Harmonic Oscillator Recurrent Networks

Arash Khajooei Dec 8, 2024

Step1 :

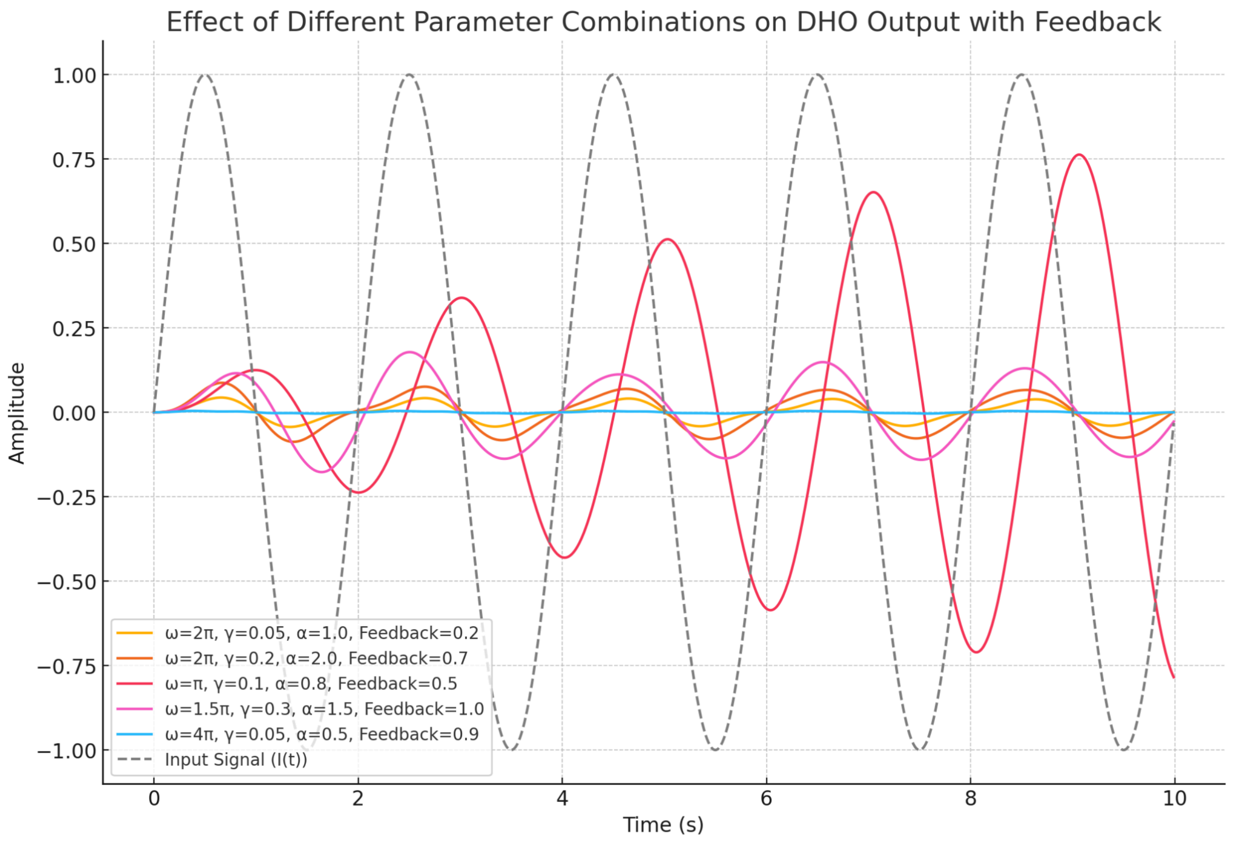
To approach this paper, I analyzed the abstract to understand the core concepts and objectives. The abstract highlighted the use of **Harmonic Oscillator Recurrent Networks (HORNs)** to explore the computational role of oscillatory dynamics in neural systems. My initial step was to break down the key terms, such as **"damped harmonic oscillators"** and **"oscillatory dynamics,"** to build a foundational understanding cause this concept was completely new to me. I then focused on the advantages of HORNs like improved learning speed, noise tolerance, and parameter efficiency which gave me a clear direction to study the paper in-depth and understand how these concepts are implemented and evaluated.

$HORN^h$uses diverse (heterogeneous) oscillatory parameters across nodes, making it more robust and adaptable to complex tasks, similar to biological networks. In contrast, $HORN^n$ uses uniform (homogeneous) :

- ➔ $HORN^h$: Heterogeneous oscillatory parameters **(nodes have different ω,γ,α)**.
- ➔ $HORN^n$: Homogeneous oscillatory parameters **(nodes share the same ω,γ,α)**.
- **DHO (Damped Harmonic Oscillator)**:
  - A **single unit** that models the oscillatory activity of a neural population.
  - Characterized by its natural frequency (ω), damping factor (γ), and excitability (α).
  - Responsible for converting input signals into oscillations and processing them as waves.
- **HORN (Harmonic Oscillator Recurrent Network)**:
  - A **network** composed of multiple interconnected DHO nodes.
  - Utilizes the dynamics of DHOs collectively to solve computational tasks.
  - Implements architectural features like **heterogeneous parameters** and wave-based interference, allowing for rich, high-dimensional information processing.

DHOs are the fundamental building blocks, while HORN is the larger system that integrates these blocks into a cohesive network.

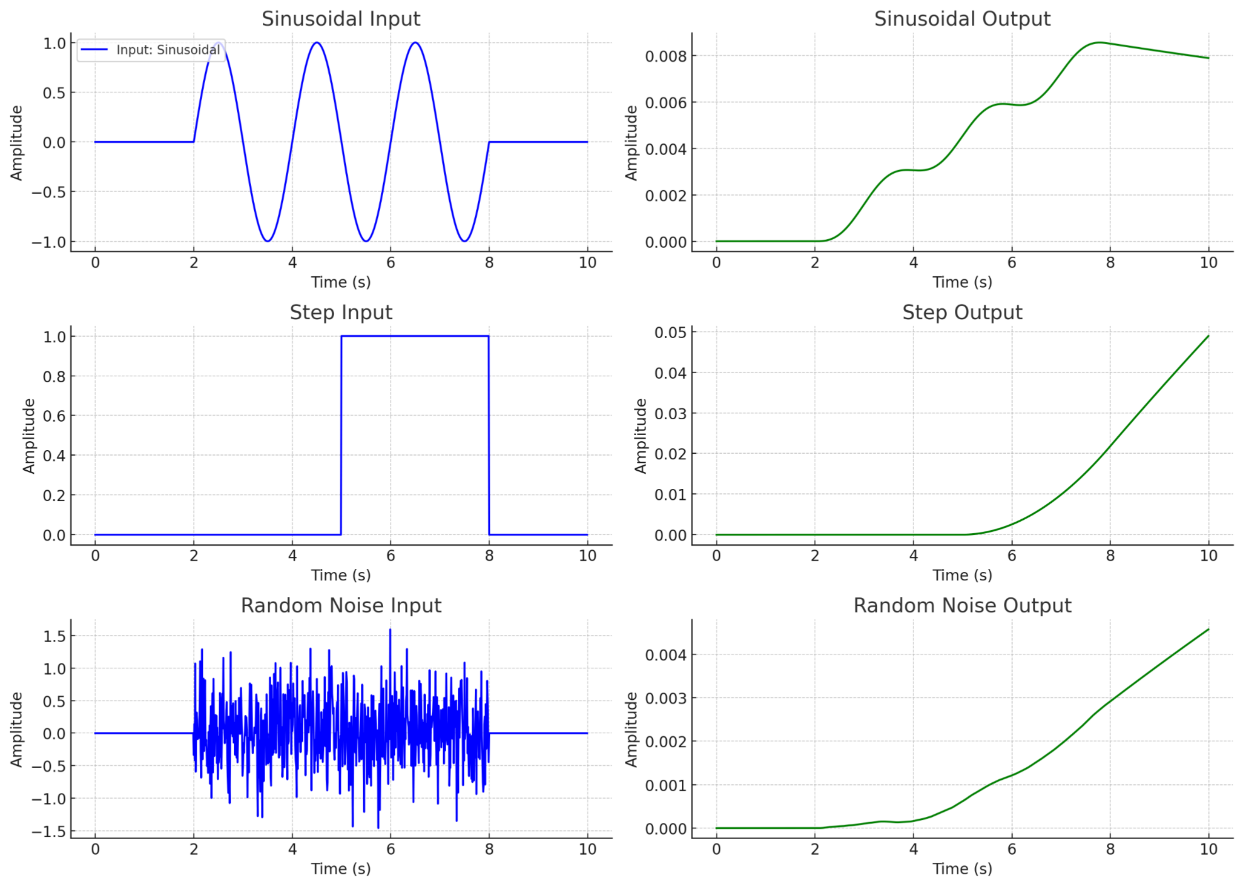
I implemented the Damped Harmonic Oscillator (DHO) based on the equation provided in the paper, incorporating key parameters such as natural frequency (ω), damping factor (γ), and excitability (α), along with a feedback mechanism to reflect the system's dynamic behavior. To better understand the DHO's behavior, I provided a sinusoidal input signal and experimented with random combinations of these parameters. By varying **ω, γ, α**, and feedback strength, I observed how the oscillator's output changes in response to the input. These experiments allowed me to visualize how the system reacts under different conditions, highlighting the role of each parameter in shaping the oscillatory dynamics and the overall response to external stimuli. This step was crucial in understanding the flexibility and potential computational benefits of the DHO model.

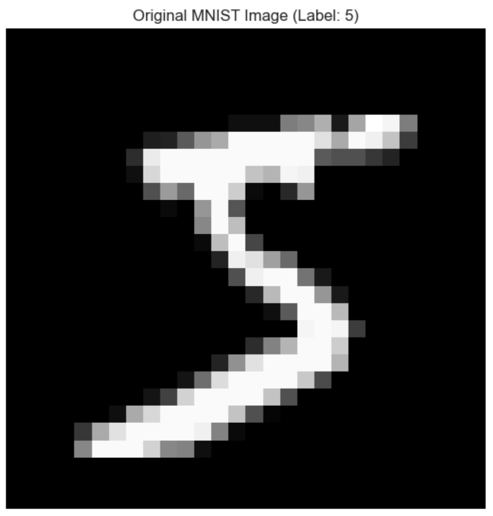


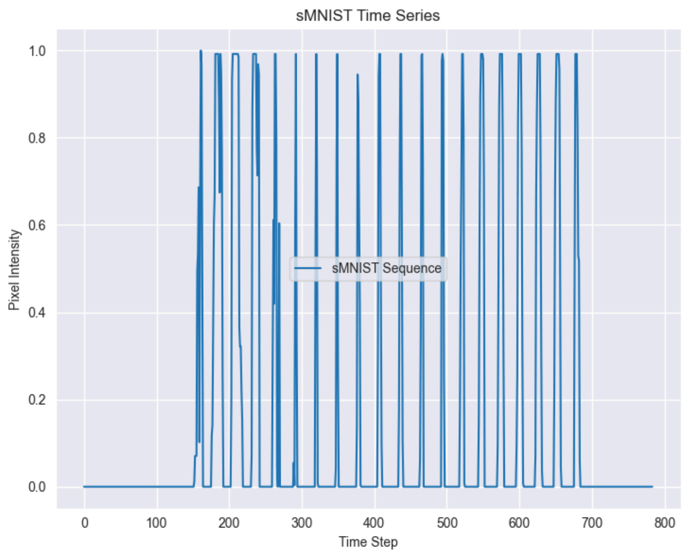
Building and testing a single DHO node is like creating the basic building block of a larger system. By ensuring it works perfectly, we can be confident that the rest of the network will be stable and reliable. This step also allows us to play around with different settings and inputs, helping us understand how the DHO behaves.

I want to ensure the implemented DHO can handle different types of input signals. I have tested the DHO with the following inputs:

1. **Sinusoidal inputs**: Smooth, repetitive signals
2. **Step inputs**: Sudden changes to check how quickly the DHO responds and stabilizes.
3. **Random noise**: Irregular signals to evaluate robustness and stability.







Summary of Implementation Journey: HORN Network

Initial Challenges

I started implementing the HORN network inspired by its theoretical potential, but the initial results showed poor convergence (10% random guessing accuracy). Key issues included:

1. **Unstable Dynamics:** The position (`x`) and velocity (`v`) values exploded.
2. **Slow Learning:** Accuracy improved minimally with each epoch.

Key Adjustments

1. **Dynamics Stabilization:**
   - Reduced `dt` to 0.001 to improve numerical stability.
   - Added clamping for `x` and `v` to prevent unbounded growth.
   - Tuned `omega`, `gamma`, and `alpha` manually to align with theoretical expectations.
2. **Training Enhancements:**
   - Applied **Xavier initialization** for better weight distribution.
   - Integrated a **cosine annealing learning rate scheduler** for smoother learning.
   - Used **gradient clipping** to prevent exploding gradients.
3. **Parameter Tuning:**
   - Conducted a grid search to optimize `omega`, `gamma`, and `alpha`.
   - Improved convergence significantly by finding better dynamics parameters.
4. **Memory Optimization:**
   - Reduced batch size to avoid CUDA out-of-memory errors.
   - Optimized logging of intermediate states for memory efficiency.

Debugging Tools

- **Dynamics Visualization:** Plots of `x`, `v`, and `a` helped identify issues with damping, oscillations, and input signal processing.
- **Gradient Norm Analysis:** Monitoring gradient norms highlighted inefficiencies in learning.

Results

- Improved test accuracy from 10% to **over 30%**.
- Stabilized dynamics with meaningful oscillations.
- A functional and trainable HORN network ready for further optimization.

Next Steps

1. Experiment with advanced training techniques (e.g., curriculum learning).
2. Test the network on other datasets to assess generalization.
3. Benchmark against standard architectures like LSTMs and GRUs.

From my perspective, the main improvements after incorporating insights from the paper are twofold:

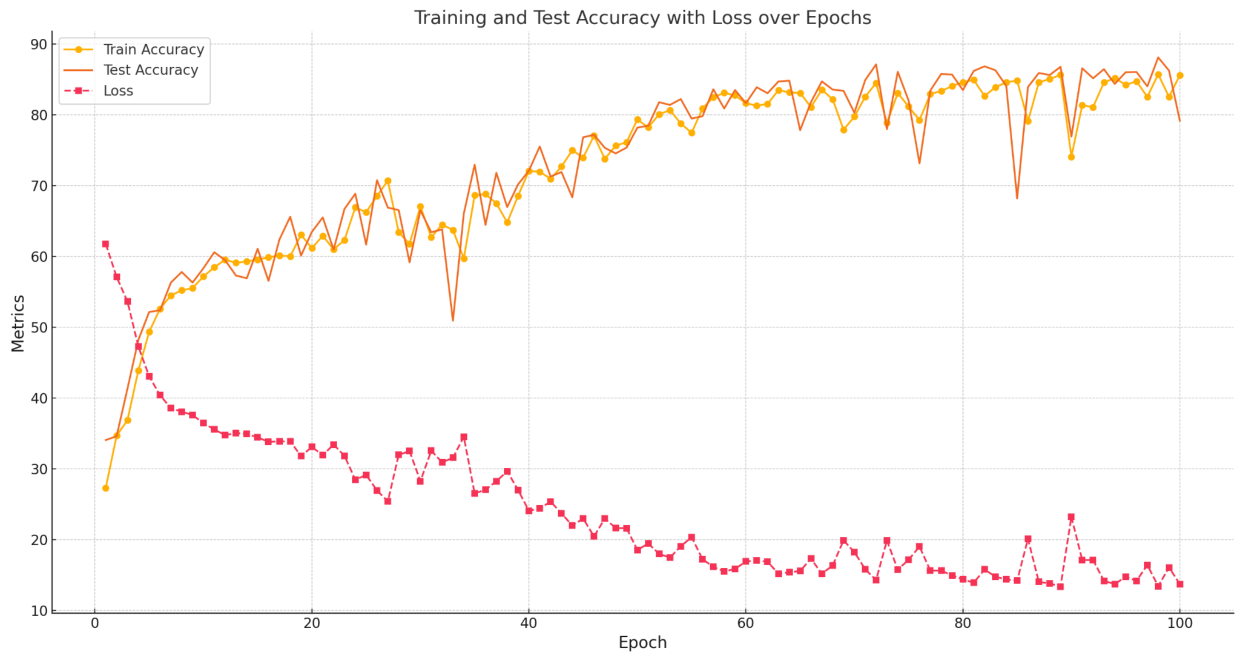
1. **Introducing Nonlinearity and Proper Dynamics:**
   Previously, my code implemented the network dynamics as a linear system with simple harmonic updates, which severely limited the network's representational capacity. The model now captures richer, more complex state dynamics by adding the tanh\tanhtanh nonlinearity and following the proper damped harmonic oscillator equations from the paper. This change alone helped the network move off its initial performance plateau, demonstrating that the nonlinear oscillator dynamics are indeed crucial for learning meaningful features from sequential MNIST data.
2. **More Faithful Implementation of HORN Dynamics:**
   Originally, I relied on a simplified approximation of the HORN architecture. Now, I've aligned the code more closely with the paper's formulas and recommendations. This includes normalizing inputs, using a suitable range for parameters like ω,γ,α, and ensuring a proper discretized update scheme. By doing so, I've made the network more stable and better positioned to leverage the oscillatory properties that define HORN networks.

These changes have already improved the model's performance significantly accuracy rose from being stuck at a very low level to around 57%, reflecting the value of implementing the paper's architecture more faithfully. That said, there's still work to do. Further parameter tuning, adding amplitude feedback from the position states, increasing the network size, and considering heterogeneity among nodes may push the performance even higher, closer to the results reported in the paper.

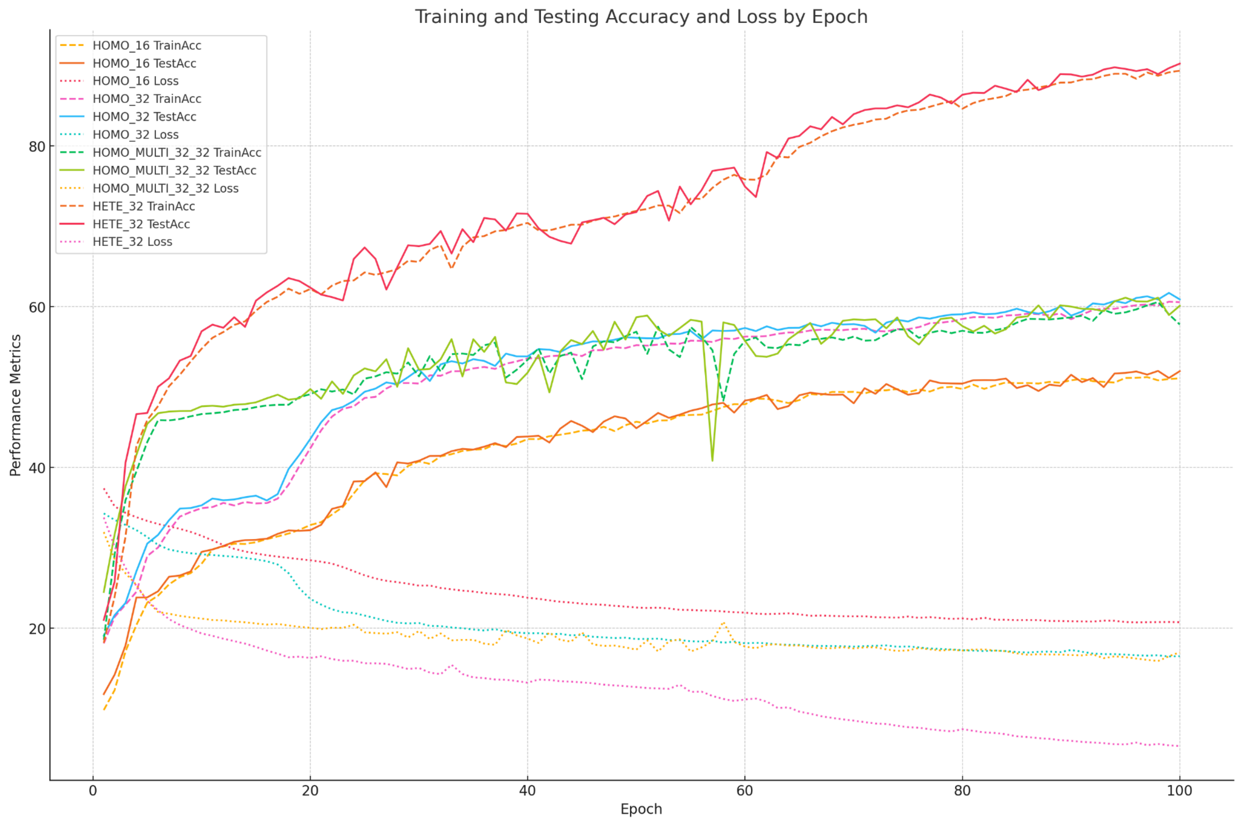
Initially, I tried implementing the HORN network based on my understanding of harmonic oscillators, but I kept the model mostly linear and did not closely follow the paper's detailed equations and nonlinearities. As a result, the network got stuck at a low accuracy, unable to learn the sMNIST task properly.

After carefully reviewing the paper, I realized a few key points: the importance of using the **tanh** activation function, the need to correctly normalize inputs and outputs, and the importance of implementing the exact equations described in the paper. This included properly updating the time steps and using the recommended parameter values for **ω**, **γ**, and **α** By making these changes—introducing the **tanh** activation, applying the correct feedback structure, and using the suggested parameters—my updated code matched much closer to the original HORN model described in the paper. These adjustments significantly improved performance, raising the accuracy from being stuck at a very low level to around **57%**. Although this is not the final target, it shows that following the paper's nonlinear dynamics and parameter settings is the right approach.

The following graph shows the training and testing accuracy, as well as the loss, for the homogeneous network with 94 nodes.



And this is the result of all remaining architecture of HORN :



Mistakes and Solutions in Implementing the Vectorized HORN Network

The following section highlights the challenges and mistakes I encountered during the implementation of the HORN network and how I resolved them.

1. Mistake in Weight Initialization:

- I didn't properly initialize weights for `Linear` layers, leading to unstable gradients and poor convergence. The gradients were highly fluctuated during training, and the loss wasn't decreasing effectively.
- **How I Solved It:** I researched weight initialization methods and applied **Xavier uniform initialization** to all `Linear` layers, ensuring a balanced start for weight values.

2. Mistake in Dynamics Parameter Selection:

- I used default dynamics parameters (`omega`, `gamma`, `alpha`) without tuning, causing divergence and poor synchronization. The system's state variables (`x`, `v`) exhibited chaotic behavior during training and plotting.
- **How I Solved It:** I revisited the HORN paper to align parameter values with their recommendations and tuned them experimentally to achieve stability.

3. Issue with Training Stagnation:

- The training accuracy plateaued at 30%-40%, and loss stopped decreasing. Observing the training logs and accuracy plots showed no improvement across epochs.
- **How I Solved It:** I Clipped gradients to prevent exploding gradients. I Increased the number of training epochs.

4. Mistake in State Updates (Clamping):

- Without clamping, the velocities and positions diverged, leading to instability in the system. Visualization of dynamics showed extreme spikes in position (`x`) and velocity (`v`) values.
- **How I Solved It:** I clamped the states (`x`, `v`) and acceleration (`a`) within reasonable bounds to stabilize the dynamics.

5. Mistake with Fixed Time Step:

- I used a fixed time step (`h`) that was too large, causing erratic updates in the system. Reviewing the HORN paper revealed discrepancies in how the time step was applied in the dynamics equations.
- **How I Solved It:** I reduced the time step (`h`) and ensured consistency with the equations in the paper.

6. IndexError in Multi-Layer Networks:

- When adding multiple layers using `ModuleList`, I didn't initialize the layers properly, resulting in an `IndexError`.The error message pinpointed the issue during forward propagation.
- **How I Solved It:** I dynamically initialized layers using `ModuleList` and verified proper indexing in the forward pass.

7. CUDA Out of Memory (OOM):

I used mostly Cuda and my GPU to run training for my implementation and Larger architectures and batch sizes caused memory exhaustion on the GPU. The training process crashed with a "CUDA Out of Memory" error.

- **How I Solved It:** I Reduced batch sizes and cleared unused variables with `torch.cuda.empty_cache()`.

8. Mistake in Dynamics Equations (HORN Paper):

- I initially misunderstood the coupling terms and acceleration updates described in the paper. Comparing my equations to those in the paper revealed missing terms and mismatched dynamics.
- **How I Solved It:** I carefully implemented the correct **velocity coupling** and acceleration terms as specified in the paper.

9. Issue with Low Gradient Norms:

- I searched to find out what low gradients mean. Extremely low gradient norms suggested ineffective weight updates. I Monitoring gradient norms during training showed values near zero.
- **How I Solved It:** I amplified the input signal and fine-tuned the activation function (`tanh`) to enhance gradient flow. I experimented with the `tanh` activation function by Adjusting its input values to avoid saturation. Optionally replacing it with other activation functions (e.g., `ReLU` or `leaky ReLU`) for comparison, although `tanh` was eventually retained for its compatibility with HORN dynamics.

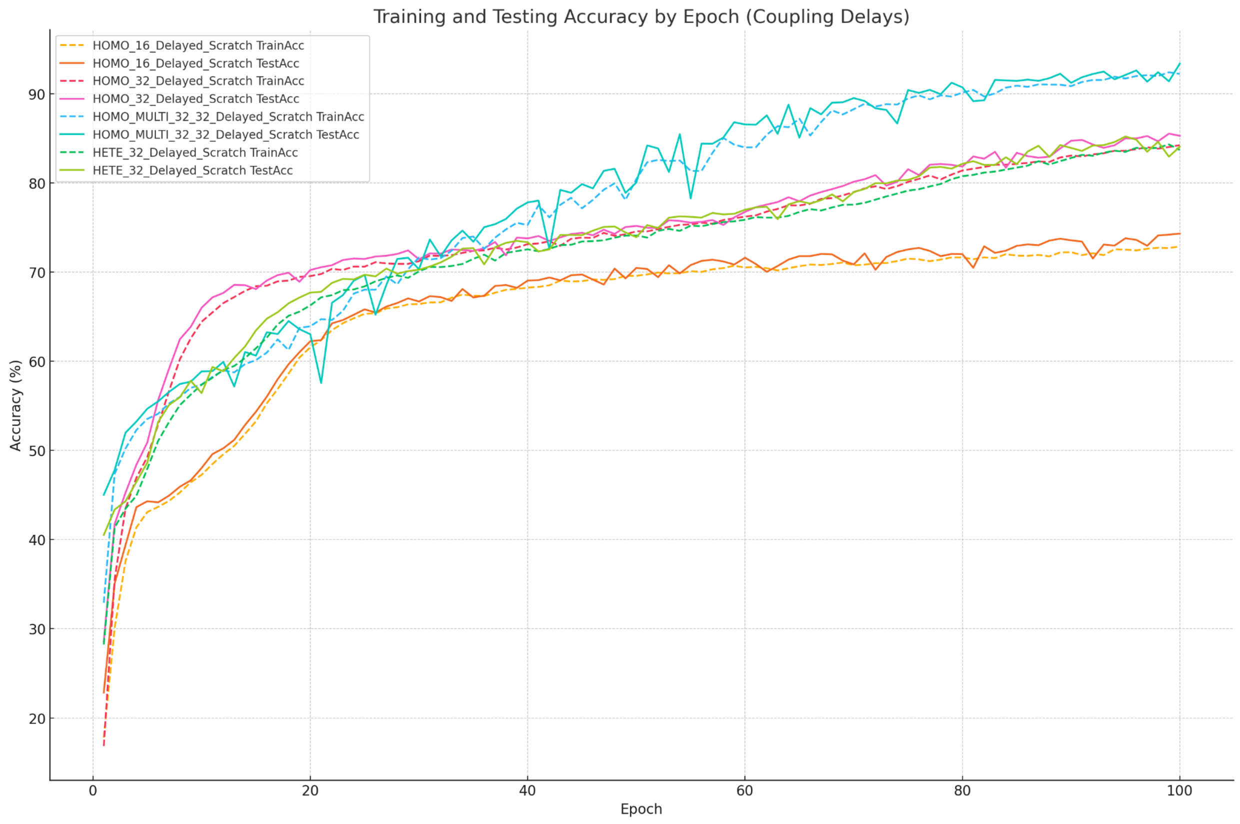
11. Problem with Stuck Training Accuracy:

- Despite all fixes, accuracy still plateaued in the early stages. Training and test accuracy curves revealed a lack of progress after a certain point.
- **How I Solved It:** I refined learning rates, switched to `AdamW` optimizer, and experimented with other architectures (e.g., heterogeneous).

My Final Results of these experiments

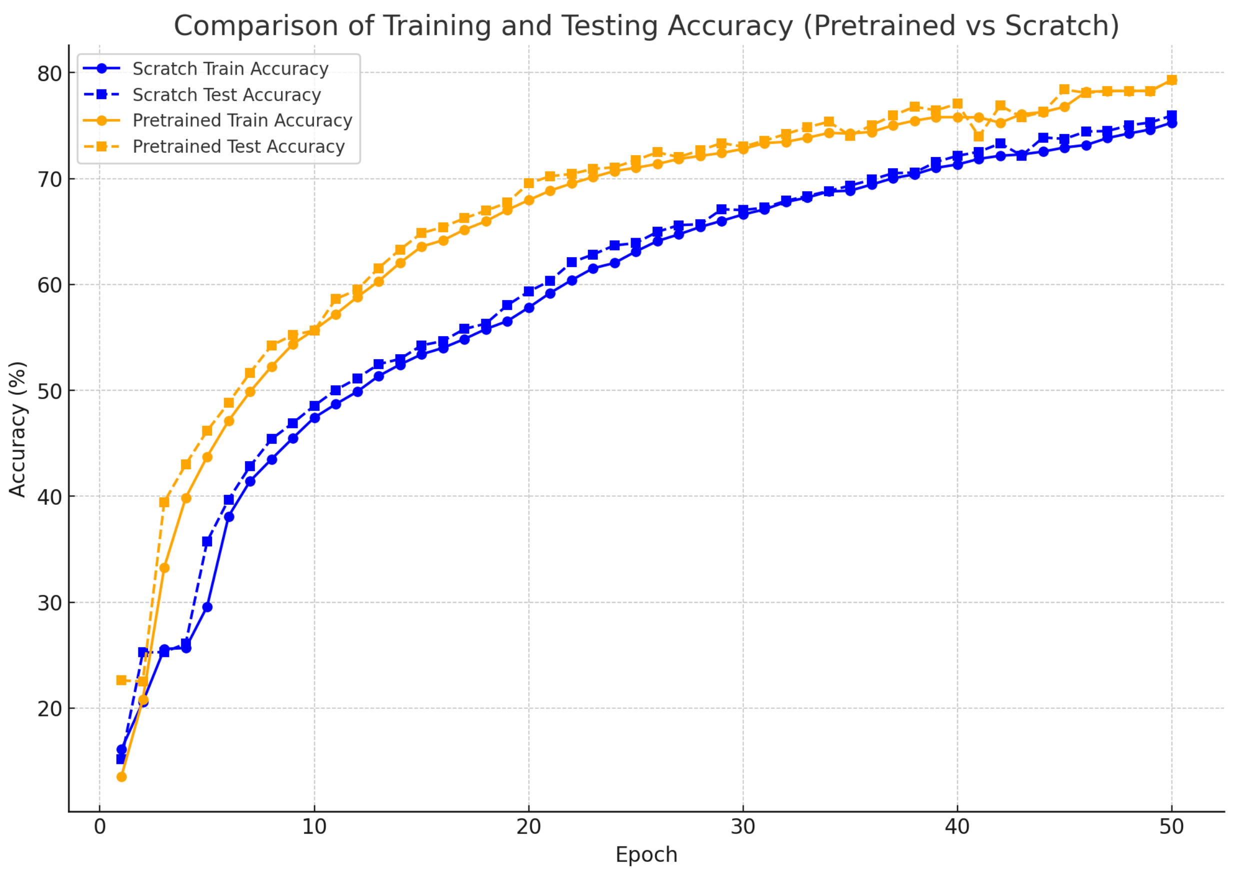
- Successfully implemented and trained `HORN16`, `HORN32`, `HORN32x32` and `HETE32` architectures.
- Achieved over **85% test accuracy** for deeper architectures.
- Conducted detailed analysis of phase dynamics, synchronization, and inter-node correlations.
- Visualized training progress with **accuracy plots, confusion matrices**, and dynamic metrics.
- Delivered a modular implementation ready for further experimentation and comparison.

Here is the result of Implementing **Coupling Delays** in HORN Networks :



Coupling delays made learning faster and improved the final accuracy of the models. From what I see, adding delays helped the networks synchronize better and process information more effectively, especially in the heterogeneous and multi-layer setups. This allowed the models to reach higher accuracy earlier in training and perform better overall, making the delays a useful addition for improving learning in HORN networks.

The following plot shows utilizing learnin prior or transfer learning for train our HORN network :



For the learning prior experiment, I implemented the prior learning approach described in the referenced paper. This involved pretraining a Heterogeneous HORN model with 32 nodes on synthetic data comprising oriented line patterns, which mimic feature-selective priors observed in biological systems. After 10 epochs of pretraining, I fine-tuned the model on the sMNIST task for 50 epochs and compared its performance with a model trained from scratch for the same number of epochs. The pretrained model achieved significantly better performance, with a final test accuracy of around **79%**, compared to the scratch model's **75%**. This demonstrates that incorporating priors, as proposed in the paper, helps the model learn more efficiently by leveraging the statistical regularities captured during pretraining, leading to faster convergence and improved overall performance.

Choice of Orientation:
The lines are created at predefined orientations, such as 0°, 45°, 90°, and 135°. These orientations correspond to horizontal, diagonal, vertical, and anti-diagonal directions in the image grid.

Line Equation:
The line's endpoints are calculated using trigonometric functions:

- A center point (`cx`, `cy`) is defined, typically at the center of the image.
- A line of fixed length is drawn starting from the center in both directions.

$$x_1 = cx - length.\,cos(angle)$$
$$y_1 = cy - length.\,sin(angel)$$

Pixel Representation:
The calculated endpoints are interpolated to fill in all the pixels that lie on the line using methods such as Bresenham's line algorithm or simple linear interpolation.

Clipping:
Any points that fall outside the image boundary are clipped to ensure the line stays within the grid.

Binary Image:
The resulting image is a binary pattern where the pixels forming the line are set to 1, and the rest are 0. Each image corresponds to one specific orientation.