

8 Supplementary Materials

8.1 Effect of Surrogate Gradients on Gradient Alignment

When training spiking neural networks (SNNs), the non-differentiability of the spiking function is often handled by replacing it with a smooth surrogate function. In this section, we formalize how the slope of the surrogate gradient affects weight updates in deeper networks, supporting the main paper’s findings on cosine similarity decay. We replace the binary spike with a sigmoid to obtain a tractable closed-form baseline, however, we believe the qualitative findings hold for spiking neurons as well.

Network Setup. We consider an ANN feedforward network with two hidden layers and sigmoid activations. Let $\mathbf{x} \in \mathbb{R}^n$ denote the input vector. Each layer performs an affine transformation followed by a nonlinearity:

$$\begin{aligned} \mathbf{z}_1 &= \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1, & \mathbf{a}_1 &= \sigma(\mathbf{z}_1) \\ \mathbf{z}_2 &= \mathbf{W}_2 \mathbf{a}_1 + \mathbf{b}_2, & \mathbf{a}_2 &= \sigma(\mathbf{z}_2) \\ \mathbf{z}_3 &= \mathbf{W}_3 \mathbf{a}_2 + \mathbf{b}_3, & \mathbf{a}_3 &= \mathbf{z}_3 \end{aligned}$$

Here, $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid activation. The output layer is a linear layer, not followed by an activation.

Backpropagation and Surrogates. During backpropagation, we can compute δ for a neuron i in the layer l (going from 1 to 3 for the input to output layer) as:

$$\delta_i^l = \left(\sum_{j=1}^{n_{l+1}} W_{ij}^{(l+1)} \delta_j^{(l+1)} \right) \cdot \sigma'(z_i^l)$$

now the weight update is computed as:

$$\frac{\partial L}{\partial W_{ji}^{(l)}} = a_j^{(l-1)} \cdot \delta_i^l$$

When computing the gradient of the sigmoid, we replace the true derivative $\sigma'(z)$ with a surrogate gradient:

$$\tilde{\sigma}'_k(z) = \frac{\sigma'(kz)}{k} = \cdot \sigma(kz) \cdot (1 - \sigma(kz))$$

where k controls the slope of the surrogate. Since the surrogate gradient involves both scaling the input by k (to sharpen the activation) and differentiating with respect to that input, the resulting gradient scales with k as well, due to the chain rule. Without compensating for this, gradient magnitudes can explode as k increases. We therefore divide by k to stabilize gradient flow regardless of the steepness.”

Bias in Gradient Magnitude. The gradient of the second layer weights becomes:

$$\frac{\partial L}{\partial W_{ji}^{(2)}} = a_j^{(1)} \cdot \left(\sum_{h=1}^{n_3} W_{ih}^{(3)} \delta_h^{(3)} \right) \cdot \tilde{\sigma}'_k(z_i^{(2)})$$

Compared to the true gradient using $\sigma'(z)$, the surrogate gradient generally yields:

$$\tilde{\sigma}'_k(z) \geq \sigma'(z) \quad \text{for most } z$$

This introduces a positive bias in gradient magnitude, as:

$$\text{bias} = \mathbb{E} \left[\frac{\tilde{\partial L}}{\partial W_{ji}^{(2)}} - \frac{\partial L}{\partial W_{ji}^{(2)}} \right] > 0$$

unless $k = 1$, which recovers the original derivative.

Effect on Deep Layers. In deeper networks, this overestimation accumulates. For example, the first-layer gradient becomes:

$$\frac{\partial \tilde{L}}{\partial W_{ji}^{(1)}} = x_j \cdot \left[\sum_{g=1}^{n_2} W_{ig}^{(2)} \cdot \left(\sum_{h=1}^{n_3} W_{gh}^{(3)} \delta_h^{(3)} \cdot \tilde{\sigma}'_k(z_g^{(2)}) \right) \right] \cdot \tilde{\sigma}'_k(z_i^{(1)})$$

, where the $\tilde{\sigma}'$ are larger than their true counterparts. Compared to its counterpart using the true derivative, the surrogate gradient not only increases magnitude but distorts direction.

Besides increasing magnitude, the surrogate gradient also distorts direction. To quantify this, we compute the cosine similarity between gradients using surrogate and true derivatives:

$$\text{cosine similarity} = \frac{\nabla W \cdot \nabla \tilde{W}}{\|\nabla W\| \cdot \|\nabla \tilde{W}\|}$$

As shown in [Figure 1c](#) of the main paper, cosine similarity decays toward 0 in deeper layers when using shallow slopes (e.g., $k = 1$). This indicates that surrogate-based gradients become increasingly misaligned with the true gradient, effectively randomizing weight updates.

8.2 Simulator and Training Details

The design of the reward structure and termination conditions plays a critical role in shaping the learned policy. Overly strict rewards can prevent learning altogether, while rewards that are too lenient often lead to unsafe or suboptimal behavior. In practice, tuning the reward function is crucial for the success of reinforcement learning algorithms [\[31\]](#).

The reward function is described as:

$$r_t = C_{rs} - C_{rp} \|p_t - p_{\text{des}}\|^2 - C_{rv} \|v_t - v_{\text{des}}\|^2 - C_{rq} \|q_t - q_{\text{des}}\|^2 - C_{ra} \|a_t - C_{rab}\|^2 \quad (9)$$

where:

- $p_t \in \mathbb{R}^3$ is the current position (x, y, z) , p_{des} is the desired hover position
- $v_t \in \mathbb{R}^3$ is the linear velocity (v_x, v_y, v_z) , v_{des} is the desired velocity
- $q_t \in \mathbb{R}^3$ represents the orientation angles (θ, ϕ, ψ) , q_{des} is the desired orientation
- $a_t \in \mathbb{R}^4$ is the action (motor commands)
- C_{rs} is the survival bonus to discourage early termination
- C_{rab} is a fixed action baseline offset

To encourage stable and progressively more precise control, a reward curriculum is applied by linearly increasing the penalties during training. The initial and final values are summarized in [Table 5](#)

	C_{rp}	C_{rv}	C_{ra}	C_{rq}	C_{rs}	C_{rab}
Start	1.0	0.01	0.14	0.25	1.0	0.667
End	3.5	0.10	0.50	0.25	1.0	0.667

Table 5: Reward parameters used during training. Penalties increase linearly across epochs, updated in six steps.

Drone Dynamics. The simulated drone uses a simple physics model. Motor thrust is derived from RPM via a second-order polynomial:

$$T = c_0 + c_1 \cdot \text{rpm} + c_2 \cdot \text{rpm}^2$$

The motor dynamics are modeled using a first-order low-pass filter:

$$\Delta \text{rpm} = \frac{\text{rpm}_{\text{des}} - \text{rpm}_{\text{curr}}}{\tau}$$

where τ represents the motor time constant. Body-frame dynamics are integrated and then transformed to the world frame, as described by Eschmann et al. [\[14\]](#).

Table 6: Training hyperparameters for all methods.

Parameter	BC	TD3BC	TD3BC+JSRL [Ours]	TD3
Common Parameters				
Hidden sizes	[256, 128]	[256, 256]	[256, 128]	[256, 128]
Learning rate	1e-3	1e-3	1e-3	1e-3
Buffer size	1M	1M	2M	2M
Slope start	2	2	2	2
Slope schedule	adaptive	adaptive	adaptive	adaptive
Scheduling order	3	3	3	3
Method-Specific Parameters				
Warm-up steps	50	50	50	–
Policy noise	0.0	0.0	0.2	–
Noise clip	–	0.5	0.5	–
α (TD3BC)	–	2.0	2.0	–
τ (Target)	–	0.01	0.01	0.01
γ (Discount)	–	0.99	0.99	0.99
λ (BC coef)	–	–	0.2	–
BC decay factor	–	–	0.99	–
Training epochs	300	300	1000	1000
Steps per epoch	–	–	2M	1M

Training Hyperparameters. Table 6 provides an overview of the hyperparameters used across all methods. Common parameters are listed first, followed by method-specific values.

8.3 Energy Consumption Analysis

Although we do not have access to neuromorphic hardware small enough to fit on the Crazyflie, the energy consumption can be estimated using the method proposed by Davies *et al.* [30], also used in [32], we see that the total energy per inference would be 9.7×10^{-5} mJ, which is in line with the result from Wang *et al.* [32] (normalized per timestep), and also in line with real-world measurements from Paredes-Valles *et al.* [33], who measured 7×10^{-3} mJ for a much larger and deeper vision network deployed on a neuromorphic system.

An overview of this calculation is given below.

Table 7: Simulation Parameters and Energy Values

Parameter	Value
Energy per synaptic spike op P_s	23.6 (pJ)
Within-tile spike energy P_w	1.7 (pJ)
Energy per neuron update P_u	81 (pJ)
Layer sizes $[N_{in}, N_1, N_2, N_{out}]$	18, 256, 128, 4
Activation Sparsity AS	0.79

The total energy per inference is estimated as:

$$\begin{aligned}
E &= [N_1 \cdot (P_u + N_{in} \cdot P_w)] \\
&\quad + [P_s \cdot (1 - AS) \cdot N_1 + N_2 \cdot (P_u + N_1 \cdot P_w)] \\
&\quad + [P_s \cdot (1 - AS) \cdot N_2 + N_{out} \cdot (N_2 \cdot P_w)] \\
&\approx 9.7 \times 10^{-5} \text{ mJ}
\end{aligned}$$

Note: The multiply operations of the input encoding are ignored. The output is a linear layer that accumulates spikes, without explicit output neurons, therefore no energy for neuron updates P_u are accounted for.

Discussion

A comparison of a conventional method (traditional controller or RNN) versus our method on the Teensy would not be fair. For realizing the energy efficiency, our method depends on neuromorphic hardware that leverages sparsity and asynchronous compute, while conventional methods would not be able to profit as much. Therefore, the hardware implementation section of our paper aims at demonstrating the feasibility of neuromorphic control, not at showing improved energy efficiency.

8.4 Pseudo Code: TD3BC+JSRL (Sequence-Based Training)

Algorithm 1 TD3BC+JSRL for Online Spiking Policy Training

```

1: Initialize: Replay buffer  $\mathcal{D}$ , environment  $\mathcal{E}$ , guiding policy  $\pi_{\text{ctrl}}$ 
2: Initialize: Spiking actor  $\pi_\theta$ , critics  $Q_{\phi_1}, Q_{\phi_2}$ , target networks  $\pi_{\theta'}, Q_{\phi'_1}, Q_{\phi'_2}$ 
3: Hyperparameters:  $\alpha$  (TD3BC weight),  $\lambda_{\text{BC}}$  (BC coefficient),  $\tau$  (soft update rate),  $\gamma$  (discount),  $\sigma_{\text{explore}}$  (exploration noise),  $t_{\text{warmup}}$  (warm-up steps),  $\beta$  (BC decay)
4: for each training iteration do
5:   // Collect trajectories
6:   for each rollout do
7:     Reset environment:  $s_0 \sim \mathcal{E}$ 
8:     for timestep  $t = 0$  to  $T - 1$  do
9:       if  $t < t_{\text{warmup}}$  then
10:         $a_t \leftarrow \pi_{\text{ctrl}}(s_t)$  ▷ Privileged guiding policy
11:         $\pi_\theta(s_t)$  ▷ Warm up spiking hidden states
12:       else
13:         $a_t \leftarrow \pi_\theta(s_t) + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma_{\text{explore}})$ 
14:         $a_t \leftarrow \text{clip}(a_t, -2, 2)$ 
15:       end if
16:       Execute  $a_t$ , observe  $(s_{t+1}, r_t, d_t)$ 
17:     end for
18:     Slice trajectory into overlapping sequences  $\tau = \{(s_t, a_t, r_t, d_t, s_{t+1})\}_{t=1}^L$  with stride  $\Delta$ 
19:     Add sequences  $\tau$  to  $\mathcal{D}$ 
20:   end for
21:   // Network updates
22:   for each gradient step do
23:     Sample mini-batch of sequences  $\{\tau_i\}_{i=1}^B$  from  $\mathcal{D}$ 
24:     for each sequence  $\tau = \{(s_t, a_t, r_t, d_t, s_{t+1})\}_{t=1}^L$  do
25:       Critic update: for all timesteps  $t \in [1, L]$ 
26:         $y_t = r_t + \gamma(1 - d_t) \min_j Q_{\phi'_j}(s_{t+1}, \pi_{\theta'}(s_{t+1}))$ 
27:         $\phi_j \leftarrow \phi_j - \nabla_{\phi_j} \|Q_{\phi_j}(s_t, a_t) - y_t\|^2$ , for  $j = 1, 2$ 
28:         $\hat{a}_t \leftarrow \pi_\theta(s_t); Q_t \leftarrow Q_{\phi_1}(s_t, \hat{a}_t)$ 
29:        Actor update: only for  $t \geq t_{\text{warmup}}$ 
30:         $\mathcal{L}_{\text{BC}} = \|a_t - \hat{a}_t\|^2$ 
31:         $\lambda = \alpha / \text{mean}(|Q_t|)$ 
32:         $\theta \leftarrow \theta - \nabla_\theta (-\lambda Q_t + \lambda_{\text{BC}} \mathcal{L}_{\text{BC}})$ 
33:      end for
34:      Soft update targets:
35:       $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$ 
36:       $\phi'_j \leftarrow \tau\phi_j + (1 - \tau)\phi'_j$  for  $j = 1, 2$ 
37:    end for
38:    Decay BC coefficient:  $\lambda_{\text{BC}} \leftarrow \beta\lambda_{\text{BC}}$ 
39:    if curriculum learning enabled then
40:       $\mathcal{E}.\text{update\_curriculum}()$ 
41:    end if
42:  end for
43: return trained spiking policy  $\pi_\theta$ 

```
