DIFFERENCE PREDICTIVE CODING FOR TRAINING SPIKING NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Predictive coding networks (PCNs) offer a local-learning alternative to backpropagation in which layers communicate residual errors, aligning well with biological computation and neuromorphic hardware. In this work we introduce *Difference Predictive Coding* (DiffPC), a spike-native PC formulation for spiking neural networks. DiffPC replaces dense floating-point messages with sparse ternary spikes, provides spike-compatible target and error updates, and employs adaptive threshold schedules for event-driven operation. We validate DiffPC on fully connected and convolutional architectures, demonstrating competitive performance on MNIST (99.3%) and Fashion-MNIST (89.6%), and outperforming a backpropagation baseline on CIFAR-10. Crucially, this performance is achieved with high communication sparsity, reducing data movement by over two orders of magnitude compared to standard predictive coding. DiffPC thus establishes a faithful, hardware-aligned framework for communication-efficient training on neuromorphic platforms.

1 INTRODUCTION

The error backpropagation algorithm has been fundamental to the success of deep learning, yet its core mechanisms are widely considered biologically implausible Salvatori et al. (2023). Key limitations include the requirement for global error signals—where synaptic updates depend on information transmitted across multiple layers, far beyond locally available signals—and the reliance on continuous-valued communication and gradients, in contrast to the brain's use of discrete, event-driven signals N'dri et al. (2024). These discrepancies create a significant gap between conventional artificial neural networks (ANNs) and biological neural systems.

Artificial intelligence does not need to replicate biology—airplanes do not flap their wings—but certain biological properties are worth emulating. Examples include the brain's energy efficiency and its ability to perform robust and adaptive computation with sparse, noisy, and low-precision signals N'dri et al. (2024). These observations align with the development of neuromorphic systems, which address the limitations of conventional von Neumann architectures by co-locating memory and computation to reduce data movement Al Abdul Wahid et al. (2024), thereby enabling substantially lower energy consumption—a key biological property worth emulating. Within this hardware setting, Spiking Neural Networks (SNNs) provide a natural computational model: information is represented not by continuous activations but by discrete spikes, as observed in the brain Olshausen & Field (1996; 2004). In SNNs, neurons communicate through sparse, asynchronous events rather than dense, synchronous updates Mainen & Sejnowski (1995); Cox et al. (2000), making them intrinsically well suited for low-power implementation.

One promising framework for training such systems is Predictive Coding (PC). Originating from neuroscience, PC theorizes that the brain functions as a prediction machine, continuously generating top-down predictions of sensory input while bottom-up signals convey only the residual prediction errors Rao & Ballard (1999); Friston (2005); Spratling (2017); Huang & Rao (2011); Keller & Mrsic-Flogel (2018). Importantly, PC relies on local learning rules, where synaptic updates depend only on the activity of adjacent pre- and post-synaptic neurons. This locality makes PC highly compatible with the parallel and distributed organization of neuromorphic hardware N'dri et al. (2024); Salvatori et al. (2023).

However, despite its theoretical alignment with neuromorphic principles, the standard formulation of PC faces practical computational challenges. To infer neural activities, PC networks typically perform an iterative settling process, requiring multiple forward and backward passes of information to converge for a single input. In standard implementations, this process relies on dense, floating-point message passing, resulting in a computational overhead that notably exceeds that of backpropagation Rosenbaum (2022). This reliance on dense, continuous communication during the iterative phase can offset the efficiency gains sought by deploying SNNs on event-driven hardware. Therefore, while PC offers a solution to the global transport problem of backpropagation, its standard formulation does not fully exploit the sparsity and efficiency of neuromorphic substrates.

Combining SNNs with PC-based training is a natural research direction to address these issues Lan et al. (2022); Wacongne et al. (2012); Boerlin et al. (2013); Ororbia (2023). In this work, we introduce *Difference Predictive Coding* (DiffPC), an algorithm that reformulates the predictive coding framework for native implementation in SNNs. DiffPC seeks to address the communication overhead of standard PC by replacing dense, floating-point message passing with sparse, ternary spike-based communication. By employing spike-compatible state update rules and adaptive threshold schedules, DiffPC ensures that computation and message passing are event-driven, occurring when necessary to correct prediction errors. Our results indicate that DiffPC achieves accuracy matching or exceeding that of standard predictive coding networks (PCNs) and backpropagation trained models on benchmark datasets, while reducing the number of transmitted bits by more than two orders of magnitude compared to standard PC baselines.

- We propose *Difference Predictive Coding* (DiffPC), a spike-native framework based on novel update rules that transmit incremental state updates via sparse ternary spikes rather than broadcasting the full state, resulting in reduced communication costs.
- We introduce an adaptive threshold scheduling mechanism that enables the discrete spiking network to closely approximate the dynamics of standard continuous predictive coding with fewer timesteps.
- We empirically validate DiffPC on fully connected and convolutional architectures, demonstrating that it matches the accuracy of standard predictive coding and matches or exceeds that of Backpropagation on MNIST, Fashion-MNIST, and CIFAR-10, while reducing the number of transmitted bits by more than two orders of magnitude compared to standard predictive coding baselines.

2 Related works

Spiking neural networks. SNNs compute with discrete events and update their state only upon spike arrivals, yielding sparse, asynchronous processing that maps well to neuromorphic substrates and modern accelerators Pfeiffer & Pfeil (2018); Tavanaei et al. (2019). This event-driven operation provides a natural match to the parallel, low-power architecture of neuromorphic hardware such as TrueNorth Akopyan et al. (2015), Loihi Davies et al. (2018), Loihi 2 Intel Corporation (2021b), and SpiNNaker Furber et al. (2014). Beyond neuromorphic-vision benchmarks, deep SNNs now achieve competitive accuracy on static datasets when equipped with convolutional backbones and carefully engineered neuron and normalization layers Hu et al. (2024).

Several software frameworks have been developed to simulate and train SNNs, including Brian2 Stimberg et al. (2019), NEST Gewaltig & Diesmann (2007), SpikingJelly Fang et al. (2023), and LAVA Intel Corporation (2021a). In this work we use LAVA to verify that our methods are compatible with the Intel hardware chip Loihi 2 Intel Corporation (2021b), but also provide a PyTorch implementation for easy verification and faster runtimes.

SNN training. A key challenge in SNN learning is that spike generation is non-differentiable, preventing direct application of backpropagation. Contemporary approaches can be divided into three main families, each with distinct accuracy, latency, and efficiency trade-offs Hu et al. (2024).

(i) ANN→SNN conversion. In this approach, a ReLU ANN is trained with backpropagation and mapped to an SNN under a rate or latency coding assumption Cao et al. (2015). Practical pipelines reduce activation–rate mismatch via weight and threshold normalization Diehl et al. (2015) or reset-by-subtraction Rueckauer et al. (2017), and further tighten equivalence using quantization mapping Hu et al. (2023), clip-floor-shift activation Bu et al. (2023), and post-training parameter calibration

Li et al. (2024). Conversion scales well and can match ANN accuracy with few timesteps, but training remains off-chip with dense floating-point communication, and residual conversion error or latency may erode energy benefits.

(ii) Direct surrogate-gradient training. By treating SNNs as recurrent systems unrolled over time, the zero derivative of the spike function is replaced with a smooth surrogate, enabling backpropagation through time (BPTT) Wu et al. (2018); Neftci et al. (2019). Representative methods include temporal-loss formulations (SpikeProp) Bohte et al. (2002) and time-based error reassignment (SLAYER) Shrestha & Orchard (2018). Later advances improved optimization and representation by learning neuron dynamics such as time constants Fang et al. (2021). Significant progress was also made in normalizing membrane dynamics across time via threshold-dependent scaling (tdBN) Zheng et al. (2021), time-varying parameter decoupling (BNTT) Kim & Panda (2021), input rescaling for uniform temporal distributions (TEBN) Duan et al. (2022), or direct membrane potential regulation (MPBN) Guo et al. (2023). Finally, other methods mitigate surrogate mismatch via gradient re-weighting (TET) Deng et al. (2022), information maximization objectives (IM-Loss) Guo et al. (2022), or learnable surrogate shapes Lian et al. (2023). These methods reach state-of-the-art accuracy with $\mathcal{O}(1-8)$ timesteps, but they still rely on global backward signals and dense communication, which can limit viability of on-chip training.

(iii) Local plasticity. Purely local rules, such as spike-timing-dependent plasticity (STDP) and reward-modulated variants, are aligned with both biology and hardware constraints. However, they typically require auxiliary classifiers and tend to underperform on complex tasks Diehl & Cook (2015); Kheradpisheh et al. (2018); Ororbia (2023).

Overall, these approaches highlight a central trade-off in SNN training: methods that achieve the highest accuracy typically rely on dense, non-local signals or off-chip training, while methods that are fully local and spiking have yet to consistently match this performance on complex tasks.

Predictive coding. Predictive coding (PC) has recently emerged as an alternative to backpropagation that is both biologically motivated and compatible with event-driven computation. In PC, each layer generates predictions of activity in the layer below, while only the residual prediction errors are communicated forward Rao & Ballard (1999); Friston (2005); Spratling (2017); Huang & Rao (2011). PC has been developed into a computational framework with formal links to backpropagation and variational inference Millidge et al. (2021); Rao & Ballard (1999). A central advantage is its use of local learning rules: synaptic updates depend only on the activity of adjacent pre- and post-synaptic neurons, making the framework well suited for distributed neuromorphic implementation N'dri et al. (2024); Salvatori et al. (2023).

Several works have sought to integrate PC with SNNs Lan et al. (2022); Ororbia (2023); Lee et al. (2024). The PC-SNN algorithm formulates predictive coding in time-to-first-spike (TTFS) encoding, where each neuron spikes at most once Lan et al. (2022). This achieves unmatched energy efficiency in terms of spikes, but runtime scales exponentially with input precision (2^B timesteps for B-bit input) and must be predefined due to the single-spike restriction. Additionally, their training schema remains reliant on transmission of dense floating point numbers and is done on GPU. Recent work toward creating purely spiking predictive coding frameworks Ororbia (2023); Lee et al. (2024) has made significant progress. However, to evaluate performance on discriminative benchmarks like MNIST, these frameworks adopt a hybrid approach. The spiking network is first trained in a purely unsupervised manner to learn representations. Subsequently, its weights are frozen, and a separate, non-spiking linear classifier is trained post-hoc on rate-coded activities extracted from the network's final layer. This reliance on an external, non-spiking component for the final classification step means the reported accuracies do not reflect the performance of an end-to-end spiking system, complicating a direct assessment of their utility for fully neuromorphic deployment.

Relation to DiffPC. Event-driven 'gradient-by-spikes' approaches approximate backpropagation by discretizing gradients into spikes Bohte et al. (2000); Cai et al. (2024), which enables training and inference using only spikes. We apply a similar approach in our proposed Difference Predictive Coding. Distinct from prior work, DiffPC integrates these concepts through three key mechanisms: (1) a spike-based message passing protocol that adapts sparse ternary communication specifically for predictive coding error propagation; (2) a difference-based update rule that triggers communication only upon state changes to minimize redundancy; and (3) a cyclic threshold scheduler designed to accelerate the convergence of discrete spiking states toward continuous PCN targets.

3 BACKGROUND

3.1 SPIKING NEURAL NETWORKS

SNNs are a class of artificial neural networks that mimic the behavior of biological neurons more closely than conventional neural networks. Neurons in spiking neural networks communicate through discrete spikes, or action potentials, when their membrane potential V reaches a certain activation threshold T_0. The output of the neuron i is a function of the potential $s_i(V_i)$. The neuron model utilized in this work is based on the difference equation of the Integrate-and-Fire neuron model:

$$V_i(t+1) = V_i(t) - T_\theta s_i(V_i(t)) + \sum_j w_{ij} s_j(t), \quad V_i(0) = b_i \ , \tag{1}$$

where $V_i(t)$ is the integration variable (membrane potential), T_θ is the threshold, b_i is the bias and w_{ij} is the weight of the synapse connecting the input neuron j to the neuron i. The spike activation function $s_i(t) \in \{-1, 0, 1\}$ is

$$s_i\left(V_i(t)\right) := \begin{cases} 1 & \text{if } V_i(t) \geq T_\theta, \\ -1 & \text{if } V_i(t) \leq -T_\theta, \\ 0 & \text{otherwise} \ . \end{cases} \tag{2}$$

Activation of the Integrate-and-Fire-neuron output causes a voltage drop (damping) in the difference equation (1).

3.2 PREDICTIVE CODING

In the context of neural networks, PC proposes that each layer generates predictions of the activity in the next layer. The next layer then computes the error between the forward propagated prediction and the target activity. This error signal is then propagated backward[1] to update both the targets and synaptic weights of the previous layer. The learning objective is to reduce the overall prediction error in the network. Unlike the separate forward and backward passes of conventional deep learning.

Predictive Coding Networks (PCNs) involve a bidirectional local flow of information; predictions of current targets in one direction and prediction errors in another (Figure 1). The target activity is updated using the received prediction error and the synaptic weights using the target activity. The updates are local, which is a substantial difference from the backpropagation algorithm, where updates depend on a single error signal calculated at the final output and propagated backward through the entire network.

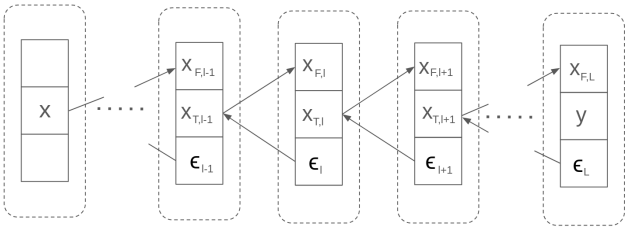


Figure 1: The structure of a multi-layer Predictive Coding Network (PCN). Each neuronal unit, bounded by the dashed line, consists of the target activity $\mathbf{x}_T$, prediction $\mathbf{x}_F$, and prediction error ϵ. The arrows indicate the flow of information between layers $l = 0, 1, \ldots, L$, where the feedforward path carries the predictions from $\mathbf{x}_{T,l}$ to $\mathbf{x}_{F,l+1}$, and the feedback path conveys the prediction errors ϵ_l which are used to update $\mathbf{x}_{T,l-1}$. Computations and updates can be asynchronous.

We review the PC principles for a conventional Multi-Layer Perceptron network (MLP) of L dense layers to the input $\mathbf{x}$ to output $\mathbf{y}$. The input and output layers are indexed as $l = 0$ and $l = L$, and between them are the hidden layers $l = 1$ to $L-1$. Predictive Coding (PC) makes use of each layer's

[1] In the PCN literature the terms top/down are more common, but we opt forward/backward for consistency with the deep learning terminology.

current activation vectors $\mathbf{x}_{T,l}$ ('T' referring to 'Target'). In addition, the layers include vectors $\mathbf{x}_{F,l}$ (F: 'forward') that are the predictions generated by the previous layer $l-1$. Ideally, the predictions and the targets should be the same.

Energy function. Layer l predictions are calculated from the target activity of the layer $l-1$

$$\mathbf{x}_{F,l} = \mathbf{W}_l \phi(\mathbf{x}_{T,l-1}) \ , \tag{3}$$

where $\mathbf{W}_l \in \mathbb{R}^{N_l \times N_{l-1}}$ are the weights, N is the number of neurons, and $\phi(\cdot)$ is an activation function. The difference between the targets and predictions is fed back as the prediction error

$$\epsilon_l = \mathbf{x}_{T,l} - \mathbf{x}_{F,l} \ . \tag{4}$$

The errors from all layers are summed to compute the 'free-energy' of a network and PC operates by minimizing this free-energy function

$$\mathcal{F} = \sum_{l=1}^{L} \|\epsilon_l\|_2^2 = \sum_{l=1}^{L} (\mathbf{x}_{T,l} - \mathbf{x}_{F,l})^2 \ . \tag{5}$$

Update steps. The main difference between PC and the gradient descent error backpropagation is that PC alternates between the two updates, the target and weight updates, which are computed locally and can be asynchronous. The synaptic weight update is

$$\mathbf{W}_l^{(t+1)} = \mathbf{W}_l^{(t)} - \alpha \frac{\partial \mathcal{F}}{\partial \mathbf{W}_l} = \mathbf{W}_l^{(t)} + \alpha \epsilon_l \phi(\mathbf{x}_{T,l-1})^\top \ , \tag{6}$$

where α is the learning rate, and the prediction update is

$$\mathbf{x}_{T,l}^{(t+1)} = \mathbf{x}_{T,l}^{(t)} - \gamma \frac{\partial \mathcal{F}}{\partial \mathbf{x}_{T,l}} = \begin{cases} \mathbf{x}_{T,l}^{(t)} - \gamma \left(\epsilon_l - \mathbf{W}_{l+1}^\top \epsilon_{l+1} \odot \phi'(x_{T,l}) \right), & \text{for } l < L, \\ \mathbf{x}_{T,l}^{(t)} - \gamma \epsilon_l, & \text{for } l = L \end{cases} \ , \tag{7}$$

where γ is the prediction learning rate and $\odot$ is the Hadamard product. Because updates only occur between adjacent layers, they must be performed iteratively to allow information to propagate throughout the entire network. See Appendix 7.1 for an intuitive explanation of predictive coding.

4 The difference predictive coding algorithm (DiffPC)

In this section, we propose the Difference Predictive Coding (DiffPC) algorithm that implements the standard PC on SNNs. To ensure that our algorithm can be deployed on a neuromorphic chip, we used the instruction set of the Intel Loihi 2 neuromorphic chip. The algorithm was verified in the official simulator because access to the actual Loihi 2 hardware is limited to Intel partners. We provide the full simulator code and the pseudocode is given in Algorithm 1.

4.1 Algorithm overview

To adapt the predictive coding framework from Section 3.2 for spiking neural networks, its floating-point computations and information transfer must be converted into discrete spikes (Algorithm 1). In this formulation, all information transmitted during the learning steps takes the form of a sequence of ternary values (-1, 0, 1).

In DiffPC, each unit maintains a *target state* x_T and an *actual state* x_A. The target state x_T represents the desired (target) activity, while the actual state x_A attempts to follow x_T, aiming to minimize the difference between them. Their difference is reduced incrementally by steps proportional to an adaptive threshold T_θ. The difference-based adjustments are communicated as spikes to subsequent layers, which integrate the incoming spiking information.

Two error variables, e_T and e_A, function in the same way and represent the errors of the two states. e_T is the target error and e_A is the actual error that attempts to align with e_T. The error e_A is adjusted by steps proportional to T_θ, which are then transmitted to subsequent layers as spikes. Since the threshold T_θ determines the step size of these updates, its schedule is critical for convergence. In the next section, we will introduce specific scheduling strategies.

Feed Forward Initialization. Before the iterative DiffPC process begins (as detailed in Algorithm 1), the network undergoes a feed forward initialization phase. In this phase, input spikes are propagated through the layers in a single pass without feedback error calculation. This rapidly establishes an initial estimate for the target activities $\mathbf{x}_T$ and predictions $\mathbf{x}_F$, reducing the number of subsequent iterative steps required for convergence. This phase effectively mimics a standard feed-forward SNN inference step to prime the network state and can be implemented by utilizing graded spikes on the Loihi 2 chip. See section 7.2 of the appendix for a more detailed explanation of the algorithm.

Algorithm 1 DiffPC Algorithm for Spiking Neural Network Training

Input: Spike signals s_{in}, s_e
Process parameters: threshold $T_\theta(t)$, learning rate $\gamma(t)$, weight lr α
Initialize: x_F, x_T, x_A, e_T, e_A, e_B, s_A, s_e

Feed Forward Initialization: Propagate input to prime x_F
for each time step t **do**
  $s^l_{\text{in}} \leftarrow W^l s^{l-1}_A$ ▷ *Receive spike input*
  $x^l_F \leftarrow x^l_F + s^l_{\text{in}} \cdot T_\theta(t-1)$ ▷ *Update forward prediction*
  if $\gamma^l(t) > 0$ **then**
    $e^l_T \leftarrow x^l_T - x^l_F$ ▷ *Compute target error*
  $x^l_T \leftarrow x^l_T + \gamma^l(t) \cdot (-e^l_T + (x^l_T > 0) \odot e^l_B)$ ▷ *Update Target Activity*
  if $\gamma^l(t) > 0$ **then**
    $e^l_T \leftarrow x^l_T - x^l_F$ ▷ *Update target error*
  $s^l_A \leftarrow \text{sign}(x^l_T - x^l_A) \odot \left(\left|x^l_T - x^l_A\right| > T_\theta(t)\right)$ ▷ *Generate spikes*
  $s^l_A \leftarrow s^l_A \odot \left(x^l_A + s^l_A \cdot T_\theta(t) > 0\right)$ ▷ *'Spiking ReLU'*
  $x^l_A \leftarrow x^l_A + T_\theta(t) \cdot s^l_A$ ▷ *Update Actual Activity*
  s_out.send(s^l_A) ▷ *Send state spikes*
  ▷ *Propagate Error:* ◁
  e_out.send(s^l_e) ▷ *Send error spikes*
  e_in$^l \leftarrow (W^{l+1})^\top s^{l+1}_e$ ▷ *Receive error input*
  $e^l_B \leftarrow e^l_B + T_\theta(t-1) \cdot$ e_in^l ▷ *Accumulate incoming errors*
  $s^l_e \leftarrow \text{sign}(e^l_T - e^l_A) \odot \left(\left|e^l_T - e^l_A\right| > T_\theta(t)\right)$ ▷ *Generate error spikes*
  $e^l_A \leftarrow e^l_A + T_\theta(t) \cdot s^l_e$ ▷ *Update actual error*
$W^l \leftarrow W^l + \alpha\, e^l_T\, \phi\big(x^{l-1}_T\big)^\top$ ▷ ***Update Weights***

4.2 T_θ AND γ SCHEDULES

We present a *cyclic scheduler*, which allows for accurate approximation of the standard PC algorithm,

$$T_\theta(t) = \frac{2^m}{2^{t \bmod n}}, \quad \gamma(t) = g(t \bmod n) \ , \tag{8}$$

where

$$g(x) = \begin{cases} \gamma, & \text{if } x = 0 \\ 0, & \text{otherwise} \end{cases} \ , \tag{9}$$

where $n, t \in \mathbb{N}^+$ denote the cycle length and timestep index, respectively, $m \in \mathbb{Z}$ sets the initial magnitude of the threshold via $T_\theta(0) = 2^m$, and $t \bmod n$ denotes the modulo operation. The set of n steps starting with $\gamma(t) = \gamma$ is referred to as γ-cycle and a set of timesteps during which we train the network with a single input and output pair is called an iteration. A single iteration therefore consists of multiple γ-cycles.

To theoretically motivate the precision of this approach, we establish the following bound on the quantization error demonstrating that, in the absence of error, the spiking states exponentially converge to their floating-point targets within a cycle.

Theorem 4.1. *Suppose that the target activity for layer l, denoted as x_T^l, satisfies $|x_T^l - x_A^l| < 2^{m+1}$ and $x_T^l > 0$. Then, after one γ-cycle of n timesteps of a cyclic scheduler, the difference between the target activity and the actual activity $|x_T^l - x_A^l|$ is less than 2^{m+1-n}.*

Proof. See Appendix 7.3. □

Though this method is able to attain great precision in the approximation of standard PC when we set n large, it comes at the cost of extra timesteps and spikes. In practice, we observe that as the PC network converges, the changes in e_T and x_T become smaller. Thus we should also scale the cyclic scheduler to be smaller as the network converges allowing us to use smaller n and still attain high accuracies. This is the motivation behind the cyclic decay scheduler defined as

$$T_\theta(t) = d(t \bmod T)\frac{2^m}{2^{t \bmod n}}, \quad \gamma(t) = g(t \mod n) \ , \tag{10}$$

where $d(t)$ is a decreasing function. We set

$$d(t) = \left(1 - (1-a)\frac{t}{T}\right) \ , \tag{11}$$

where $a \in (0, 1]$ such that $d(t \bmod T) \in (0, 1]$ and T is the length of the iteration. In addition to the cyclic decay scheduler, we introduce the *constant decay scheduler*, defined as

$$T_\theta(t) = d(t \bmod T)\, c, \quad \gamma(t) = g(t \bmod n), \tag{12}$$

where $c \in \mathbf{R}^+$. This schedule maintains a fixed threshold c that is scaled by the decay function $d(t)$, thereby reducing the update size over the course of an iteration.

5 Experiments

5.1 Data and settings

We evaluate our method using Multi-Layer Perceptron (MLP) architectures as MLPs provide a clean and well-studied baseline for predictive coding, and we extend our evaluation to Convolutional Neural Networks (CNNs) on the CIFAR-10 dataset to demonstrate the method's applicability to convolutional networks. For empirical validation, we use the MNIST, Fashion-MNIST, and CIFAR-10 benchmarks. MNIST and Fashion-MNIST comprise 60,000 training and 10,000 test grayscale images of size 28×28 across 10 classes. CIFAR-10 consists of 50,000 training and 10,000 test color images of size 32×32 across 10 classes. We train fully connected networks with one or two hidden layers for the simpler tasks, utilizing dropout. For CIFAR-10, we utilize a convolutional architecture, consisting of two convolutional layers with 5×5 kernels and stride 2 (with 10 and 5 filters respectively), followed by three fully connected layers. All models are optimized using AdamW. Data augmentation includes random translation jitter for MNIST and random horizontal flips for CIFAR-10. We assess performance by test-set classification accuracy and by spike efficiency, quantified as the average number of activity and error spikes per neuron per sample during training. Our CIFAR-10 implementation is based on code from Millidge et al. (2020); Whittington & Bogacz (2017).

Selected baselines For comparison, we focus on Convolutional and MLP networks and include both conventional floating-point and spiking implementations. In addition, we report results from spiking networks trained with alternative learning rules beyond predictive coding, providing context on how our approach compares to state-of-the-art non-PC methods

5.2 Results

Classification accuracy On MNIST, DiffPC achieves high accuracy that matches previously reported results for non-convolutional spike-based methods. For example, DiffPC-L attains 99.3%

accuracy and DiffPC-S reaches 98.2%, placing them on par with or above several recent SNN models, as shown in Table 1. On Fashion-MNIST, which presents a greater challenge, DiffPC also achieves competitive accuracy (Table 2). Finally, on CIFAR-10, DiffPC demonstrates effective scaling to convolutional architectures; DiffPC-Long achieves 65.6% accuracy, surpassing the standard backpropagation baseline of 63.5%, while DiffPC-Efficient reaches 63.3% (Table 4).

Communication efficiency Communication efficiency provides further insight into the advantages of DiffPC. On modern hardware, the energy cost of moving data is often comparable to, and in many workloads higher than, the cost of arithmetic operations Horowitz (2014); Lian et al. (2023). Because memory access and interconnect traffic can be orders of magnitude more energy-intensive than a multiply–accumulate, the number of floating-point values transmitted during training and inference is a key proxy for communication energy. In addition, spiking implementations require computation to unfold in discrete timesteps, and the number of timesteps needed for convergence strongly predicts runtime performance Li et al. (2023).

Table 3 reports the average number of bits transmitted per neuron during error propagation and the corresponding timestep counts on the MNIST task. Standard backpropagation transmits 32 bits per neuron in a single timestep, while predictive coding (PC-SE) requires 960 bits across 15 timesteps. SNN-based predictive coding (PC-SNN) is similar as they use floating point numbers during the training stage of their network. In contrast, DiffPC achieves orders-of-magnitude improvements: DiffPC-L transmits only 0.18 bits (0.09 spikes) per neuron on average across 120 timesteps, and DiffPC-S reduces this further to 0.08 bits (0.04 spikes) per neuron across 75 timesteps. These results demonstrate that DiffPC combines competitive accuracy with substantially improved efficiency in terms of communication.

We observe similar trends on the CIFAR-10 dataset, as detailed in Table 4. Here, we compare two configurations: DiffPC-Long, which utilizes a scheduler cycle length of $n = 16$, and DiffPC-Efficient, which employs a shorter cycle length of $n = 12$. Both configurations run for 15 cycles per sample. While the convolutional architecture utilizes higher-fidelity error messaging compared to the MLP used for MNIST, DiffPC retains a substantial efficiency advantage. DiffPC-Long requires only 1.9 bits per neuron, and DiffPC-Efficient further reduces this to 0.7 bits. Although these values are higher than those for MNIST, they remain significantly lower than the 32 bits required by backpropagation or the 960 bits used by standard PC, demonstrating that the communication sparsity of DiffPC scales effectively to convolutional networks.

Table 1: Comparison of the Test Accuracy of Different SNN and PC Models on the MNIST dataset. (FC denotes fully connected layers)

Method	**Network Architecture**	**Acc. (%)**
Backpropagation	784FC-1024FC-512FC-10FC	99.3
PC-SE (Standard PC) (Pinchetti et al. (2024))	784FC-128FC-128FC-128FC-10FC	98.3
STiDi-BP (Mirsadeghi et al. (2021))	40C5-P2-1000FC-10FC	99.2
SSTDP (Liu et al. (2021))	784FC-300FC-10FC	98.1
PC-SNN (Lan et al. (2022))	784FC-200FC-10FC	98.1
SRC-RNN (De Geeter et al. (2024))	784FC-512FC-512FC-512FC-10FC	98.4
FastSNN (Taylor et al. (2022))	784FC-1000FC-10FC	97.9
FastSNN (Taylor et al. (2022))	32C5-P2-64C5-P2-1000FC-10FC	99.3
DiffPC-L (Ours)	784FC-1024FC-512FC-10FC	99.3
DiffPC-S (Ours)	784FC-400FC-10FC	98.3

Table 2: Comparison of the Test Accuracy of models on the Fashion-MNIST dataset.

Method	**Network Architecture**	**Acc. (%)**
FastSNN (Taylor et al. (2022))	784FC-1000FC-10FC	89.1
FastSNN (Taylor et al. (2022))	32C5-P2-64C5-P2-1000FC-10FC	90.6
SRC-RNN (De Geeter et al. (2024))	784FC-512FC-512FC-512FC-512FC-512FC-10FC	88.5
DiffPC-M (Ours)	784FC-1000FC-10FC	89.6
DiffPC-S (Ours)	784FC-400FC-10FC	89.2

Table 3: Average bits transferred during the error propagation stage of different models, along with the timesteps used on the MNIST task. (fp: floating-point operations; sp: spikes)

Method	Ops	Network Architecture	Bits/N	Timesteps
Backpropagation	fp	784FC-1024FC-512FC-10FC	32	1
PC-SE (Pinchetti et al. (2024))	fp	784FC-1024FC-512FC-10FC	960	15
PC-SNN (Lan et al. (2022))	fp	784FC-200FC-10FC	960	15
DiffPC-L (Ours)	sp	784FC-1024FC-512FC-10FC	0.18	120
DiffPC-S (Ours)	sp	784FC-400FC-10FC	0.08	75

Table 4: Comparison of Test Accuracy and Efficiency (average bits per neuron during error propagation) on the CIFAR-10 dataset. (fp: floating-point; sp: spikes)

Method	Ops	Network Architecture	Acc. (%)	Bits/N	Timesteps
Backpropagation	fp	10C5S2-5C5S2-50FC-30FC-10FC	63.5	32	1
PC-SE (Pinchetti et al. (2024))	fp	10C5S2-5C5S2-50FC-30FC-10FC	65.3	960	15
DiffPC-Long (Ours)	sp	10C5S2-5C5S2-50FC-30FC-10FC	65.6	1.9	240
DiffPC-Efficient (Ours)	sp	10C5S2-5C5S2-50FC-30FC-10FC	63.3	0.7	180

Numerical precision – We evaluated the numerical precision of *DiffPC* by comparing the final states x_T obtained with standard predictive coding (PCN) and with our method. To quantify the approximation, we measured the absolute difference between the hidden-layer activations produced by the two algorithms.

In this experiment, we used a fixed multilayer perceptron (MLP) with architecture $128{-}200{-}10$. For each trial, we initialized the synaptic weights randomly but shared them between PCN and *DiffPC*, ensuring that differences arise solely from the approximation scheme. As inputs, we used i.i.d. random vectors sampled uniformly from $[-1, 1]^{128}$ and as output we similarly had i.i.d. random vectors sampled uniformly from $[-1, 1]^{10}$. We repeated the evaluation over 300 random weight initializations and inputs.

For each random trial, we computed the absolute difference between the final states x_T of PCN and *DiffPC*. We observed that the error depends systematically on the scheduler parameters. Specifically, the error is larger when the number of approximation steps n is small and the limit-decay value a is large, and it decreases consistently as n increases and a decreases. This trend was robust across weight initializations and random inputs, showing that approximation precision can be tuned directly through scheduler parameters.

This constitutes a general test of numerical fidelity within the specified architecture and activation function for three reasons. First, it eliminates dataset-specific structure and labels, so the comparison probes only the update rules rather than task semantics. Second, by combining random bounded inputs with many random weights, it explores a wide region of the state space. Third, the use of shared weights across both algorithms isolates the approximation error from any modeling differences.

The results seen in Table 5 demonstrate that *DiffPC* provides a close approximation of standard PCN dynamics under random input conditions, confirming that the method faithfully reproduces PCN across a broad range of states for the given architecture.

Table 5: Mean absolute difference between DiffPC and standard PC, averaged over three seeds. The value after the $\pm$ symbol represents the sample standard deviation. Lower is better.

n \ a	1.0	0.5	0.25	0.1
3	0.1506 ± 0.0051	0.0627 ± 0.0014	0.0292 ± 0.0014	0.0168 ± 0.0004
4	0.0750 ± 0.0018	0.0312 ± 0.0005	0.0158 ± 0.0005	0.0106 ± 0.0003
5	0.0373 ± 0.0006	0.0169 ± 0.0006	0.0102 ± 0.0004	0.0083 ± 0.0006
6	0.0197 ± 0.0005	0.0107 ± 0.0005	0.0083 ± 0.0005	0.0075 ± 0.0006
7	0.0117 ± 0.0002	0.0085 ± 0.0005	0.0075 ± 0.0006	0.0072 ± 0.0006

6 Conclusion

In this work, we presented Difference Predictive Coding, a learning framework that reformulates standard predictive coding for native implementation in spiking neural networks. By replacing dense floating-point communication with sparse, event-driven ternary spikes, DiffPC addresses the data movement bottleneck that typically constrains on-chip training.

Our results on MNIST, Fashion-MNIST, and CIFAR-10 indicate that DiffPC approximates continuous predictive coding dynamics with high precision. Crucially, it achieves competitive classification accuracy while greatly reducing the number of transmitted bits compared to backpropagation and standard predictive coding baselines. These findings suggest that DiffPC offers a viable pathway for spiking based learning on neuromorphic systems.

Looking forward, a primary direction for future research in this domain is the evaluation of DiffPC on significantly deeper architectures. Recent advancements, such as μ-PC Innocenti et al. (2025a), have shown that predictive coding can scale to deep ResNets when inference dynamics are stabilized. Since DiffPC is designed as a faithful discretization of PC, it is reasonable to hypothesize that these stabilization techniques would transfer to this spike-based framework. A valuable extension of this work would be to quantify the layer-wise deviation between DiffPC and continuous PC states in deep networks, establishing how the spike-communication window must scale to maintain approximation fidelity.

A complementary future direction concerns temporally correlated data. Recent work shows that when inputs evolve smoothly over time, PC inference can be warm-started from previous states, reducing inference iterations by half and substantially lowering the number of weight updates Zadeh-Jousdani et al. (2025). Prototype-based continual-learning methods similarly demonstrate that reducing update frequency yields large energy benefits on neuromorphic hardware such as Loihi 2 Hajizada et al. (2024; 2025). Since DiffPC is intrinsically event-driven—remaining silent during steady states and emitting spikes only on changes—leveraging temporal priors may further reduce both inference steps and plasticity operations. Quantifying these temporal-sparsity benefits represents another natural extension of the present work.

Finally, beyond algorithmic scalability and temporal experiments, transitioning DiffPC from simulation to physical neuromorphic hardware remains a critical milestone. Deployment on platforms such as Intel Loihi 2 would allow for the assessment of the protocol under real-world hardware constraints and provide a rigorous verification of its potential energy efficiency advantages.

References

Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, 34(10):1537–1557, 2015.

Seham Al Abdul Wahid, Arghavan Asad, and Farah Mohammadi. A survey on neuromorphic architectures for running artificial intelligence algorithms. *Electronics*, 13(15):2963, 2024.

M Boerlin, CK Machens, and S Denève. Predictive coding of dynamical variables in balanced spiking networks. *PLoS computational biology*, 9(11):e1003258, 2013.

Sander M Bohte, Joost N Kok, and Johannes A La Poutré. Spikeprop: backpropagation for networks of spiking neurons. In *ESANN*, volume 48, pp. 419–424. Bruges, 2000.

Sander M Bohte, Joost N Kok, and Han La Poutre. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48(1-4):17–37, 2002.

Tong Bu, Wei Fang, Jianhao Ding, PengLin Dai, Zhaofei Yu, and Tiejun Huang. Optimal ann-snn conversion for high-accuracy and ultra-low-latency spiking neural networks. *arXiv preprint arXiv:2303.04347*, 2023.

Zhengyu Cai, Hamid Rahimian Kalatehbali, Ben Walters, Mostafa Rahimi Azghadi, Roman Genov, and Amirali Amirsoleimani. Advancing image classification with phase-coded ultra-efficient

spiking neural networks. In *2024 IEEE international symposium on circuits and systems (ISCAS)*, pp. 1–5. IEEE, 2024.

Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1):54–66, 2015.

Charles L Cox, Winfried Denk, David W Tank, and Karel Svoboda. Action potentials reliably invade axonal arbors of rat neocortical neurons. *Proceedings of the National Academy of Sciences*, 97 (17):9724–9728, 2000.

Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

Florent De Geeter, Damien Ernst, and Guillaume Drion. Spike-based computation using classical recurrent neural networks. *Neuromorphic Computing and Engineering*, 4(2):024007, 2024.

Shikuang Deng, Yuhang Li, Shanghang Zhang, and Shi Gu. Temporal efficient training of spiking neural network via gradient re-weighting. *arXiv preprint arXiv:2202.11946*, 2022.

Peter U Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in computational neuroscience*, 9:99, 2015.

Peter U Diehl, Daniel Neil, Jonathan Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing. In *2015 International joint conference on neural networks (IJCNN)*, pp. 1–8. ieee, 2015.

Chaoteng Duan, Jianhao Ding, Shiyan Chen, Zhaofei Yu, and Tiejun Huang. Temporal effective batch normalization in spiking neural networks. *Advances in Neural Information Processing Systems*, 35:34377–34390, 2022.

Wei Fang, Zhaofei Yu, Yanqi Chen, Timothée Masquelier, Tiejun Huang, and Yonghong Tian. Incorporating learnable membrane time constant to enhance learning of spiking neural networks. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 2661–2671, 2021.

Wei Fang, Yanqi Chen, Jianhao Ding, Zhaofei Yu, Timothée Masquelier, Ding Chen, Liwei Huang, Huihui Zhou, Guoqi Li, and Yonghong Tian. Spikingjelly: An open-source machine learning infrastructure platform for spike-based intelligence. *Science Advances*, 9(40):eadi1480, 2023.

Karl Friston. A theory of cortical responses. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 360(1456):815–836, 2005.

Steve B Furber, Francesco Galluppi, Steve Temple, and Luis A Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, 2014.

Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4): 1430, 2007.

Yufei Guo, Yuanpei Chen, Liwen Zhang, Xiaode Liu, Yinglei Wang, Xuhui Huang, and Zhe Ma. Im-loss: information maximization loss for spiking neural networks. *Advances in Neural Information Processing Systems*, 35:156–166, 2022.

Yufei Guo, Yuhan Zhang, Yuanpei Chen, Weihang Peng, Xiaode Liu, Liwen Zhang, Xuhui Huang, and Zhe Ma. Membrane potential batch normalization for spiking neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 19420–19430, 2023.

Elvin Hajizada, Balachandran Swaminathan, and Yulia Sandamirskaya. Continual learning for autonomous robots: A prototype-based approach, 2024. URL `https://arxiv.org/abs/2404.00418`.

Elvin Hajizada, Danielle Rager, Timothy Shea, Leobardo Campos-Macias, Andreas Wild, Eyke Hüllermeier, Yulia Sandamirskaya, and Mike Davies. Real-time continual learning on intel loihi 2, 2025. URL `https://arxiv.org/abs/2511.01553`.

Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pp. 10–14. IEEE, 2014.

Yangfan Hu, Qian Zheng, Xudong Jiang, and Gang Pan. Fast-snn: Fast spiking neural network by converting quantized ann. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45 (12):14546–14562, 2023.

Yangfan Hu, Qian Zheng, Guoqi Li, Huajin Tang, and Gang Pan. Toward large-scale spiking neural networks: A comprehensive survey and future directions. *arXiv preprint arXiv:2409.02111*, 2024.

Yan Huang and Rajesh PN Rao. Predictive coding. *Wiley Interdisciplinary Reviews: Cognitive Science*, 2(5):580–593, 2011.

Francesco Innocenti, El Mehdi Achour, and Christopher L Buckley. μpc: Scaling predictive coding to 100+ layer networks. *arXiv preprint arXiv:2505.13124*, 2025a.

Francesco Innocenti, El Mehdi Achour, and Christopher L. Buckley. μpc: Scaling predictive coding to 100+ layer networks, 2025b. URL https://arxiv.org/abs/2505.13124.

Intel Corporation. Intel advances neuromorphic computing with loihi 2 and lava software framework. https://www.intel.com/content/www/us/en/newsroom/news/intel-advances-neuromorphic-loihi-2.html, September 2021a.

Intel Corporation. Loihi 2 Technology Brief. Technical report, Intel Corporation, 2021b. URL https://www.intel.com/content/www/us/en/research/neuromorphic-computing-loihi-2-technology-brief.html. Accessed: 2025-09-24.

Georg B Keller and Thomas D Mrsic-Flogel. Predictive processing: a canonical cortical computation. *Neuron*, 100(2):424–435, 2018.

Saeed Reza Kheradpisheh, Mohammad Ganjtabesh, Simon J Thorpe, and Timothée Masquelier. Stdp-based spiking deep convolutional neural networks for object recognition. *Neural Networks*, 99:56–67, 2018.

Youngeun Kim and Priyadarshini Panda. Revisiting batch normalization for training low-latency deep spiking neural networks from scratch. *Frontiers in neuroscience*, 15:773954, 2021.

Mengting Lan, Xiaogang Xiong, Zixuan Jiang, and Yunjiang Lou. Pc-snn: Supervised learning with local hebbian synaptic plasticity based on predictive coding in spiking neural networks. *arXiv preprint arXiv:2211.15386*, 2022.

Kwangjun Lee, Shirin Dora, Jorge F Mejias, Sander M Bohte, and Cyriel MA Pennartz. Predictive coding with spiking neurons and feedforward gist signaling. *Frontiers in Computational Neuroscience*, 18:1338280, 2024.

Yuhang Li, Tamar Geller, Youngeun Kim, and Priyadarshini Panda. Seenn: Towards temporal spiking early-exit neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 63327–63342. Curran Associates, Inc., 2023. doi: 10.5555/3666122.3668886. URL https://papers.nips.cc/paper_files/paper/2023/hash/9a3f3c16c1d4b89b3b4e2f24f44f5cb4-Abstract-Conference.html.

Yuhang Li, Shikuang Deng, Xin Dong, and Shi Gu. Error-aware conversion from ann to snn via post-training parameter calibration. *International Journal of Computer Vision*, 132(9):3586–3609, 2024.

Shuang Lian, Jiangrong Shen, Qianhui Liu, Ziming Wang, Rui Yan, and Huajin Tang. Learnable surrogate gradient for direct training spiking neural networks. In *IJCAI*, pp. 3002–3010, 2023.

Fangxin Liu, Wenbo Zhao, Yongbiao Chen, Zongwu Wang, Tao Yang, and Li Jiang. Sstdp: Supervised spike timing dependent plasticity for efficient spiking neural network training. *Frontiers in Neuroscience*, 15:756876, 2021.

Zachary F Mainen and Terrence J Sejnowski. Reliability of spike timing in neocortical neurons. *Science*, 268(5216):1503–1506, 1995.

Beren Millidge, Alexander Tschantz, and Christopher L Buckley. Predictive coding approximates backprop along arbitrary computation graphs. *arXiv preprint arXiv:2006.04182*, 2020.

Beren Millidge, Anil Seth, and Christopher L Buckley. Predictive coding: A theoretical and experimental review. *arXiv preprint arXiv:2107.12979*, 2021.

Maryam Mirsadeghi, Majid Shalchian, Saeed Reza Kheradpisheh, and Timothée Masquelier. Stidibp: Spike time displacement based error backpropagation in multilayer spiking neural networks. *Neurocomputing*, 427:131–140, 2021.

Antony W N'dri, William Gebhardt, Céline Teulière, Fleur Zeldenrust, Rajesh PN Rao, Jochen Triesch, and Alexander Ororbia. Predictive coding with spiking neural networks: a survey. *arXiv preprint arXiv:2409.05386*, 2024.

Emre O Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *IEEE Signal Processing Magazine*, 36(6):51–63, 2019.

Bruno A Olshausen and David J Field. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature*, 381(6583):607–609, 1996.

Bruno A Olshausen and David J Field. Sparse coding of sensory inputs. *Current opinion in neurobiology*, 14(4):481–487, 2004.

Alexander Ororbia. Spiking neural predictive coding for continually learning from data streams. *Neurocomputing*, 544:126292, 2023.

Michael Pfeiffer and Thomas Pfeil. Deep learning with spiking neurons: Opportunities and challenges. *Frontiers in neuroscience*, 12:409662, 2018.

Luca Pinchetti, Chang Qi, Oleh Lokshyn, Gaspard Olivers, Cornelius Emde, Mufeng Tang, Amine M'Charrak, Simon Frieder, Bayar Menzat, Rafal Bogacz, et al. Benchmarking predictive coding networks–made simple. *arXiv preprint arXiv:2407.01163*, 2024.

Rajesh PN Rao and Dana H Ballard. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1):79–87, 1999.

Robert Rosenbaum. On the relationship between predictive coding and backpropagation. *Plos one*, 17(3):e0266102, 2022.

Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in neuroscience*, 11:682, 2017.

Tommaso Salvatori, Ankur Mali, Christopher L Buckley, Thomas Lukasiewicz, Rajesh PN Rao, Karl Friston, and Alexander Ororbia. A survey on brain-inspired deep learning via predictive coding. *arXiv preprint arXiv:2308.07870*, 2023.

Sumit B Shrestha and Garrick Orchard. Slayer: Spike layer error reassignment in time. *Advances in neural information processing systems*, 31, 2018.

Michael W Spratling. A review of predictive coding algorithms. *Brain and Cognition*, 112:92–97, 2017.

Marcel Stimberg, Romain Brette, and Dan FM Goodman. Brian 2, an intuitive and efficient neural simulator. *elife*, 8:e47314, 2019.

Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural networks*, 111:47–63, 2019.

Luke Taylor, Andrew King, and Nicol Harper. Robust and accelerated single-spike spiking neural network training with applicability to challenging temporal tasks. *arXiv preprint arXiv:2205.15286*, 2022.

C Wacongne, J-P Changeux, and S Dehaene. A neuronal model of predictive coding accounting for the mismatch negativity. *Journal of Neuroscience*, 32(11):3665–3678, 2012.

James CR Whittington and Rafal Bogacz. An approximation of the error backpropagation algorithm in a predictive coding network with local hebbian synaptic plasticity. *Neural computation*, 29(5): 1229–1262, 2017.

Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. Spatio-temporal backpropagation for training high-performance spiking neural networks. *Frontiers in neuroscience*, 12:331, 2018.

Darius Masoum Zadeh-Jousdani, Elvin Hajizada, and Eyke Hüllermeier. Efficient online learning with predictive coding networks: Exploiting temporal correlations, 2025. URL `https://arxiv.org/abs/2510.25993`.

Hanle Zheng, Yujie Wu, Lei Deng, Yifan Hu, and Guoqi Li. Going deeper with directly-trained larger spiking neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pp. 11062–11070, 2021.

7 Appendix

7.1 Intuition of Predictive Coding Training

To clarify the mechanics of the learning process, we provide a step-by-step intuition of how a Predictive Coding Network (PCN) learns to classify inputs. Unlike Backpropagation, which calculates gradients of a loss function with respect to weights, PCN frames learning as an energy minimization problem involving local neuronal activities.

The training process for a single input-label pair $(\mathbf{x}, \mathbf{y})$ proceeds as follows:

1. **Prediction (Forward Pass):** The input $\mathbf{x}$ is clamped to the input layer. The network propagates activity forward layer-by-layer to generate a prediction at the output layer.
2. **Constraint (Clamping):** During training, the output layer is clamped to the correct label $\mathbf{y}$. This immediately creates a prediction error at the output layer (since the network's initial guess likely did not match $\mathbf{y}$).
3. **Relaxation (Backward Error Flow):** This is the core of PC. The error at the output layer implies that the *penultimate* layer's activity was "wrong." This error flows backward, pulling the hidden layer neurons away from their original values toward states that *would have* produced the correct output. This happens iteratively across all layers. The network "relaxes" into a low-energy state where the activities are consistent with both the input and the correct label.
4. **Weight Update (Learning):** Once the neuron activities have shifted to this better configuration, the synaptic weights are updated locally. The update rule effectively says: "Change the weight so that next time, this input naturally produces this 'better' hidden activity."

7.2 Algorithm breakdown

Update forward prediction – Forward prediction $\mathbf{x}_F$ is updated using the incoming spike signal $\mathbf{s}_A$ from the previous layer,

$$\mathbf{x}_F^{(t+1)} \leftarrow \mathbf{x}_F^{(t)} + \mathbf{W}^l \mathbf{s}_A^{l-1} \cdot T_\theta(t-1) \ . \tag{13}$$

Update Target Activity and Generate Spikes

The core of the DiffPC algorithm lies in iteratively updating the target activity vector $\mathbf{x}_T$ for each layer and communicating changes via spikes. The process begins by adjusting $\mathbf{x}_T$ to minimize

prediction error, followed by generating spikes based on the discrepancy between this new target and the layer's current state.

First, the target activity $\mathbf{x}_T$ is updated based on both the local prediction error $\mathbf{e}_T$ and the error propagated from the subsequent layer, which is accumulated in $\mathbf{e}_B$. This update, performed only when the learning rate $\gamma(t)$ is active, is defined as:

$$\mathbf{x}_T \leftarrow \mathbf{x}_T + \gamma(t) \cdot (-\mathbf{e}_T + (\mathbf{x}_T > 0) \odot \mathbf{e}_B) \quad .$$

This rule closely mirrors the standard PC update in Equation 7. The term $-\mathbf{e}_T$ corrects for local prediction error, while the second term incorporates feedback from the next layer. The element-wise condition $(\mathbf{x}_T > 0)$ serves as the derivative of the ReLU activation function, ensuring that updates are only applied to active neurons. During training, the target activities of the input and output layers are clamped to the provided data and labels, respectively. During inference, only the input layer is clamped.

Next, the algorithm generates spikes to communicate the necessary adjustments for bringing the layer's *actual* state, $\mathbf{x}_A$, in line with the newly updated *target* state, $\mathbf{x}_T$. Instead of transmitting dense floating-point values, DiffPC sends sparse ternary spikes. A spike is generated only if the magnitude of the difference between the target and actual activity for a given neuron exceeds the adaptive threshold $T_\theta(t)$.

The activity spike vector $\mathbf{s}_A$ is computed as follows:

$$\mathbf{s}_A = \text{sign}(\mathbf{x}_T - \mathbf{x}_A) \odot (|\mathbf{x}_T - \mathbf{x}_A| > T_\theta(t)) \quad , \tag{14}$$

where $\odot$ denotes the Hadamard product. The $\text{sign}(\cdot)$ function determines the spike's polarity (+1 or -1), while the comparison operator produces a binary mask, ensuring that spikes are only generated when the required update is significant. This event-driven mechanism ensures that communication is sparse, as spikes are only transmitted to correct meaningful deviations from the target state.

Spiking ReLU – To implement a non-linear transfer function similar to the Rectified Linear Unit (ReLU) in conventional neural networks, we propose a masking operation that effectively prevents the actual neural activity $\mathbf{x}_A$ from becoming negative.

$$\mathbf{s}_A^+ \leftarrow \mathbf{s}_A \odot (\mathbf{x}_A + \mathbf{s}_A \cdot T_\theta(t) > 0) \quad . \tag{15}$$

The spiking ReLU ensures that only correction spikes in $\mathbf{s}_A$ maintaining $\mathbf{x}_A \geq 0$ are allowed, effectively implementing a ReLU-like activation function. We can also implement a clipped ReLU activation function in a similar manner by setting an additional constraint:

$$\mathbf{s}_A^+ \leftarrow \mathbf{s}_A \odot (1 > \mathbf{x}_A + \mathbf{s}_A \cdot T_\theta(t) > 0) \quad . \tag{16}$$

Update activation – The spiking ReLU produces a simple update step to update the actual activity $\mathbf{x}_A$:

$$x_A \leftarrow x_A + T_\theta(t) \cdot \mathbf{s}_A^+ \quad . \tag{17}$$

The activation update operation adjusts $\mathbf{x}_A$ towards $\mathbf{x}_T$.

Error encoding in spikes – The target update is the same as in the standard PC in Sec. 3.2.

The target error vector ϵ_l is computed using (4). Similarly to the original PC, this error term received from the following layer serves as a measure of how well the current forward prediction matches the target activity. However, unlike the standard PC we cannot directly use the update rule (7) since the error is encoded in the form of spikes. Instead, after computing the target error $e_T = \epsilon_l$, the DiffPC algorithm generates error spikes $\mathbf{s}_e$ based on the difference between $\mathbf{e}_T$ and the actual error $\mathbf{e}_A$,

$$\mathbf{s}_e = \text{sign}(\mathbf{e}_T - \mathbf{e}_A) \odot (|\mathbf{e}_T - \mathbf{e}_A| > T_\theta(t)) \quad . \tag{18}$$

The spikes $\mathbf{s}_e$ are then sent to the following layers as error signals.

Accumulate incoming errors – Errors from the next layers are integrated into the network using the accumulated error vector e_B. This vector represents the sum of the incoming error signals weighted by the threshold $T_\theta(t)$,

$$e_B \leftarrow e_B + T_\theta(t) \cdot \text{e\_in} \quad ,$$

where e_in denotes the incoming error signals from the next layer. The accumulated error vector e_B helps refine the target activity x_T by incorporating feedback from different layers of the network.

The actual error e_A is then updated using previously generated error spikes s_e,

$$e_A \leftarrow e_A + T_\theta(t) \cdot s_e \ .$$

Weight Update Mechanism – The update rule for a single sample, derived from minimizing free energy, is computed as $\Delta W_{ij} \propto e_{T,i} \cdot \phi(x_{T,j})$, where $e_{T,i}$ is the post-synaptic error state and $x_{T,j}$ is the pre-synaptic activity state. This computation is compatible with neuromorphic hardware like Loihi 2, which supports fixed-precision multiplication and accumulation of local variables.

7.3 Proof of Convergence

Here we provide the proof for Theorem 4.1 regarding the convergence of the cyclic scheduler.

Proof. Consider the cyclic scheduler where $T_\theta(0) = 2^m$. At the first timestep $t = 0$, if $2^{m+1} > |x_T^l - x_A^l| > 2^m$, then x_A^l is updated by 2^m. Consequently, after the update, we have:

$$|x_T^l - x_A^l| < 2^m.$$

The same trivially holds if $|x_T^l - x_A^l| < 2^m$ already held on the first timestep. At the next timestep $t = 1$, with $T_\theta(1) = 2^{m-1}$, the difference $|x_T^l - x_A^l|$ can again be reduced by 2^{m-1} if it exceeds 2^{m-1}. Repeating this process over n timesteps, each reduction step halves the threshold compared to the previous timestep from which the result follows by induction. □

7.4 Hyperparameter Analysis

To understand the impact of our key scheduler hyperparameters, the cycle length n and the decay factor a, we performed an extensive grid search. The results, visualized in Figure 2, reveal a clear trade-off between classification accuracy, communication cost (spikes), and runtime (timesteps).

Figure 2a shows that, with a fixed decay (a=1.0), increasing the cycle length n generally improves performance. On both datasets, the accuracy gains diminish as performance saturates for sufficiently large n. However, this accuracy gain comes at a direct cost. By definition, a larger n value increases the number of timesteps per iteration, which in turn increases both the total runtime and the number of spikes transmitted.

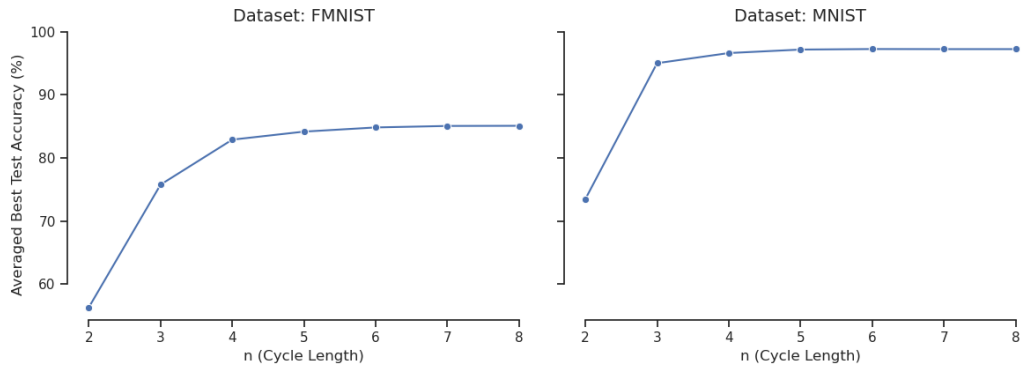
The role of the adaptive decay parameter a is to tune the precision of spike-based communication within a fixed number of timesteps. Figure 2b illustrates this trade-off effectively. For smaller cycle lengths, reducing a from 1.0 to smaller values provides a notable accuracy boost. This performance gain is achieved by allowing the adaptive threshold to decrease more over the iteration, which in turn generates more spikes to represent the error signals with higher fidelity. This can increase communication cost, but does not increase the runtime as the number of timesteps per iteration is fixed by n.

Crucially, the benefit of a smaller a diminishes as n increases. For large n, the performance is already high and stable, and varying a has little to no effect on the final accuracy. This suggests that a long cycle length n already provides sufficient timesteps for the network's states to converge with high precision. In this regime, the fine-tuning offered by the adaptive decay (a¡1.0) becomes redundant, as the inherent precision of the long spike train is already maximal for the task.

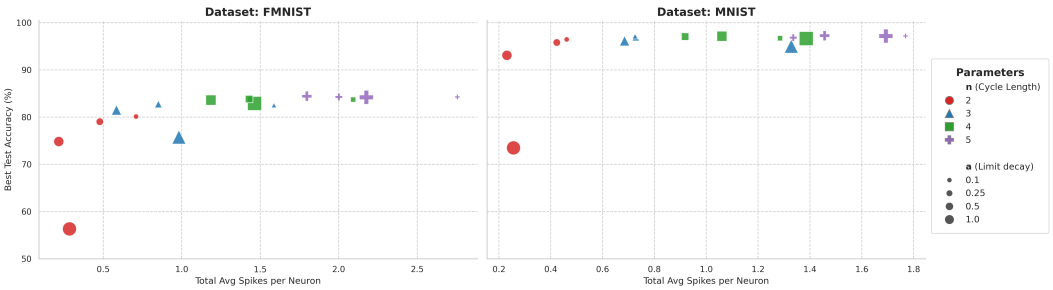
7.5 Hyperparameter Selection Heuristics

Choosing optimal hyperparameters for DiffPC, as with many complex models, is a non-trivial task without a closed-form solution. However, we have identified several heuristics that provide a strong starting point for tuning the network for a new task.

From standard predictive coding theory, a functional network requires a minimum number of relaxation steps, typically at least twice the depth of the network, to allow information to propagate fully between the input and output layers (see e.g. A.3.2 of Innocenti et al. (2025b)). This principle



(a) Effect of cycle length n (see Eq. 8) on test accuracy with fixed decay (a=1.0). Performance increases with n on both MNIST and Fashion-MNIST, though returns diminish for sufficiently large n.



(b) Trade-off between test accuracy and communication cost (spikes). Varying the decay parameter a (marker size) can improve accuracy, particularly for smaller cycle lengths n (marker shape/color), at the cost of increased spike activity.

Figure 2: Analysis of scheduler hyperparameters n and a on MNIST and Fashion-MNIST. The top figure isolates the effect of n, while the bottom figure shows the interplay between n, a, accuracy, and spike cost.

provides a useful guideline for the minimum number of timesteps required. For DiffPC, we offer the following more specific guidance.

Choosing m For the cyclic scheduler, a robust choice is m=2 when using a clipped activation function like ReLU6. The parameter m sets the initial and largest threshold value in a cycle, which is 2^m. If this value is significantly larger than the maximum possible activation (e.g., 6 for ReLU6), the initial timesteps will generate no spikes, as the difference $|x_T - x_A|$ will never exceed the threshold. Setting m=2 yields an initial threshold of 4, which is on the same order of magnitude as the activation range, ensuring that the spike generation process is active from the beginning of the cycle.

Choosing n and a The parameters n and a jointly control the trade-off between runtime, communication cost, and precision. A practical approach to tuning them is a two-step process:

1. **Find an effective cycle length n.** First, set a=1.0 (disabling adaptive decay) and incrementally increase n while monitoring test accuracy. Continue until performance saturates, establishing a baseline for the required precision.

2. **Optimize for efficiency.** Once a saturation point n_{sat} is found, one can attempt to reduce the cycle length to $n_{\text{new}} = n_{\text{sat}} - k$ for some small integer k, thereby reducing runtime. To compensate for the potential loss of precision, the decay factor can be set to $a \approx 1/2^k$.

The reasoning for this two-step process is as follows. Reducing the cycle length by k steps removes the k timesteps that have the smallest, and therefore most precise, threshold values. To compensate, a smaller value of a is used to scale down the entire threshold schedule within the new, shorter cycle. The heuristic $a \approx 1/2^k$ is specifically chosen because it ensures that the final, smallest threshold in the new n_{new}-step cycle is approximately equal to what the final threshold was in the original n_{sat}-step cycle with a=1.0. This approach aims to recover the necessary representational fidelity while benefiting from a shorter runtime.

8 Table of Notation

Table 6 summarizes the mathematical symbols used in the Difference Predictive Coding (DiffPC) algorithm.

Table 6: Nomenclature and Symbols

Symbol	Description
Shared Variables (Standard PC & DiffPC)	
l	Layer index, $l \in \{0, \ldots, L\}$.
$\mathbf{W}^l$	Synaptic weight matrix connecting layer $l-1$ to l.
$\mathbf{x}_F$	**Forward Prediction**. The prediction generated by the previous layer.
$\mathbf{x}_T$	**Target Activity**. The ideal state calculated to minimize prediction energy.
ϵ	**Prediction Error**. The difference between target and prediction ($\mathbf{x}_T - \mathbf{x}_F$).
$\gamma(t)$	Inference learning rate at time t.
DiffPC-Specific States (Spiking Implementation)	
$\mathbf{x}_A$	**Actual Activity**. The discrete state that tracks the shared target $\mathbf{x}_T$, updated via spikes.
$\mathbf{s}_A$	**Activity Spikes**. Ternary spikes $\{-1, 0, 1\}$ communicating changes in $\mathbf{x}_A$.
$\mathbf{e}_T$	**Target Error**. The local error variable (functionally equivalent to ϵ in this context).
$\mathbf{e}_A$	**Actual Error**. The discrete state that tracks $\mathbf{e}_T$, updated via spikes.
$\mathbf{s}_e$	**Error Spikes**. Ternary spikes communicating changes in the error state.
$\mathbf{e}_B$	**Backward Error**. The error signal accumulated from layer $l+1$.
Scheduler & Thresholds	
$T_\theta(t)$	Adaptive firing threshold at time t.
m	Scheduler magnitude parameter (sets max threshold 2^m).
n	Scheduler cycle length (periodicity of the steps).
a	Decay factor for the cyclic decay scheduler.

9 Event-Driven Response to Input Changes

To demonstrate the event-driven nature of our method, we conducted an experiment to measure the network's spiking activity in response to a changing input. We presented a static input image from the test set to a trained DiffPC network and monitored the total number of activity spikes (s_A) across all layers over time. After an initial period of 25 timesteps, we introduced an abrupt change by shifting the input image by a single pixel.

The results, averaged over 1000 different input images, are shown in Figure 3. Initially, there is a burst of spiking activity as the network processes the new image. This activity quickly subsides, and the network becomes nearly silent as its internal state converges to a stable representation of

the static input. At timestep 25 when the input is shifted, the network immediately responds with another burst of spikes, which then decays as it settles into a new stable state.

This behavior highlights a key feature of DiffPC: computation is performed only when necessary to process new or changed information. For applications where inputs may remain static for periods of time, this event-driven property suggests the potential for energy savings by eliminating redundant processing, making the approach highly suitable for energy-constrained neuromorphic hardware.

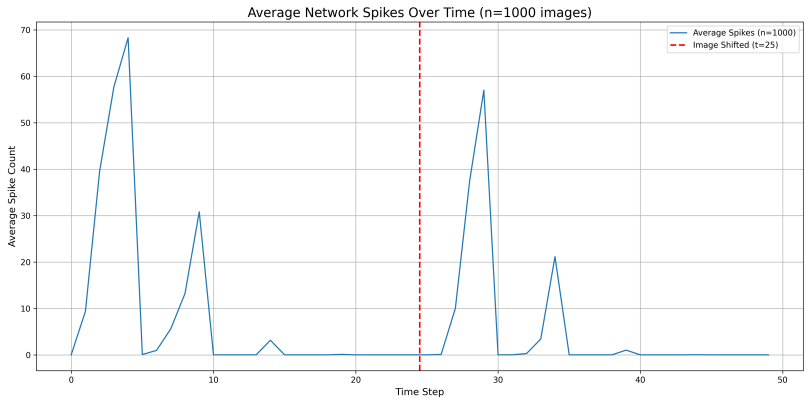


Figure 3: Event-driven spiking in a trained DiffPC network. The network shows an initial burst of spikes when an image is presented, then falls silent. A second burst of activity is triggered precisely at timestep 25, when the input image is shifted by one pixel, demonstrating that the network only computes in response to change.

Reproducibility Statement

We are committed to ensuring the reproducibility of our research. The complete PyTorch implementation of the Difference Predictive Coding algorithm, including model architectures, schedulers, and training procedures, will be made available as supplementary material. Our core method is detailed in Section 7.2, with a step-by-step breakdown provided in the Appendix. The source code includes the exact configurations and hyperparameters used to generate all reported results, including classification accuracy on MNIST (Table 1) and Fashion-MNIST (Table 2), and the communication efficiency analysis (Table 3). Further, the codes used to generate the CIFAR-10 results will also be made public. All experiments were conducted using standard public datasets, and the specific data processing pipelines are explicitly defined within our implementation.

The Use of Large Language Models (LLMs)

In the preparation of this manuscript, Large Language Models (LLMs) were utilized as a general-purpose assistive tool. The authors take full responsibility for all content, ensuring its scientific accuracy and originality. The specific roles of the LLMs are outlined below:

- **Writing Assistance:** LLMs were employed to improve the language and clarity of the manuscript. This included refining sentence structures, correcting grammatical errors, and ensuring overall readability. The core scientific ideas, arguments, and conclusions presented are entirely the work of the authors.
- **Literature Discovery:** LLMs were used as a tool to aid in the literature review process by suggesting potentially related academic papers and summarizing established concepts. All works cited in this paper were subsequently retrieved, read, and critically evaluated by the authors to verify their relevance and accuracy.
- **Coding Support:** LLMs assisted in the software development process by generating boilerplate code, helping to debug specific code segments, and suggesting algorithmic opti-

mizations. The overall design of the experiments, the core logic of the implementation, and the final analysis were conceived and performed by the authors.