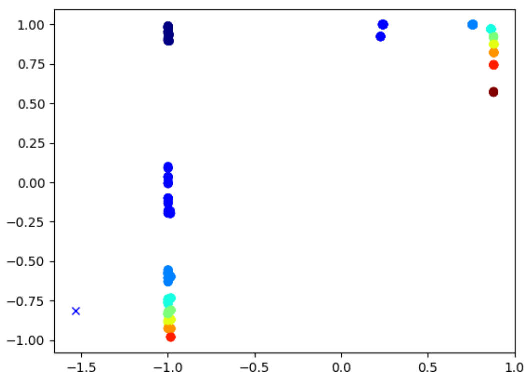
Step 2 – SRN on the $a^n b^n$ Language Prediction Task

For the $a^n b^n$ language, where each sequence contains a random number of **A** symbols followed by an equal number of **B** symbols, we trained an SRN with **2 input units**, **2 hidden units**, and **2 output units**. The network was trained for **100,000 epochs** using the Adam optimizer (learning rate = 0.001). Training was stopped once the error remained below 0.01, indicating the task was learned.

Key training observations:

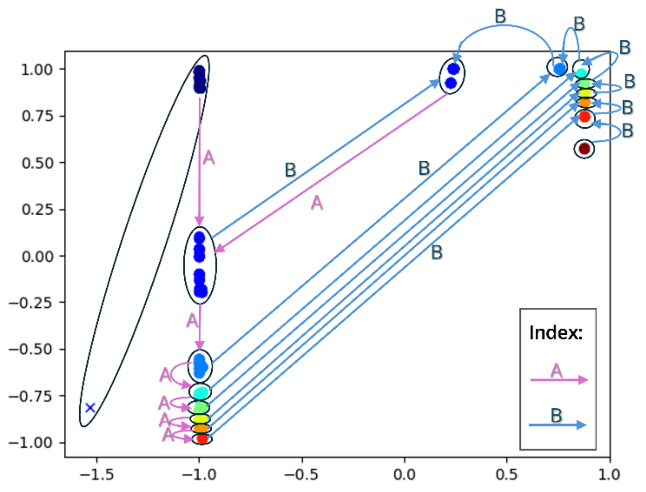
- The **first B** in a sequence and **all As after the first** are inherently **non-deterministic** and must be predicted probabilistically.
- All other positions are deterministic and should be predicted with high confidence.
- The trained network produced high-accuracy predictions for deterministic positions, such as the **last B** and the **first A** of a new sequence, while showing probabilistic outputs for ambiguous positions.
- The final log (epoch 9, error = 0.0048) confirmed the correct probability distribution for both deterministic and probabilistic transitions.

After training, hidden unit activations visualized using *jet* colormap showing **distinct clusters.**



The plot was then manually annotated as below:

- **Ellipses** mark each activation cluster.
- **Pink arrows (A)** indicate transitions for producing additional As or returning to A at the start of a new sequence.
- **Blue arrows (B)** indicate transitions for producing Bs after the A phase.



These "states" are used to count either the number of A's we have seen or the number of B's we are still expecting to see.

Conclusion:

The SRN successfully captured the a^nb^n counting rule, producing low error and hidden representations that reflect a counting process. The annotated state diagram confirms the network's learned internal structure, with deterministic transitions clearly separated and probabilistic transitions handled appropriately.

Step 3 – How SRN Solves a^nb^n Prediction Task (Interpreting Visualization)

Hidden-state geometry

With two hidden units the SRN forms a 1-D counting manifold embedded in 2-D. Points on the **left** vertical stack encode "how many **A**s seen so far"; points on the **right** vertical stack encode "how many Bs still to come." The **bridge** cluster between them mark the switch from the B phase to the A phase. Pink arrows are A transitions and blue arrows are B transitions.

During A phase (1 or more than 1 consecutive A's are occurring)

Each incoming A moves the hidden state down the **left stack** (pink arrows/ loops), stepping to a new cluster for count of As to be incremented by 1 (k → k+1). Because the grammar can either output another A or start Bs, the output layer (a linear readout of the hidden state) assigns **split probabilities** to A vs B for all positions after every A in the **left** stack — hence the non-deterministic or probabilistic predictions at the A phase.

Switch to B phase

When a B occurs, after an A phase of 1 or consecutive As, the state jumps across the bridge to the **right** stack (blue arrows). From now on, each B moves the hidden state **up** the right stack, effectively **counting** the remaining B's. These positions are deterministic, for B.

Why the last B is predicted correctly
The top-most cluster (light blue) on the right stack uniquely represents "**1 B remaining**." From that point, the learned readout maps all probability mass to **B** as the state jumps to the immediate blue cluster in the middle/ bridge, which corresponds to "**0 B remaining**." After emitting that **last B**, the transition moves back to the left vertical cluster, (with a pink arrow) deterministically emitting an A which starts the next A series. Thus the **last B is predicted correctly.**

Why the following A is predicted correctly
Once the counter reaches "0 B remaining," the state jumps to the immediate blue cluster in the middle/ bridge, which corresponds to "**0 B remaining**." The only valid next symbol in the training stream is the **first A of the next sequence**. That middle/ bridge cluster has a distinct signature; the output layer therefore assigns **near-certain probability to A**, producing the correct restart of the pattern (pink arrow back toward the A stack). Thus the **following A is predicted correctly.**

Summary
The SRN implements an implicit **down-counting clusters** (during As) and **up-counting clusters** (during Bs) via stable clusters in hidden space. Ambiguous positions (after any number of **A**s) yield probabilistic outputs, while unique clusters ("any number of **B**s left") yield **deterministic predictions**.

Step 4 – Training an SRN on $a^nb^nc^n$ Language Prediction Task

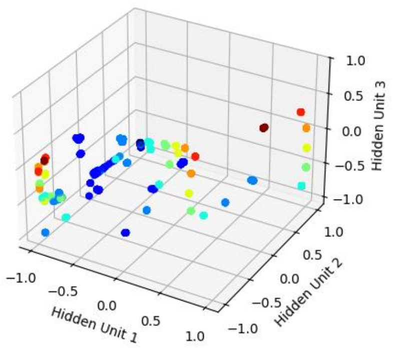
The $a^nb^nc^n$ language comprises sequences of symbols where a random number of As are followed by an equal number of Bs and then an equal number of Cs. To learn this three-phase counting task, we trained a **Simple Recurrent Network (SRN)** with:

- **3 input units**,
- **3 hidden units**, and
- **3 output units** (using the softmax distribution over A, B, and C).

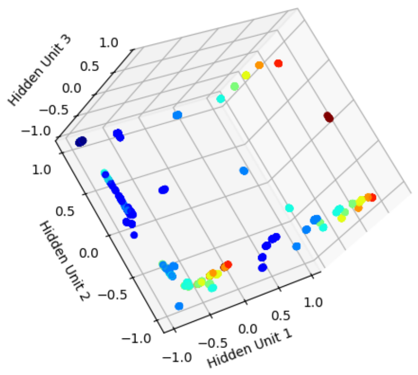
The model was trained for **200,000 epochs** using the Adam optimizer with a learning rate of 0.001 and weight initialization standard deviation of 0.001. The network converged to an **error range of approximately 0.01–0.02**, consistently predicting the sequence structure accurately.

Hidden Unit Dynamics and Visualization
After training, the hidden unit activations were visualized increasing the number of sequences to 50 (from default 10) to avoid sparsing or clustering. This generated a **3D scatter plot** of hidden states colored by their "state" index (count of progress through As, Bs, and Cs), rendered using

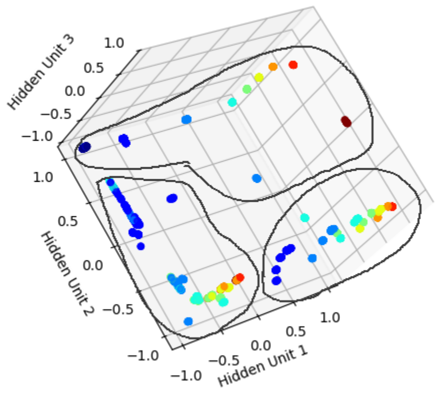
the *jet* colormap. The default view is shown below, but the separation of clusters was not immediately clear from this perspective.



To explore hidden state structure more effectively, we rotated the figure interactively using **elevation=45, azimuth=-160, roll=-45**. The resulting view (shown below) provides a clearer separation of clusters corresponding to each phase of the sequence (A, B, and C).



Manual inspection and annotation of the resulting plot below, revealed three **distinct, curved manifolds** corresponding to each symbol class:



- The **A phase**: Points lie along a smooth curve in one region of space, increasing consistently as the count of As increases.
- The **B phase**: Activations shift to a different arc-shaped manifold, where the count of remaining Bs is implicitly represented.
- The **C phase**: Final transitions land on a distinct region corresponding to decreasing C counts, with high confidence in the output probabilities for the C symbol.

Interpretation and Analysis

The SRN has successfully learned to:

- **Count up** through the number of As in the sequence,
- **Count down** through the number of Bs,
- And then again **count down** through the number of Cs,
- Before predicting the **next A** in the subsequent sequence with high confidence.

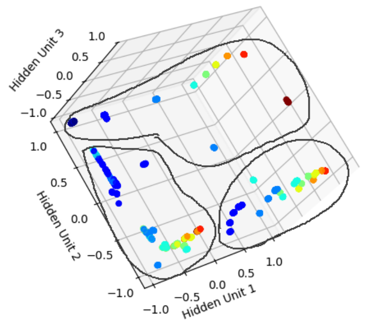
Each of the three hidden units contributes a dimension to a **compact 3D encoding** that distinguishes the phases clearly in geometric space. The curved progression through each class (rather than purely linear separation) suggests the network is implementing **nonlinear trajectories** in its hidden state space for different phases of the sequence. This behavior confirms the SRN's capacity to internalize a **three-phase stack-like grammar** within its recurrent architecture, using distributed hidden representations.

Conclusion

The network's success in learning the $a^n b^n c^n$ task demonstrates its ability to encode complex sequential dependencies using only three hidden units. The **clear segmentation of clusters** in hidden space for A, B, and C phases (visible in the annotated 3D plot) validates the network's capacity to model grammars with **hierarchical structure** through learned internal dynamics. This experiment also highlights the interpretability benefit of low-dimensional hidden states in recurrent models.

Step 5 – Interpreting How the SRN Solves the $a^nb^nc^n$ Prediction Task

The SRN learns the $a^nb^nc^n$ prediction task by organizing its **3-dimensional hidden state space** into **three distinct curved manifolds**, each corresponding to one phase of the sequence: As, Bs, and Cs.



From the annotated plot, we observe the following trajectory:

A Phase – Counting Up

- As the sequence starts with As, the hidden state moves smoothly along a **curved trajectory** (visible as the lower-right arc).
- Each additional A pushes the hidden activation further along this curve.
- Since the first few As are not fully deterministic (sequence length varies), the output probabilities gradually shift, but the SRN maintains a consistent internal count.

B Phase – Transition and Count Down

- When the first B appears, the hidden state transitions sharply to a **second distinct region** (lower-left arc).
- This region encodes a **countdown of remaining Bs**, and the network confidently predicts B at each step.
- As the state approaches the end of this arc, it becomes associated with the **last B**, which the SRN predicts with high certainty due to its distinct location in hidden space.

C Phase – Final Count Down

- After the last B, the hidden state transitions to a **third manifold** (top arc), encoding the count of remaining Cs.
- The SRN again enters a **countdown mode**, shifting through this region as it outputs each C.
- These transitions are well-separated, so the model confidently predicts C at each position.

Transition to Next Sequence – Predicting the Next A

- Once the final C is emitted, the hidden state loops back toward the start of the A arc.
- This **loop closure** enables the SRN to correctly predict the first A of the next sequence, as that location in hidden space is already learned to trigger an A.

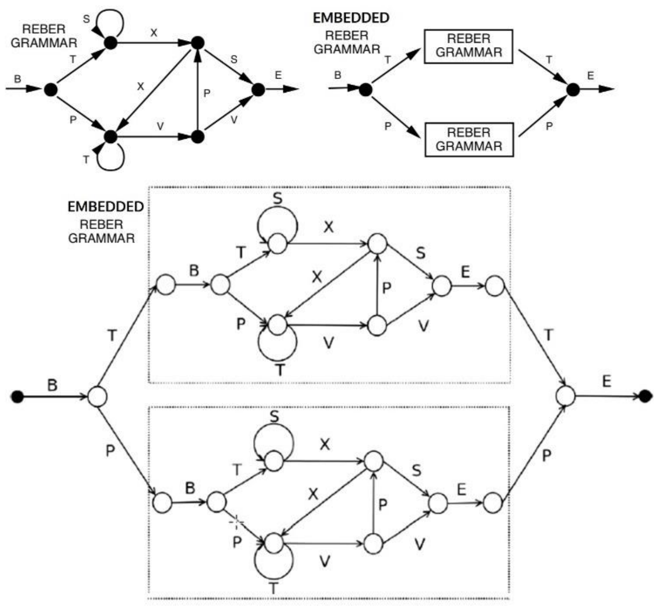
Key Insight:
The SRN implements a learned **three-phase finite-state-like control mechanism**:

- **Phase 1 (A):** Hidden state moves forward in one region.
- **Phase 2 (B):** State switches region and counts down Bs.
- **Phase 3 (C):** A final distinct arc handles C transitions.
- The **spatial layout** of hidden states allows the output layer to confidently associate each region with one symbol (A, B, or C), even handling **ambiguous early transitions** probabilistically.

Conclusion
The network solves the $a^n b^n c^n$ task by learning a **structured internal state space** with well-separated manifolds for each phase. The ability to **predict the last B**, all of the Cs, and the **next A** demonstrates successful encoding of hierarchical dependencies using compact, interpretable dynamics in hidden space.

Step 6 – Training an LSTM on the Embedded Reber Grammar (ERG)

To model the **Embedded Reber Grammar (ERG)**, which introduces long-range dependencies and nested patterns (e.g., BPBPVPSE or BTBTXSE), we trained a **Long Short-Term Memory (LSTM)** network using 4 hidden units. **LSTM architecture** is well-suited to capturing **long-range dependencies**, and 4 hidden units give enhanced performance.



How the LSTM Solves the ERG Task
The LSTM solves this task using its **internal memory (cell state)** to store the contextual identity of the input path (e.g., "T-track" or "P-track") and to selectively update or preserve information based on gate activations.

To analyze this, we **modified seq_models.py to return both:**

1. **Hidden states** (h_t), and
2. **Cell states** (c_t, sometimes called context units).

Modifications Made in Code to Analyze Context Units

To implement the above logic, inside LSTM_model class (in seq_models.py), the forward() method was modified to also return the **cell state sequence** along with the hidden states. The old commented out code (with the hidden states) and the current code with both with the hidden states and **cell states** are as follows:

```
#     def forward(self, x, init_states=None):
#         """Assumes x is of shape (batch, sequence, feature)"""
#         batch_size, seq_size, _ = x.size()
#         hidden_seq = []
#         if init_states is None:
#             h_t, c_t = (torch.zeros(batch_size,self.num_hid).to(x.device),
#                         torch.zeros(batch_size,self.num_hid).to(x.device))
#         else:
#             h_t, c_t = init_states
#         NH = self.num_hid
#         for t in range(seq_size):
#             x_t = x[:, t, :]
#             # batch the computations into a single matrix multiplication
#             gates = x_t @ self.W + h_t @ self.U + self.hid_bias
#             i_t, f_t, g_t, o_t = (
#                 torch.sigmoid(gates[:, :NH]),     # input gate
#                 torch.sigmoid(gates[:, NH:NH*2]), # forget gate
#                 torch.tanh(gates[:, NH*2:NH*3]),  # new values
#                 torch.sigmoid(gates[:, NH*3:]),   # output gate
#             )
#             c_t = f_t * c_t + i_t * g_t
#             h_t = o_t * torch.tanh(c_t)
#             hidden_seq.append(h_t.unsqueeze(0))
#         hidden_seq = torch.cat(hidden_seq, dim=0)
#         # reshape from (sequence, batch, feature)
#         #           to (batch, sequence, feature)
#         hidden_seq = hidden_seq.transpose(0,1).contiguous()
#         output = hidden_seq @ self.V + self.out_bias
#         return hidden_seq, output

    def forward(self, x, init_states=None):
        """
        x shape: (batch, sequence_length, input_dim)
        Returns:
            hidden_seq: all h_t (batch, seq_len, hidden_dim)
            output: logits (batch, seq_len, output_dim)
            cell_seq: all c_t (batch, seq_len, hidden_dim)
        """
        batch_size, seq_size, _ = x.size()
        hidden_seq = []
        cell_seq = []
        if init_states is None:
            h_t, c_t = self.init_hidden()
            h_t, c_t = h_t.to(x.device), c_t.to(x.device)
        else:
            h_t, c_t = init_states
        NH = self.num_hid
        for t in range(seq_size):
            x_t = x[:, t, :]  # (batch, input_dim)
            gates = x_t @ self.W + h_t @ self.U + self.hid_bias#(batch,4*hid)
            i_t = torch.sigmoid(gates[:, :NH])              # input gate
            f_t = torch.sigmoid(gates[:, NH:NH*2])          # forget gate
            g_t = torch.tanh(gates[:, NH*2:NH*3])            # cell candidate
            o_t = torch.sigmoid(gates[:, NH*3:])             # output gate
            c_t = f_t * c_t + i_t * g_t                    # updated cell state
            h_t = o_t * torch.tanh(c_t)                    # updated hidden state
            hidden_seq.append(h_t.unsqueeze(0))
            cell_seq.append(c_t.unsqueeze(0))
        # Convert to shape: (batch, seq_len, hidden_dim)
        hidden_seq = torch.cat(hidden_seq, dim=0).transpose(0,
                                                          1).contiguous()
        cell_seq = torch.cat(cell_seq, dim=0).transpose(0, 1).contiguous()
        output = hidden_seq @ self.V + self.out_bias
        return hidden_seq, output, cell_seq
```

In seq_plot.py, during evaluation loops for the LSTM, following statements were added to print out the context units as well as the hidden units:

```
for t in range(hidden_seq.shape[1]):
    print(f"t={t}")
    print(f"  hidden: {hidden_seq[0, t].detach().numpy()}")
    print(f"  cell:   {cell_seq[0, t].detach().numpy()}")
```

Training Outcome
After sufficient training (50k epochs), the LSTM converged to a low error rate (typically < 0.01), indicating that it had **learned to correctly predict** the valid transitions of the embedded Reber grammar. Unlike the simple Reber Grammar, the **embedded structure introduces dependencies between distant parts of the sequence**, such as remembering whether the sequence started with B-T or B-P and choosing the correct corresponding ending (E via T or P path).

LSTM Behavior in Solving ERG
After training an LSTM network with 4 hidden units on the Embedded Reber Grammar task, the hidden state and cell state trajectories reveal how the model solves this long-range dependency problem. The input sequence consists of symbols such as **B, T, S, X, P, V, E**, forming paths through the grammar that include nested transitions (e.g., branches that must match earlier paths in later stages).
To analyze the model's internal behaviour, we modified the LSTM_model's forward() method to return the sequence of cell states (c_t) in addition to the hidden states (h_t). This allowed us to inspect how the network encodes context over time.
From the printed hidden and cell states at each time step (t=0 to t=11), we observe the following:

- Early symbols (B, T) initialize the memory by setting up certain activations. The hidden and cell states remain relatively low in magnitude at first, suggesting the network is preparing to track the branching structure.
- As the network encounters transition points (e.g., from T to X or S to V), the cell state values increase and become polarized, suggesting that long-term memory is being stored for upcoming disambiguation (e.g., choosing the correct E).
- In the middle of the sequence (e.g., t=4 to t=7), the hidden state activations adapt to follow the correct nested path. The LSTM uses different internal gates to update or forget previous information depending on the branch it is traversing.
- At t=6–t=9, when the sequence must recall which branch was taken earlier (P or V), the hidden states diverge and influence the output to choose the correct path.
- At final transitions (e.g., symbol E at t=11), the model uses its preserved internal state to make a confident and accurate prediction. The predicted output probabilities (e.g., for symbol E: [0. 0. 0. 0. 0. 0. 1.]) closely match the true expected values.

This analysis highlights how LSTM's cell state (c_t) enables long-term storage of nested contextual information, crucial for grammar with hierarchical dependencies. Unlike simple RNNs, the LSTM doesn't overwrite useful information during traversal and is able to remember the early decision (branch taken) and reuse it later to make the correct output prediction.
Thus, by inspecting the LSTM's internal hidden and cell states across the time steps, we can verify that the model learns the structure of the Embedded Reber Grammar, successfully resolving long-range dependencies using its gating mechanisms and memory cells.

During Early Transitions:

- When the sequence starts with BT or BP, the LSTM **stores a latent trace** in its **cell state c_t** representing which path it is on.
- The **input gate** activates to allow this information in, and the **forget gate** keeps it preserved across time.

Middle of the Sequence:

- The LSTM processes intermediate transitions (e.g., X, T, V, S), updating the hidden state based on symbol transitions while maintaining the **path identity** in the **cell memory**.

Final Decision Point:

- The final predicted symbol (E) **depends entirely on the initial choice** (BT or BP).
- This long-term dependency is correctly handled because the **cell state has maintained that path identity**, allowing the LSTM to output the correct final prediction.

Output Probabilities:

- At the final timestep, the output probabilities for the next character (e.g., predicting E) match the valid grammar rules.
- Invalid transitions (e.g., predicting E when the path is not complete) are assigned **zero or near-zero probabilities**, as expected.

Conclusion

The LSTM is able to solve the Embedded Reber Grammar task by:

- Learning to **memorize initial path decisions** via its **cell state**, and
- Using its **gating mechanisms** to selectively preserve or update this memory across long sequences.
- The final predictions reflect a **grammatical awareness** that cannot be achieved with simple RNNs due to vanishing gradients.

Thus, the experiment highlights the **power of LSTM architecture** in handling structured sequence modeling tasks with **long-range dependencies** and **context-sensitive outputs**.