Hardware-Aware Ternary Neural Networks with Accuracy Recovery on Two-State Silicon Nanowire Memristive Crossbars

Ali Pilehvar Meibody[1], Junrui Chen[1,*],Aferdita Xhameni[2,3],Antonio Lombardo[2,3] and Sandro Carrara

[1]Bio/CMOS Interfaces Laboratory (BCI), École Polytechnique Fédérale de Lausanne (EPFL), Neuchâtel, Switzerland.

[2]London Center for Nanotechnology, University College London, WC1E 7JE, UK
[3]Department of Electronic and Electrical Engineering, University College London, London WC1E 7JE

[*]Correspondence: junrui.chen@epfl.ch

Supplementary Information

S1 Ternary Surrogate-Gradient Optimization

The ternary neural network (TNN) employed in this work constrains the synaptic weights used during forward computation to the discrete set $\{-1, 0, +1\}$. Each ternary layer maintains a latent full-precision trainable weight w, while the corresponding weight used during the forward pass is obtained according to

$$w_t = \begin{cases} +1, & w > \Delta, \\ 0, & |w| \leq \Delta, \\ -1, & w < -\Delta, \end{cases} \tag{1}$$

where w_t denotes the ternary weight and Δ is the ternarization threshold. In the present implementation, $\Delta = 0.05$.

Because the ternary quantization operation in Eq. (1) is piecewise constant, its exact derivative with respect to the latent weight is zero almost everywhere and undefined at the transition boundaries $w = \pm\Delta$. Direct use of this derivative would therefore prevent effective gradient-based optimization of the latent full-precision parameters. During backpropagation, the exact derivative of the ternary quantizer was consequently replaced by a surrogate-gradient function $g(w)$:

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\partial \mathcal{L}}{\partial w_t} g(w), \tag{2}$$

where $\mathcal{L}$ denotes the training loss.

Seven surrogate-gradient formulations were investigated under identical TNN architecture and training conditions. In every case, the forward computation used the exact ternary representation defined in Eq. (1); only the backward approximation used to optimize the latent full-precision weights was modified.

S1.1 Plain Straight-Through Estimator

The plain straight-through estimator (STE) directly propagates the incoming gradient through the ternary quantization operation:

$$g_{\text{Plain}}(w) = 1. \tag{3}$$

Accordingly,

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\partial \mathcal{L}}{\partial w_t}. \tag{4}$$

This formulation allows unrestricted gradient propagation independently of the magnitude of the latent weight.

S1.2 Clipped Straight-Through Estimator

A clipped STE was evaluated to restrict gradient propagation to a finite interval of the latent-weight space:

$$g_{\text{Clipped}}(w) = \begin{cases} 1, & |w| \leq 1, \\ 0, & |w| > 1. \end{cases} \tag{5}$$

Thus,

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\partial \mathcal{L}}{\partial w_t} \mathbb{I}\left(|w| \leq 1\right), \tag{6}$$

where $\mathbb{I}(\cdot)$ denotes the indicator function. Latent weights outside the interval $[-1, 1]$ therefore receive no gradient through the ternary quantization operation.

S1.3 Scaled Straight-Through Estimator

The scaled STE modifies the magnitude of the gradient propagated through the quantizer using a constant scaling coefficient s:

$$g_{\text{Scaled}}(w) = s. \tag{7}$$

The corresponding gradient is

$$\frac{\partial \mathcal{L}}{\partial w} \approx s \frac{\partial \mathcal{L}}{\partial w_t}. \tag{8}$$

Multiple values of s were evaluated to determine the influence of gradient scaling on TNN optimization.

S1.4 Bi-Real-Style Triangular Surrogate

A piecewise-linear triangular approximation was also investigated:

$$g_{\text{Tri}}(w) = \max\left(2 - 2|w|, 0\right). \tag{9}$$

This surrogate assigns its largest gradient magnitude to latent weights close to the origin and progressively reduces the gradient as $|w|$ increases, reaching zero outside its finite support region.

S1.5 Tanh/IR-Net-Style Surrogate

A smooth surrogate based on the derivative of a scaled hyperbolic tangent function was evaluated:

$$g_{\mathrm{Tanh}}(w) = \beta \left[1 - \tanh^2(\beta w)\right], \tag{10}$$

where β controls the concentration of the gradient around the origin. Increasing β produces a narrower gradient-support region, whereas smaller values produce a broader profile.

S1.6 Threshold-Window Surrogate

Because ternarization introduces two explicit transition boundaries at $w = -\Delta$ and $w = +\Delta$, a threshold-window surrogate was evaluated to concentrate the gradient around these transitions:

$$g_{\mathrm{TW}}(w) = \begin{cases} 1, & |w - \Delta| \le r \quad \text{or} \quad |w + \Delta| \le r, \\ 0, & \text{otherwise}, \end{cases} \tag{11}$$

where r denotes the half-width of the gradient-support region surrounding each ternarization threshold.

S1.7 Smooth Threshold-Window Surrogate

Finally, a continuous threshold-aware surrogate was evaluated by centering triangular profiles at the positive and negative ternarization boundaries:

$$g_{\mathrm{STW}}(w) = \max\left[\max\left(1 - \frac{|w - \Delta|}{r}, 0\right), \max\left(1 - \frac{|w + \Delta|}{r}, 0\right)\right]. \tag{12}$$

The gradient reaches its maximum at $w = \pm\Delta$ and decreases linearly as the latent weight moves away from the corresponding transition boundary. The parameter r determines the width of the gradient-support region.

Among the investigated formulations, the smooth threshold-window surrogate provided the selected optimization configuration for subsequent TNN training. A window parameter of $r = 0.7$ was used in the final TNN implementation.

S2 Binary Neural Network Training

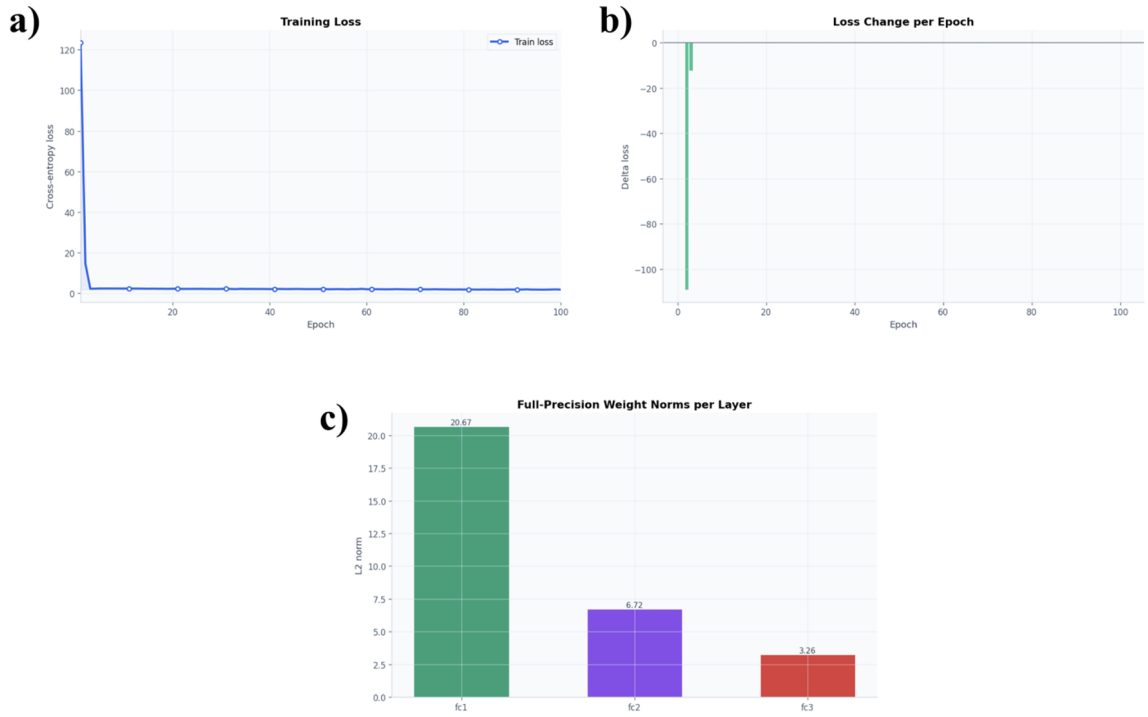


Fig. S2. Training results of Binary Neural Network (BNN) on the MNIST dataset. (a) Training loss evolution. (b) Loss change per epoch. (c) Full-precision weight norms for each network layer.

S3 TNN Surrogate-Gradient Training Comparison

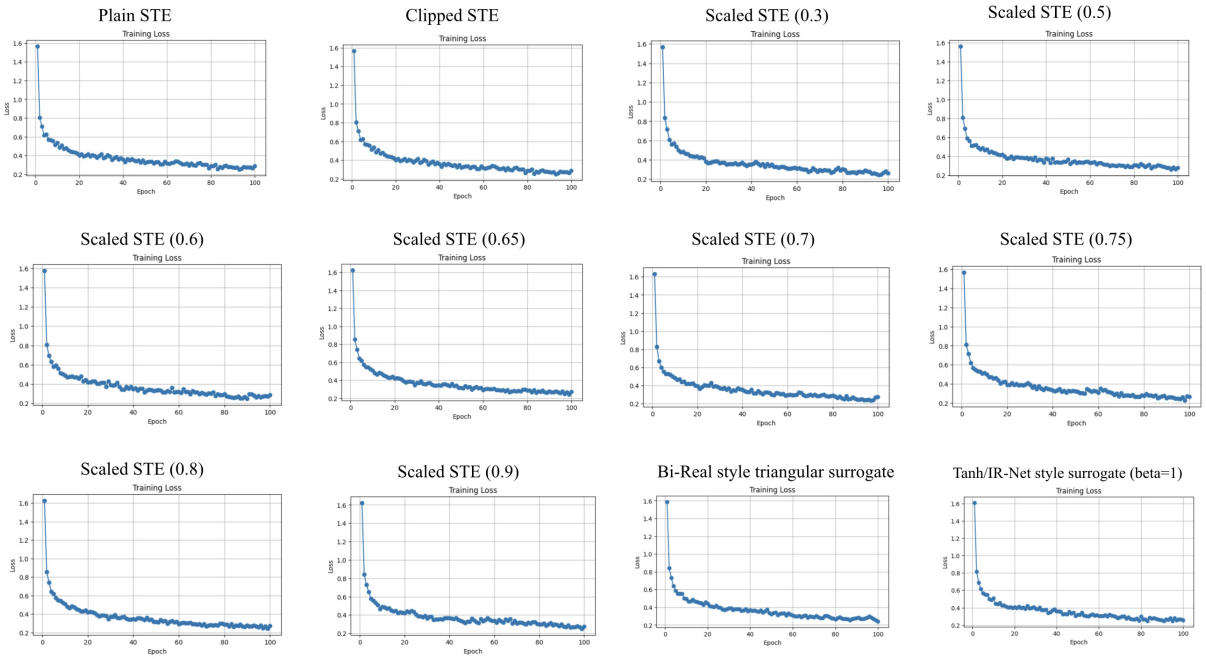


Fig. S3. Training-loss evolution of the initial surrogate-gradient methods evaluated for TNN training. The comparison includes Plain STE, Clipped STE, multiple Scaled STE configurations with $s = 0.3$–0.9, the Bi-Real-style triangular surrogate, and the Tanh/IR-Net-style surrogate with $\beta = 1$.

S4 Extended Surrogate-Gradient Training Comparison

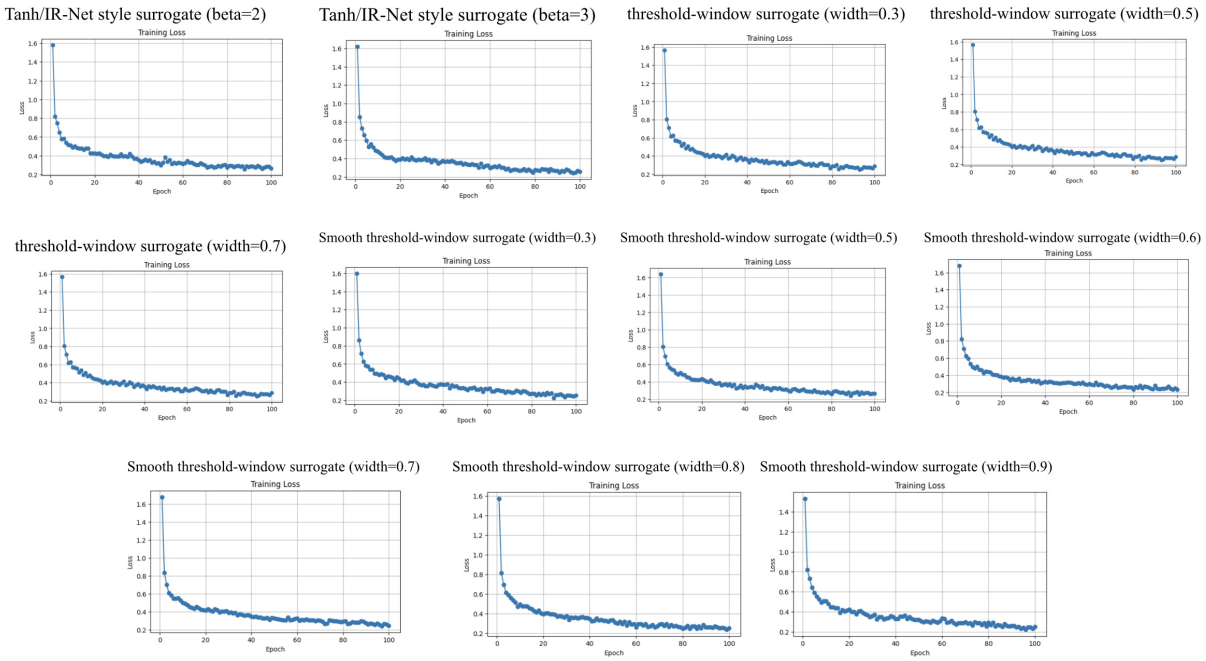


Fig. S4. Training-loss curves for additional surrogate-gradient configurations evaluated in this work, including Tanh/IR-Net-style surrogates with $\beta = 2$ and $\beta = 3$, threshold-window surrogates with different window widths, and smooth threshold-window surrogates with widths ranging from 0.3 to 0.9.

S5 Surrogate-Gradient Parameter Sensitivity

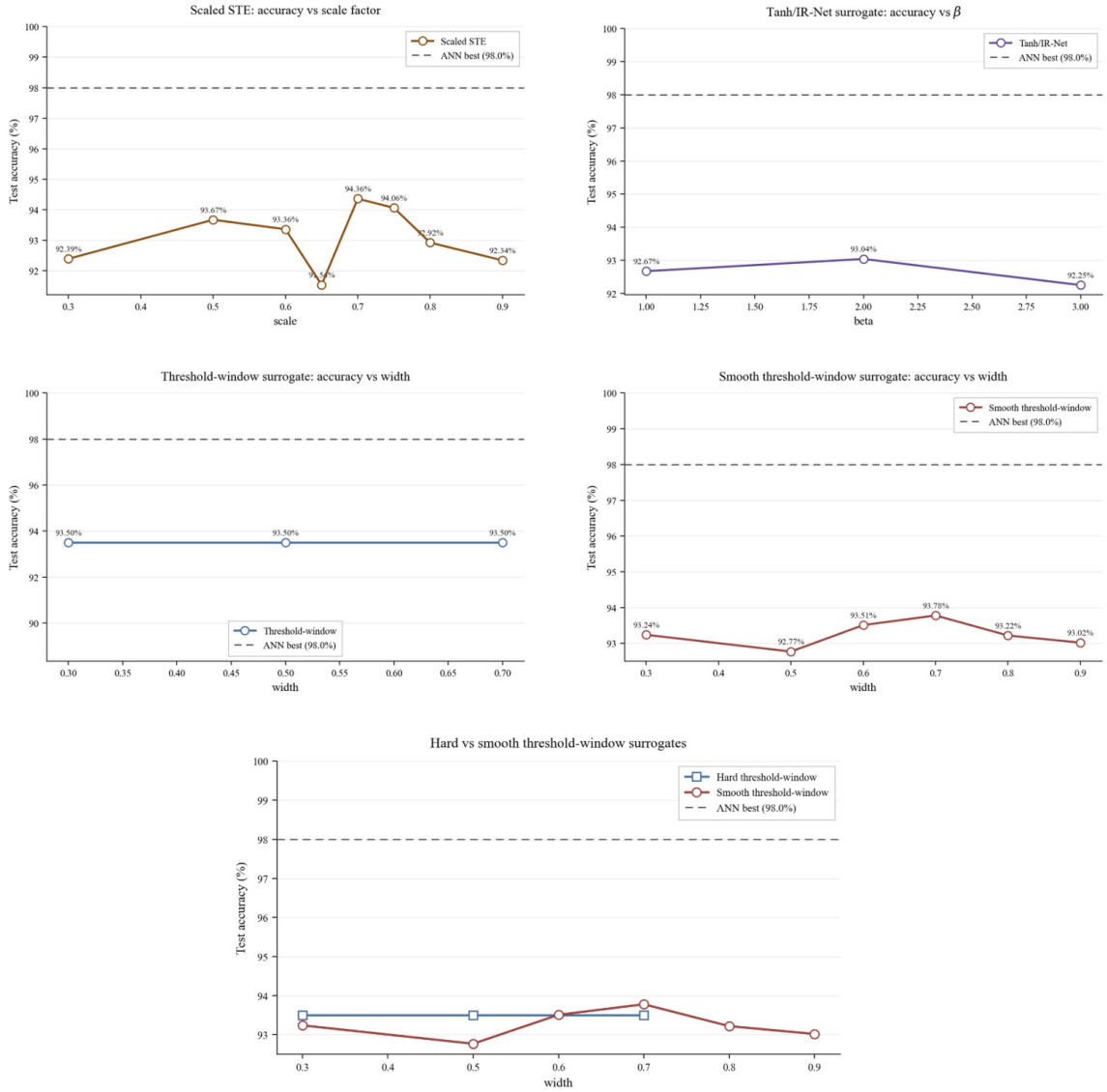


Fig. S5. Parameter-sensitivity analysis of the evaluated surrogate-gradient methods. The figure shows the influence of the scaling factor in Scaled STE, the parameter β in the Tanh/IR-Net surrogate, the window width in the threshold-window surrogate, and the window width in the smooth threshold-window surrogate on the final test accuracy. A direct comparison between hard and smooth threshold-window surrogates is also included. The dashed line indicates the ANN baseline accuracy of 98.0%.

S6 Selected TNN Surrogate-Gradient Candidates

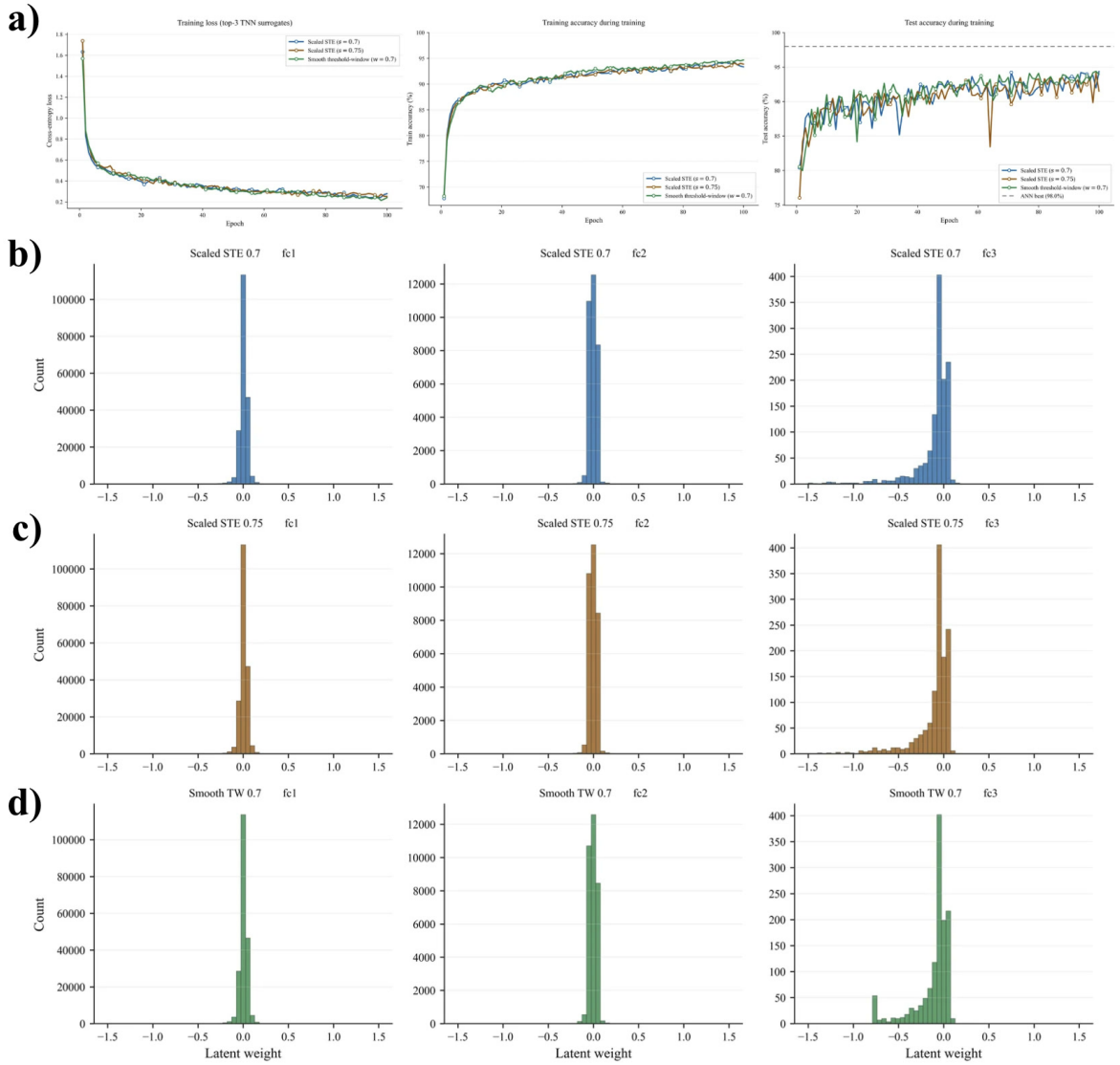


Fig. S6. Detailed comparison of the three selected TNN surrogate-gradient candidates: Scaled STE with $s = 0.7$, Scaled STE with $s = 0.75$, and the smooth threshold-window surrogate with $r = 0.7$.

S7 Ideal CrossSim Accuracy Validation

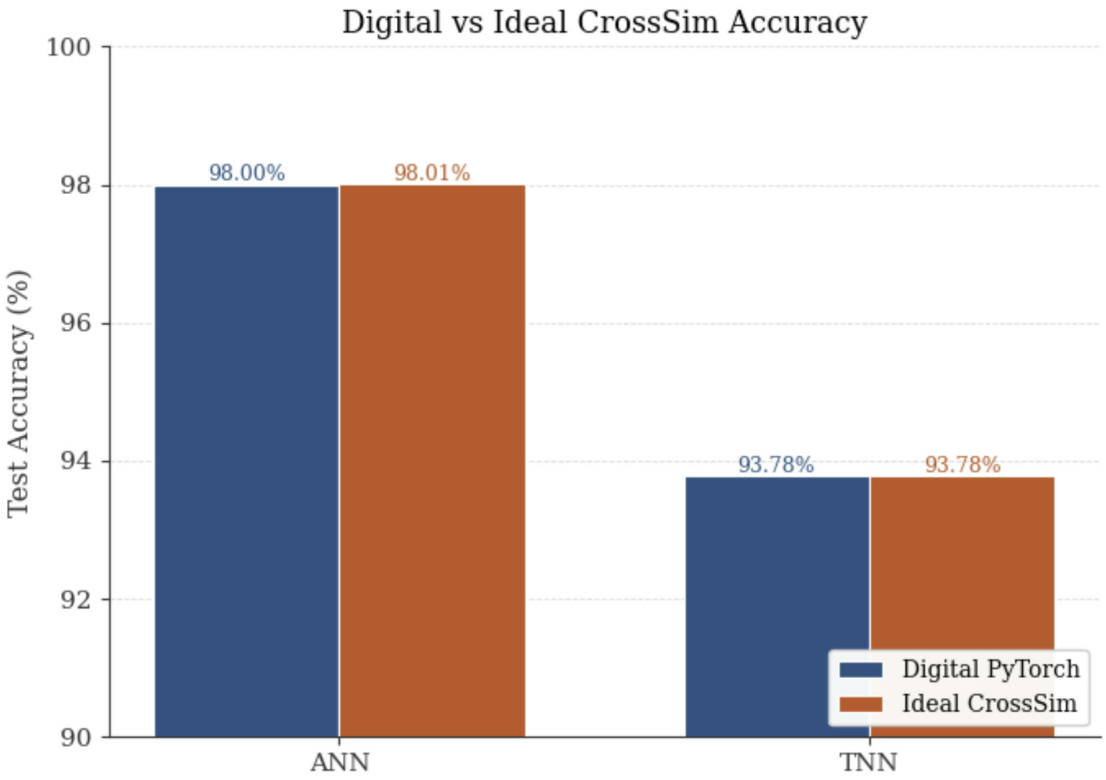


Fig. S7. Comparison of test accuracy obtained from the original PyTorch implementations and their corresponding ideal CrossSim implementations.

S8 Ideal CrossSim Confusion Matrices

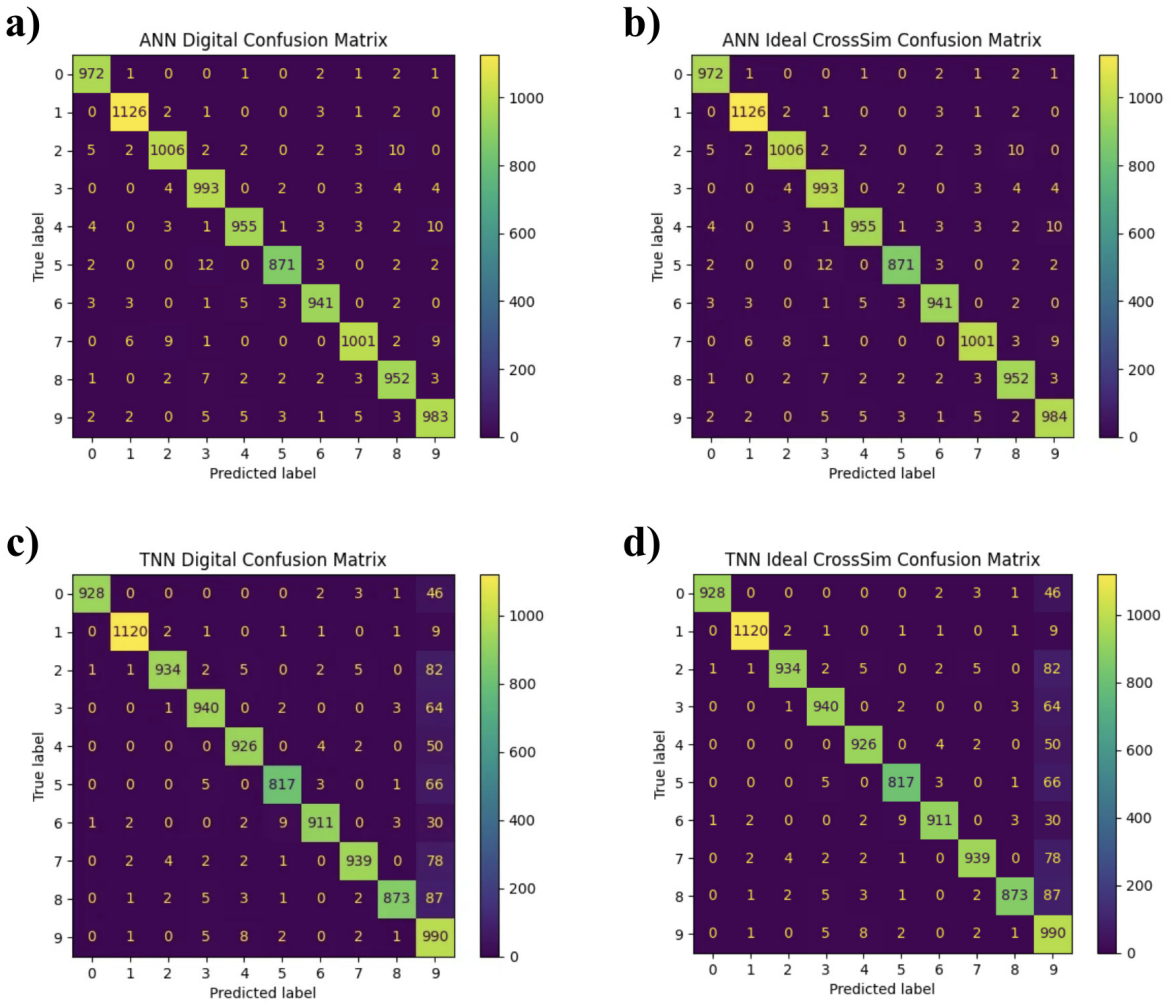


Fig. S8. Confusion matrices for digital and ideal CrossSim implementations. (a) ANN implemented in PyTorch. (b) ANN implemented in ideal CrossSim. (c) TNN implemented in PyTorch. (d) TNN implemented in ideal CrossSim.

S9 ANN Weight-Mapping Validation

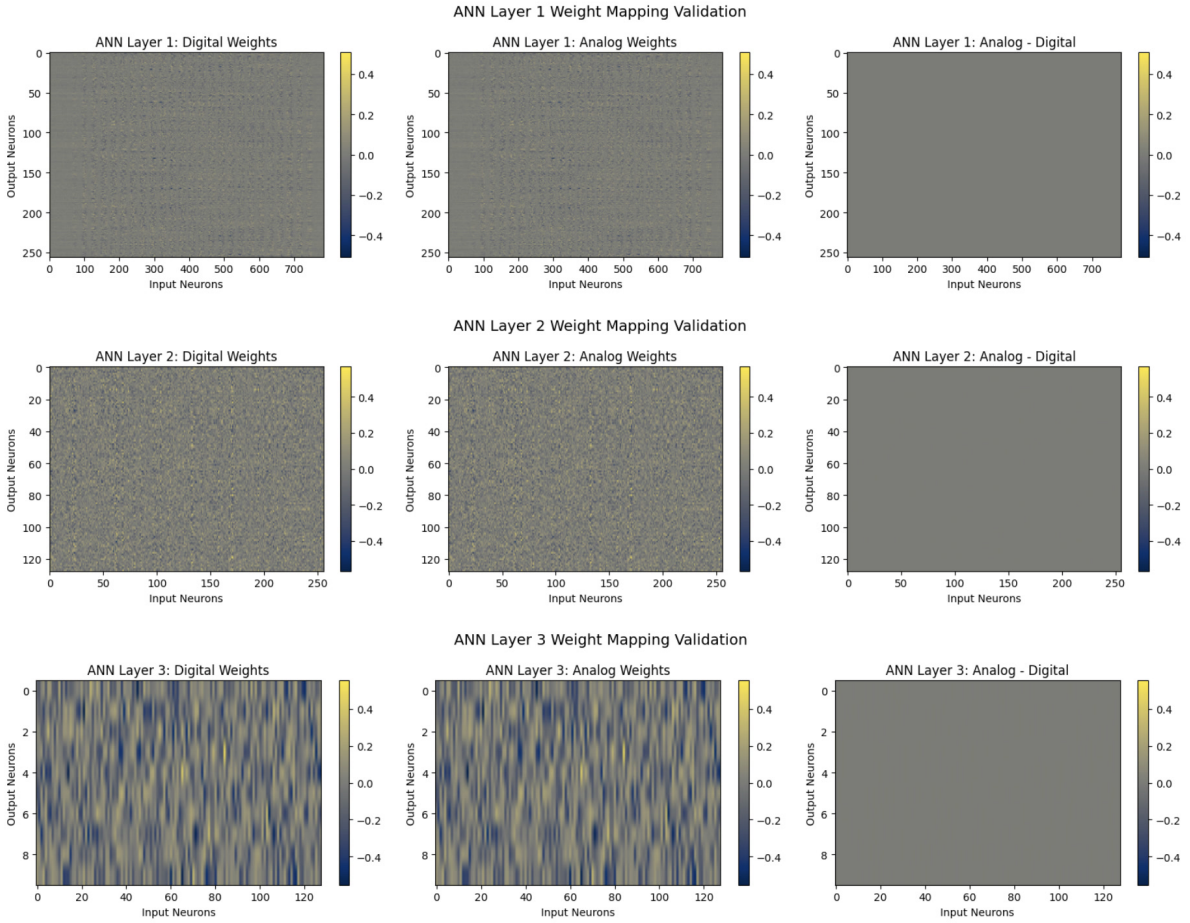


Fig. S9. Validation of ANN weight mapping between PyTorch and ideal CrossSim. For each layer, the left column shows the original digital weights, the middle column shows the reconstructed CrossSim weights, and the right column shows the element-wise difference.

S10 TNN Weight-Mapping Validation

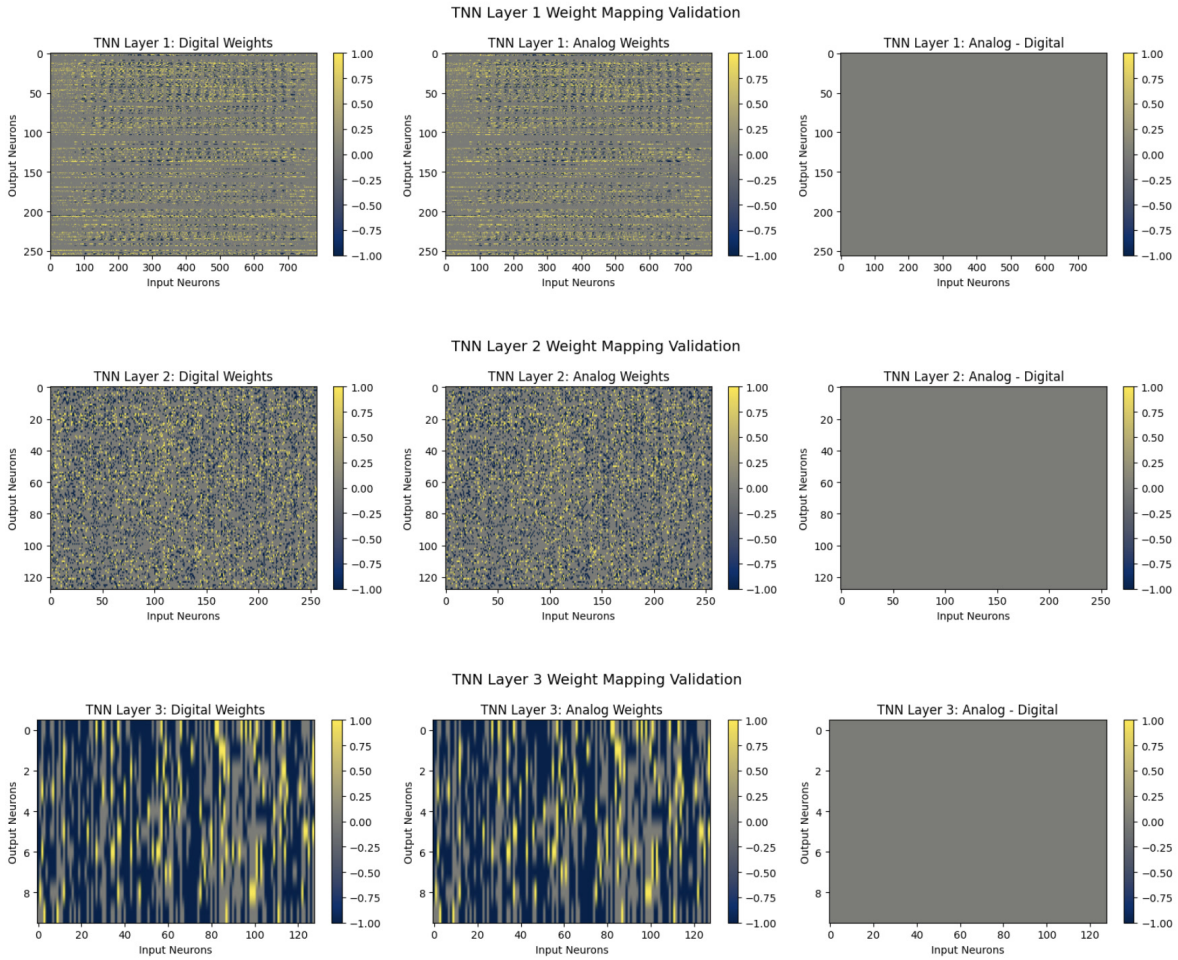


Fig. S10. Validation of TNN weight mapping between PyTorch and ideal CrossSim. For each layer, the left column shows the original ternary weights, the middle column shows the reconstructed CrossSim weights, and the right column shows the difference matrix.

S11 Bit-Sliced Conductance Mapping of FC1

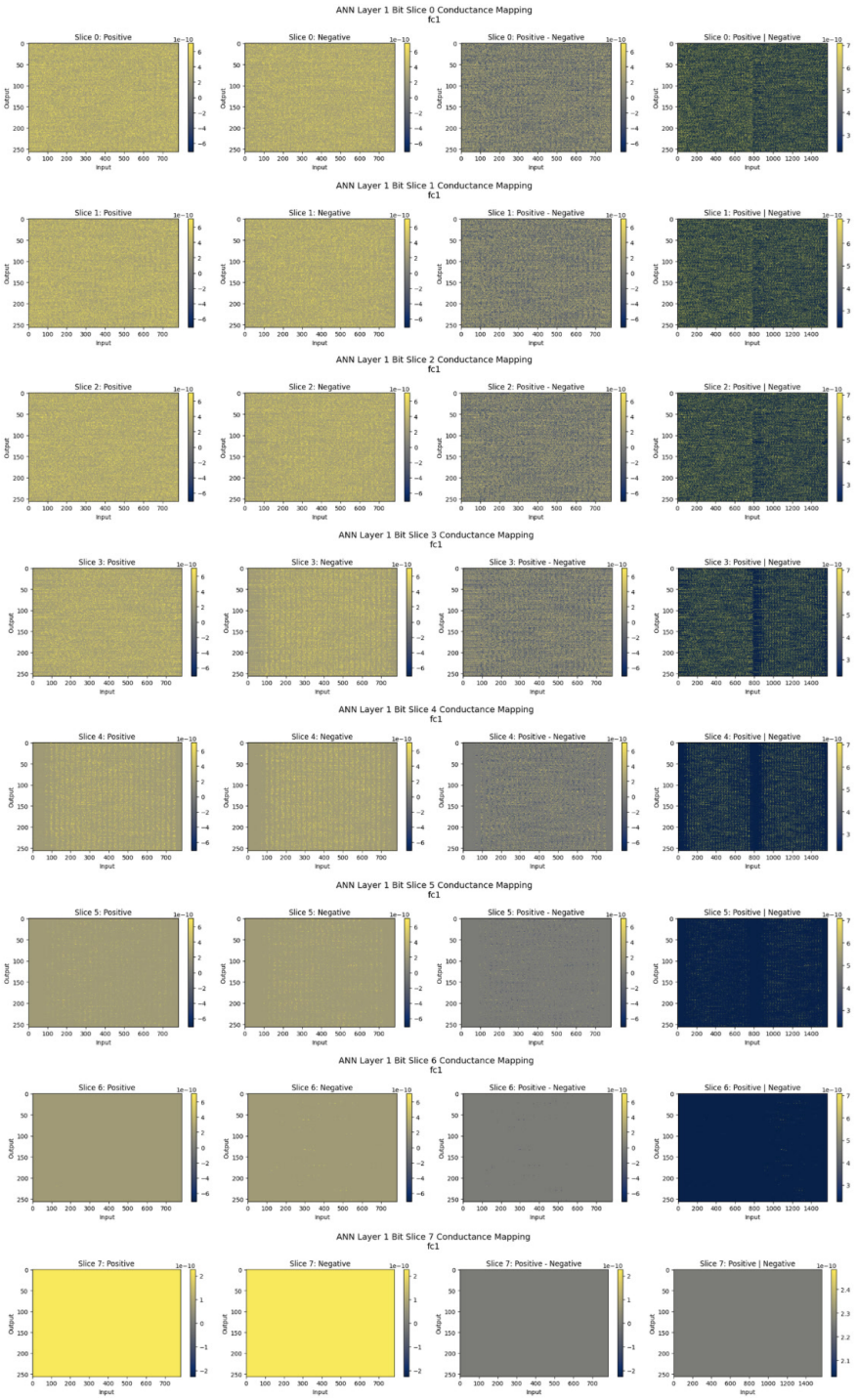


Fig. S11. Conductance mapping of the first ANN layer (FC1) using the bit-sliced crossbar implementation. For each of the eight weight-bit slices, the positive conductance array, negative conductance array, differential representation, and combined crossbar mapping are shown. Each bit slice requires separate positive and negative arrays, resulting in sixteen physical crossbar arrays for one signed 8-bit network layer.

S12 Bit-Sliced Conductance Mapping of FC2

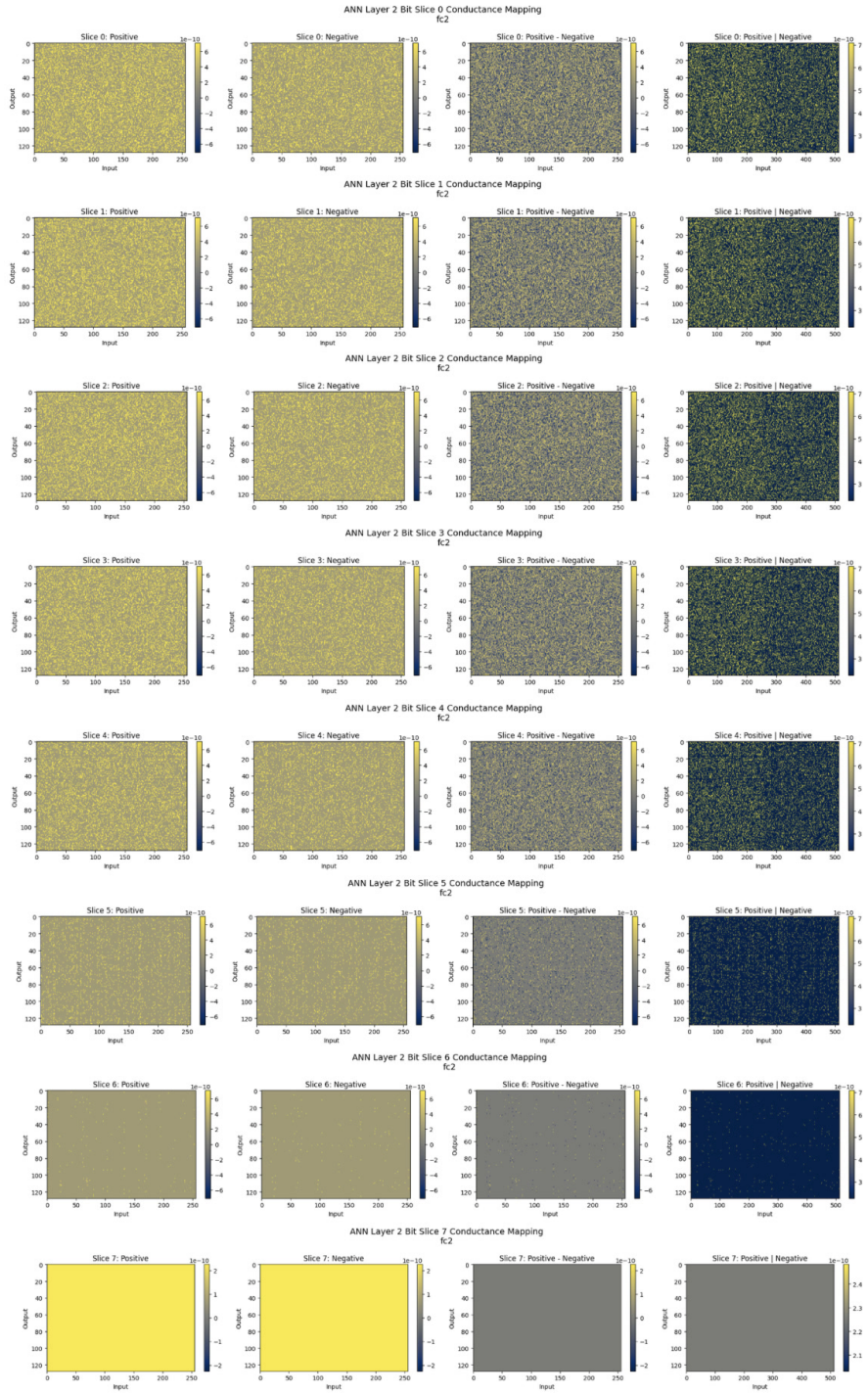


Fig. S12. Bit-sliced conductance mapping of the second ANN layer (FC2). Each row corresponds to one weight-bit slice and shows the positive array, negative array, differential conductance representation, and combined crossbar representation used to implement signed 8-bit weights using binary memristive devices.

S13 Programming-Error Model Validation

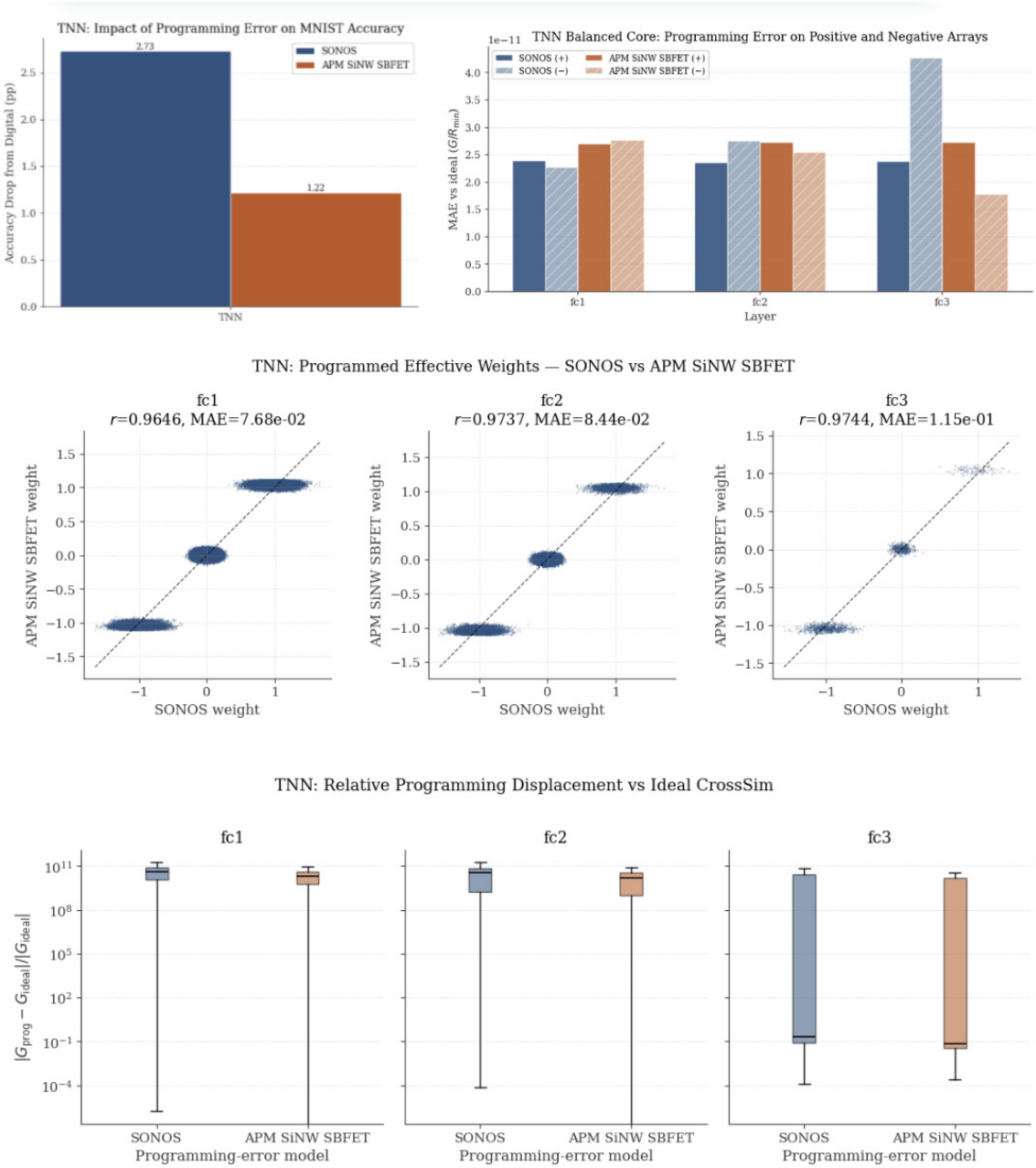


Fig. S13. Validation of the proposed APM SiNW SBFET programming-error model against the reference SONOS model. The top row compares the resulting MNIST accuracy degradation and the mean absolute programming error of the positive and negative arrays across all network layers. The middle row shows the correlation between programmed effective weights obtained using the SONOS and APM SiNW SBFET models for FC1, FC2, and FC3. The bottom row presents the distributions of relative programming displacement from the ideal conductance values.

S14 Statistical Evaluation of Programming Variability

To evaluate the statistical variation in inference accuracy arising from stochastic device programming, a Monte Carlo analysis was performed using 30 independent programming-error realizations of the experimentally calibrated APM SiNW SBFET model. The trained TNN weights, network architecture, and MNIST test dataset were kept unchanged across all realizations. Only the stochastic programming errors applied to the memristive crossbar devices were resampled, while ADC/DAC quantization, conductance drift, read noise, and parasitic interconnect resistance were disabled.

The ideal CrossSim TNN achieved a classification accuracy of 93.78%. Across the 30 independently sampled programming realizations, the mean accuracy was 92.80% with a sample standard deviation of 0.20 percentage points, corresponding to an average accuracy reduction of 0.98 percentage points. The observed accuracies ranged from 92.40% to 93.16%. The original single-seed result of 92.56% was reproduced and falls within the observed distribution.

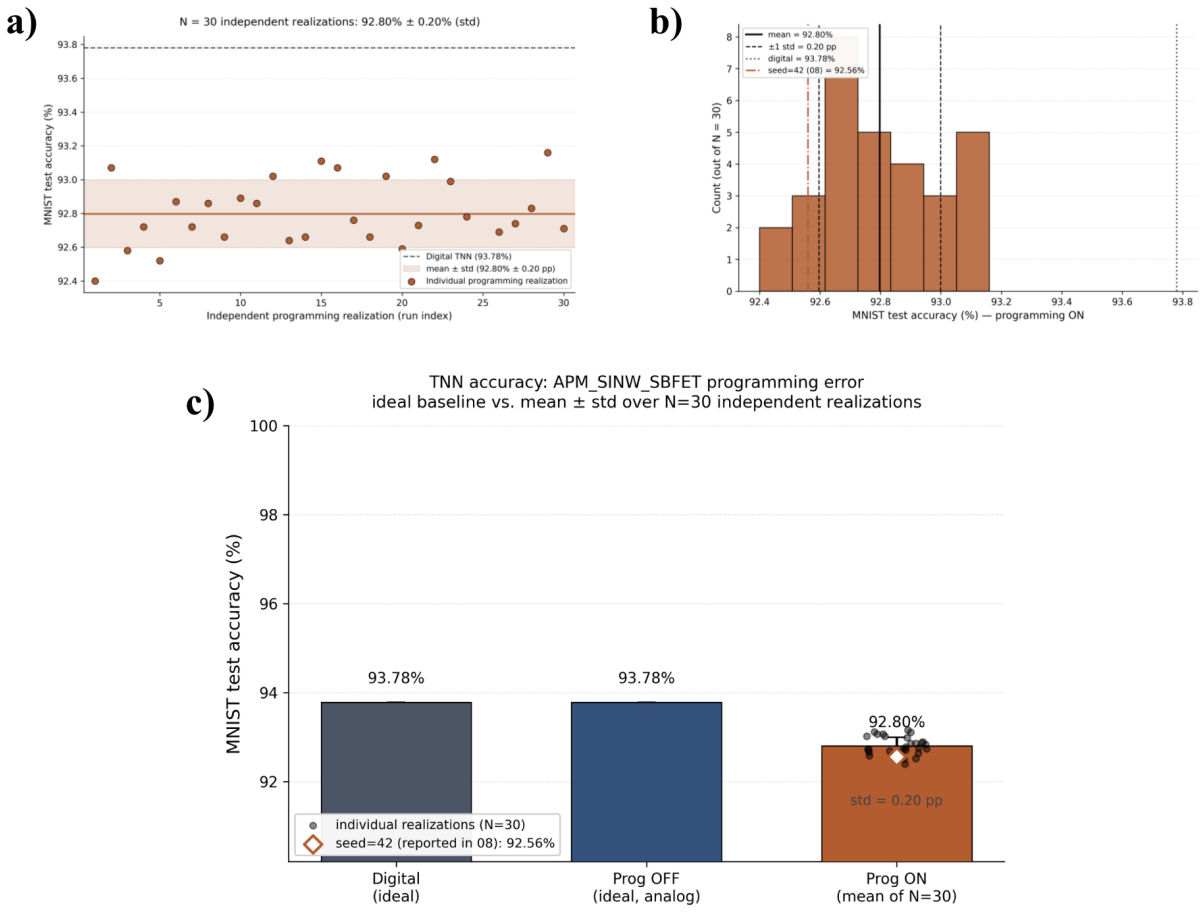


Fig. S14. Monte Carlo evaluation of programming-induced accuracy variability in the SiNW-based TNN crossbar implementation. (a) Classification accuracy across 30 independently sampled programming-error realizations, with the mean accuracy and one-standard-deviation band indicated. (b) Distribution of classification accuracies, showing the ensemble mean, standard-deviation bounds, ideal TNN reference, and the original seed-42 result. (c) Comparison of the digital TNN, ideal CrossSim TNN without programming variability, and programming-aware TNN accuracy averaged over 30 realizations. Error bars indicate one sample standard deviation, individual markers represent separate programming realizations, and the original seed-42 result is highlighted. The Monte Carlo analysis yielded an accuracy of $92.80\% \pm 0.20$ percentage points (mean ± standard deviation), compared with the ideal CrossSim accuracy of 93.78%.

S15 ON-State Conductance Drift

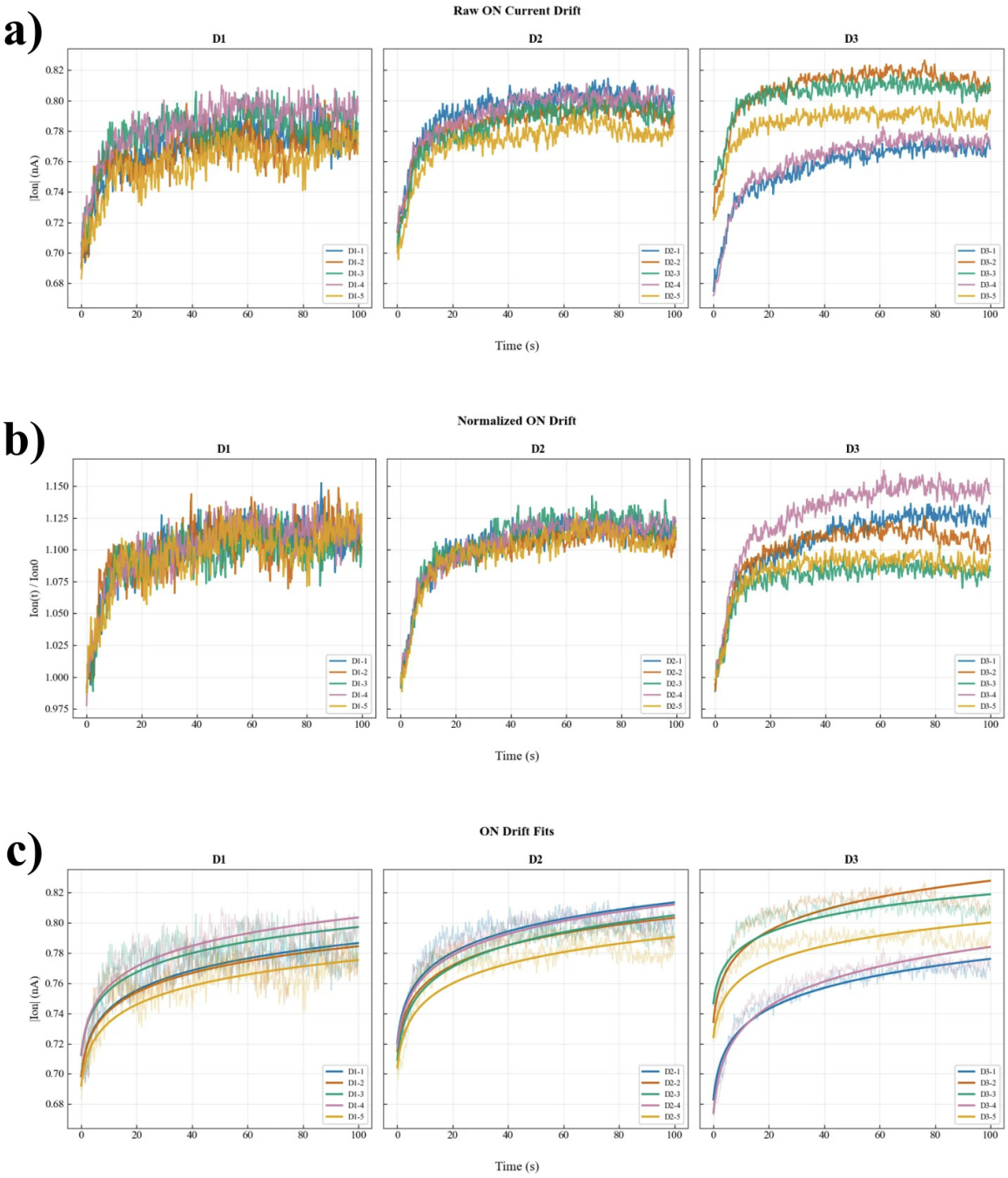


Fig. S15. ON-state (LRS) drift characterization. (a) Raw measured current drift for three devices with five repeated measurements each over a period of 100 s. (b) Normalized ON-state drift curves. (c) Power-law fits applied to the normalized ON-state measurements.

S16 OFF-State Conductance Drift

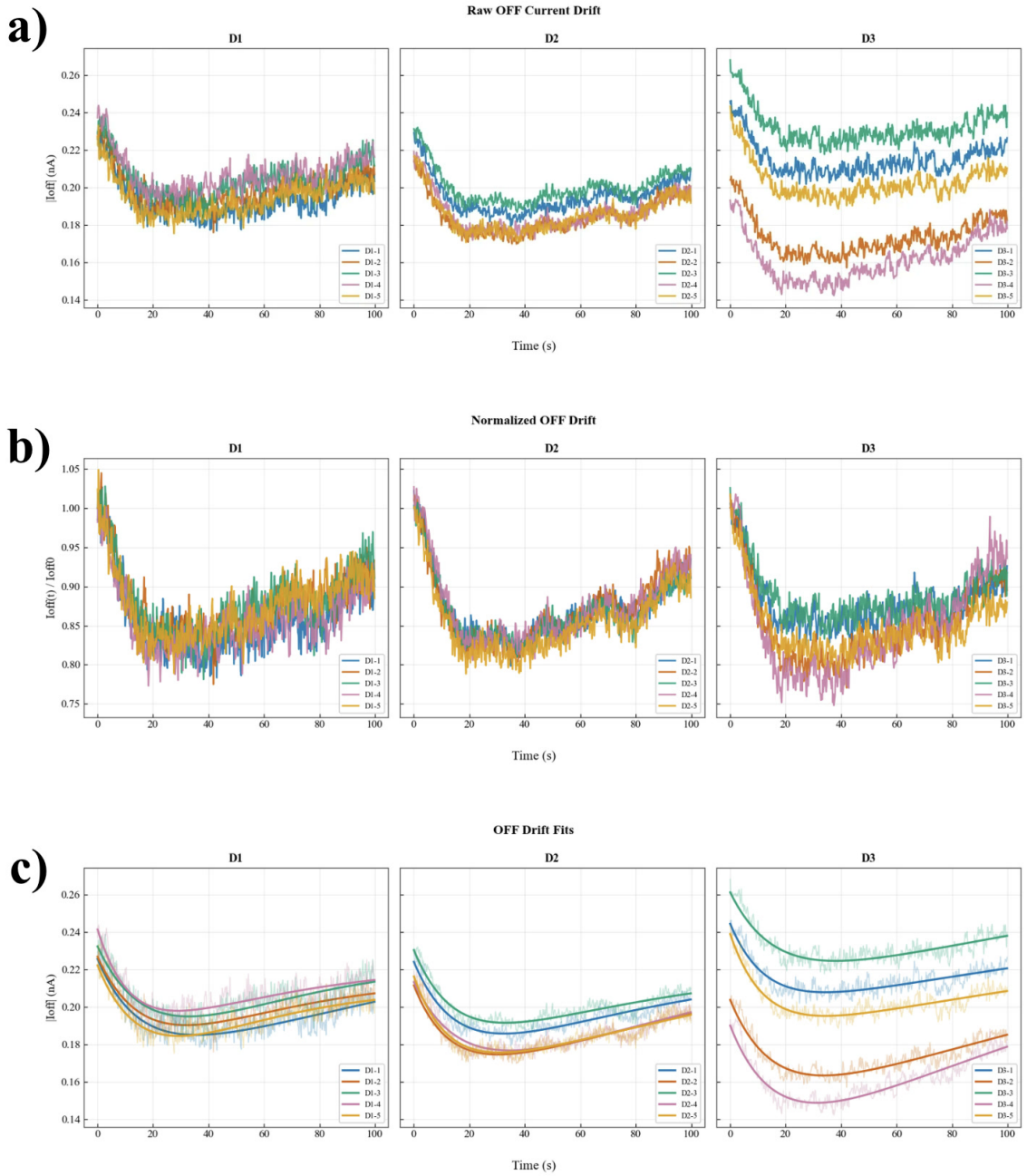


Fig. S16. OFF-state (HRS) drift characterization. (a) Raw measured current drift for three devices with five repeated measurements each over a period of 100 s. (b) Normalized OFF-state drift curves. (c) Double-exponential fits applied to the normalized OFF-state measurements.

S17 Conductance Drift in FC1

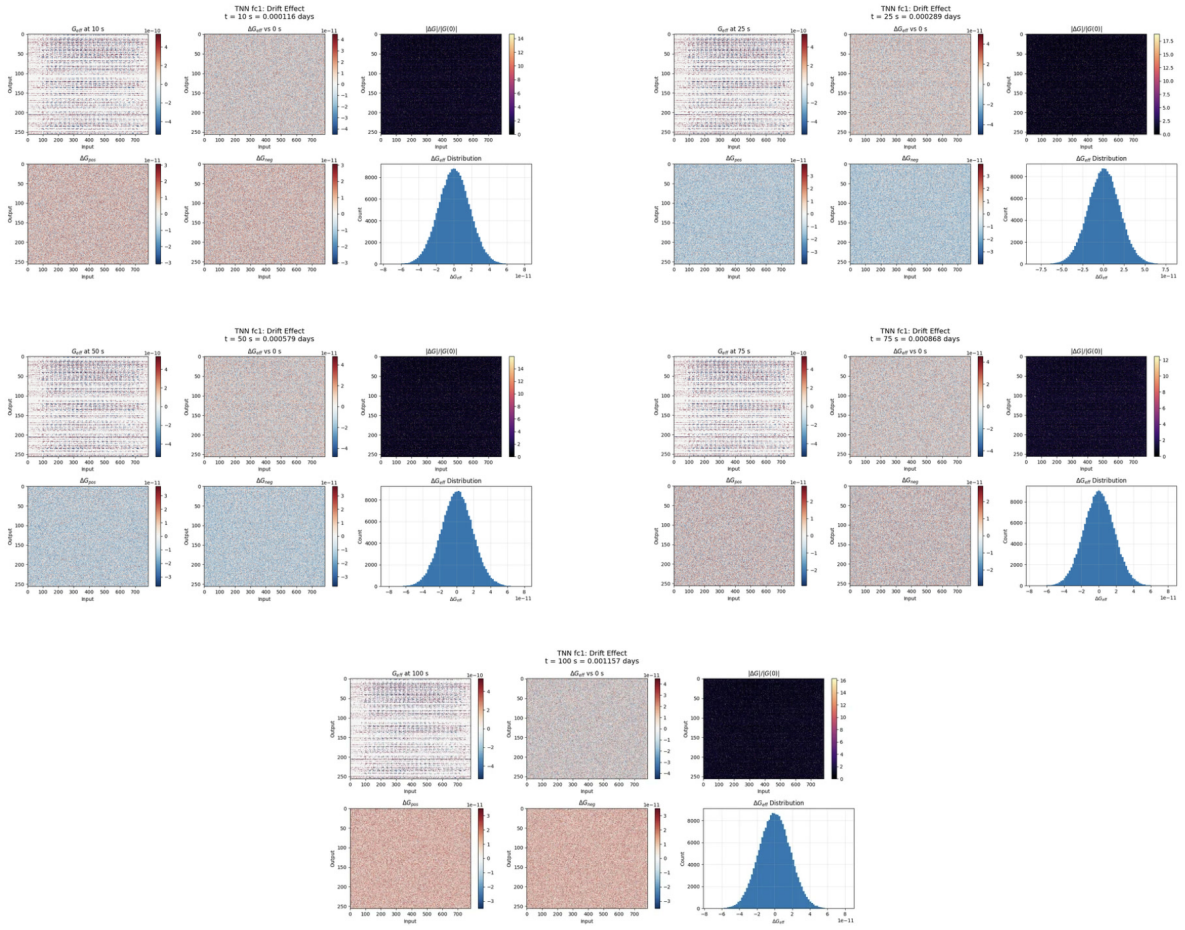


Fig. S17. Drift-induced conductance variations in the first network layer (FC1) at retention times of 10, 25, 50, 75, and 100 s. For each time point, the effective conductance matrix, conductance change relative to the initial state, positive-core conductance variation, negative-core conductance variation, and the corresponding distribution of conductance changes are shown.

S18 Conductance Drift in FC2

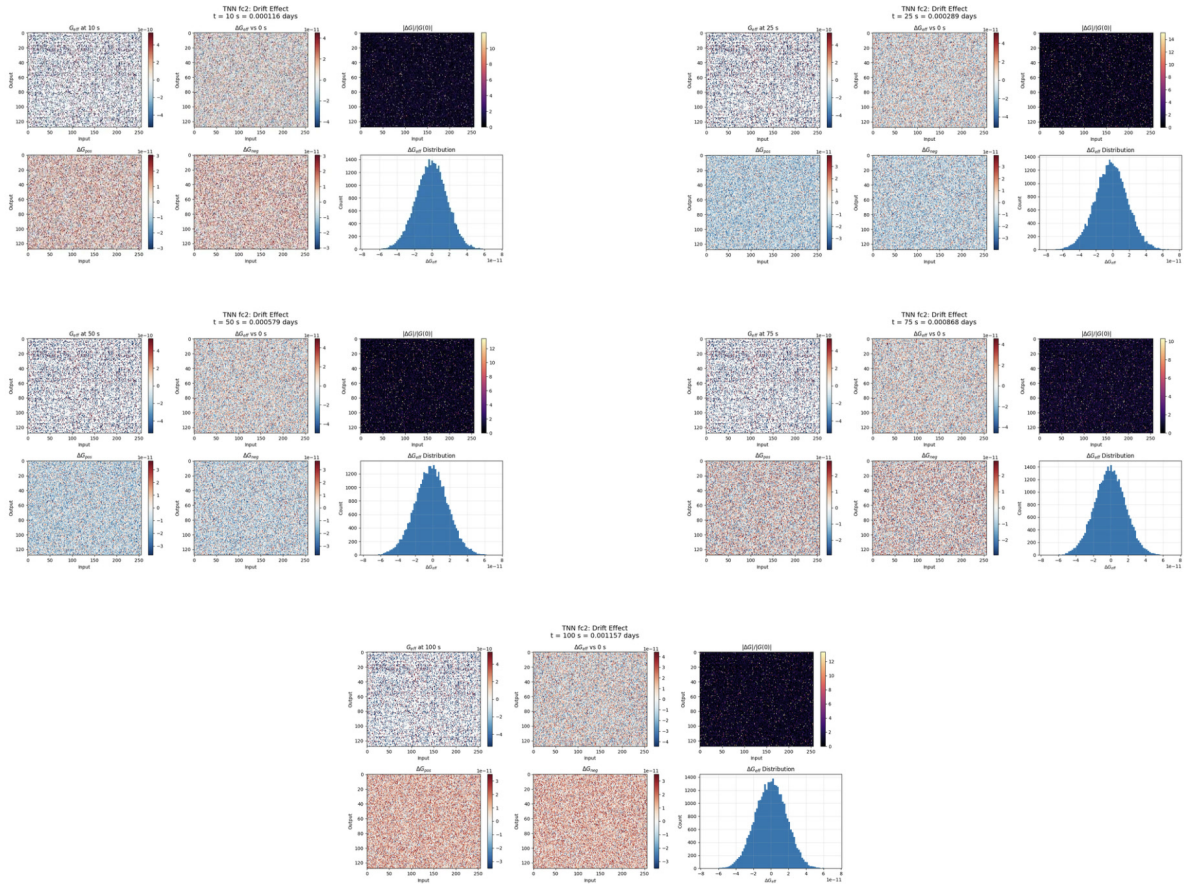


Fig. S18. Conductance-drift visualization for the second network layer (FC2) at different retention times. The effective conductance matrix, conductance changes relative to the initial state, positive-core conductance variation, negative-core conductance variation, and the corresponding distributions of conductance changes are shown.

S19 Conductance Drift in FC3

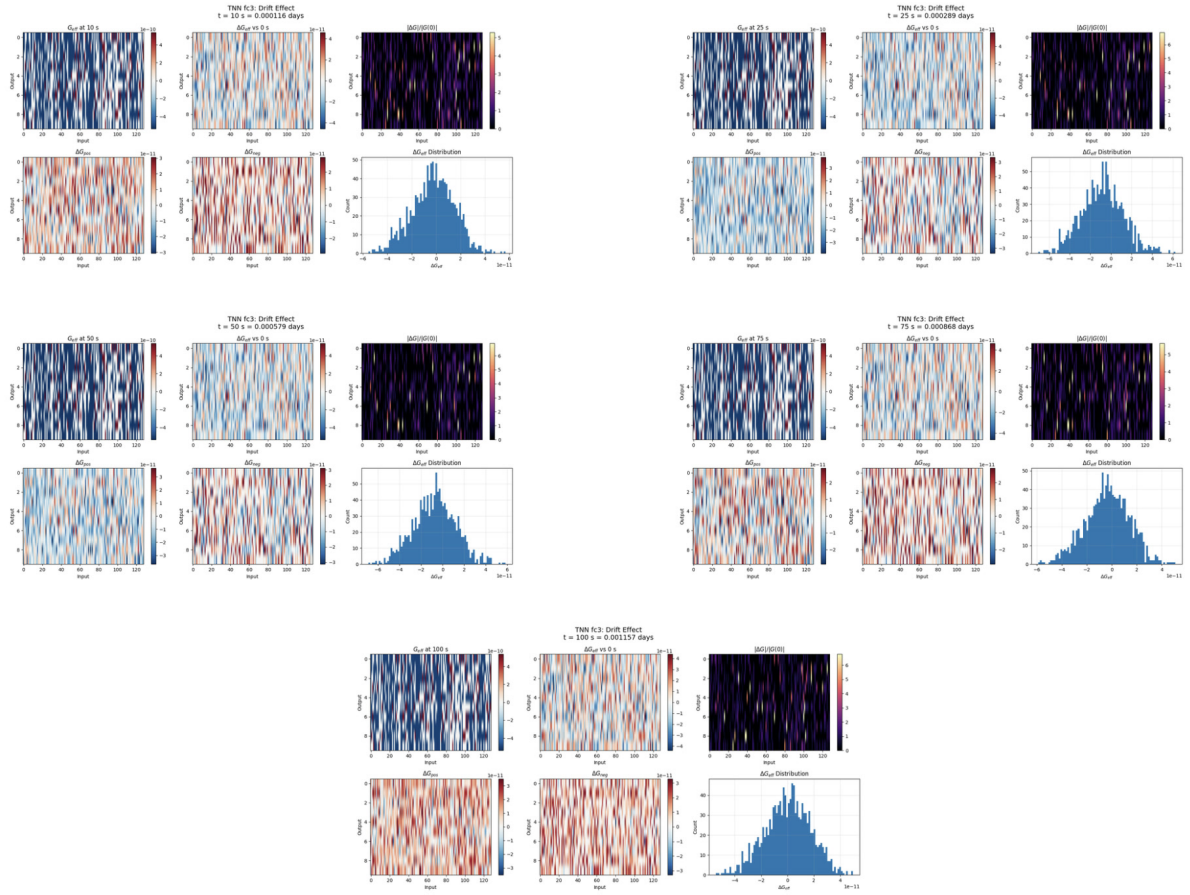


Fig. S19. Drift-induced conductance variations in the third network layer (FC3) at retention times of 10, 25, 50, 75, and 100 s. The effective conductance matrix, conductance change relative to the initial state, positive-core conductance variation, negative-core conductance variation, and the corresponding distributions of conductance changes are shown.

S20 Effect of Parasitic Resistance

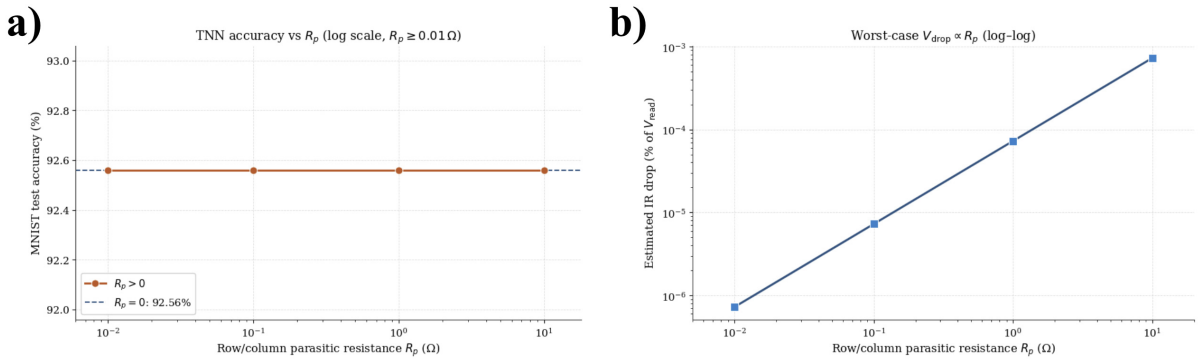


Fig. S20. Impact of parasitic wire resistance on TNN performance. (a) Classification accuracy as a function of row/column parasitic resistance. (b) Estimated worst-case IR-drop as a function of parasitic resistance.

S21 Device-Aware Training

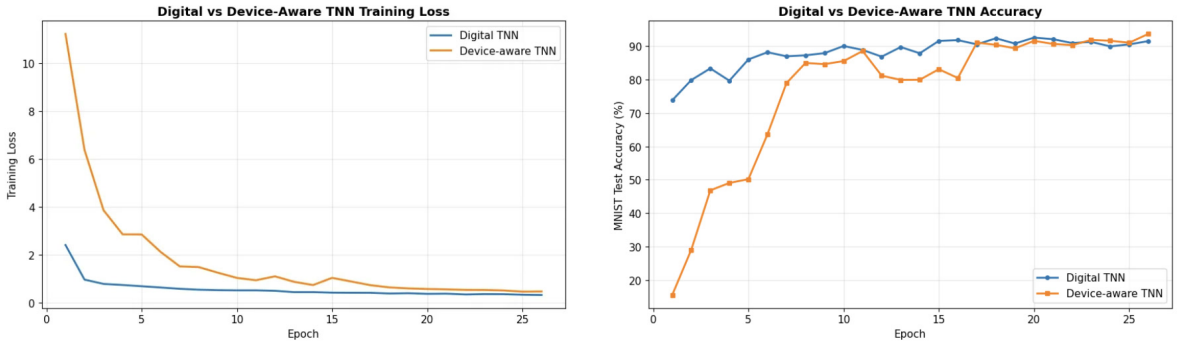


Fig. S21. Training evolution of the digital TNN and the device-aware TNN.