Comparing the Hebbian Rule and Predictive Coding Results

Jan 10, 2025

Arash Khajooei

Explanation of the Architecture for PC Layer

The **Predictive Coding (PC) Layer** is designed to model neural activity by minimizing a predictive error.

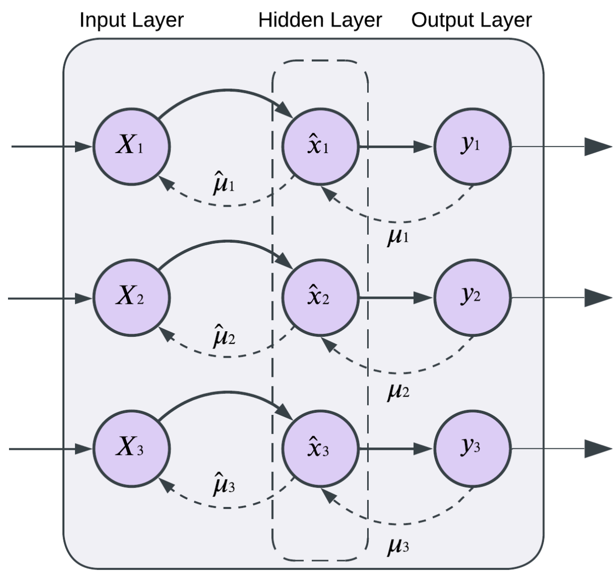
1. **Core Components:**
   - **Latent Variable (*x*):** Represents the hidden state of the layer.
   - **Prediction (*μ*):** Represents the input prediction from a higher or lower layer.
   - **Error Calculation** $(\mu - x)^2$**:** Measures how far off the latent variable is from the prediction.
2. **Functionality:**
   - **Energy Minimization:** The layer computes an energy value based on the error, which drives the latent variable ***x*** to adapt and reduce the discrepancy with ***μ***.
   - **Training Mode:** During training, the layer updates the latent variable ***x*** iteratively using the energy function.
3. **Structure in Code:**
   - **Forward Pass:** Computes the energy and updates the latent variable ***x***.
   - **Backward Pass:** Allows gradients to flow for updating other parts of the network.

Hebbian Mode in Code

When operating in **Hebbian mode**, the layer performs weight updates based on **Hebbian learning principles**. Here's a simplified explanation:

- **What Hebbian Learning Does:**
  - *"Neurons that fire together, wire together."*
  - The layer strengthens the connections between input (***x***) and output (**y**) neurons based on their correlation.
- **Code Mechanism:**
  - **Weight Update Rule:**
    - $\Delta W_{ij} = \mathrm{lr\_hebb} \cdot (\mathrm{post}_j \cdot \mathrm{pre}_i)$ where:
      - (W_{ij}): Weight between input i and output j.
      - (post_j): Output neuron activation.
      - (pre_i): Input neuron activation.

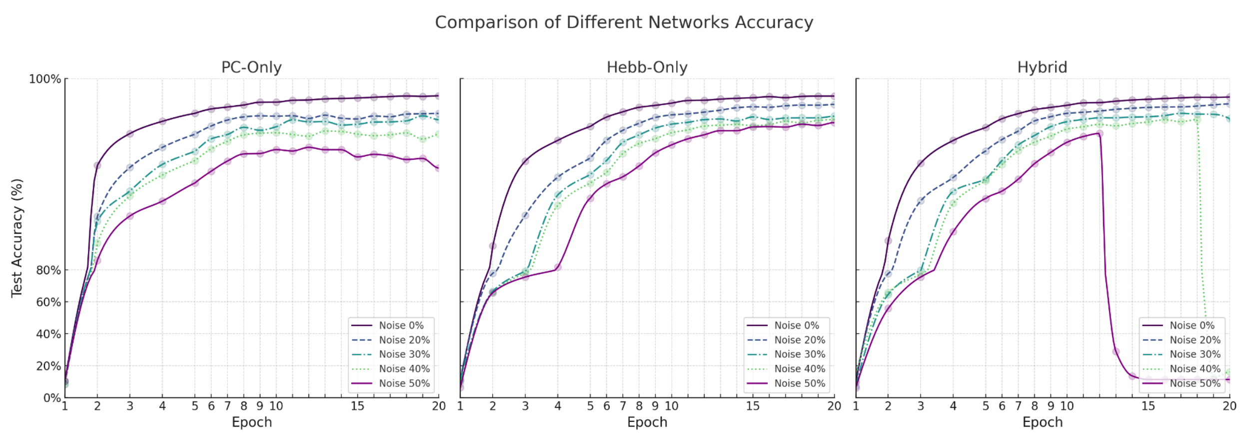
- $(\mathrm{lr\_hebb})$: Hebbian learning rate.
    - Regularization is added to prevent weights from growing excessively or becoming unstable.
- **Hebbian Code Process:**
    - The layer records the input **(pre_syn)** and output **(post_syn)** activations.
    - After the forward pass, it updates weights using the correlation of these activations.
    - Regularization clips weights or applies decay for stability.



Explanation of Components in the Diagram

1. **Input Layer (X_1, X_2, X_3):**
    - These represent the raw input signals or features entering the PC network.
    - Each input (X_i) corresponds to a specific neuron or feature in the layer.
2. **Hidden Layer ($\hat{x}_1, \hat{x}_2, \hat{x}_3$):**
    - These are **latent states** $(\hat{x}_i)$ in the PC layer.
    - Each latent state is optimized iteratively to minimize the **energy** (prediction error) between the predicted signal (μ_i) and the latent state $(\hat{x}_i)$.
3. **Output Layer (y_1, y_2, y_3):**
    - Represents the final outputs of the layer, which are passed to subsequent layers or used for comparison with a target signal.
    - y_i is computed based on the latent states $(\hat{x}_i)$ after error minimization.

4. **Predictions (μ_1, μ_2, μ_3):**
   - These are the predicted signals, generated based on the outputs of higher or lower layers.
   - The PC layer minimizes the error between μ_i (predicted) and $\hat{x}_i$ (latent state).
5. **Feedback and Error Minimization:**
   - **Dashed Arrows ($\mu \to \hat{x}$):** Represent the flow of predictions (μ_i) into the latent states ($\hat{x}_i$).
   - **Solid Arrows ($X \to \mu$ and $\hat{x} \to y$):** Represent the forward pass, where raw input signals (X_i) or updated latent states are passed along to generate predictions or outputs.



Analysis of Each Architecture Result on Noisy MNIST:

1. Speed of Convergence:

- **PC-Only**:
  - This architecture converges quickly to a high accuracy and has the **fastest convergence**, particularly for lower noise levels (e.g., 0% and 20% noise). By epoch 5, substantial performance improvements have already been made, especially under low-noise conditions.
  - **Strength**: Faster convergence compared to others in low-noise scenarios.
- **Hebb-Only**:
  - This architecture converges more slowly than PC-Only but shows steady improvements over time, especially in higher noise levels. It achieves competitive performance compared to PC-only architecture.
  - **Weakness**: Slower initial learning, but stable progression.
- **Hybrid**:

- The hybrid model shows a good speed to reach acceptable accuracy however in high noise percentages like 40% and 50% it collapses maybe some consideration is needed which I should take in my implementation.

2. Accuracy Across Noise Levels:

- **Low Noise (0-20%)**:
  - **PC-Only** performs the best in low-noise settings. It achieves slightly higher accuracy than other architectures, especially in the 0% and 20% noise scenarios.
  - **Hybrid** also performs well but doesn't outperform PC-Only in these conditions.
  - **Hebb-Only** lags in accuracy under low noise.
- **Medium Noise (30-40%)**:
  - **Hybrid** architecture starts to dominate as the noise level increases. Its combination of Hebbian updates and predictive coding enables it to maintain high accuracy where PC-Only and Hebb-Only begin to degrade.
  - **PC-Only** performs reasonably but shows a noticeable drop compared to Hybrid.
  - **Hebb-Only** starts closing the gap with PC-Only but still lags.
- **High Noise (50%)**:
  - **Hybrid** architecture collapsed under high noise conditions but in other scenarios, it performed well.
  - **Hebb-Only** performs better than PC-Only at high noise levels, likely due to its stability and local learning mechanisms.

Summary:

- **Best Accuracy at Low Noise**: **PC-Only**, with slightly better performance for 0-20% noise.
- **Best Accuracy at High Noise**: **Hebb-Only**, demonstrating robustness and adaptability.
- **Hebb-Only** is a slower learner but is a solid choice for handling high-noise scenarios with steady performance improvements over time.