Exploration of Spiking Neural Networks for Learning Tasks

Abstract

This report details the implementation and investigation of two distinct Spiking Neural Network (SNN) architectures for different learning paradigms. The first implementation showcases a foundational SNN built from scratch using NumPy, focusing on neuron dynamics (Leaky Integrate-and-Fire, LIF) and a biologically inspired learning rule, Reward-modulated Spike-Timing Dependent Plasticity (R-STDP). The second, more advanced implementation, utilizes the `snnTorch` library to construct a convolutional SNN feature extractor integrated into a Prototypical Network for few-shot image classification on the MNIST dataset. This report provides a comprehensive overview of the methodologies, architectural choices, and observed behaviors for both systems, highlighting the versatility and challenges associated with SNNs in contemporary machine learning.

1. Introduction

Artificial Neural Networks (ANNs) have achieved remarkable success across diverse machine learning tasks. However, their computational principles differ significantly from those of biological brains. **Spiking Neural Networks (SNNs)** represent a third generation of neural networks that aim to bridge this gap by mimicking the event-driven, asynchronous communication found in biological neurons. Instead of continuous activation values, SNNs communicate through discrete electrical impulses called "spikes." This event-driven nature offers potential advantages in terms of energy efficiency and the inherent ability to process spatio-temporal information naturally.

This report explores two distinct SNN implementations:

1. **A basic, custom-built SNN with Reward-modulated Spike-Timing Dependent Plasticity (R-STDP):** This implementation focuses on the fundamental components of SNNs, including Leaky Integrate-and-Fire (LIF) neuron models and a form of biologically plausible synaptic plasticity for learning. It serves as an educational foundation for understanding SNN mechanics.
2. **An advanced SNN-based Prototypical Network for Few-Shot Learning:** This system addresses the challenge of **few-shot learning (FSL)**, where models must learn to classify new categories from a very limited number of examples. By employing a convolutional SNN as a feature extractor within a meta-learning framework (Prototypical Networks), this implementation investigates how SNNs can be leveraged for rapid adaptation to novel tasks.

The objective of this report is to detail the design, implementation, and observed performance of these two SNN models, providing insights into their operational principles and potential applications.

2. Theoretical Foundations

2.1 Leaky Integrate-and-Fire (LIF) Neuron Model

The Leaky Integrate-and-Fire (LIF) model is a simplified yet widely used model for simulating the dynamics of biological neurons. It describes the evolution of a neuron's membrane potential (Vm) over time. When Vm reaches a threshold (Vth), the neuron fires a spike, and its potential is reset to a resting potential (Vrest) for a refractory period.

The sub-threshold dynamics of the LIF neuron are typically governed by:

$$\tau \frac{dV_m}{dt} = -(V_m - V_{rest}) + I_{syn}$$

where τ is the membrane time constant, and Isyn is the synaptic input current. After firing a spike at time tspike, the membrane potential is reset:

$$V_m(t_{spike}^{+}) = V_{rest}$$

and remains at Vrest for a defined refractory period.

2.2 Spike-Timing Dependent Plasticity (STDP) and Reward-Modulated STDP (R-STDP)

Spike-Timing Dependent Plasticity (STDP) is a biologically plausible learning rule that adjusts the strength of synaptic connections based on the relative timing of pre-synaptic and post-synaptic spikes.

- If a pre-synaptic spike *precedes* a post-synaptic spike within a short time window, the synapse is strengthened (Long-Term Potentiation, LTP).
- If a post-synaptic spike *precedes* a pre-synaptic spike, the synapse is weakened (Long-Term Depression, LTD).

The weight change (Δw) often follows an exponential decay based on the time difference (Δt=tpost −tpre):

$$\Delta w = \begin{cases} A_{LTP} \cdot \exp(\Delta t/\tau_{+}) & \text{if } \Delta t > 0 \text{ (LTP)} \\ A_{LTD} \cdot \exp(\Delta t/\tau_{-}) & \text{if } \Delta t < 0 \text{ (LTD)} \end{cases}$$

where ALTP,ALTD are learning rates, and τ+,τ− are time constants.

Reward-Modulated STDP (R-STDP) extends traditional STDP by incorporating a global reward signal. Synaptic potentiation or depression is scaled by a reward factor, allowing the network to learn behaviors that lead to positive rewards and avoid those that lead to negative outcomes. This aligns with reinforcement learning principles.

2.3 Prototypical Networks

Prototypical Networks are a metric-learning approach for few-shot learning. The core idea is to learn an embedding function fϕ(x) that maps input samples x into a feature space. In this space, each class c is represented by a prototype vector pc, which is typically the mean of the embedded support examples for that class:

$$\mathbf{p}_c = \frac{1}{|S_c|} \sum_{(\mathbf{x}_i, y_i) \in S_c} f_\phi(\mathbf{x}_i)$$

where Sc is the set of support examples for class c.

Classification of a query example xq is performed by assigning it to the class whose prototype is nearest in the embedding space, typically using Euclidean distance:

$$\text{classify}(\mathbf{x}_q) = \text{argmin}_c\, d(f_\phi(\mathbf{x}_q), \mathbf{p}_c)$$

where d(⋅,⋅) is a distance function (e.g., Euclidean distance). The network is trained by minimizing the distance between query embeddings and their correct class prototypes, while maximizing the distance to incorrect ones.

2.4 Surrogate Gradient Learning for SNNs

A major challenge in training SNNs using gradient-based optimization (like backpropagation) is the non-differentiable nature of the spiking activity (a hard thresholding function). **Surrogate gradient methods** address this by replacing the discontinuous spiking function with a continuous, differentiable approximation during the backward pass, while retaining the true spiking behavior in the forward pass. Common surrogate gradient functions include sigmoid, arctan, or fast sigmoid, each with tunable slopes to control the approximation.

3. Implementation 1: Custom SNN with Reward-Modulated STDP

This section describes a basic SNN implemented from scratch using NumPy, demonstrating fundamental SNN principles and R-STDP.

3.1 Architecture Overview

The SNN consists of three layers:

- **Input Layer:** `N_input = 100` neurons.
- **Hidden Layer:** `N_hidden = 50` neurons.
- **Output Layer:** `N_output = 10` neurons.

The connections are fully connected between the input and hidden layers, and between the hidden and output layers. Synaptic weights (`w_in_hidden`, `w_hidden_out`) are randomly initialized.

3.2 Neuron Model Details

Each neuron in the hidden and output layers is modeled as a Leaky Integrate-and-Fire (LIF) neuron.

- **Resting Potential (Vrest):** -65.0 units
- **Threshold Potential (Vth):** -58.0 units
- **Membrane Time Constant (τ):** 10.0 milliseconds
- **Refractory Period:** 5 milliseconds (during which the neuron cannot fire after spiking and its membrane potential is held at Vrest).

The `lif` function updates the membrane potential based on input current, checks for spike generation, resets potential upon spiking, and manages the refractory period.

3.3 Input Generation

Input to the SNN is simulated using randomly generated spike trains. For the initial NumPy-based example, a simple rate-coded input is created:

- `input_rate = 100 / 1000` (100 Hz), meaning each input neuron has a 10% chance of spiking at each time step.
- `sim_time = 500` milliseconds, simulated over `time_steps = 500` (with `dt = 1ms`).

Later in the notebook, a `generate_spike_pattern` function is introduced to create more structured, synthetic spike patterns for different class labels, allowing for a basic classification task. These patterns have a periodic spiking component with added noise, and are reshaped to a flat vector of size (time_steps, 1156) for a larger input layer example.

3.4 Learning Rule: Reward-Modulated STDP (R-STDP)

The network learns through R-STDP, applied to both `Input → Hidden` and `Hidden → Output` synaptic connections. The learning mechanism implicitly used in the training loop (e.g., `np.einsum`) performs a form of Hebbian-like update scaled by a reward signal.

The update rule for weights is implemented as:

$$w_{ij} \leftarrow w_{ij} + \eta \cdot \text{reward} \cdot \left(\sum_t \text{pre}_i(t) \cdot \text{post}_j(t) \right)$$

where η is the learning rate (0.01), reward is a global feedback signal (1.0 for correct, -0.2 for incorrect), and prei(t) and postj(t) are the spiking activities of presynaptic neuron i and postsynaptic neuron j at time t. Weights are clipped between 0 and 2.0 to prevent unbounded growth.

3.5 Simulation and Results

The simulation runs for `sim_time` milliseconds. At each time step, input spikes are fed, hidden and output neurons are updated using the LIF model, and then R-STDP is applied to modify the synaptic weights.

Initial executions of this custom NumPy SNN with synthetic inputs (e.g., random rate coding) produce raster plots of hidden and output layer spikes, illustrating the temporal firing patterns of the neurons. Without explicit task-based training, these initial runs demonstrate the basic dynamics but not sophisticated learning.

Further experiments with the `generate_spike_pattern` function and a 3-class classification setup show the R-STDP in action over multiple episodes. The network attempts to classify spike patterns by summing output spikes and taking the argmax. The reward signal (1.0 for correct prediction, -1.0 or -0.2 for incorrect) modulates weight updates. The observed accuracies in the notebook output (e.g., "Accuracy = 1/3", "Query set accuracy: 43.33%") indicate that this basic R-STDP mechanism can induce some learning, though it might be slow or require more sophisticated tuning for robust classification.

4. Implementation 2: SNN-based Prototypical Networks for Few-Shot Learning with `snnTorch`

This section details a more advanced SNN implementation focused on few-shot learning, utilizing the `snnTorch` library built on PyTorch.

4.1 Architecture Overview: SNN Feature Extractor

The core of this system is `SNNFeatureExtractor`, a convolutional SNN responsible for transforming input images into a discriminative feature space.

- **Input Layer:** Handles `NUM_CHANNELS = 1` (grayscale MNIST images) of size `IMG_SIZE = 28x28`.
- **Convolutional Blocks:** Two convolutional blocks, each consisting of:
    - `nn.Conv2d`: Standard convolutional layer.
    - `snn.Leaky`: Leaky Integrate-and-Fire neuron layer.

    - `nn.MaxPool2d`: Max pooling layer.
- **Feature Flattening:** The output from the second pooling layer is flattened.
- **Fully Connected Layer:** A `nn.Linear` layer maps the flattened features to a 64-dimensional feature vector.
- **Final LIF Layer:** Another `snn.Leaky` neuron layer processes the output of the linear layer.

4.2 Neuron Model and Surrogate Gradients

The snnTorch library's snn.Leaky neuron model is used, with a BETA = 0.95 decay rate. Critically, to enable gradient-based training, surrogate gradients are employed:

$$\text{spike\_grad} = \text{surrogate.fast\_sigmoid}(\text{slope}{=}\text{SPIKE\_GRAD\_BETA})$$

where SPIKE_GRAD_BETA = 5 controls the steepness of the approximation. This allows standard backpropagation to compute gradients through the otherwise non-differentiable spiking events.

The `forward` pass simulates the SNN for `NUM_STEPS = 50` time steps. During this simulation, the membrane potentials of the LIF neurons are initialized and updated. The feature vector for an input image is derived by accumulating and averaging the membrane potentials of the final LIF layer (`self.lif3`) over all time steps. This average membrane potential serves as the static embedding.

4.3 Dataset and Task Generation

- **Dataset:** The standard MNIST handwritten digit dataset is used for few-shot learning.
- **Few-Shot Task Generation:** A custom `MNISTFewShotDataset` class handles the episodic meta-training and meta-testing:
    - **N-way:** `NUM_CLASSES_PER_TASK = 5` distinct classes are randomly selected for each task.
    - **K-shot:** `N_SHOT = 5` examples per selected class form the **support set**. These are used to compute class prototypes.
    - **Query:** `N_QUERY = 15` examples per selected class form the **query set**. These are used to evaluate the model's performance and calculate the loss.
    - The model is meta-trained on `NUM_META_TRAIN_TASKS = 2000` such tasks and meta-tested on `NUM_META_TEST_TASKS = 50` unseen tasks.

4.4 Prototypical Network Logic

The `SNNFeatureExtractor` provides embeddings for the Prototypical Network:

- **Prototype Calculation:** For each task, the SNN extracts features from the support set images. For each class c in the task, its prototype pc is computed by taking the mean of the SNN-extracted feature vectors of all N_SHOT examples for that class.

- **Distance-Based Classification:** For each query image, its SNN-extracted feature vector is compared to all prototypes using **Euclidean distance**. The query image is assigned the label of the closest prototype.
- **Loss Function:** `nn.CrossEntropyLoss()` is applied to the negative Euclidean distances (which serve as logits). This loss encourages query examples to be closer to their correct class prototype and further from incorrect ones.

4.5 Meta-Training and Evaluation Procedure

- **Meta-Training Loop:** The `train_meta_model` function iterates over `NUM_META_TRAIN_TASKS`:
    1. A new few-shot task is generated.
    2. Support and query images are passed through the `SNNFeatureExtractor`.
    3. Prototypes are computed from support features.
    4. Loss and accuracy are calculated on the query set using prototypical network logic.
    5. `loss.backward()` and `optimizer.step()` update the SNN feature extractor's weights using Adam optimizer with `LEARNING_RATE = 1e-3`.
- **Meta-Testing Loop:** The `test_meta_model` function evaluates the trained model on new, unseen tasks from the test dataset. No gradients are computed during testing (`torch.no_grad()`). The average accuracy across `NUM_META_TEST_TASKS` is reported.

4.6 Results

The `snnTorch` based few-shot learning model demonstrates learning capability. During meta-training, the loss generally decreases and accuracy improves over tasks. For instance, the provided console output shows:

- `Task 100/1000 | Loss: 0.2064 | Accuracy: 93.12%`
- `Task 200/1000 | Loss: 0.1771 | Accuracy: 93.59%`

This indicates that the SNN feature extractor is learning to produce effective embeddings for few-shot classification. The meta-testing accuracy reflects the model's ability to generalize to new classes it has not explicitly seen during training.

5. Discussion and Comparative Analysis

The two implementations highlight different aspects of SNN research:

Custom NumPy SNN (Implementation 1):

- **Goal:** Fundamental understanding of LIF neuron dynamics and R-STDP.
- **Complexity:** Low-level, manual implementation of neuron models and learning rules.
- **Learning:** Basic R-STDP, demonstrating a form of biologically inspired reinforcement learning. The observed accuracies indicate that while it learns, it's not designed for high-performance classification out-of-the-box.

- **Data:** Simple synthetic spike patterns.

`snnTorch`-based Prototypical Network (Implementation 2):

- **Goal:** Address few-shot learning using modern SNN frameworks and gradient-based optimization.
- **Complexity:** Higher-level, leverages `snnTorch` for efficient SNN building and PyTorch's auto-differentiation.
- **Learning:** Surrogate gradient-based backpropagation for end-to-end training of the SNN feature extractor within a meta-learning context. More powerful and scalable for complex tasks.
- **Data:** MNIST dataset, prepared into episodic few-shot tasks.

Strengths:

- **Implementation 1:** Excellent for pedagogical purposes, offering direct control and insight into SNN internal workings.
- **Implementation 2:** Demonstrates a practical application of SNNs in a challenging modern AI task (few-shot learning). The use of `snnTorch` simplifies development while maintaining SNN advantages. The Prototypical Network provides an elegant and effective meta-learning framework.

Challenges and Limitations:

- **Both:** SNNs inherently introduce temporal dynamics, requiring decisions on input encoding (e.g., rate coding, latency coding) and simulation time steps.
- **Implementation 1:** The manual R-STDP rule, while biologically inspired, might require extensive tuning for performance and scalability. The limited output accuracy suggests its primary role is illustrative rather than competitive.
- **Implementation 2:** Reliance on surrogate gradients, though effective, deviates from strict biological plausibility. Tuning hyperparameters (`BETA`, `SPIKE_GRAD_BETA`, `NUM_STEPS`) can be more complex than for ANNs. The simulation over multiple time steps adds computational overhead during training compared to feedforward ANNs.

Biological Inspirations and Analogies:

The meta-learning setup, particularly with Prototypical Networks, offers a compelling analogy to how biological brains, especially the hippocampus, enable rapid learning from limited examples. The SNN's dynamic processing and ability to learn generalizable features for new tasks resonate with the brain's capacity for rapid memory formation and flexible adaptation.

6. Conclusion

This report has detailed two distinct SNN implementations: a fundamental NumPy-based SNN with R-STDP for educational purposes, and a more advanced `snnTorch`-based convolutional SNN integrated into a Prototypical Network for few-shot image classification. The latter successfully demonstrates the viability of training SNNs using surrogate gradients to learn robust feature representations that enable rapid adaptation to novel classes. While challenges remain in terms of training complexity and hyperparameter tuning, SNNs continue to be a promising area of research due to their biological plausibility and potential for energy-efficient AI.

7. Future Work

Future directions for this work could include:

- **Exploring More Complex Few-Shot Benchmarks:** Applying the `snnTorch` Prototypical Network to datasets like miniImageNet or Omniglot to test scalability and generalization across more diverse and challenging image categories.
- **Optimizing SNN Encoding Schemes:** Investigating advanced input encoding methods (e.g., precise spike timing, burst coding) to potentially improve information representation and network performance.
- **Hybrid SNN/ANN Architectures:** Combining SNN layers with traditional ANN layers to leverage the strengths of both paradigms for specific tasks.
- **Energy Consumption Analysis:** Quantifying the potential energy benefits of the SNN architecture, especially if deployed on neuromorphic hardware, and exploring methods for sparse spiking to enhance efficiency.
- **Robustness Studies:** Evaluating the SNN-based few-shot system's resilience to adversarial attacks or noisy inputs, a potential advantage of event-driven processing.
- **Alternative Meta-Learning Algorithms:** Integrating SNN feature extractors with other meta-learning algorithms, such as MAML (Model-Agnostic Meta-Learning), to compare adaptation strategies.
- **In-depth R-STDP Tuning:** For the NumPy SNN, a more extensive hyperparameter search and analysis of its learning dynamics could provide deeper insights into its capabilities for simple pattern recognition.