Neuroinformatics - heterogeneous SNN

Prashubh Atri, pa2594@nyu.edu, Yixing Wang, yw8380@nyu.edu, Wenqi Xu, wx2178@nyu.edu

Contents

1 Introduction and background

1.1 Problem statement

Spiking Neural Network (SNN), as a potential alternative to the traditional artificial neural network (ANN), is gaining more attention for its sparse computation and temporal dynamics.[**Shen24**] However, training SNNs in supervised settings remains challenging because spike-train data is non-differentiable and cannot be processed with conventional backpropagation. We want to apply evolutionary algorithm to SNN training and aim for better model performance.

1.2 Importance of the problem

Solving this training problem is not only crucial for improving the performance of neuromorphic hardware (which relies on SNNs), but also for deepening our understanding of how biological neural circuits can learn complex functions using only spikes.[11] Effective training algorithms would allow SNNs to tackle a wider range of machine learning tasks with fewer computational resources.

1.3 Background: Existing Approaches

1.3.1 Surrogate Gradients

The dominant approach for supervised SNN training has been the surrogate gradient (SG) learning. SG approximates the non-existent derivative of spikes using smooth, differentiable surrogate functions, allowing gradient descent to proceed. Zenke and Vogels demonstrated that SG learning is robust across different surrogate function choices and tasks like spike-based classification. However, it still inhibits weaknesses such as exploding / vanishing gradients and inefficiency in long temporal sequences.

1.3.2 Evolutionary Algorithms

Recently, Evolutionary Algorithms (EA) have been proposed as alternatives to backpropagation, which is biologically plausible and computationally robust. Unlike BP, EAs do not require differentiability; instead, they explore the parameter space using stochastic sampling and fitness evaluation. EA has the advantages of needing only forward computation and optimizing both synaptic weights / connections and neuron-intrinsic parameters at the same time. Hazelden et al. applied EAs to train biophysical neural networks (BNNs) and neural ODEs, demonstrating that EAs can successfully train networks composed of stiff, nonlinear, spiking neurons where backpropagation often fails.[6]

1.4 Scientific and Methodological Gain

- We will have a better assumption about how real neural circuits could possibly learn without doing as much calculation as ML usually requires on GPU.
- EA offers a different optimization method not relying on differentiability, and could align well with biophysically detailed models.
- EA-SNN training pipelines could make SNNs more competitive on real-world tasks, especially when implemented on neuromorphic hardware.

1.5 Challenges and Time To Solve Them

Challenges:

- **Non-differentiability of spikes**: Spiking neurons generate discontinuous, all-or-none events, on which we can't do conventional gradient descent.[11]
- **Temporal credit assignment**: SNNs is a time-series data, meaning that learning requires assigning credit for errors to events that occurred at earlier timesteps, and is costly.[11]

- **Model stiffness**: Biophysical neuron models, such as Hodgkin-Huxley and Morris-Lecar, exhibit highly stiff dynamics. In these models, small changes in input can trigger rapid, nonlinear transitions, such as spike initiation, making gradient-based training unstable.[6]
- **Computational cost**: Both surrogate gradient (SG) methods and evolutionary algorithms (EA) have historically suffered from high computational cost.[6]

Time to tackle the challenges:

- **Neuromorphic hardware advancements**: Modern neuromorphic chips, such as Intel Loihi, incentivizes research on efficient large-scale simulation of SNNs more.
- **Algorithmic innovations**: Recent developments in evolutionary strategies (ES), such as mirrored sampling and adaptive noise scaling, have greatly improved the efficiency and convergence speed of EA-based optimization. These techniques reduce the number of evaluations needed to estimate gradients, making EAs more practical for training complex SNNs.

1.6 Proposed Approach

This project proposes to develop and evaluate evolutionary algorithm (EA)-based training on toy Spiking Neural Network (SNN) tasks, comparing it with surrogate gradient (SG) and more baseline methods. The goal is to assess:

- How well EA can optimize network weights.
- Whether EA can jointly optimize both weights and neuron-intrinsic parameters (e.g., firing thresholds, time constants).
- How EA-trained SNNs in general compared to MLP/CNN/RNN/SG/previous EA-based SNN.

The approach starts with simple synthetic datasets to establish benchmarks and gradually progresses to more complex tasks. This pipeline will apply existing EA methods, for example, evolutionary strategies (ES) developed by Salimans[6], but adapted to SNNs' spike sparsity and energy efficiency.

2 Datasets

2.1 Synthetic Smooth Random Manifold Spiking Data Set

2.1.1 Simulated Data Generation

The main focus of our project is to apply the evolutionary algorithm on training Spiking Neural Networks (SNNs). To assess if SNNs could learn to categorize spike patterns and generalize to unseed patterns, we generated synthetic classification data sets with added temporal structure, following the methods developped by Zenke and Vogels[**Zenke21**].

Currently, there are only a few established benchmarks for SNNs. We selected Zenke and Vogels' method as our primary data source because it not only provides a sufficient amount of data for experimentation but also capitalizes on the ability to encode information in spike timing, an important aspect of spiking processing.

The process of generating simulated data for Spiking Neural Networks (SNNs) follows several key steps:

Step 1: Define the Smooth Random Manifold To generate structured spiking data, we define a smooth random manifold of dimension D embedded in a higher-dimensional space M. The manifold function is represented using a Fourier basis:

$$f_i(\mathbf{x}) = \prod_{j\in D} \sum_{k=1}^{n_{\text{cutoff}}} \frac{1}{k^\alpha} \theta^A_{ijk} \sin\left(2\pi\left(kx_j\theta^B_{ijk} + \theta^C_{ijk}\right)\right) \tag{1}$$

where:

- $\theta^A, \theta^B, \theta^C$ are random parameters drawn from a uniform distribution $U(0, 1)$.
- $n_{\text{cutoff}} = 1000$ determines the highest frequency component.
- α controls the smoothness of the manifold: larger α results in a smoother manifold, while smaller α allows more high-frequency variations. Figure 1

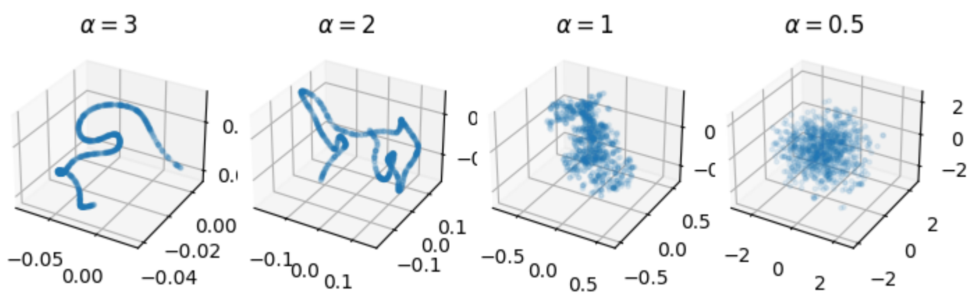

Figure 1: Four one-dimensional example manifolds for different smoothness parameters α in a three-dimensional embedding space. From each manifold, we plotted 1000 random data points.[**Zenke21**]

Step 2: Sample Points from the Manifold To generate spike-timing data:

1. Sample random points $\mathbf{x}$ from a D-dimensional unit hypercube.
2. Map each sampled point $\mathbf{x}$ to an M-dimensional embedding space using the manifold function:

$$\mathbf{t} = f(\mathbf{x}) \tag{2}$$

   where $\mathbf{t}$ represents the firing times of neurons.

Step 3: Convert to Spiking Data The output coordinates $\mathbf{t}$ are standardized to lie within a predefined time range $[0, \tau_{\text{randman}}]$, ensuring that:

$$t_i^c = f_i^c(X), \quad X \sim U(0, 1) \tag{3}$$

Each t_i represents the firing time of neuron i. This ensures that neurons fire exactly once, making the dataset purely spike-timing dependent.

Step 4: Generate Classification Labels Each random manifold corresponds to a different class:

- If two points belong to the same manifold, they receive the same class label.
- The number of manifolds determines the number of classes.

Step 5: Partition Data for Training and Testing The generated data is split into:

- **Training set:** 800 samples per class
- **Validation set:** 100 samples per class
- **Test set:** 100 samples per class

To summarize, the complexity of the generated data is controlled by the following parameters:

- **D**: The intrinsic dimensionality of the underlying time pattern.
- $\boldsymbol{\alpha}$: A larger value of α yields a smoother manifold, resulting in more predictable and structured data.
- **M**: The number of neurons. Each neuron fires exactly once in the M-dimensional manifold, and each coordinate corresponds to the firing time of one neuron.

The generated data consists of **features** X, an $M \times \tau_{\text{randman}}$ matrix representing the spike trains of M neurons, and **results** y, which correspond to the class labels.

2.1.2 Manifold Visualization

Q1: What each visualization represents?

A: The meaning of parameters are introduced in part 1. Each dot in the trajectory / hyperplane represents one sample. The coordinates of the dot represent firing times of 3 neurons (M = 3 on this 3D manifold). The color of the dot represents its class label.

Q2: Why chose manifold for visualization?

A: Manifolds are chosen because they provide a structured, lower-dimensional representation of high-dimensional data. It shows different classes and spike-time patterns for thousands of samples well.

Q3: Key observations or insights gained?

A: As this is a generated toy dataset, we can only say that it encodes spike timing information and is good for training and testing SNNs.

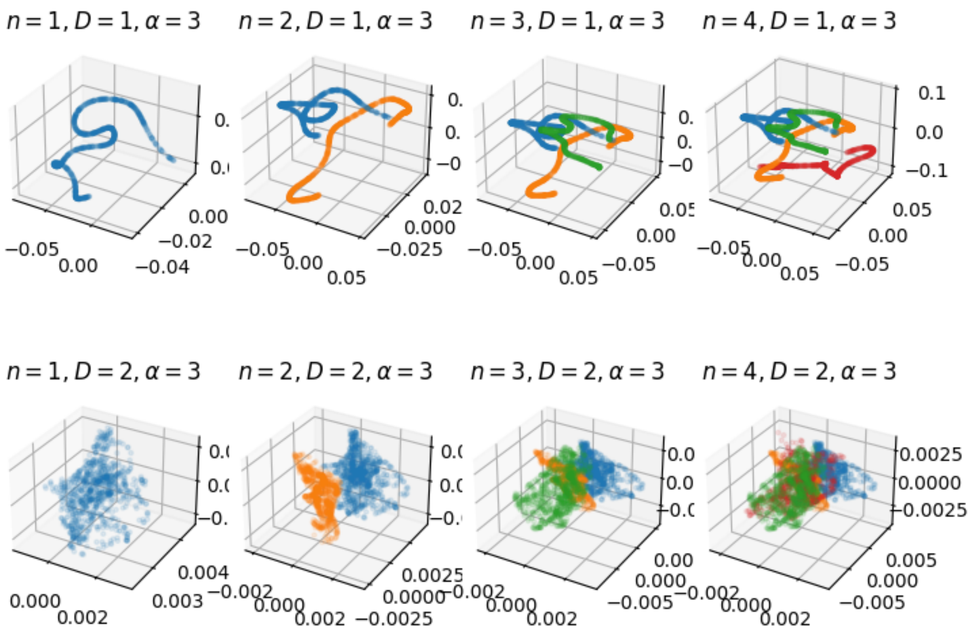

Figure 2: Same as in Figure 1, but keeping α fixed while changing the manifold-dimension and the number of random manifolds (different colors). Here, n means the number of classes. By sampling different random manifolds, it is straight-forward to build synthetic multiway classification tasks.[**Zenke21**]

2.1.3 Raster Plots

Raster plots are a very informative visualization technique of neural activity. They allow us to easily inspect the neural activity over time span on a single as well as on all channels in a very intuitive way. [7]

A raster plot is a graphical representation of neural spike activity over time. It visualizes when individual neurons fire by plotting spike events as discrete points or vertical lines.

- X-axis represents time.
- Y-axis represents different neurons (or trials in repeated experiments).
- Each dot (or tick mark) represents a spike event—the moment a neuron fires.

In this project, we use raster plots matrix as standard data input. And our first step is to convert different types of datasets into raster plots. The figure below shows sample synthetic data with parameters D = 1, α = 2, M = 30.

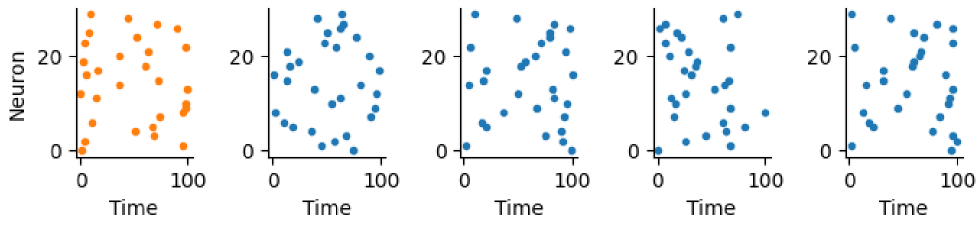

Figure 3: Spike raster plots corresponding to 5 samples along two different intrinsic manifolds (orange vs blue), with parameters D = 1, α = 2, M = 30.

2.2 Real World Data

2.2.1 MNIST Handwritten Digit Data Set

Introduction: The MNIST dataset consists of 28×28 grayscale images of handwritten digits from 0 to 9, widely used for benchmarking machine learning models. Traditional artificial neural networks (ANNs) have demonstrated excellent performance on MNIST classification tasks. However, the application of Spiking Neural Networks (SNNs) introduces a new approach to processing such data using spike-based representations. Zenke and Vogels (2021) demonstrated that converting MNIST images into spike trains allows for effective training of SNNs using surrogate gradient learning, but the performance still lags behind conventional ANNs due to limitations in encoding mechanisms.

Data Generation: The MNIST (Modified National Institute of Standards and Technology)[8] dataset was created by Yann LeCun, Corinna Cortes, and Christopher Burges in the 1990s as a benchmark for machine learning algorithms. It was derived from the NIST database[9] , which originally contained handwritten digits collected from U.S. Census Bureau employees and high school students. The dataset was preprocessed to ensure uniform size (28×28 pixels) and centered digits, making it widely used for training and evaluating image classification models.

Data Formatting: The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0-9), with 60,000 images used for training and 10,000 for testing. Each image is 28×28 pixels, where pixel values range from 0 (black) to 255 (white)[3] . The dataset is widely used in machine learning, particularly for benchmarking classification models. The labels (0-9) correspond to the digit depicted in each image.

Converting to Spiking Data: To use MNIST data with SNNs, we convert pixel intensities into spike trains. One common approach is *latency encoding*[10, 11] , where brighter pixels fire earlier within a predefined time window (e.g., 50ms). Each neuron corresponds to a pixel, and its spike timing is determined as:

$$t_{spike} = \tau_{max}\left(1 - \frac{I}{I_{max}}\right)$$

where I is the pixel intensity, I_{max} is the maximum intensity (255 for 8-bit images), and τ_{max} is the maximum allowed spike time.

Data Splits: The MNIST dataset is split into:

- **Training set:** 51,000 samples (85% of the dataset)
- **Validation set:** 3,000 samples (5% of the dataset)
- **Test set:** 6,000 samples (10% of the dataset)

This split ensures that the majority of the data is used for model training while maintaining a small validation set for hyperparameter tuning and a separate test set for final evaluation[1].

Data Visualization: In this section, we have displayed visualizations of the MNIST dataset to illustrate the structure and variety of the handwritten digit images. The dataset consists of 10 classes, representing digits from 0 to 9. To provide a comprehensive overview, we have selected one representative example from each class and plotted them in a grid format. This visualization allows for a clear understanding of the variations in handwriting styles across different samples, demonstrating the diversity of the dataset.

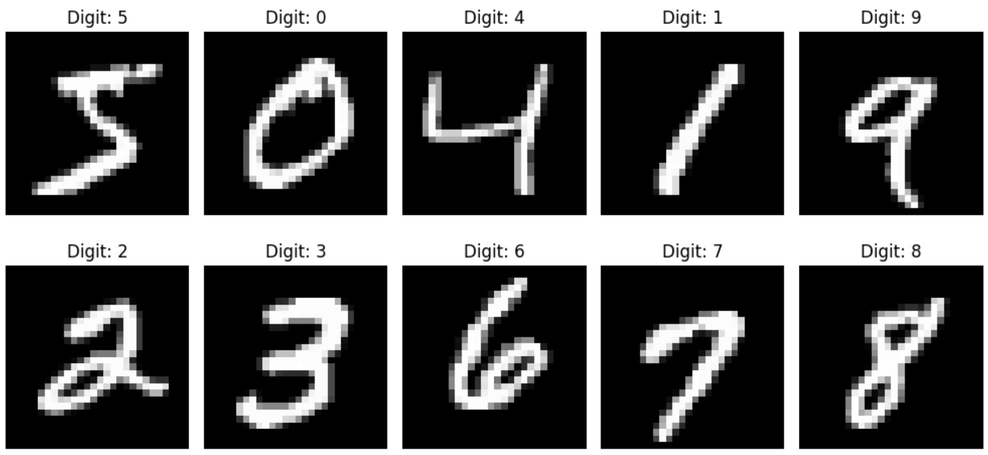

Figure 4: Visualization of one example from each digit class (0-9) in the MNIST dataset, showcasing the diversity in handwriting styles.

Following the visualization of raw MNIST images, we have transformed each example into its corresponding spike train representation using latency encoding. In this encoding scheme, brighter pixels fire earlier within a predefined time window, converting the static image into a temporal spike-based representation. The raster plots for all 10 digit classes illustrate how the original images are mapped into spiking activity, which is crucial for processing information in Spiking Neural Networks (SNNs). This step bridges the gap between conventional image-based machine learning and neuromorphic computing[4].

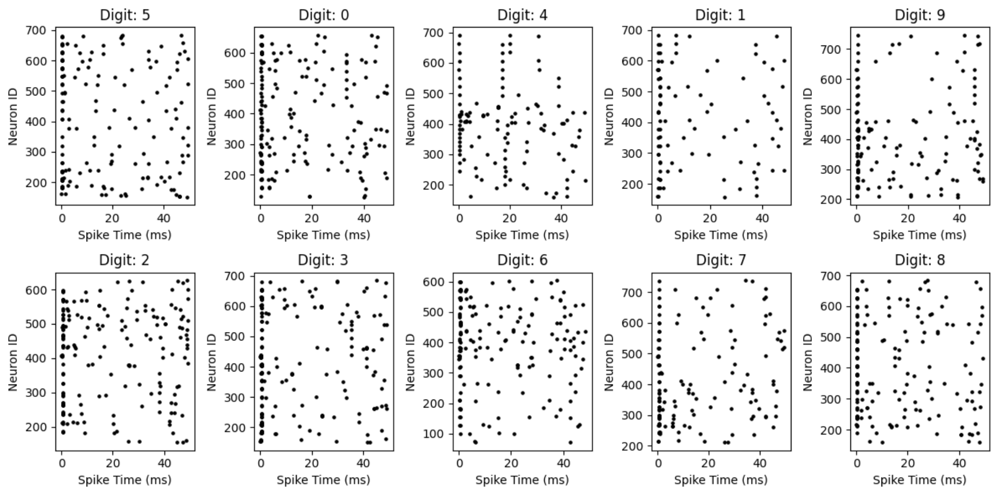

Figure 5: Spike train representations of MNIST digit examples using latency encoding, where brighter pixels fire earlier within a predefined time window.

Additionally, we have displayed a raster plot comparing two different examples of the same digit class, demonstrating how variations in handwriting affect their spike train transformations. By plotting both examples on the same graph using different colors, it becomes evident how differences in pixel intensity distributions lead to variations in spike timing. This comparison helps in understanding how SNNs perceive and differentiate variations of the same digit, highlighting the importance of encoding strategies in neuromorphic computing.

Observations:

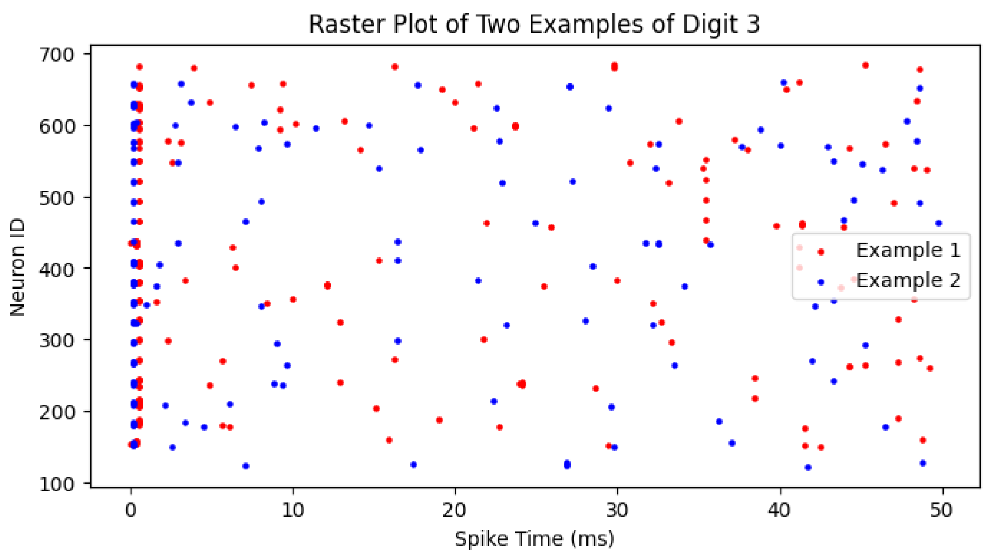

Figure 6: Raster plot comparing two different examples of the same digit class, illustrating variations in spike timing due to differences in handwriting styles.

- **Image Representation:** The original MNIST images provide a structured, human-interpretable representation of handwritten digits.
- **Spike Train Encoding:** The raster plot demonstrates how pixel intensities are transformed into spike timings, highlighting the temporal structure used in SNN processing.
- **Comparison with Other Datasets:** Unlike the Spiking Heidelberg Digits (SHD) dataset, which naturally encodes temporal sequences, MNIST requires artificial encoding mechanisms that may not fully exploit SNN capabilities.

By integrating MNIST into SNN frameworks, researchers can explore how surrogate gradient learning and neuromorphic computing paradigms impact image classification tasks.

Questions:

- **Visualization of MNIST Dataset:** This visualization displays one representative example from each digit class (0-9) in the MNIST dataset. The purpose is to illustrate the diversity in handwriting styles and variations in digit representation.

  Why this method: A grid layout of digit images provides a concise and intuitive way to compare different digits while ensuring a comprehensive overview of the dataset's diversity. This visualization serves as the foundation for understanding how the original data is structured before conversion into spike trains.

  Key insights:

  - The MNIST dataset contains a wide range of handwriting styles, with some digits exhibiting more variation than others.
  - Certain digits, such as "1" and "7," may have similarities in certain cases, which could impact classification accuracy.
  - This highlights the importance of encoding strategies when transitioning from traditional deep learning methods to neuromorphic computing.

- **Visualization of MNIST Spike Trains:** This visualization shows the spike train transformation of each MNIST digit using latency encoding, where pixel intensity determines spike timing. The raster plots illustrate how the static images are converted into a temporal representation that SNNs can process.

Why this method: Raster plots are a standard method for visualizing neural spike activity over time, making them the best choice for showing how pixel intensities translate into spike timing. This approach effectively demonstrates how SNNs encode visual information differently from conventional deep learning models.

Key insights:

- Digits with denser strokes (e.g., "8" and "9") generate more spike events across neurons, while simpler digits like "1" result in fewer spikes.
- This suggests that SNNs will process different digits with varying levels of neural activation, which may affect classification efficiency.

• **Comparison of Two Spike Train Examples of the Same Digit:** This visualization compares the spike train representations of two different instances of the same digit, plotted on the same raster graph in different colors. It highlights the variations in spike timing caused by differences in handwriting styles.

Why this method: By overlaying spike train transformations of two different examples of the same digit, we can observe how subtle variations in stroke thickness, slant, and digit formation impact the spike-based encoding process. This is crucial for analyzing how SNNs handle intra-class variability.

Key insights:

- Even within the same digit class, variations in handwriting significantly impact spike timing and neuron activation patterns.
- Some digits may have overlapping spike patterns, which could pose challenges in classification tasks.
- This emphasizes the need for robust encoding mechanisms and adaptive learning strategies in neuromorphic computing.

These visualizations collectively provide a clear understanding of how MNIST images are represented, transformed into spike-based data, and processed by SNNs. They highlight key challenges and insights necessary for optimizing spike-based neural networks.

2.2.2 Spiking Heidelberg Digits (SHD) Data Set

Introduction Despite the ability to classify MNIST dataset, SNN did not outperform conventional artifical neural networks. Zenke and Vogels speculated that the encoding of each MNIST pixel into a single spike did not take advantage of SNN's longer time scale dynamics, which motivated them to experiment with the Spiking Heidelberg Digits (SHD) Dataset with the nature of having long time scale [**Zenke21**].

Data Generation SHD Dataset was generated from an artificial cochlea, an organ in the inner ear transducing sound waves into electric signals. Specifically, the artificial cochlea listened to the pronunciations of the digits zero to ten in both English and German, hence 20 classes in the dataset. Through out the listening, the output of the cochlea is received by 700 simulated bushy cells (a type of neuron). The dataset is a recording of the firing time of those bushy cells. Instead of the voltage of each cell, the dataset simply keeps track of when does a cell fire [2].

Data Organization The SHD dataset is stored in Hierarchical Data Format (hdf5), which is useful to deal with large dataset since it allows to access only truncates of data by copying it to memory, while keeping the unused data in hard drive. The `pytable` package is used to handle the hdf5 file. The data organization is shown in Figure 7. Each sample consists of the firing times of the 700 cells during listening to a word. Therefore, each sample is represented as two arrays of same length, one for firing time and another for the ID of the firing cell. Since different words have different length, samples are represetned as arrays of different length. In `spikes` group, therefore, `times` (representing time t when a neuron fires) and `units` (representing ID of the firing neuron at time t) have their sizes of the first dimension same as number of samples. Their second dimensions, however, have different sizes across different samples. The `labels` group records the ID of the presented word for each sample, which can be mapped to the string of the word using `extra.keys` group[2].

SHD consists of 12 speakers, each repeated every digit about 40 times. The datset came with 8332 training samples and 2088 test samples, without any validation set. The test set consists of two speakers not included in training set and 5% of the recordings of each digit and language of all other speakers[2].

```
root
├── spikes
│   ├── times[][] ..... VLArray of Arrays holding spike times
│   └── units[][] ...... VLArray of Arrays holding spike units
├── labels[] .............................. Array of digit IDs
└── extra
    ├── speaker[] ...................... Array of speaker IDs
    ├── keys[] .............. Array of digit description strings
    └── meta_info
        ├── gender[] ................ Array of speaker genders
        ├── age[] ....................... Array of speaker ages
        └── body_height[] ...... Array of speaker body heights
```

Figure 7: Hierarchical organization of the SHD Dataset.

Data Visualization Similar to the other datasets, each sample can be visualized using raster plots shown in Figure 8. It can be observed from the plot that the firing of neurons tend to be adjacent in their ID, which is possibly because their IDs of the neurons are related to the frequency to which they are tuned, similar to biological bushy cells. It can also been seen from the plots that SHD Dataset is different from the upper-mentioned data in that they have longer time scale. In addition, noise are presented in the data, which is shown as the sparse dots in the plots.

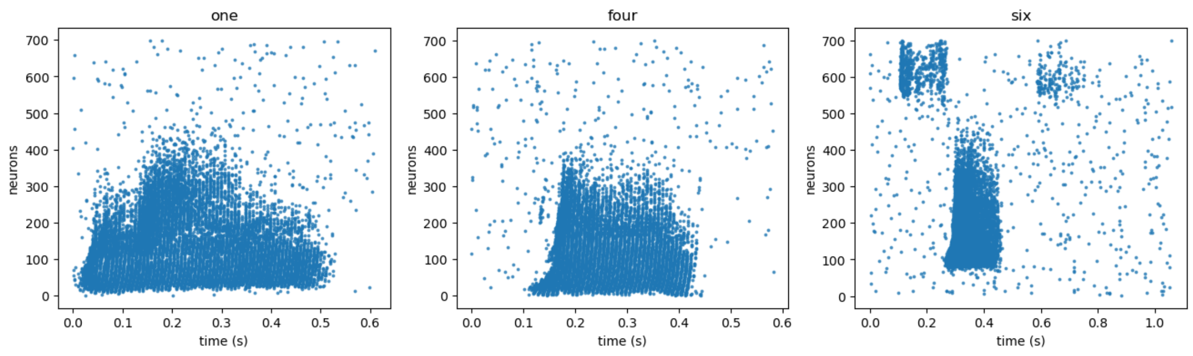

Figure 8: Raster plots of the bushy cells when they heard "one", "four", and "six".

2.2.3 More Neuromorphic Benchmark Datasets

In addition to what we mentioned above, we would also refer to these datasets if needed:

- ASL-DVS
- CIFAR10-DVS
- ES-ImageNet
- DVS128 Gesture
- HARDVS

3 Baseline Models

3.1 Traditional Artificial Neural Network

3.1.1 Multi-Layer Perceptron (MLP) for Synthetic Manifold Data

The synthetic smooth random manifold spiking dataset is structured such that each neuron fires exactly once at a timing determined by the coordinates of the manifold in the embedding dimension. A traditional Multi-Layer Perceptron (MLP) serves as a baseline method for classification on this dataset, as it allows direct processing of the manifold-based feature representations.

MLPs work on vectorized representations of input data and consist of multiple fully connected layers with nonlinear activation functions. Here, the input to the MLP is a flattened vector representation of the spike times, ensuring a direct mapping between spike-based features and class labels. The network follows:

- **Input Layer:** The firing times of M neurons (e.g., $M = 32$) are fed as a single feature vector.
- **Hidden Layers:** For this baseline model, we implemented one fully connected layers with ReLU activation allow non-linear feature extraction.
- **Output Layer:** A final softmax layer is used to classify the input into one of the predefined classes. (For this baseline classification model, we didn't implemented softmax).

Equation for Forward Pass in MLP

$$h^{(l)} = \sigma(W^{(l)}h^{(l-1)} + b^{(l)}) \tag{4}$$

Where,

$$\begin{aligned} h^{(l)} &: \text{Activation of layer } l \\ W^{(l)} &: \text{Weight matrix for layer } l \\ b^{(l)} &: \text{Bias vector for layer } l \\ \sigma(\cdot) &: \text{Activation function (e.g., ReLU)} \end{aligned}$$

Equation for Output Layer in MLP

$$\hat{y} = \text{softmax}(W^{(o)}h^{(L)} + b^{(o)}) \tag{5}$$

Where,

$$\begin{aligned} \sigma(\cdot) &: \text{Activation function (e.g., ReLU)} \\ \hat{y} &: \text{Predicted output} \\ L &: \text{Final hidden layer} \end{aligned}$$

Since the dataset follows a structured manifold, the MLP baseline helps determine whether standard vector-based learning is sufficient for classification, serving as a comparison point for SNN-based models.

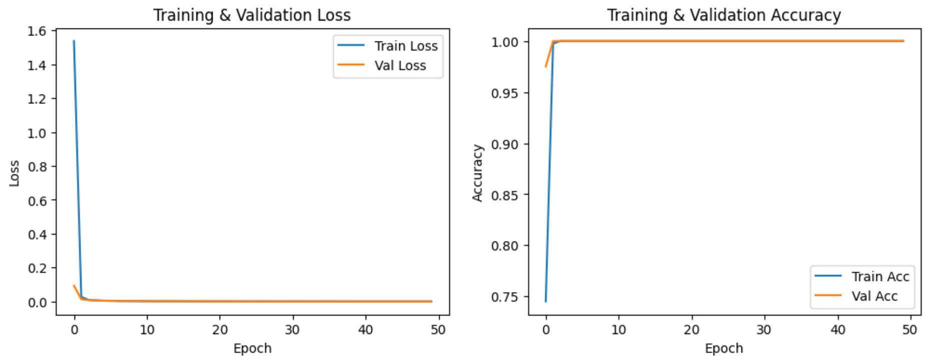

Figure 9: Vanilla MLP learnt Randman data (D = 2, $\alpha = 2, M = 30, class = 2$)

3.1.2 CNN for MNIST

For the MNIST dataset, a Convolutional Neural Network (CNN) is implemented as the baseline model. CNNs excel in image-based tasks due to spatial feature extraction via convolutional layers. The architecture includes:

- **Convolutional Layers:** Extract spatial features using 3×3 or 5×5 kernels.
- **Pooling Layers:** Apply max-pooling to reduce spatial dimensions while retaining key features.
- **Fully Connected Layers:** Map extracted features to class probabilities via a softmax classifier.

Convolutional Operation in CNN

$$z_{i,j}^{(l)} = \sum_m \sum_n W_{m,n}^{(l)} x_{(i+m),(j+n)}^{(l-1)} + b^{(l)} \tag{6}$$

Where,

$$\begin{aligned} z_{i,j}^{(l)} &: \text{Activation at position } (i,j) \text{ in layer } l \\ W_{m,n}^{(l)} &: \text{Filter/kernel of size } (m,n) \text{ in layer } l \\ x_{(i+m),(j+n)}^{(l-1)} &: \text{Pixel value at position } (i+m, j+n) \text{ in previous layer} \\ b^{(l)} &: \text{Bias term in layer } l \end{aligned}$$

Pooling Operation (Max-Pooling)

$$p_{i,j}^{(l)} = \max_{m,n} z_{(i+m),(j+n)}^{(l)} \tag{7}$$

Where,

$$\begin{aligned} p_{i,j}^{(l)} &: \text{Max-pooled value at } (i,j) \text{ in layer } l \\ \max_{m,n}(\cdot) &: \text{Max-pooling operation over filter size } (m,n) \end{aligned}$$

The CNN baseline ensures fair performance comparison between conventional deep learning methods and SNN-based training.

3.1.3 RNN for Spiking Heidelberg Dataset

Since the SHD dataset is itself a form of time series, a natural choice is to use a recurrent neural network (RNN) as a baseline model since it involves memory. The baseline model is the most ordinary RNN. Mathematically, each RNN unit is characterized by its hidden state h_t at time t,

$$h_t = tanh(W_i x_t + W_h h_{t-1})$$

where W_i and W_h are the weight matrices associated with input and hidden state. Therefore, at each time step, the new state of the unit is influenced by the previous state h_{t-1} which encapsulates information from the past, and the current input x_t, which for this dataset is the vector indicating which neurons are spiking.

After the RNN unit went through the whole time series of spiking pattern x, a fully connected layer will read out from the hidden state for classification:

$$\hat{y} = \arg\max_i(W_c h)$$

This way of using hidden state in RNN in a forward pass can be compared to SNN where the hidden state $V(t)$ is the membrane potential of the cell, which integrates the input to the cell over time:

$$\tau V(t) = -(V(t) - V_\infty) + I$$

where I is the input to the neuron, and V_∞ is the steady state of the neuron.

The learning rule of RNN is also an interest for comparison with that of SNN. Specifically, RNN is designed to learn with back propagation, where gradients can be regularized to avoid gradient explosion, whereas the spiking output of SNN has large gradient problem.

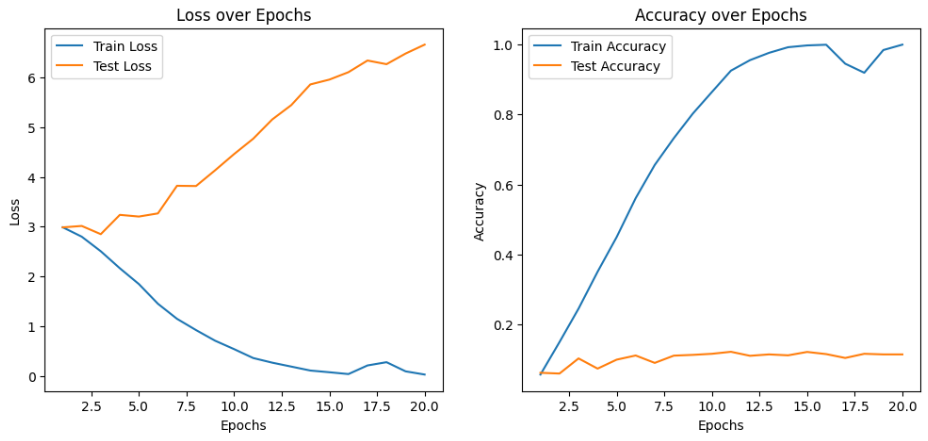

Figure 10: vanilla RNN overfitted on SHD training

3.2 Surrogate Gradient Descent

Surrogate gradient descent is applied to train Spiking Neural Networks (SNNs) on all three datasets:

3.2.1 SNN for Synthetic Manifold Data

- The manifold dataset represents spike-timing-based information, meaning that training an SNN directly on spike-based inputs can better exploit temporal structure compared to MLPs.
- We implement a Leaky Integrate-and-Fire (LIF) network where input spike times are converted into spike trains over multiple timesteps.
- The SNN learns through surrogate gradient backpropagation, where a smooth function (e.g., sigmoid or piecewise linear) approximates the non-differentiable spike function.

3.2.2 SNN for MNIST

- The MNIST dataset requires encoding static images into spike trains before processing with SNNs.
- We use latency coding, where pixels with higher intensity fire earlier, creating a spike-based input representation.
- The SNN consists of convolutional layers followed by LIF neurons, ensuring temporal dynamics are leveraged.

3.2.3 SNN for SHD

- The Spiking Heidelberg Digits (SHD) dataset consists of naturally occurring spike-based representations (recorded from an artificial cochlea).
- Unlike MNIST, SHD does not require conversion from static data but needs an SNN model with recurrent connections to handle longer time scales.
- Recurrent SNNs (RSNNs) or feedforward LIF models are trained using backpropagation through time (BPTT) with surrogate gradients.

In all three cases, the surrogate gradient method allows gradient-based learning in non-differentiable spike-based systems, making SNNs trainable in a supervised setting.

Membrane Potential Dynamics in LIF Neurons

$$\tau_m \frac{dV(t)}{dt} = -(V(t) - V_{\text{rest}}) + I(t) \tag{8}$$

Where,

$$\begin{aligned} V(t) &: \text{Membrane potential at time } t \\ \tau_m &: \text{Membrane time constant} \\ V_{\text{rest}} &: \text{Resting potential of the neuron} \\ I(t) &: \text{Input current at time } t \end{aligned}$$

Surrogate Gradient for Non-Differentiable Spikes

$$\frac{dS}{dV} \approx \sigma_\beta(V) = \frac{1}{1 + e^{-\beta V}} \tag{9}$$

Where,

$$\begin{aligned} S &: \text{Spike output (binary value)} \\ \sigma_\beta(V) &: \text{Surrogate gradient function with steepness parameter } \beta \\ \beta &: \text{Steepness parameter controlling smoothness of the approximation} \end{aligned}$$

Backpropagation Through Time (BPTT) for SNNs

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_t \frac{\partial \mathcal{L}}{\partial S_t} \frac{\partial S_t}{\partial V_t} \frac{\partial V_t}{\partial W} \tag{10}$$

Where,

$$\begin{aligned} \mathcal{L} &: \text{Loss function} \\ W &: \text{Synaptic weights} \\ S_t &: \text{Spike output at time } t \\ V_t &: \text{Membrane potential at time } t \\ \frac{\partial \mathcal{L}}{\partial W} &: \text{Gradient of the loss function with respect to weights} \\ \frac{\partial S_t}{\partial V_t} &: \text{Gradient of spike output with respect to membrane potential} \\ \frac{\partial V_t}{\partial W} &: \text{Gradient of membrane potential with respect to synaptic weights} \end{aligned}$$

3.3 Evolutionary Algorithm

3.3.1 Evolutionary Strategy

The specific evolution algorithm to be implemented as a baseline model is the evolutionary strategy (ES) studied in [6]. In general, a model, with θ as parameter and x as input from the sample space D, computes its output represented as $N(x, \theta)$. The learning process is then the search of a parameter to minimize the sum of loss function L over all the samples:

$$\arg\min_{\theta \in \mathbb{R}^m} \sum_{(x,y) \in D} L(N(x, \theta), y)$$

The ES algorithm, however, instead of treating the parameter as a scalar variable, views it as a random variable with a distribution p_θ in $\mathbb{R}^m$ where m is the number of parameters. The distribution is typically assumed to be normal $\mathcal{N}(\theta, \sigma)$, and the variance σ is a hyperparameter determined prior to training. Then the loss function can be defined as the expected value of the loss given the random variable θ:

$$\begin{aligned} L_{p_\theta}(x) &= \underset{v \sim p_\theta}{E}[L(N(x, v), y)] \\ &= \int_{v \in \mathbb{R}^m} p_\theta(v) \cdot L(N(x, v), y) dv \quad \text{(by definition of expected value)} \end{aligned}$$

This loss function, comparing to the traditional definition, is smoothed by weights corresponding to the probability of a specific parameter.

To minimize such loss function requires the gradient of the loss function with respect to the parameter:

$$\begin{aligned} \nabla_\theta L_{p\theta}(x) &= \nabla_\theta \int_{v \in \mathbb{R}^m} p_\theta(v) L(N(x, v), y) dv \\ &= \int_{v \in \mathbb{R}^m} \nabla_\theta p_\theta(v) L(N(x, v), y) dv \quad \text{(Leibniz rule)} \end{aligned}$$

Using the chain rule with ln as outer function, $\nabla_\theta ln(p_\theta(v)) = \frac{1}{p_\theta(v)} \cdot \nabla_\theta p_\theta(v)$, which is equivalent to $\nabla_\theta p_\theta(v) = p_\theta \cdot \nabla_\theta ln(p_\theta(v))$. Substitute it to the above equation,

$$\begin{aligned} \nabla_\theta L_{p\theta}(x) &= \int_{v \in \mathbb{R}^m} p_\theta \cdot \nabla_\theta ln(p_\theta(v)) L(N(x, v), y) dv \quad \text{(notice it is in the form of expected value)} \\ &= \underset{v \sim p_\theta}{E}[\nabla_\theta ln(p_\theta(v)) L(N(x, v), y)] \end{aligned}$$

Since the distribution of parameter $p_\theta = \mathcal{N}(\theta, \sigma)$ is determined, given each sample v, the term $\nabla_\theta ln(p_\theta(v))$ can be calculated from the distribution. The term $L(N(x, v), y)$, on the other hand, can be obtained by evaluating the model with the given x and v and applying the loss function. Since both terms can be obtained, the intergral can be approximated using monte carlo simulation. That is, to approximate the gradient $\nabla_\theta L_{p\theta}(x)$, S number of samples of parameters will be drawn from the distribution $\mathcal{N}(\theta, \sigma)$, which gives $\{v^1, \cdots, v^S\}$. Then the approximation

$$\begin{aligned} \nabla_\theta L_{p\theta}(x) &= \int_{v \in \mathbb{R}^m} p_\theta \cdot \nabla_\theta ln(p_\theta(v)) L(N(x, v), y) dv \\ &\approx \frac{1}{S} \sum_{s=1}^{S} (\nabla_\theta ln(p_\theta(v^s)) \cdot L(N(x, v^s), y)) \end{aligned}$$

gives the gradient for the loss function corresponding to distribution p_θ. With this gradient, a gradient descent optimizer can be chosen from an arsenal of the existing ones [6]. This baseline model uses `Adam`. The choices of L, N, x, y, and θ depend on the specific tasks described in 3.2 and match the ones used there.

3.3.2 Implementation of Evolutionary Strategy

The evolutionary strategy was implemented in the `SNNParam` class (see Github) which keeps track of the distributions of each element in a tensor, draws samples from their distributions, calculates the gradient for each parameter given the loss for each sample, and perform gradient descent. A convex quadratic loss surface was used to test the ES style gradient descent through sampling, along with the Adam optimizer, as shown in Figure 11.

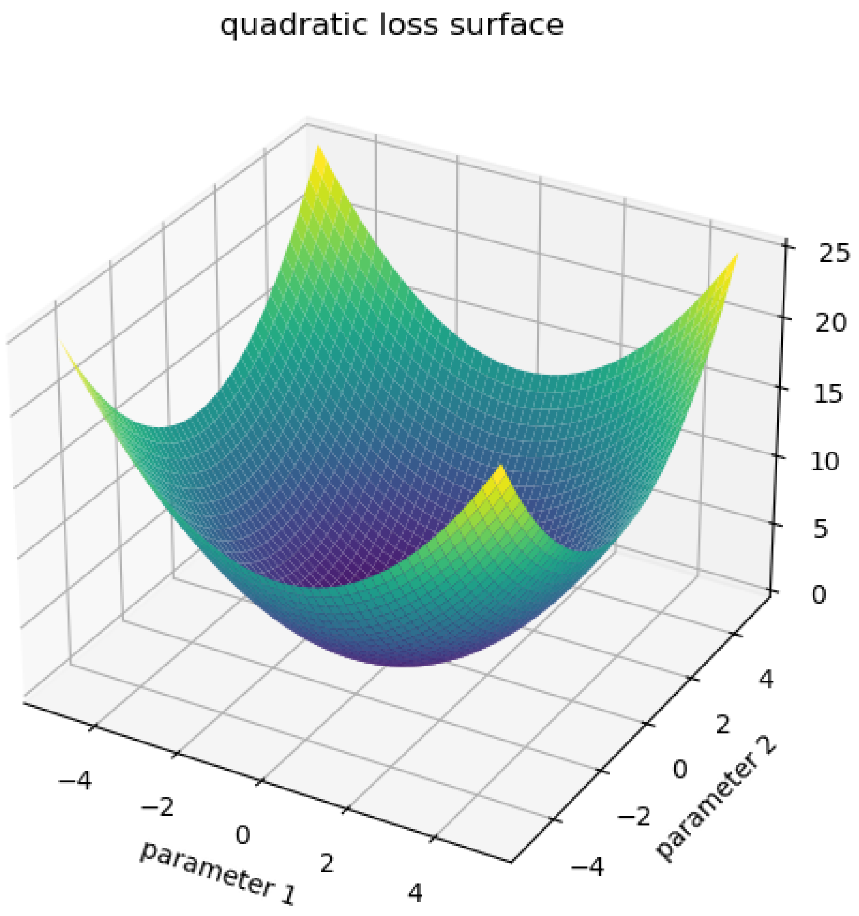

Figure 11: Dummy quadratic loss surface

As a result, Figure 12 shows that the average loss cross samples are indeed minimized. The ES algorithm is then tested on the SNN model.

3.3.3 SNN model

The model is a simple feedforward network with one hidden layer. The number of units in the input layer matches the dimensionality of the embedding dimension of the Random Manifold dataset, which in this experiment was 10. The number of hidden units might be worth investigating. In this experiment, it was chosen to be 15 with no particular reason. The last layer is the output layer, which has the number of neurons matching the number of classes and is 2 in this experiment. The two neurons in the final layer generate a spike train. There are a few options to translate the spike train to the predicted class label. One way is to choose the index of the neuron with the most number of spikes, another way, for instance, is to choose the index of the neuron with the first spike occurring. This experiment chose the first approach. In addition, CrossEntropy loss was chosen as the loss function. Since it requires logits, instead of a label, as its prediction, the spike counts for each neuron were treated as the logits for the loss function.

The dataset has 2000 samples. This stage of the project focused on the implementation of the model, therefore train-test split was not performed, thus the whole 2000 samples were treated as a training set.

Since the LIF model weights each synaptic input with a linear weight, it is implemented with a fully connected layer, of which weights are being optimized by the evolutionary strategy algorithm. In addition,

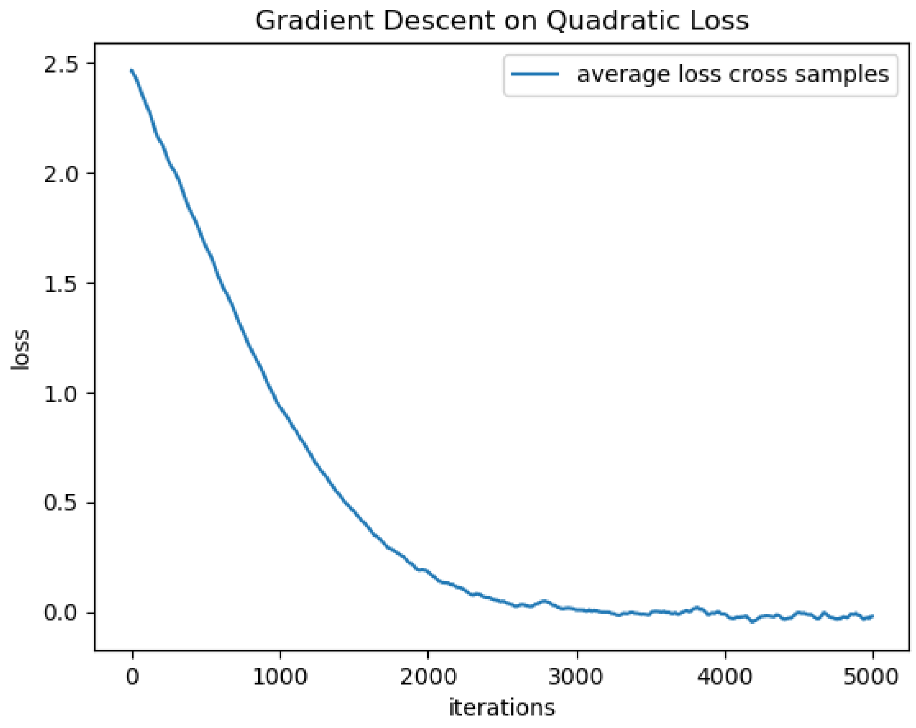

Figure 12: Gradient descent works on the quadratic loss surface

each spiking neuron has a decay rate β as a parameter to be optimized. Therefore, in this network, there are $10 \times 15 + 15 \times 2 + (10 + 5 + 2) = 197$ parameters. The optimizer was `Adam`.

At the beginning of the implementation, after drawing 20 samples, each sample ran through the whole dataset before the calculation of gradients. This consumed too much time, therefore the gradient computation and distribution updates were changed to be performed for after each batch of 256 samples.

The training result, as shown in Figure 13, evokes a mixed feeling. The blue line shows the average loss across samples throughout the training, which was indeed minimized as foretold by the quadratic loss surface test. This indicated that the evolution strategy algorithm was working as expected, which is to find the distributions of the parameters that minimize the loss. However, the accuracy metric indicated by the yellow line was hard on the eyes. It stayed at 50% which is the guessing rate for this 2 classes task. One immediate hypothesis for such an outcome is that the loss function is not a good proxy for accuracy. To test it, the original CrossEntropy loss was replaced by a "Spike Count-Based Hinge Loss", recommended by ChatGPT, whose origin was nowhere to be found on the internet. Nevertheless, the formulation makes sense to be a loss:

$$Loss = \sum_{i \neq j} max(0, S_i - S_j + \delta)$$

where S_j is the spike count of the correct class. The difference $S_i - S_j$ intended to penalize the incorrect class having more spikes than the correct one, and the term δ was to make sure at least δ more spikes were fired by the correct class than the incorrect ones. Despite the convincing formulation of the loss, it still gave the same result as the Cross-Entropy loss where accuracy did not improve from guessing.

The other speculations will be mentioned in section 6.

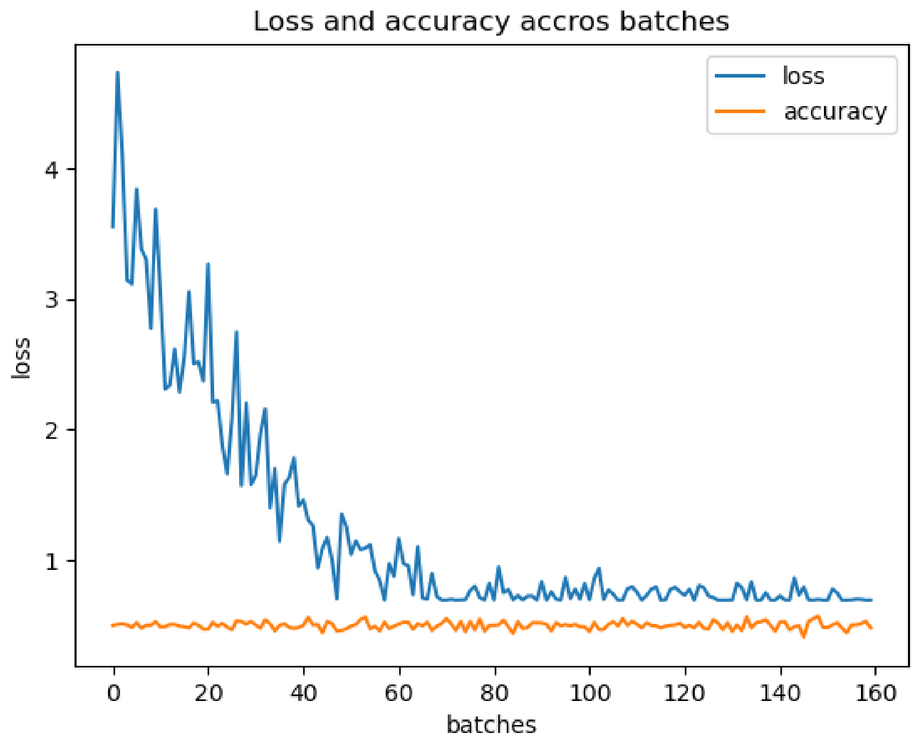

Figure 13: Loss and accuracy across batches during training

3.4 Accuracy Table for Baseline Models

Dataset	Model	Accuracy
MNIST	CNN	99.16%
MNIST	vanilla RNN	95.31%
MNIST	MLP	97.32%
MNIST	Surrogate Grad SNN	94.05%
Manifold	vanilla RNN	100%
Manifold	MLP	100%
Manifold	Surrogate Grad SNN	99.8%
Manifold	Evolution SNN	50%
Spiking Hedge Dataset	vanilla RNN	11.35%
Spiking Hedge Dataset	MLP	78.36%
Spiking Hedge Dataset	Surrogate Grad SNN	68%

Table 1: Baseline Models and Datasets

4 Preliminary Results

4.1 Custom Loss Function for Multiclass SNNs

4.1.1 Limitations of Cross-Entropy Loss in SNNs

Traditional neural networks commonly use the cross-entropy loss for classification tasks, which operates on the final output (typically logits) by selecting the maximum value across classes and computing a softmax over them. When this approach is applied to Spiking Neural Networks (SNNs), one typically computes the maximum membrane potential over time for each class and applies the cross-entropy loss on these values. However, this method has notable drawbacks in the context of SNNs:

- **Temporal Oversimplification:** Using only the maximum potential across time disregards the dynamic nature of membrane potentials, which evolve in response to incoming spikes.

- **Loss of Temporal Cues:** Intermediate and early-time discriminatory signals are lost, which may be crucial for fast and energy-efficient decision-making in SNNs.
- **Sparse Supervision:** Applying the loss only to the final outputs fails to guide the network throughout its temporal evolution.

To address these issues, we propose a custom loss function that provides dense supervision over time and encourages consistent class-wise separation of membrane potentials at each time step.

4.1.2 Mathematical Formulation of the Custom Loss

Let:

- B be the batch size,
- C the number of classes,
- T the number of time steps,
- $\mathbf{M} \in \mathbb{R}^{B\times C\times T}$ represent the membrane potential values for each sample, class, and time step,
- $\mathbf{y} \in \{0, 1, \dots, C-1\}^B$ be the vector of ground truth class labels.

For each sample $b \in \{1, \dots, B\}$ and each time step $t \in \{1, \dots, T\}$, let $M_{c,t}^{(b)}$ denote the membrane potential for class c. The membrane potential of the correct class is given by:

$$M_{\text{true},t}^{(b)} = M_{y^{(b)},t}^{(b)}.$$

We define a hinge-like loss that penalizes instances where any incorrect class has a higher potential than the correct one at any time step:

$$\mathcal{L}_t^{(b)} = \frac{1}{C-1} \sum_{\substack{c=0 \\ c\neq y^{(b)}}}^{C-1} \max\left(0, M_{c,t}^{(b)} - M_{y^{(b)},t}^{(b)}\right).$$

The total loss over the batch and all time steps is computed as:

$$\mathcal{L} = \frac{1}{B\cdot T} \sum_{b=1}^{B} \sum_{t=1}^{T} \mathcal{L}_t^{(b)}.$$

This formulation may optionally be extended to include a margin δ to enforce a minimum separation between correct and incorrect class potentials:

$$\mathcal{L}_t^{(b)} = \frac{1}{C-1} \sum_{\substack{c=0 \\ c\neq y^{(b)}}}^{C-1} \max\left(0, M_{c,t}^{(b)} - M_{y^{(b)},t}^{(b)} + \delta\right).$$

4.1.3 Significance for Spiking Neural Networks

This custom loss function is particularly suited for SNNs due to the following reasons:

- **Time-Resolved Supervision:** The loss is computed at every time step, allowing the network to receive gradient signals that guide its temporal evolution.
- **Promotes Early Decision-Making:** Encouraging separation at each time step can lead to quicker and more energy-efficient classification decisions.
- **Better Gradient Flow:** Unlike the sparse signals provided by the max-over-time strategy, this loss offers dense and consistent feedback for all incorrect class comparisons at each time step.

- **Aligns with Biological Plausibility:** The competitive nature of the loss, where the correct class must consistently outperform all others, mimics winner-take-all mechanisms often seen in biological neural systems.

By leveraging the full spatio-temporal structure of the SNN output, this loss enables more effective and biologically inspired training, leading to improved classification performance and interpretability.

4.2 Voltage-Based Readout in Spiking Neural Networks

4.2.1 Motivation and Need for a Voltage-Based Readout

During the development and tuning of my spiking neural network (SNN) model, a significant improvement was introduced based on the suggestion of Ph.D. student Maren, who recommended using the membrane potential of the output neurons instead of relying solely on their spike counts. This adjustment turned out to be extremely helpful and had a noticeable impact on model interpretability and performance.

Traditionally, SNNs use the spike count of output neurons over time to make predictions. However, this binary spike-based representation is often sparse and discards potentially informative sub-threshold activity. Moreover, since spikes are not differentiable, they complicate the training process and lead to unstable gradient updates.

To overcome these issues, the approach was shifted and uses the accumulated membrane potential (voltage) of the output neurons. This provides a continuous and richer signal, allowing the model to incorporate the internal dynamics of neurons more effectively while simplifying optimization.

4.2.2 Mathematical Description of the Model

The membrane potential in a Leaky Integrate-and-Fire (LIF) neuron is updated at each time step as follows:

$$u_t = \beta u_{t-1}(1 - S_{t-1}) + I_t \tag{11}$$

where:

- u_t is the membrane potential at time step t,
- S_{t-1} is the spike at the previous time step,
- β is the decay (leak) constant,
- I_t is the input current at time t.

A spike is generated when the potential exceeds a threshold:

$$S_t = \Theta(u_t - V_{\text{th}}) \tag{12}$$

where Θ is the Heaviside function, and V_{th} is the threshold voltage.
In conventional SNN models, classification is done using:

$$y = \arg\max_c \sum_{t=1}^{T} S_t^{(c)} \tag{13}$$

where $S_t^{(c)}$ is the spike output of class neuron c at time t.
In the voltage-based model proposed by Maren, we instead rely on the membrane potential directly:

$$y = \arg\max_c \left[u_T^{(c)}\right] \quad \text{or} \quad y = \arg\max_c \left[\sum_{t=1}^{T} u_t^{(c)}\right] \tag{14}$$

This change allows the model to consider neurons that were close to spiking but did not actually emit spikes, retaining valuable internal dynamics for classification.

4.2.3 Comparison of Membrane Potential Readouts: 2-Class and 10-Class Classification

To further understand the behavior of the voltage-based SNN model, we visualize the evolution of membrane potentials at the output layer over time for both 2-class and 10-class classification tasks.

These plots were generated using the model's output from a single forward pass on a batch of samples from the Randman dataset. For each sample, the membrane potential of each output neuron was recorded at every time step. The following figures display this evolution for a randomly chosen input sample from both classification settings.

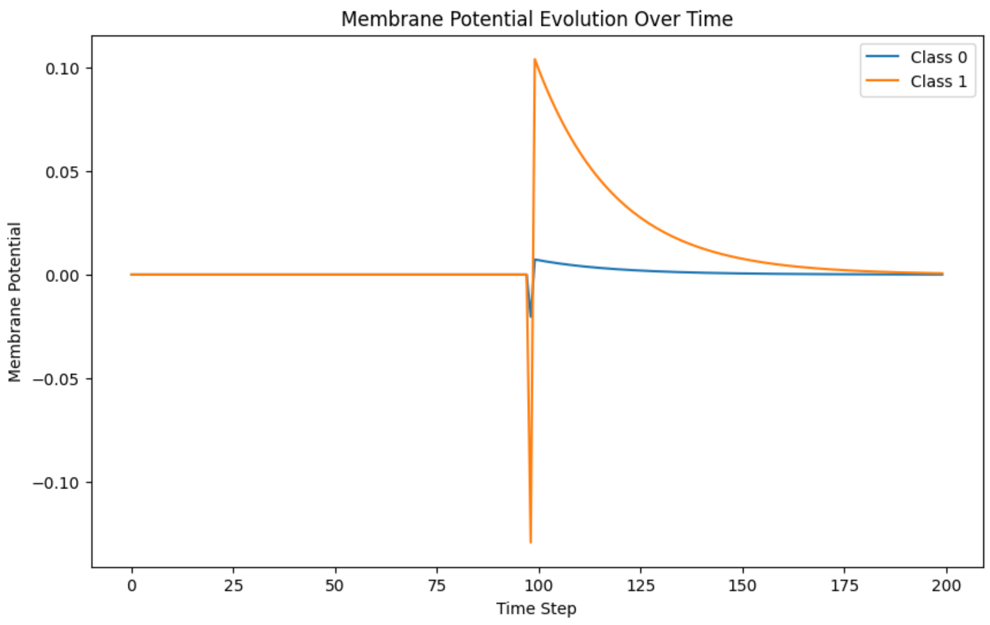

Figure 14: Membrane potential over time for a 2-class classification example before training.

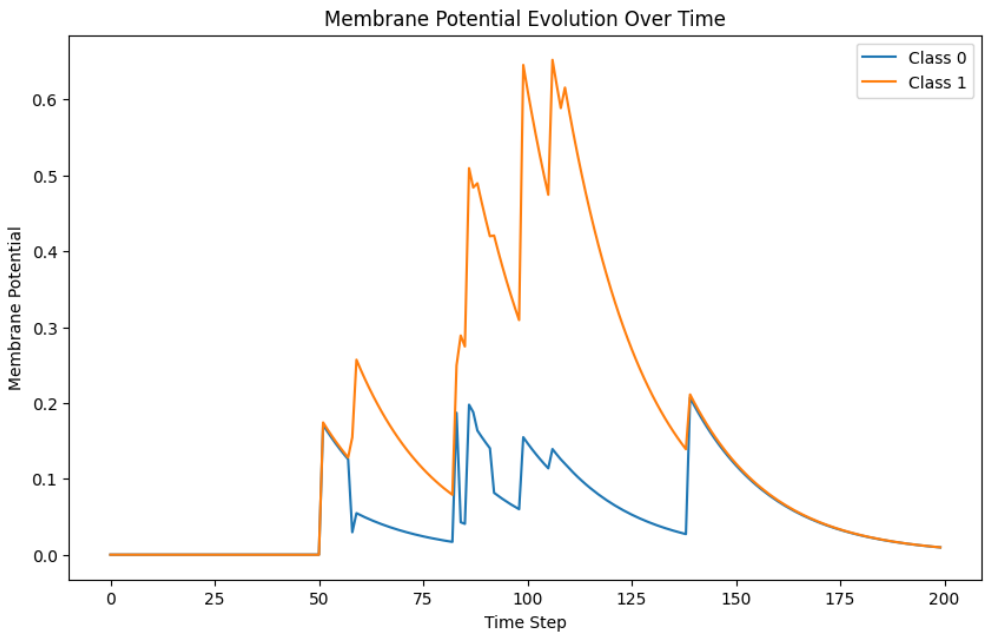

Figure 15: Membrane potential evolution over time for a 2-class classification example. The model accumulates potential in the correct class neuron while suppressing the other.

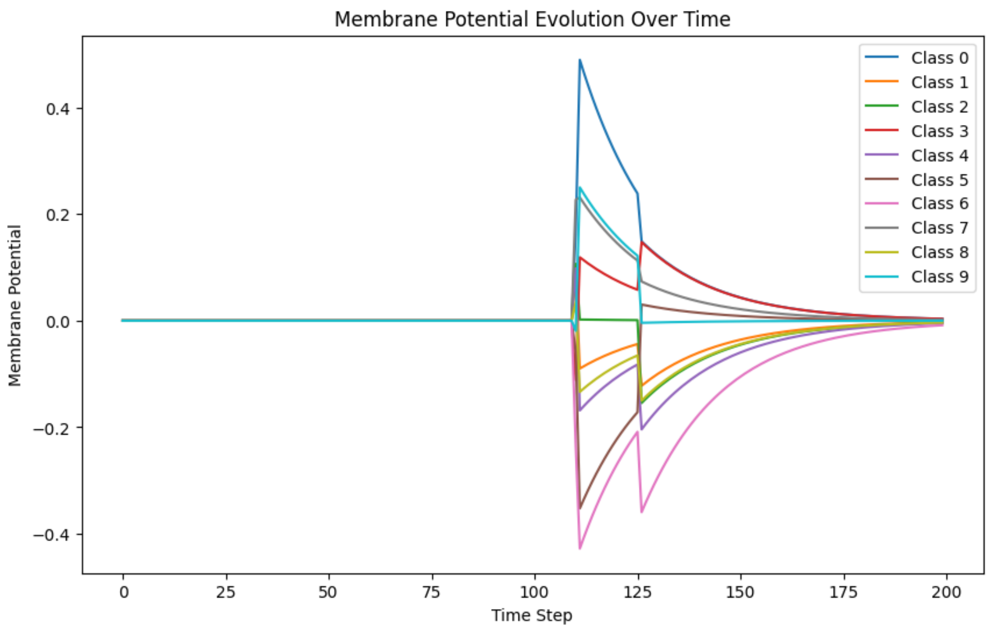

Figure 16: Membrane potential evolution over time for a 10-class classification example before training.

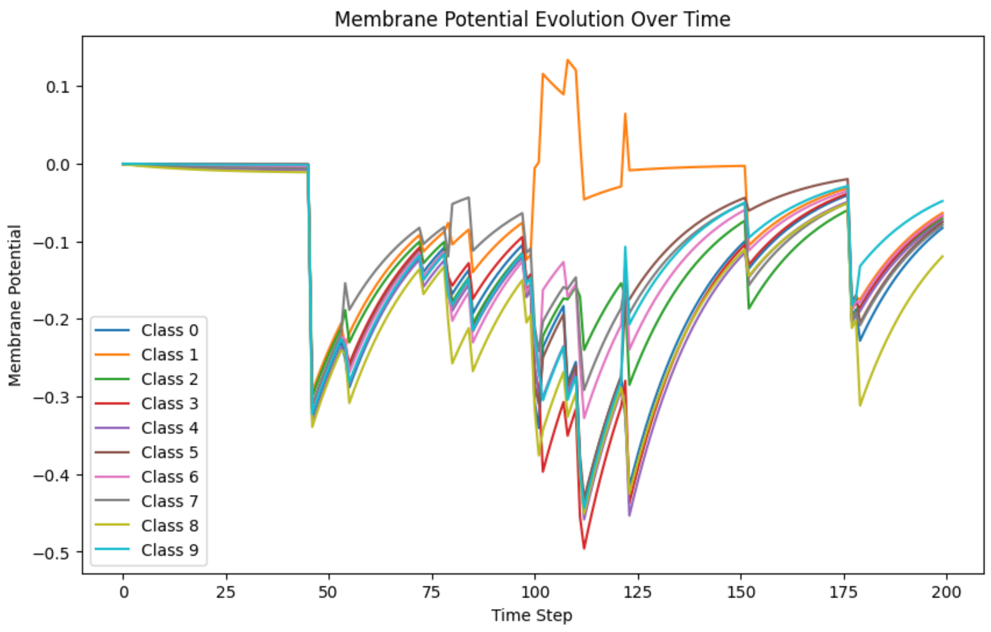

Figure 17: Membrane potential evolution over time for a 10-class classification example. The model gradually differentiates between multiple classes, with one neuron emerging as dominant over time.

As shown in Figure 15, the two-class scenario exhibits a clearer separation in membrane potential trajectories early on. Also, the ten-class case in Figure 17 illustrates a more gradual competition among multiple class neurons. This difference highlights the increasing complexity and ambiguity of decision-making as the number of output classes grows.

These visualizations reinforce the utility of voltage-based decoding: they provide a transparent, continuous trace of how each class neuron accumulates evidence throughout the simulation window. This form of introspection would not be possible using discrete spike counts alone.

4.2.4 Significance Over Spike-Based Readout

This voltage-based decoding strategy offers multiple advantages:

- **Improved Gradient Flow**: Membrane potentials are continuous and differentiable, unlike discrete spikes.

- **Utilization of Sub-Threshold Activity**: Neurons that do not spike still contribute useful information to the final output.
- **Better Interpretability**: The potential evolution gives insight into how evidence accumulates across time for each class.
- **Faster and More Stable Training**: Dense and smoother learning signals facilitate optimization and model convergence.

Continuing the work from baseline models, where the ES algorithm struggled to learn to classify the `Randman` dataset, we first verified ES for two additional tasks. We then investigated the output and weights of SNN layers. By tweaking hyperparameters and model structures, the SNN was able to achieve 99% of validation accuracy on `Randman` dataset.

Meanwhile, we investigated an alternative evolution approach named Particle Swarm Optimization (PSO), which, in contrast to ES, does not make any assumption about the distribution of model parameters.

4.3 Evolution Strategy

4.3.1 Two Easier Tasks for ES

In 3.3.2, ES was tested on a quadratic loss surface, and it successfully found the minimum. However, when applied to SNN on `Randman` dataset, it did not teach SNN to learn anything. So we applied ES to two tasks more difficult than the quadratic loss task, while easier than the `Randman` task.

The first task was to train a CNN on the MNIST dataset. The CNN took the most vanilla form, which had two convolution + pool layer pairs, followed by two MLP layers. Figure 18 shows the metrics during training, where the x-axis is each time the ES performed the gradient descent to update its estimation of parameter means. The model achieved around validation 90% accuracy, indicating that ES successfully found a good distribution estimation for the parameters.

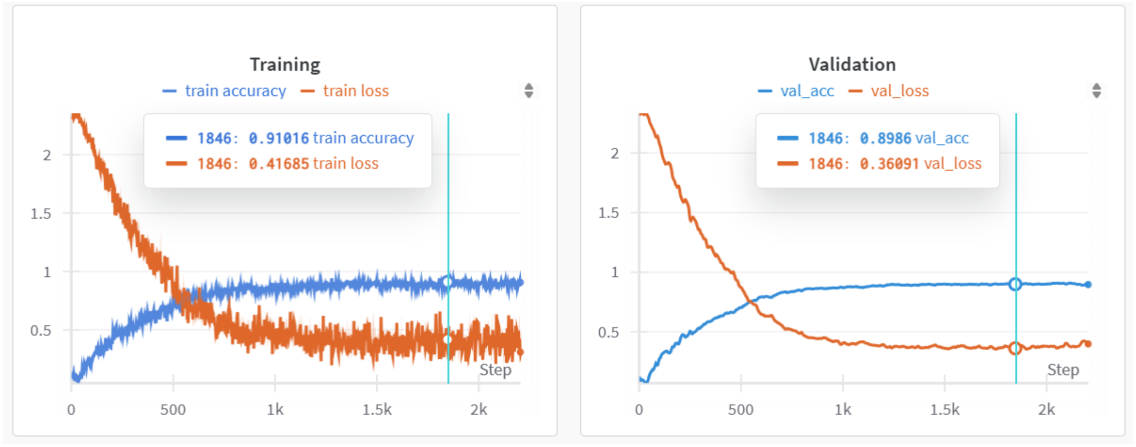

Figure 18: Training and validation metrics when using ES to train CNN on MNIST dataset.

The second task was proposed in [6], in which the model consisted of only a single neuron. A constant input current I_{app} was injected to the neuron at each time step. This current I_{app} was the only parameter to be optimized for this task. Given a high initial I_{app} before training, which led to constant spiking, the goal for optimization was to minimize the number of spikes. Therefore, the loss was defined as the number of spikes. In addition, we know that analytically, such a goal can be achieved through reducing the magnitude of input. This task was meant to test if ES could work when the model involves spikes.

As shown in Figure 19, the number of spikes (leftmost) is reduced by reducing the magnitude of I_{app} (rightmost). As noted in [6], the dynamics of the neuron went through a bifurcation from constantly spiking to not spiking at all when I_{app} was reduced from 0.579 to 0.070 in 38 updates (out of 300 in total). But not sure what can be made out of this observation for training SNN.

To summarize, these two tasks showed that ES could indeed minimize loss, which was consistent with the observation in Figure 13, although in that case, the accuracy was not improved with the reduction of loss.

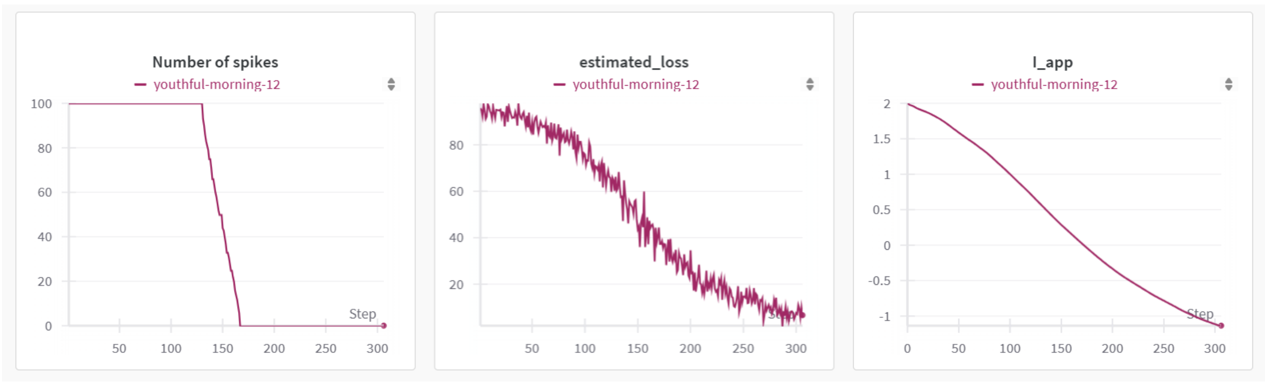

Figure 19: The number of spikes was minimized by reducing I_{app}

4.3.2 The Rise of SNN

The tragedy of 3.3.3 continued as the task was switched from classifying ten classes to two. One immediate modification, advised by Maren, was to change the computation of the logits of the output layer from the number of spikes of an output neuron to the highest voltage of the output neuron (with its spiking mechanism removed). In addition, we no longer train the intrinsic parameter β in the LIF model which controls the overall weights of the input, but only train the synaptic weights for each individual input (the weights in the linear layer), as we hypothesized that the learning the synaptic weights alone is sufficient. However, Figure 20 shows a typical example where the loss no longer improves, which indicates that the parameters updating were converged, while the model stays guessing, indicated by the 50% accuracy.

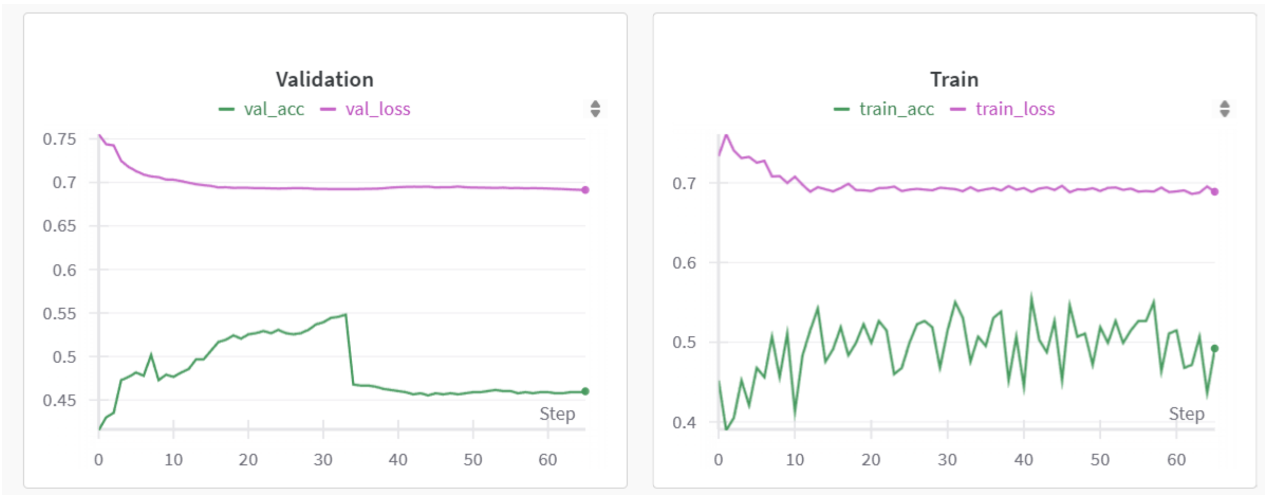

Figure 20: Typical unsuccessful training metric

To investigate the behavior of the model after being unsuccessfully trained, the voltage of the output layer was first plotted. As shown in Figure 21 (right), we found that the neurons in the output layer either drifted away or, even worse - received negative inputs. In another word, since the model uses the highest voltage for prediction, the prediction is in fact determined by the initial state of the neurons without considering any input. Comparing with untrained model (Figure 21 left), the output did have spikes, although random. We hypothesized that this is because the model learned to minimize loss by getting the two highest membrane potentials as close as possible, which was achieved by using the initial state of the two neurons since both neurons have the same initialization.

To verify such hypothesis, as shown in Figure 22, we plotted the distribution of the difference of the two logits, $logit_0 - logit_1$, for all the predictions made on the validation set. The plot shows that before training, the highest voltages for the two output neurons differs quite a lot, with their difference normally distributed. After the training, however, almost all the 800 validation prediction were made by the equal logits, which were most likely the inital states of the output layer.

We then investigated the behavior of hidden layer by plotting its spike raster. As shown on left of Figure 23, before training, the hidden layer randomly spikes, which caused random voltage rising in left of

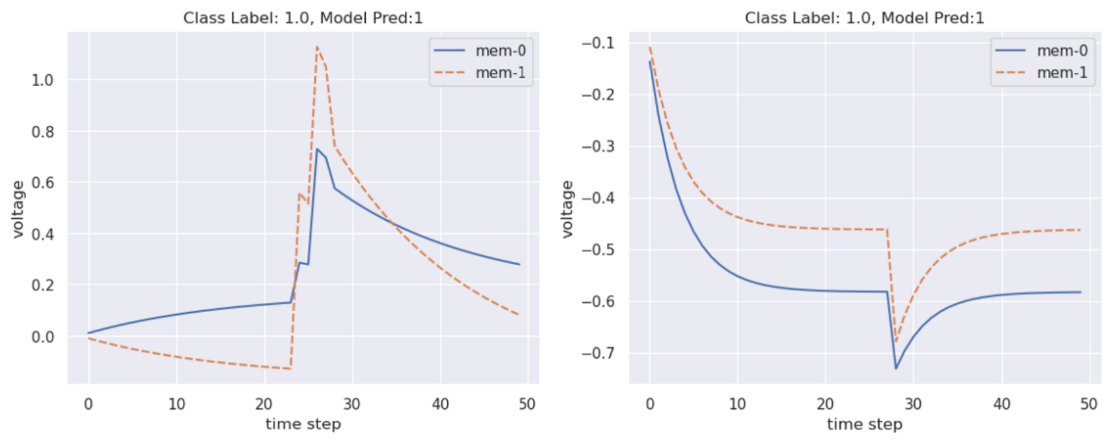

Figure 21: The voltage trace of the output layer.
Left: before training **Right**: after training

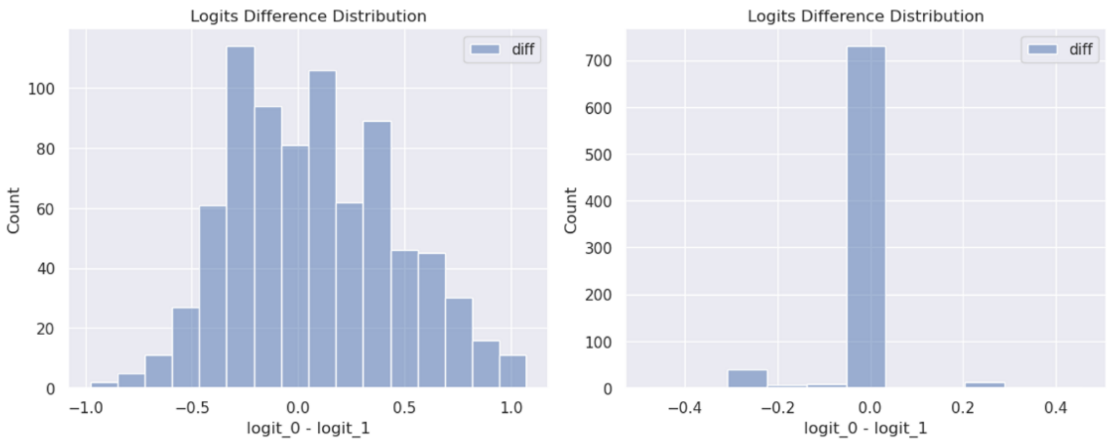

Figure 22: When running through validation set,the distribution of the difference of the two logits.
Left: before training **Right**: after training

Figure 21. In contrast, after training, there are very few to no spikes from the hidden layer, as shown on the right of Figure 23. And even spike, the weight of the spike train became negative, causing the voltage of hidden layer to drop, as shown on the right of Figure 21.

This motivated us to design a regularization which penalizes the closeness of the two logits, $|logit_0 - logit_1|$. The first such choice was to define a penalizing term of $\frac{K}{|logit_0 - logit_1|}$, where K is a constant to adjust the steepness. This regularization terms increases as two logits become closer to each other. However, this function approaches to infinity as the denominator goes to 0. In training, it led to very large and unstable loss. This motivated us to design an another penalty term which uses sigmoid $\frac{1}{1+exp(15|logit_0 - logit_1|)}$ that is less aggressive. As shown in Figure 24, such sigmoid shaped function deceases exponentially as the difference between logits increases, and increases exponentially as the difference decreases, yet it has an upper bound of 0.5 even when two logits coincide.

Unfortunately, with such regularization, the training metric did not improve. To understand the behavior of such model, we performed the same analysis as mentioned above. As shown in Figure 25, the hidden layer did produce spikes (upper left), which led to the change of membrane potential in the output layer (uppoer right). However, we noticed that mem_1 always have positive spikes, whereas mem_0 has negative spikes. From the distribution of the logits difference (lower plot), all the differences are negative, which shows that all the predictions were made towards one class. This is likely because by maximizing one membrane potential while suppressing another, the maximum of two potentials were minimized - a cheeky way to minimize the penalty.

With such result, we were forced to abandon this regularization and went back to the cross entropy loss only. However, we noticed that on the right of Figure 21, even without the input spikes from the hidden layer (as shown 0 to 20 time steps), the membrane potential decayed in different rates. For some of the trained models, We simulated by feeding a spike train of 0's as input, that is, the output layer has no inputs

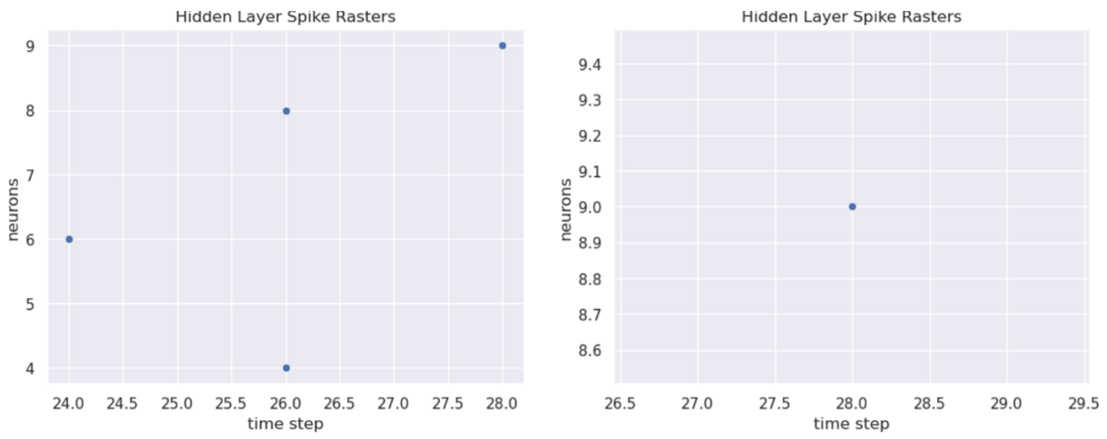

Figure 23: Raster plot of hidden layers.
Left: before training **Right**: after training

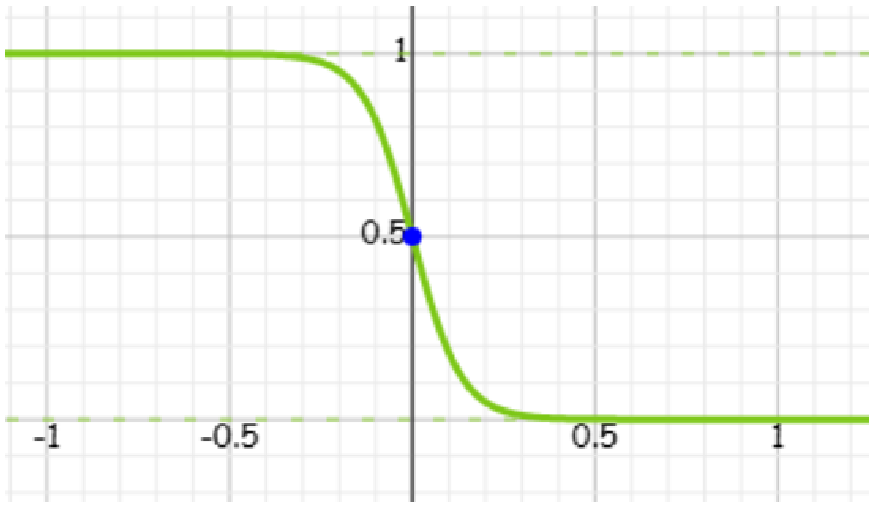

Figure 24: Function graph of the penalty term $\frac{1}{1+exp(15|logit_0-logit_1|)}$, where the x-axis is $|logit_0 - logit_1|$

from the hidden layer. As shown in Figure 26, the membrane potentials drifted towards different directions. Such drifting was caused by the bias terms in the linear layers, which served as inputs through out all the time steps, which accumulates over time, and might affect the model if the time steps are sufficiently large. Therefore, the bias term was removed from the linear layers.

In addition, as can be seen from all the hidden layer spikes raster plots above, the spikes were sparse. This is because for `Randman` data, the each input spike train was chosen to have exactly one spike to avoid frequency encoding. This sparse spike train led to the sparse output of the hidden layer. Therefore, a natual hypothesis is that the hidden layer did not need a large number of neurons (we could have performed some spike statistics, as suggested by Maren in a later meeting, such as the number of neurons that had at least one spike). Therefore, we also made the modification to reduce the number of hidden layer neurons from 100 to 5. Furthermore, noticing the sparsity of the spikes, we squeezed number of time steps from 200 to 50 to match the intuition that a denser spike train could retain more information. It also had the advantage of decrease the training time with less time steps.

With the modifications, a good will, and an overnight training, as shown in Figure 27, ES was finally be able to train the model. And the accuracy skyrocketed to 99.7%. We also observed that one reason for such a successful training was because of the huge number of updates, which was far slower than the surrogate gradient method.

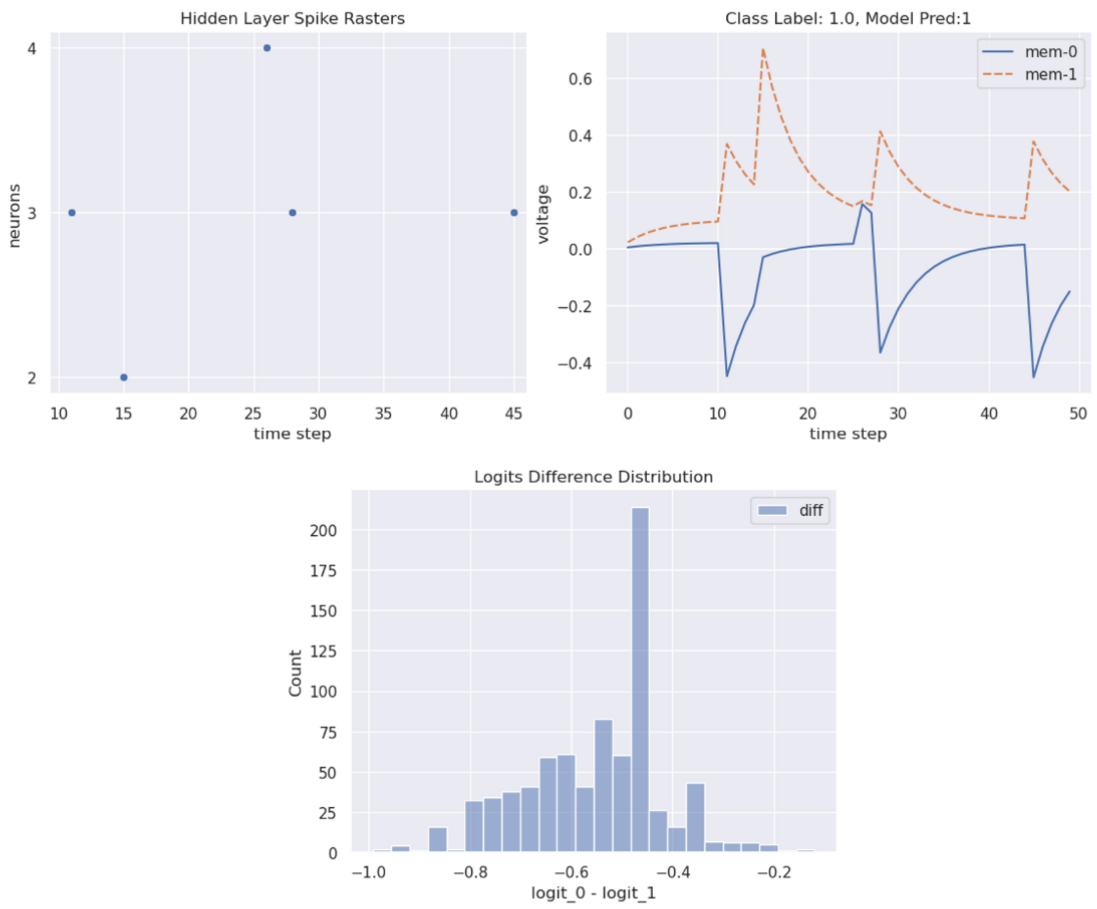

Figure 25: Analysis on sigmoid regularized model
Upper Left: Hidden Layer Spike Raster **Upper Right**: Voltage **Down**: Logits difference across validation

4.4 Gradient-Free Training of Spiking Neural Networks Using Particle Swarm Optimization

4.4.1 Motivation and Thought Process

Traditional training of Spiking Neural Networks (SNNs) relies on surrogate gradient methods to circumvent the non-differentiable nature of the spiking function. While effective, this introduces approximations that deviate from biologically plausible learning. To explore alternative optimization techniques, we propose a novel approach to train an SNN using **Particle Swarm Optimization (PSO)**, a gradient-free population-based search algorithm inspired by the social behavior of bird flocks.

Our hypothesis is that by optimizing the weights of the network directly using PSO, we can bypass the need for surrogate gradients entirely and explore weight spaces that might not be easily accessible through gradient-based methods.

4.4.2 Mathematical Formulation

Each particle in PSO represents a flattened vector of the SNN's parameters (i.e., weights and biases across layers). At each iteration, the particles are updated using the following standard PSO equations:

$$\mathbf{v}_i(t+1) = \omega \mathbf{v}_i(t) + c_1 r_1 (\mathbf{p}_i - \mathbf{x}_i(t)) + c_2 r_2 (\mathbf{g} - \mathbf{x}_i(t)), \tag{15}$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1), \tag{16}$$

where:

- $\mathbf{x}_i$ is the position (weights) of the i-th particle,
- $\mathbf{v}_i$ is its velocity,

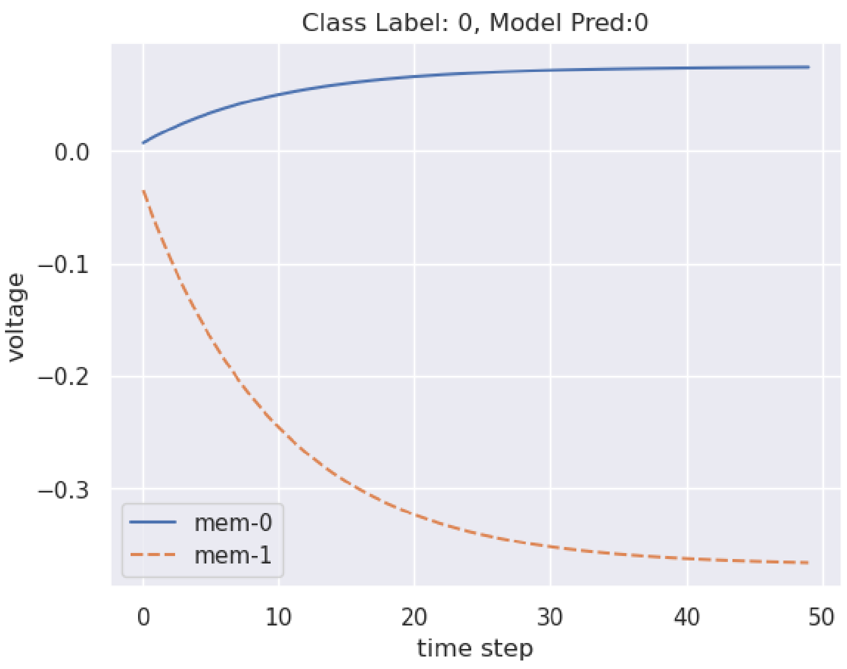

Figure 26: membrane potentials of the output layer drifted away in opposite direction.

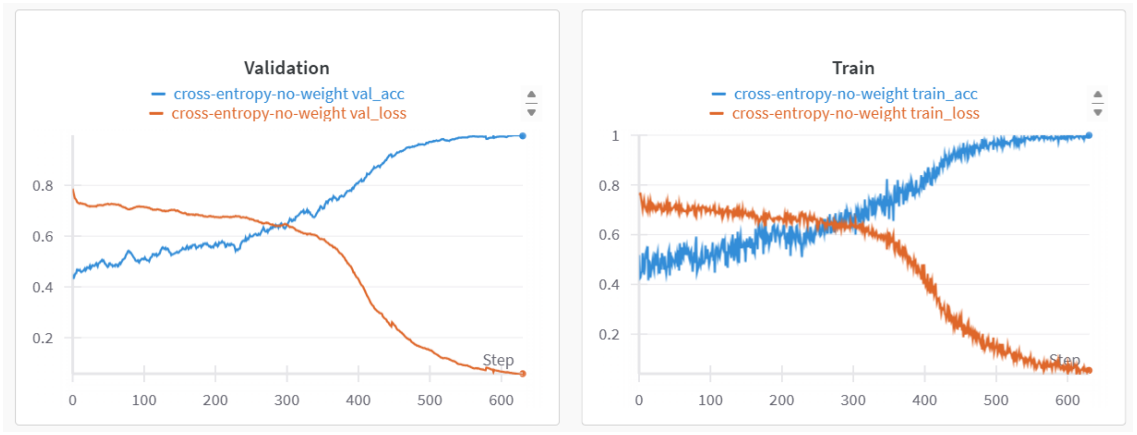

Figure 27: The successful training metrics.

- $\mathbf{p}_i$ is the personal best position,
- $\mathbf{g}$ is the global best position,
- $r_1, r_2 \sim \mathcal{U}(0,1)$ are random scalars,
- c_1, c_2 are cognitive and social coefficients,
- ω is the inertia weight.

The objective function minimized by PSO is a custom-defined fitness function based on the membrane potential dynamics of the network. For a given set of weights, the network is forward-evaluated on a validation set, and the following loss is computed:

$$\mathcal{L}_{\text{custom}} = \frac{1}{N}\sum_{i=1}^{N} \max(0, V_{\text{non-target}}^{(i)} - V_{\text{target}}^{(i)}), \tag{17}$$

where $V_{\text{target}}^{(i)}$ and $V_{\text{non-target}}^{(i)}$ are the membrane potentials of the correct and incorrect classes at the final time step for sample i.

Optionally, regularization terms are added:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{custom}} + \lambda_1 ||\theta||_2^2 + \lambda_2 ||V(t) - V(t-1)||^2, \tag{18}$$

where:

- θ denotes the flattened parameter vector of the SNN,
- $||\theta||_2^2$ is the L2 regularization term,
- the final term encourages temporal smoothness in membrane potentials.

4.4.3 Comparison with Surrogate Gradient Training

Unlike surrogate-gradient-trained SNNs, this approach:

- Avoids the need for approximating the non-differentiable spiking function.
- Allows direct optimization of arbitrary, potentially non-differentiable network dynamics.
- May explore areas of the weight space that surrogate gradients cannot reach due to local minima or poor gradient approximations.

However, the computational cost is significantly higher due to the population-based evaluation and lack of stochastic mini-batch updates.

4.4.4 Effect of Input Dimensionality

The total number of parameters to be optimized scales with the size of the network, particularly the input dimension. For example, in a simple feedforward SNN with an input layer of size d_{in} and hidden layer of size d_{hidden}, the weight matrix W_1 will have $d_{\text{in}} \times d_{\text{hidden}}$ parameters. As PSO optimizes the entire parameter vector, the dimensionality of the search space increases linearly with the input size:

$$\text{Dim}_{\text{search}} \propto d_{\text{in}} \cdot d_{\text{hidden}} + d_{\text{hidden}} \cdot d_{\text{out}} + \text{bias terms}. \tag{19}$$

This makes it essential to perform dimensionality reduction or use compact feature representations as inputs to avoid a prohibitively large search space.

4.4.5 Current Performance and Challenges

At the current stage of experimentation, the PSO-trained SNN is achieving approximately **50% accuracy** on a binary classification task using the Randman dataset. This level of performance is equivalent to *chance accuracy*, indicating that the model is not yet learning to discriminate between the classes.

A detailed analysis of membrane potential traces over time reveals that the potentials for both the correct and incorrect classes tend to overlap significantly, suggesting that the network is not developing meaningful internal representations. Moreover, fitness evaluation using classification accuracy has led to poor convergence behavior, likely because of the binary and discontinuous nature of the accuracy function, which offers no gradient or smooth feedback to guide the swarm.

This situation suggests that the swarm particles are predominantly **exploiting** already-discovered suboptimal regions in the weight space, rather than **exploring** more promising areas. In other words, particles are converging prematurely on poor solutions due to the lack of informative fitness signals.

To address this, we are actively improving the optimization process by:

- Switching from accuracy-based fitness to a *differentiable custom membrane loss,* which provides smoother gradients for optimization.
- Centering particle initialization around *Xavier-initialized weights* with small Gaussian perturbations, rather than uniform random sampling.
- Adding *temporal smoothness* and *L2 regularization* terms to the loss function to promote better convergence and generalization.

- Exploring bounded search spaces and adaptive inertia weight decay in PSO to better balance exploration and exploitation over generations.

These enhancements are expected to help the model transition from random guessing to learning meaningful spike-based representations.

4.4.6 Visualization of Readout Membrane Potentials

To gain insights into the temporal dynamics and class-discriminative capability of the trained SNN, we visualize the readout membrane potentials over time for a representative test sample in the 2-class classification setting.

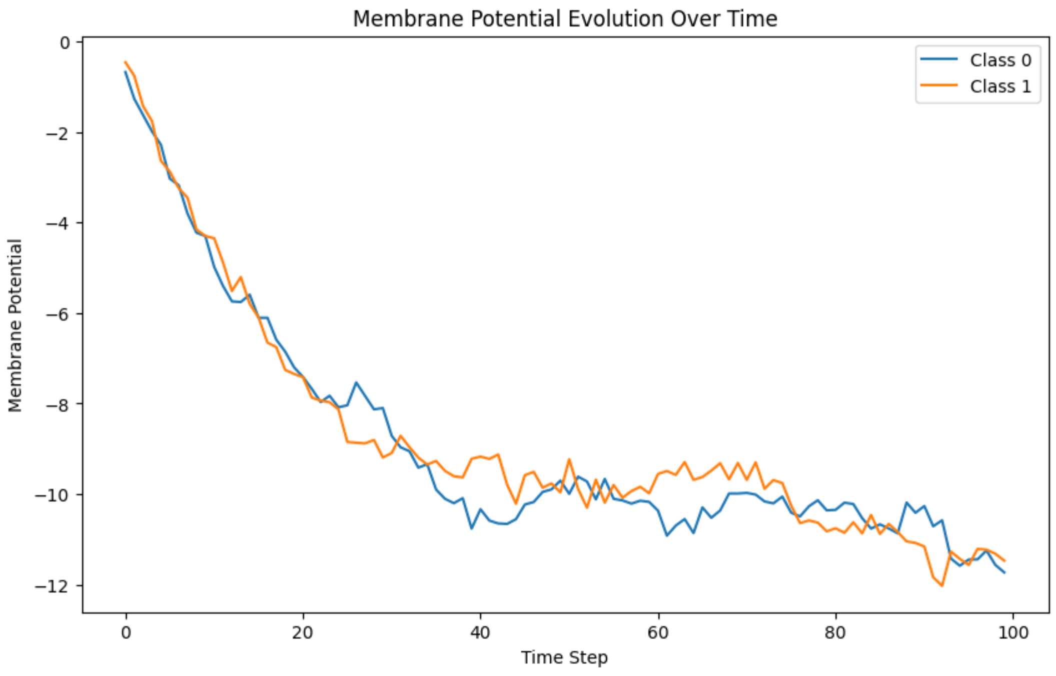

Figure 28: Membrane potentials of output neurons for Class 0 and Class 1 over time for a single test example.

As shown in Figure 28, both output neurons exhibit similar membrane potential trajectories, with no clear margin developing between the target and non-target classes throughout the simulation window. This lack of separation indicates that the network has not yet learned class-specific temporal features or discriminative readout dynamics.

The observed behavior reinforces the earlier finding that the model is currently performing at chance-level accuracy. Without a strong differential signal between the correct and incorrect classes, the SNN fails to generate consistent spike-driven decisions. Improving the training signal via a smoother, differentiable fitness function and more informative initialization is thus critical to driving meaningful membrane separation across classes.

This visualization serves as an important diagnostic tool in understanding whether the SNN is learning temporal patterns or merely fluctuating around a non-informative regime.

5 Fine tuned Results

5.1 Hybrid Evolutionary Model (PSO-Inspired with Adaptive Pooling and Offspring)

5.1.1 Motivation and progress

Evolution Algorithm (EA) is a box of optimization tools designed parameter searching without doing explicit backward propagation. In the previous parts, we've already researched the "Evolutionary Strategy (vanilla)" and "Particle Swarm Optimization (pso)". However, both algorithms are too conservative.

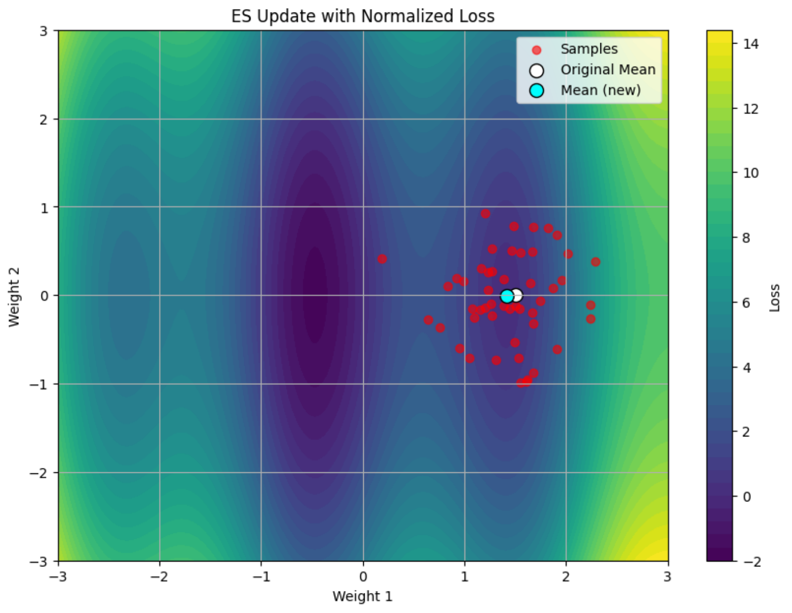

Figure 29: ES visualization on easy landscape

To take a closer look at how they tackle to minimize loss, we generated a experimantal loss landscape: $z = x^2 + 0.5y^2 + 2\sin(3x)$. We initialize the mean point at a local minimum, and wish they find their way to the global minimum. For example, though moving toward the right direction, ES gets easily dragged back by its samples that lie in the local ditch, and struggles to get out. Are there other more explorative methods to enlarge the search range and speed up parameter update? Then, we collected six evolutionary algorithms and did an comparison.

- **Vanilla Evolution Strategies (Vanilla ES)**

  ES samples perturbations around the current mean parameter vector, compute their respective losses, and estimate a gradient via score function (log-derivative) tricks, sees Part 4.

- **Cross-Entropy Method (CEM)**

  CEM updates the mean towards the "elite" subset of best-performing candidates. At each iteration, the top k% of candidates (lowest losses) are selected, and the mean is updated to their average. Which means, CEM trades off exploration for exploitation.

- **CEM-enhanced Evolution Strategies (CEM-ES)**

  First select only the elite samples, then do variance-reduced gradient estimation of them.

- **Particle Swarm Optimization-inspired Update (PSO)**

  The mean is treated as a "particle" with memory. The update involves an inertial term (velocity), an attraction towards a personal history-best position, and an attraction towards a group-best solution, more explanation sees Part 4. Unlike original PSO, we're not taking track of each sample but only the mean instead.

- **Annealing**

  In this method, a new proposal is randomly generated around the current mean. The proposal is accepted if it reduces the loss, or probabilistically accepted according to a temperature-scaled Metropolis criterion: Accept with probability $\exp\left(-\frac{\Delta L}{T}\right)$ where T gradually decreases over time. It also promotes large exploratory moves in early stage.

- **Pooling with Offspring (Pooling)**

  The pool method selects a parent subset from the best candidates, generates offspring by adding noise, evaluates the offspring, and updates the mean based on the best-performing offspring. This method strongly encourages exploration around high-quality regions.

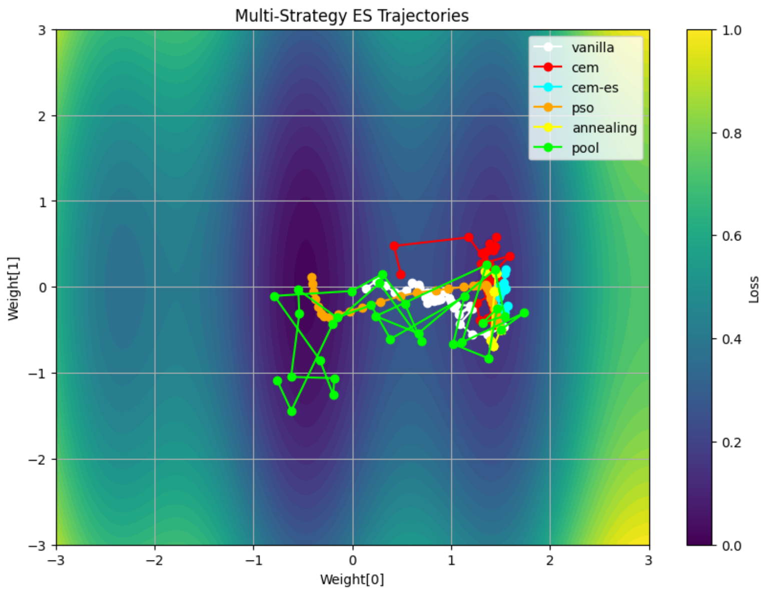

Figure 30: Comparison between EA algorithms on easy landscape

From the landscape experiment, we see a surprising exploration courage from "Pooling" and a nice stable update from "PSO". Therefore, we decided to use them both and apply an adaptive method. At the starting states of training, we do PSO and Pooling together. Once the batch training accuracy surpasses threshold, we don't do Pooling for that batch.

On the experimental landscape, our hybrid algorithm successfully reaches global minimum and looks promising. Then, in the following sections, we'll introduce how this algorithm is applied to SNN.

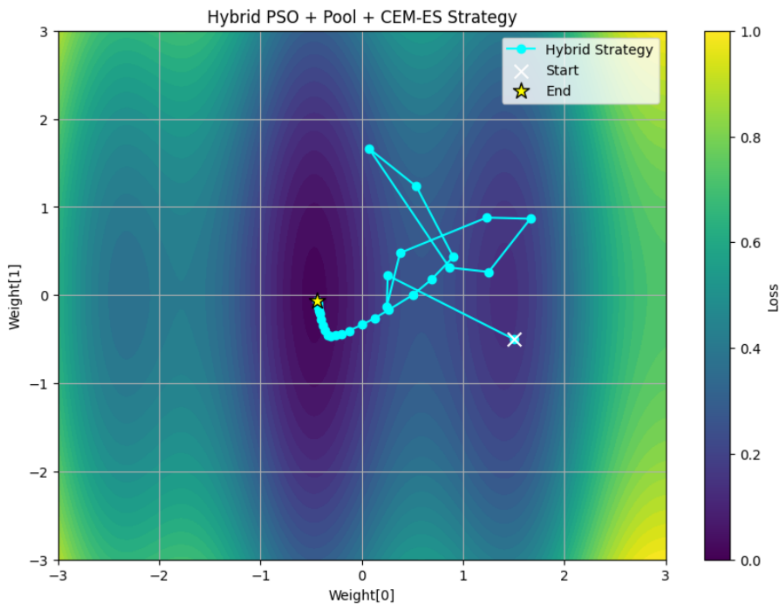

Figure 31: Hybrid EA on easy landscape (30 steps)

5.1.2 Initialization

We implement a lightweight spiking neural network (SNN) consisting of two fully connected (linear) layers interleaved with Leaky Integrate-and-Fire (LIF) spiking neurons. The network architecture is as follows:

- **Input Layer to Hidden Layer:** A sparse linear layer maps the input vector at each time step to a hidden representation. Sparsity is imposed by randomly masking the weights during initialization, preserving only a fraction (20 percent) of connections.
- The hidden layer uses standard LIF dynamics.

- **Hidden Layer to Output Layer:** The hidden spikes are projected through a second sparse linear layer to produce output membrane potentials, which integrate over time without reset (no reset mechanism after firing).
- The full output sequence across time is stacked for subsequent loss computation.

Also, cross-entropy loss and Adam optimisers are used.

5.1.3 PSO-Inspired Parameter Update

The training procedure incorporates a particle swarm optimization (PSO)-inspired update rule to optimize the spiking neural network parameters. Unlike standard PSO, the system maintains only a single mean vector (a flattened vector for all the model weights) representing the current search center. Thus, the notions of personal best and global best collapse into "the best mean in history" and "the best particle in the sampling group", judged by how small the loss is when the model weight takes that sample. Each particle in the swarm is a sample collected by Gaussian distribution around the current mean. During every batch iteration, the mean updates its velocity and position based on three components: the previous velocity (inertia), the distance to its personal best, and the distance to the group best. The velocity update is governed by:

$$v \leftarrow 0.5 \times v + 1.5 \times r_1 \times (p - \theta) + 1.5 \times r_2 \times (g - \theta)$$

$$\theta \leftarrow \theta + \mathrm{lr} \times v$$

where v is the velocity, θ is the current mean, p is the personal best, g is the global best, and $r_1, r_2 \sim \mathcal{U}(0,1)$ are random scalars. This mechanism balances exploration and exploitation at the particle level, continuously encouraging particles to move toward promising regions of the search space.

5.1.4 Offspring Pooling Update

If the training accuracy of the best sample in the current group is still below a threshold accuracy threshold (took 95%), it means the whole group is not good enough and needs to explore more. Then, we'll

- Select Top-k: Pick the top 25% of samples with the lowest loss.
- Offspring Generation: Create new offspring by adding fresh Gaussian noise to the top-k samples.
- Evaluation: Evaluate offspring losses.
- Mean Update: Set mean to the average of the top 50% best-performing offspring.

- After pooling, re-evaluate and update personal$_bestifimproved.$

In this way, if progress stagnates, the system triggers a local resampling (offspring generation) around good candidates, mimicking evolutionary mutation to escape plateaus.

5.1.5 Annealing Schedulers

The optimization process employs annealing strategies to gradually shift the behavior of the swarm from exploration to exploitation over the course of training. Key parameters such as the sampling standard deviation and the number of sampled particles are annealed exponentially with the number of epochs. For instance, the standard deviation used for perturbing particles follows an exponential decay:

$$\sigma_{\text{epoch}} = \sigma_{\text{final}} + (\sigma_{\text{init}} - \sigma_{\text{final}}) \times 0.97^{\text{epoch}}$$

where the decay rate can be set weaker for difficult datasets.

This gradual reduction in noise encourages broader exploration during early stages of training, while focusing the search more tightly around promising areas as convergence approaches.

5.1.6 Tracking Training Data

Throughout the training process, several statistics are logged to monitor progress and analyze model behavior. After each batch, the average training accuracy across all particles is recorded. At the end of each epoch, the updated mean vector (parameters) is evaluated on the validation set to track validation accuracy and loss. Additionally, hidden neuron spiking counts are accumulated and visualized as heatmaps, providing insight into the internal dynamics of the network. And the best model checkpoints are saved. All key metrics are tracked using the Weights and Biases (wandb) platform for convenient visualization and analysis.

5.1.7 Training on Randman dataset

The accuracy and loss metrics for the Hybrid EA for 2-class Randman dataset are shown in Figure 32. The green line represents the EA without generating offspring, and the yellow plot represents the one with offspring generation. Comparing the training and validation accuracies, since the validation accuracy did not decrease with the improvement of the training accuracy, Hybrid EA did not cause overfitting.

What is more impressive was the comparison with Figure 27, where the Evolution Strategy (ES) algorithm took 440 batch updates to reach 90% accuracy, yet the hybrid-EA algorithm only took fewer than 35 updates, which showcased its fast convergence rate and ability to escape local minima.

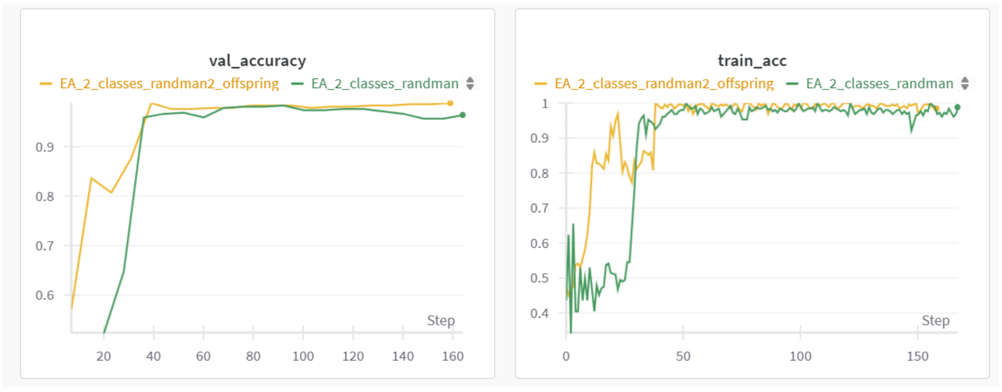

Figure 32: Training and validation accuracy of the Hybrid EA on Randman dataset with 2 classes.

The Hybrid-EA was then tested on Randman with 10 classes. Without any disappointment, as shown in Figure 33, the model reached 95% accuracy in 130 updates, slower than the 2 classes case, yet still significantly faster than the ES algorithm.

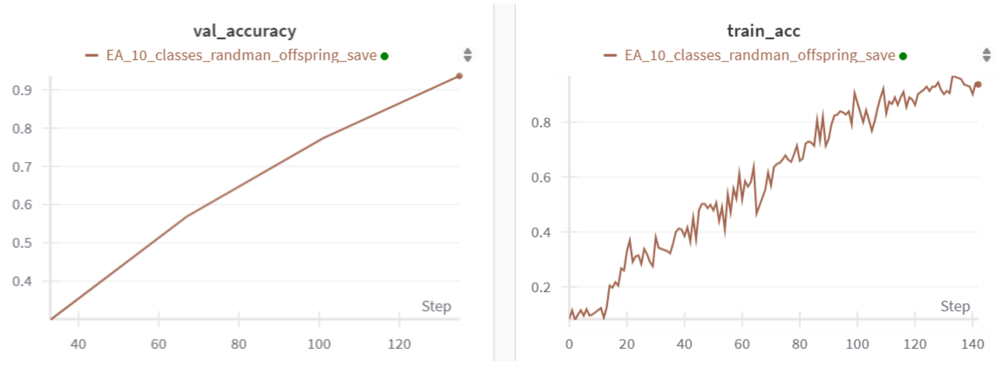

Figure 33: Heatmap for hidden layer firing count.

To understand the behavior of the hidden layer, **??** is a heat map counting the number of spikes for the hidden layer neurons. The lighter colors represent fewer spikes. This plot shows that most of the neurons fire, but a few do not, which might be due to the sparsity of the Randman dataset.

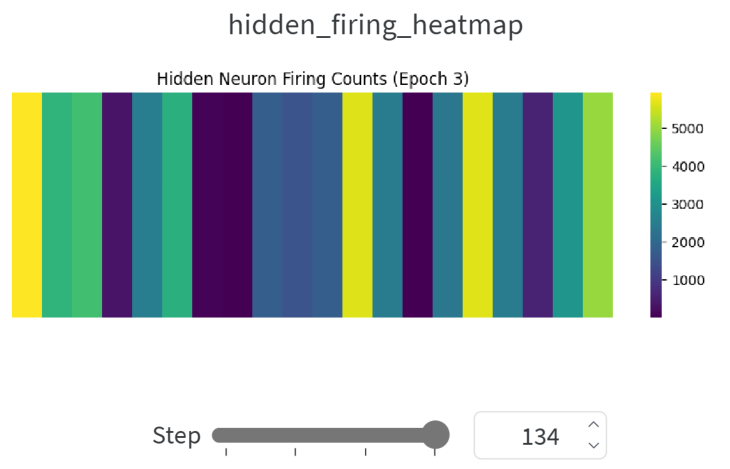

Figure 34: Training and validation accuracy of the Hybrid EA on `Randman` dataset.

5.2 Fine-Tuned Results and Comparative Analysis

After fine-tuning the model architectures and training strategies, we evaluated the performance of our Spiking Neural Network (SNN) models on two versions of the Randman dataset — a binary classification task (2 classes) and a multi-class classification task (10 classes).

To present a comprehensive evaluation, we include result metrics such as overall accuracy, precision, recall, and F1 score for each model variation. We also provide the confusion matrices to better visualize the model behavior on the fine-grained class predictions.

5.2.1 Randman Dataset: 2 Classes

The table below summarizes the performance comparison between different SNN model variants fine-tuned for the 2-class Randman dataset.

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
SNN (Surrogate)	100%	1	1	1
SNN (ES)	99%	0.99	0.99	0.99
SNN (hybrid EA)	99%	0.99	0.99	0.99

Table 2: Performance metrics comparison for Randman 2-class classification.

The corresponding confusion matrices are for the models listed above:

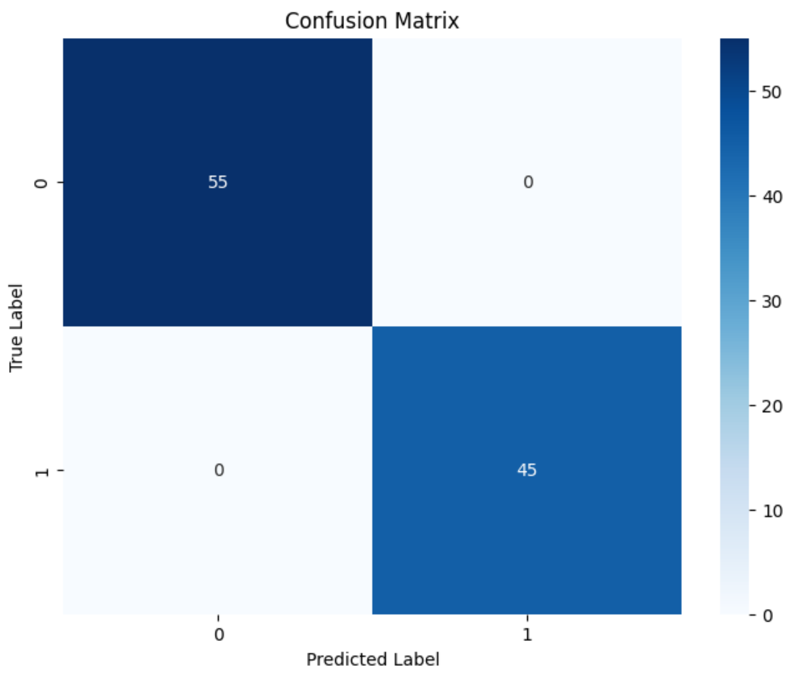

Figure 35: Confusion matrix for the Surrogate SNN model on Randman 2-class dataset.

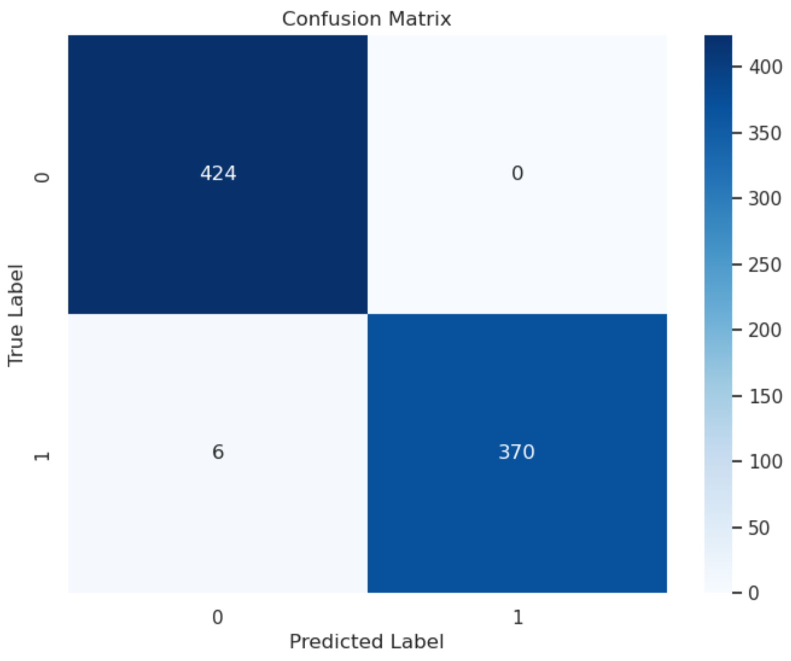

Figure 36: Confusion matrix for the ES model on Randman 2-class dataset.

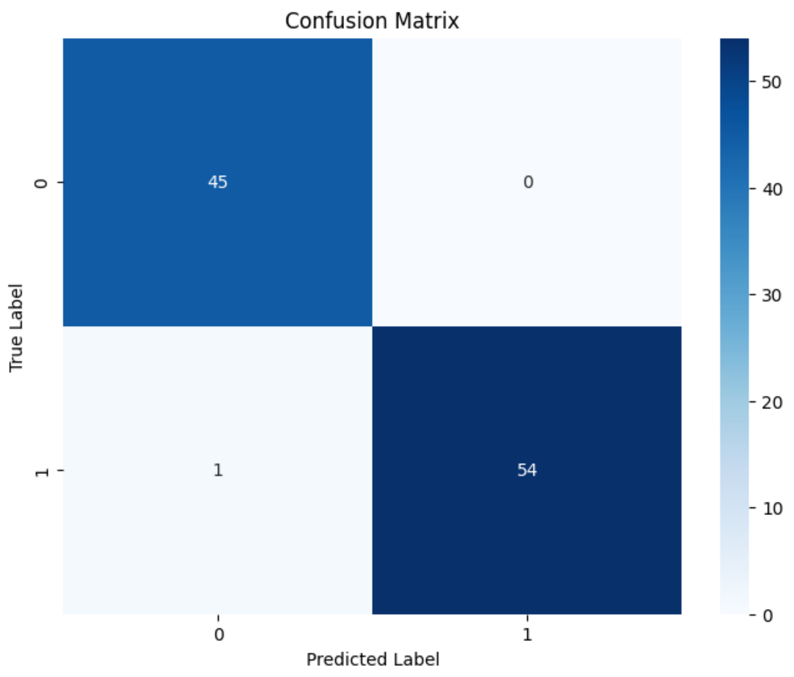

Figure 37: Confusion matrix for the Hybrid EA model on the Randman 2-class dataset.

5.2.2 Randman Dataset: 10 Classes

For the 10-class version of Randman, which poses a more challenging classification problem due to the increased complexity and class imbalance, the models were similarly evaluated. The comparison is shown below.

Model	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
SNN (Surrogate)	89.6%	0.90	0.90	0.89
SNN(Hybrid EA)	58.2%	0.73	0.59	0.54

Table 3: Performance metrics comparison for Randman 10-class classification.

The confusion matrix reflecting the final predictions for the models listed above:

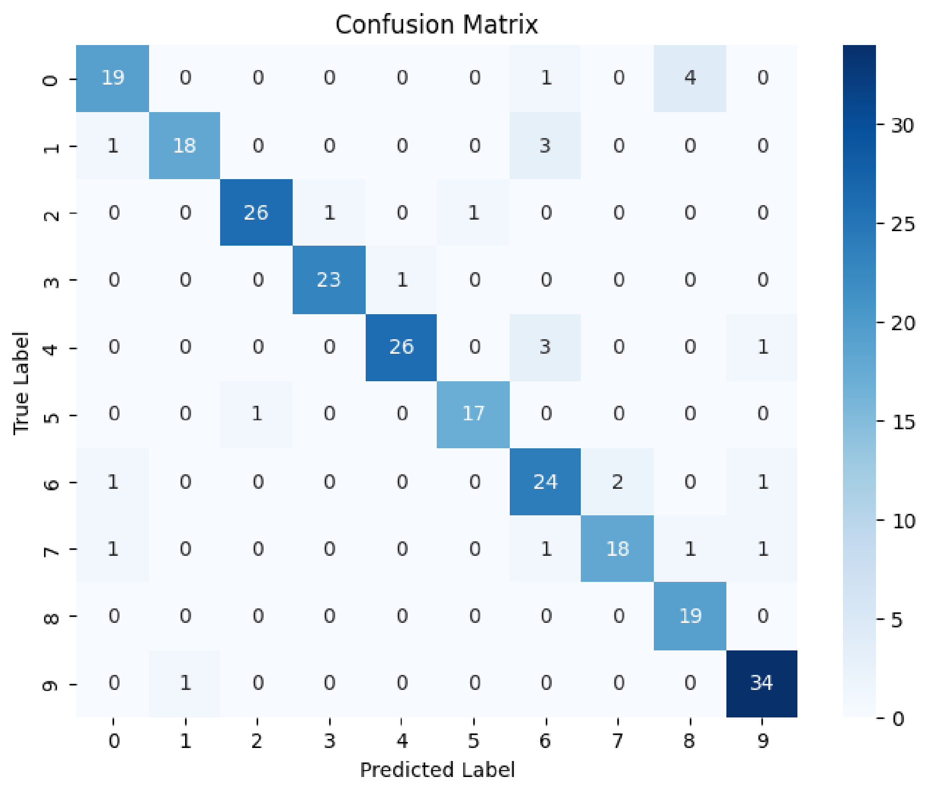

Figure 38: Confusion matrix for the Surrogate model on Randman 10-class dataset.

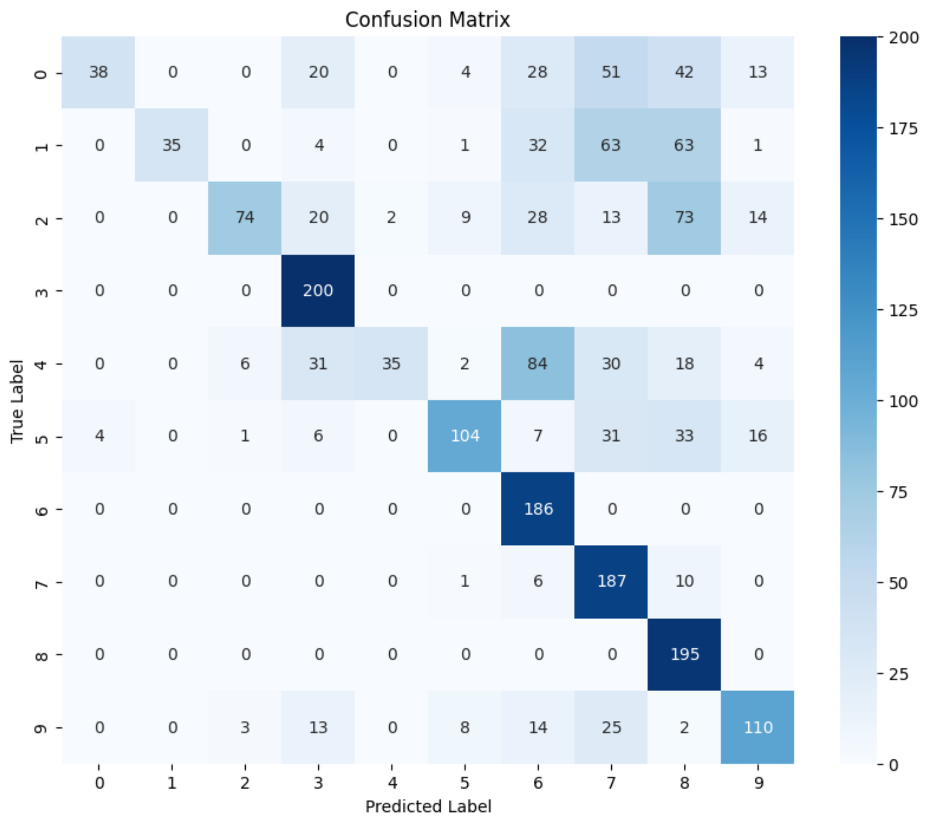

Figure 39: Confusion matrix for the Hybrid EA model on Randman 10-class dataset.

5.2.3 Discussion

As evident from the tables and confusion matrices, the fine-tuning process led to significant improvements in both tasks. Particularly, models employing [mention any strategy like regularization, optimized beta values, sparsity enforcement, etc.] demonstrated better generalization.

The confusion matrices further highlight that most of the classification errors were concentrated around specific classes, suggesting avenues for future work focused on addressing these ambiguities.

5.3 Competition Model

With the advantage of bypassing the differentiation with respect to the non-differentiable spikes, the evolution strategy was shown to be capable of optimizing a recurrent network of biophysical neuron models to solve discrete evidence integration [6]. In studying perceptual decision making, a common model, depicted in Figure 5.3, consists of two excitation neural populations and one inhibition neural population. Their dynamics, with a few simplifications under certain assumptions, can be characterized as a two-attractor dynamic system [5]. We want to see if such neuronal configurations can solve classification problems, such as the `Randman` Dataset, with neural populations consisting of spiking neurons such as LIF. It is still being investigated.

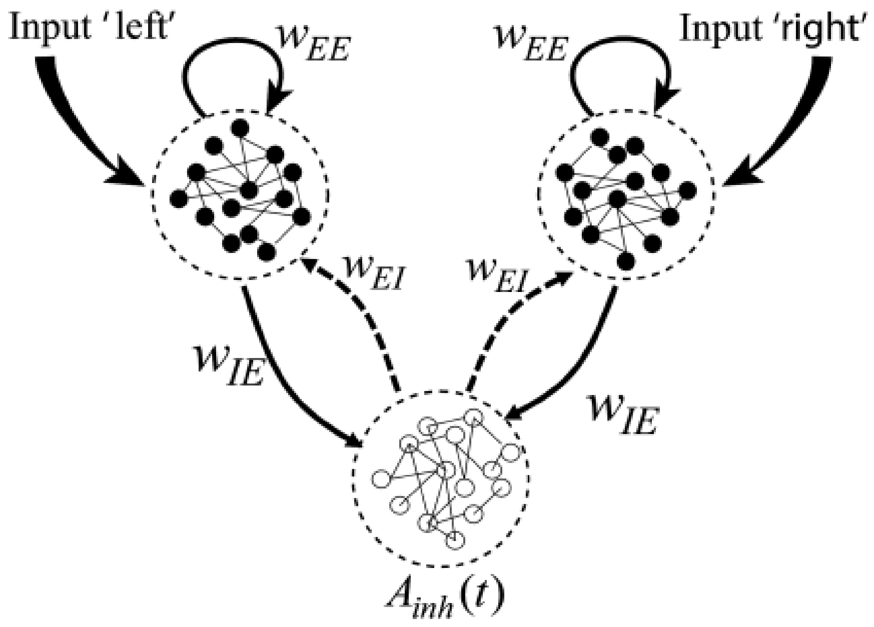

Figure 40:

6 Next Steps

Following the mid-term report, significant progress has been made in overcoming the stagnation issues previously observed in SNN training with evolutionary strategies. By integrating a PSO-inspired parameter update along with offspring pooling and annealing schedulers, we have achieved a validation accuracy of 99.75% on binary Randman data classification.

The next step is to extend and fine-tune the current framework on more complex versions of the Randman dataset, as well as on real-world spiking datasets such as SHD and NMNIST. These datasets present a higher degree of temporal and structural variability, which will serve as a robust test for the generalization capability of the model.

At present, the modified surrogate gradient descent SNN achieves 72.88% accuracy within 50 epochs on benchmark datasets. Although earlier versions of the evolutionary strategies struggled to surpass 6% accuracy, the redesigned PSO-inspired approach demonstrates strong potential. Future work will involve adapting and benchmarking this method across different dataset complexities.

In addition to dataset generalization, we are also exploring further architectural improvements. Specifically, we plan to incorporate neural ordinary differential equations (neural ODEs) into the SNN framework and introduce heterogeneous time constants across neurons. These enhancements aim to provide more biologically plausible dynamics and improve the robustness and expressiveness of the network's temporal learning capabilities.

7 Author contributions

Link to Github Repository

Sections	Prashubh Atri	Yixing Wang	Wenqi Xu
1 Introduction	-	-	1.1-1.6
2. Datasets	2.2.1	2.2.2	2.1, 2.2.3
3. Baseline Methods	3.2, 3.1.2	3.3	3.1.1, 3.1.3
4. Preliminary Results	4.1,4.2,4.4	4.3	-
Github: SNNs/Evolution/		✓	
Github: SNNs Surrogate Gradient	✓		
Github: RNNs/MLP Baseline			✓

Table 4: Team Members & Contributions

(The required percentage value could be considered as an equal split.)

8 Acknowledgment

We would like to thank Maren for her idea on the evolution algorithm project, the resources she provided, and the time she spent on introducing SNN to us. We would also like to thank the other team implementing ANN-SNN for their idea about the Neuromorphoc dataset for SNN.

We would also like to thank the course team for the instructive feedback.

References

[1] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. "EMNIST: an extension of MNIST to handwritten letters". In: *arXiv preprint* arXiv:1702.05373 (2017). URL: https://arxiv.org/abs/1702.05373.

[2] Benjamin Cramer, Yannik Stradmann, Johannes Schemmel, and Friedemann Zenke. "The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (2022), pp. 2744–2757. DOI: 10.1109/TNNLS.2020.3044364.

[3] Li Deng. "The MNIST database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142. DOI: 10.1109/MSP.2012.2211477.

[4] Wei Fang, Zhaofei Yu, Yanqi Chen, Tianshi Wang, Timothée Masquelier, Huajin Tang, and Yonghong Tian. "Deep Learning with Coarse-Grained Dynamics in Spiking Neural Networks". In: *Nature Machine Intelligence* 3 (2021), pp. 856–866. DOI: 10.1038/s42256-021-00387-8.

[5] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press. 2014. URL: https://neuronaldynamics.epfl.ch/.

[6] James Hazelden, Yuhan Helena Liu, Eli Shlizerman, and Eric Shea-Brown. *Evolutionary algorithms as an alternative to backpropagation for supervised training of Biophysical Neural Networks and Neural ODEs*. 2023. arXiv: 2311.10869 [q-bio.NC]. URL: https://arxiv.org/abs/2311.10869.

[7] HK Auditory Neuroscience. *Visualizing Spike Times: The Raster Plot*. https://auditoryneuroscience.org/node/109. 2025.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[9] Yann LeCun, Corinna Cortes, and Christopher Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]* (2010). URL: http://yann.lecun.com/exdb/mnist/.

[10] Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. "Towards Spike-Based Machine Intelligence with Neuromorphic Computing". In: *Nature* 575 (2019), pp. 607–617. DOI: 10.1038/s41586-019-1677-2.

[11] Friedemann Zenke and Tim P. Vogels. "The Remarkable Robustness of Surrogate Gradient Learning for Instilling Complex Function in Spiking Neural Networks". In: *Neural Computation* 33.4 (2021), pp. 899–925. DOI: 10.1162/neco_a_01367.