DA 527 final project on Nanowire Neural Networks for time-series processing

Soumya Savarn [1] **Ishan Chandra Gupta** [1] **Saptarshi Mukherjee** [1]

Abstract

We present an implementation and empirical study of a computational framework inspired by the physics of memristive nanowire networks, formulated within the Recurrent Neural Network (RNN) paradigm for time-series processing. Building on the Nanowire Neural Network architecture proposed by the original authors, we coded the full set of nanowire-based state-update equations and evaluated both the NW-ESN and NW-RNN variants. Our experiments replicate the authors' setup across the same regression and classification datasets, allowing us to assess the reproducibility and behaviour of the models in practice. The results obtained by our implementation are consistent with the general trends reported in the paper, reinforcing the scalability, robustness, and potential of nanowire-inspired architectures for neuromorphic time-series learning.

1. Introduction

1.1. Motivation

- Nanowire networks naturally act like recurrent systems because their memristive junctions store and update state over time.
- Neuromorphic computing benefits from physical dynamics, and nanowires physically implement RNN-like nonlinear state transitions.
- They form high-dimensional nonlinear reservoirs, making them ideal for fast, low-training-cost sequence processing (RC / ESN).
- Hybrid NW-ESN/NW-RNN models outperform classical RNNs, giving scalable, efficient, and high-accuracy time-series processing.

[*]Equal contribution [1]Indian Institute of Techology, Guwahati. Correspondence to: Soumya Savarn <s.savarn@iitg.ac.in>, Ishan Chandra Gupta <g.ishan@iitg.ac.in>, Saptarshi Mukherjee <m.saptarshi@iitg.ac.in>.

Accepted at the 1st Machine Learning for Life and Material Sciences Workshop at ICML 2024.

1.2. Nanowire Memristor Physics

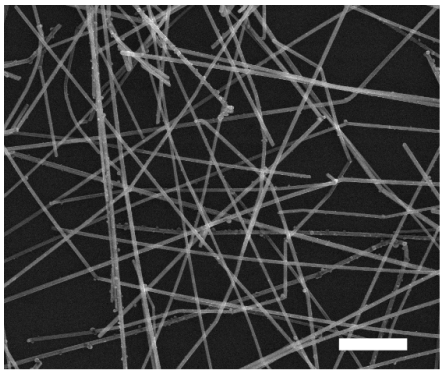

Figure 1. Image of a nanoscale self-assembled Ag nanowire network obtained through scanning electron microscopy (scale bar, 2 micrometer). This represents a memristive network, since each junction between nanowires is a memristive element and the current flow relies on the history of applied voltage and current.

- Nanowire networks are meshes of tiny metallic wires whose junctions behave like synapses.
- Their structure closely resembles brain connectivity, enabling learning and memory-like behavior.
- Each junction acts as a memristive element whose conductance changes with past electrical activity.
- These dynamics mirror ion-flow mechanisms in biological neurons, similar to the Hodgkin–Huxley model.
- As a result, nanowire networks naturally function as brain-inspired computational substrates.

1.3. Memristive Dynamics Equation

The state equation for the synaptic properties of the NW under electrical stimulation is expressed as the following potentiation–depression rate balance equation Miranda et al. [2020].

$$\frac{dg_{ij}}{dt} = \kappa_{P,ij}(V_{ij}) \cdot (1 - g_{ij}) - \kappa_{D,ij}(V_{ij}) \cdot g_{ij} \quad (1)$$

$$\kappa_{P,ij}(V_{ij}) = \kappa_{P0} \exp(+\eta_P V_{ij}) \quad (2)$$

$$\kappa_{D,ij}(V_{ij}) = \kappa_{D0} \exp(-\eta_D V_{ij}) \tag{3}$$

The authors develop Nanowire Neural Networks for time-series processing where the physical memory state g becomes the nanowire network state. To achieve this, they rewrite Eq. 1, 2 and 3 as:

$$\frac{dh}{dt} = K_p(x) \cdot (1 - h) - K_d(x) \cdot h \tag{4}$$

$$K_p = K_{p0} \cdot e^{\eta_p x}, \qquad K_d = K_{d0} \cdot e^{-\eta_d x} \tag{5}$$

$$h(t+1) = h(t) + \Delta t \left[K_p(x(t+1)) - (K_p(x(t+1)) + K_d(x(t+1)))\, h(t)\right] \tag{6}$$

Here,

- $h \in [0, 1]$ is the internal memory state of a nanowire neuron.
- K_p increases conductance(potentiation)
- K_d decreases conductance(depression)

1.4. Nanowire Neural Network Architecture

Equation 6 is then used to construct the fully connected Nanowire Neural Network (Fig.2).

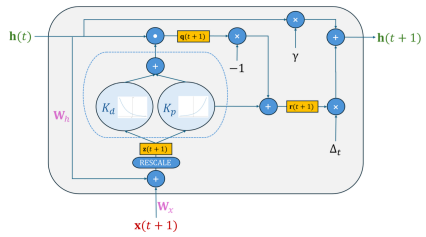

Figure 2. Nanowire Network computational graph where $h(t)$ and $h(t+1)$ (green) denote the current and next hidden state vectors, and $x(t+1)$ (red) denotes the external input vector. W_h is the recurrent weight matrix, while W_x (magenta) is the input weight matrix. Δt is the discretization time, and the non-linear operations and their trajectories are represented with dashed lines.

The dynamics of the Nanowire Neural Network are described by the following equations:

$$z(t+1) = \text{RESCALE}(W_h h(t) + W_x x(t+1) + b)\,, \tag{7}$$

$$q(t+1) = \text{diag}(K_p(z(t+1)) + K_d(z(t+1)))\, h(t), \tag{8}$$

$$r(t+1) = K_p(z(t+1)) - q(t+1), \tag{9}$$

$$h(t+1) = r(t+1)\, \Delta t + \gamma\, h(t). \tag{10}$$

where $h(t)$ is the recurrent state, $x(t)$ the external input, W_h the recurrent kernel, W_x the input-to-recurrent kernel, and b a bias vector.

The authors' proposed model admits the following fundamental set of nanowire-related hyperparameters: K_{p0}, K_{d0}, η_p, η_d, Δt, and γ. The parameter γ is introduced to ensure asymptotically stable dynamics (i.e., the Echo State Property (**?**)) when $\gamma < 1$. Moreover, for certain hyperparameter configurations, the input x directly influences the reservoir state by shifting the fixed point of Eq. (4) into a region suitable for information representation and learning.

For this reason, the affine transformation in Eq. (7) is subjected to a rescaling operation before entering the nonlinearities. This heuristic improves the stability and performance of the computational model described in Eqs. (7)–(10), and is applied component-wise as follows:

$$\text{RESCALE}(y) = \frac{(b-a)}{1 + \exp(-sy)} + a, \tag{11}$$

where $a = 0.35$, $b = 1.15$, and s is a slope parameter.

The computational graph of the proposed Nanowire Neural Network is illustrated in Figure 2. In the following, we provide two specific instances of this framework, corresponding to the Reservoir Computing (RC) paradigm and fully trainable RNN models.

2. Models

2.1. ESN

2.1.1. NW-ESN:Reservoir model

NW-ESN is the reservoir-based Nanowire Neural Network Model in which both the kernel (W_x) and the recurrent kernel (W_h) matrices are initialized under stability constraints and left untrained. As usual in RC applications, we rescale the kernel W_x, the bias b, and recurrent kernel W_h, with the input scaling (ω), the bias scaling (β), and the spectral radius (ρ), respectively, and we use them as hyperparameters of the model. Some intuitions about the parameters:

- The input scaling controls how strongly the external input influences the reservoir state, with a large ω amplifying the input, yielding non linear responses at the risk of instability while small ω increases the importance of the internal network dynamics at risk of underfitting.
- Bias scaling(β) helps shift the internal state into a useful operating region—important because nanowire nonlinearities depend sensitively on the input range.
- Spectral radius(ρ) governs the stability and memory depth of the recurrent matrix W_h. It is defined as the

largest absolute eigenvalue of W_h. Intuitively, a $\rho < 1$ implies fading memory while ρ close to 1 allows for longer memory with larger dynamics, although it does run the risk of instability.

2.1.2. Baseline ESN model

- The baseline ESN uses a standard `tanh` activation with a leaky-integrator update, i.e., $h(t+1) = (1-a)h(t) + a\ \tanh(W_h h(t) + W_x x(t))$, unlike the NW-ESN which uses nanowire-inspired K_p and K_d dynamics.
- Both W_x and W_h are randomly initialized and left untrained, similar to NW-ESN, but they do not undergo nanowire-specific rescaling or nonlinear transformations.
- The reservoir dynamics are governed only by the leaking rate and spectral radius, providing classical fading-memory behavior without potentiation–depression mechanisms.
- Overall, the regular ESN forms a standard nonlinear reservoir through `tanh`, whereas NW-ESN produces richer, physics-inspired trajectories via nanowire ODE-based updates.

2.2. RNN

2.2.1. NW-RNN: Fully-trainable model

Some ideas about this model:

- NW-RNN extends the nanowire model into a fully trainable recurrent network.
- Both the input kernel W_x and recurrent kernel W_h are optimized using backpropagation through time (BPTT), just like in standard RNNs.
- The nanowire nonlinearities K_p, K_d, and the state update equation remain fixed, providing rich, physics-inspired dynamics.
- Training adjusts how inputs and past states influence the nanowire dynamics, enabling task-specific adaptation.
- This combines the expressiveness of RNNs with the stability and biological realism of nanowire-based state transitions.

2.2.2. Baseline RNN model

Some intuitions about this model:

- The baseline RNN uses a standard `tanh` activation for state updates, whereas NW-RNN replaces this with nanowire-inspired K_p/K_d dynamics.
- Its recurrence is given by $h(t+1) = \tanh(W_h h(t) + W_x x(t))$, without any physical ODE-based update or discretization term Δt.
- The baseline RNN learns only through backpropagation over the usual neural activations, lacking the potentiation-depression behaviour encoded in NW-RNN.
- Unlike NW-RNN, the baseline RNN has no mechanism resembling the Echo State stability via γ; its dynamics depend solely on learned weights and the `tanh` nonlinearity.

3. Our Implementation

To study the behaviour of nanowire-based recurrent models, we implemented both NW-RNN and NW-ESN directly in Python, following the update equations presented in the original paper. All code was developed inside the notebooks `da527-final-project-ecg-syn-control`, `da527-final-project-quake-forda`, `da527-final-project-regression-datasets`, with each model implemented as a standalone class for clarity and modularity.

3.1. NW-RNN Implementation

The fully trainable NW-RNN architecture is implemented in the class `NanowireRNN`. This model directly encodes the nanowire update equations (Eq. 7–10) inside its `forward()` method and uses PyTorch's autograd engine to learn both recurrent and input kernels through backpropagation through time (BPTT). Unlike a standard RNN, the NW-RNN does not use a `tanh` nonlinearity; the nanowire dynamics themselves govern the state evolution. A simplified version of the forward step is:

```
def forward(self, input, hx):
    # 1. Affine term: z(t+1)
    z_t1_affine = self.W_h(hx) + self.W_x(
        input)

    # 2. Apply RESCALE
    z_t1 = self.rescale(z_t1_affine)

    # 3. Potentiation and depression terms
    Kp = self.Kp0 * torch.exp(self.eta_p *
        z_t1)
    Kd = self.Kd0 * torch.exp(-self.eta_d *
        z_t1)

    # 4. r(t+1)
    r_t1 = Kp * (1 - hx) - Kd * hx
```

```
    # 5. Next state h(t+1)
    h_t1 = self.gamma * hx + r_t1 * self.
        delta_t

    return torch.clamp(h_t1, 0.0, 1.0)
```

The NW-RNN training loop is implemented in `train_nw_rnn()`, using the Adam optimizer with early stopping. This allows the full set of nanowire parameters (W_x, W_h, and the nonlinear update dynamics) to be learned end-to-end.

3.2. From NW-RNN to NW-ESN

The NW-ESN is built by starting from the fully trainable NW-RNN architecture and *freezing* all trainable parameters. The recurrent kernel W_h and input kernel W_x are then replaced with randomly initialized matrices that are rescaled according to ESN conventions:

- **Input scaling** (ω): applied to `W_x.weight`

$$W_x \sim U(-\omega, \omega)$$

- **Bias scaling** (β): applied to `W_x.bias`

$$b \sim U(-\beta, \beta)$$

- **Spectral radius** (ρ): enforced on `W_h.weight` by normalizing with the largest eigenvalue:

$$W_h \leftarrow \frac{\rho}{\lambda_{\max}} W_h$$

These steps are implemented inside the grid-search helper `grid_search_nw_esn()`, where the model parameters are iterated over and rescaled based on their names (`W_x.weight`, `W_x.bias`, `W_h.weight`). After scaling, the nanowire dynamic update from NW-RNN remains intact, but the weights are no longer trainable—yielding the NW-ESN reservoir model.

3.3. Baseline Implementations

For comparison, we implemented two classical baselines:

- **ESNCell / ESN_Network**: Standard Echo State Network using a leaky-integrator update and a `tanh` nonlinearity:

```
h = (1 - a) * h + a * np.tanh(W_h @ h +
    W_x @ x)
```

- **SimpleRNN**: A fully trainable classical RNN:

```
h = torch.tanh(W_h(h) + W_x(x))
```

trained end-to-end via BPTT.

3.4. Experimental Setup and Reproducibility

We evaluated all four models—NW-ESN, NW-RNN, ESN, and RNN—on the same datasets used in the original paper: FordA, ECG5000, SyntheticControl, Earthquakes, FloodModeling1, and FloodModeling2. The notebook includes utility functions such as `load_ucr()`, normalization helpers, and complete training/evaluation pipelines for both classification and regression tasks.

Hyperparameter sweeps were conducted using `grid_search_esn()` and `grid_search_nw_esn()` to replicate the model-selection procedures described in the paper.

Overall, our implementation preserves the structure and behaviour of the original nanowire models while remaining fully reproducible and easily extendable through the provided notebooks.

4. Results and Analysis

4.1. About the datasets

Dataset	Train Size	Test Size	TS Length
FordA Dau et al. [2019]	3601	1320	500
ECG5000 Dau et al. [2019]	500	4500	140
SyntheticControl Dau et al. [2019]	300	300	60
Earthquakes Dau et al. [2019]	322	139	512
FloodModeling1 Tan et al. [2021]	471	202	266
FloodModeling2 Tan et al. [2021]	389	167	266

Figure 3. Dataset details, the first 4 are classification tasks, the last 2 are regression tasks

- **FordA:** Engine sensor readings used to distinguish healthy vs. faulty operating conditions. The sequences are long (about 500 time steps) and noisy, making this a realistic and challenging binary time-series classification task.
- **ECG5000:** ECG heartbeat recordings from the PhysioNet BIDMC database. A 5-class classification problem with subtle waveform variations and class imbalance, commonly used to evaluate medical time-series models.
- **SyntheticControl:** A classic synthetic dataset containing six clean, noise-free temporal patterns. The short sequences (length 60) make it ideal for studying baseline pattern separation behaviour.
- **Earthquakes:** Seismic sensor measurements used to detect earthquake events. This binary task is difficult due to strong class imbalance and long, noisy sequences, providing a good test of rare-event detection.
- **FloodModeling1:** A real-world multivariate regression dataset for predicting river water-level dynamics.

It includes environmental variables such as rainfall and soil moisture, useful for evaluating long-term dependency learning.

- **FloodModeling2:** A more complex and extended version of FloodModeling1 with richer inputs and longer sequences. It serves as a challenging benchmark for testing temporal models in hydrological forecasting.

4.2. Results of our implementation

Task	ESN	NW-ESN	RNN	NW-RNN
FordA(↑)	0.494	0.558	0.486	**0.692**
ECG5000(↑)	0.838	0.820	**0.932**	0.928
SyntheticControl(↑)	0.450	0.730	**0.963**	0.947
Earthquakes(↑)	0.712	**0.763**	0.741	0.748
FloodModeling1(↓)	0.1038	**0.0177**	0.0180	0.0184
FloodModeling2(↓)	0.0913	0.0959	0.0182	**0.0143**

Table 1. Performance comparison across tasks. Arrows indicate whether higher (↑) or lower (↓) values are better. ↑ denotes accuracy, ↓ denotes root mean squared error

4.3. Links to our notebooks:

- ECG5000, SyntheticControl(by Saptarshi Mukherjee)
- FordA, Earthquake(by Ishan Chandra Gupta)
- FloodModeling1 & 2(by Soumya Savarn)

4.4. Link to the youtube video

Youtube link

5. Conclusion

Across the six benchmark datasets, our implementation shows that the nanowire-based models behave competitively with their classical counterparts. NW-RNN performs particularly well on FordA and FloodModeling2, where it achieves the best scores, and it remains close to the standard RNN on ECG5000 and SyntheticControl. NW-ESN consistently improves over the standard ESN on most tasks, most notably on Earthquakes and FloodModeling1, highlighting the benefits of incorporating nanowire-inspired dynamics even without training the recurrent weights. While classical RNNs still obtain the strongest overall results on several datasets, the nanowire models demonstrate strong performance relative to their architectural simplicity, offering a promising alternative for lightweight sequence modelling.

Acknowledgements

Our implementation of the nanowire-based recurrent models is directly inspired by and adapted from the formulation presented in Nanowire Neural Networks for Time-Series Processing by Pistolesi et al. (NeurIPS 2024 Workshop). We thank the authors for making their methodology accessible and for providing the foundational ideas that guided our reproduction and analysis.

References

Pistolesi, V., Ceni, A., Milano, G., Ricciardi, C., and Gallicchio, C. Nanowire neural networks for time-series processing. In *NeurIPS 2024 Workshop on Machine Learning with New Compute Paradigms*, 2024.