Implementation Roadmap for Spiking Decision Transformer Novel Contributions

Prepared by Vishal Pandey

June 28, 2025

Abstract

This document provides a step-by-step LaTeX-formatted roadmap for integrating five novel modules into the Spiking Decision Transformer (SNN-DT) codebase. Each phase describes the high-level design, implementation steps, and validation strategy.

Contents

1 Phase 1: Adaptive, Data-Driven Temporal Windows

1.1 High-Level Design

- Learn a per-token gate:
$$g_i = \sigma\big(w_g^\top x_i + b_g\big) \in [0, 1].$$
- Define token-specific window length:
$$T_i = \lceil T_{\max} \cdot g_i \rceil.$$
- Unroll each token's LIF projection over its own T_i.

1.2 Implementation Steps

1. **Model:** In `SpikingSelfAttention`, add
```
self.window_gate = nn.Linear(hidden_dim, 1)
```
followed by a `sigmoid`.
2. Compute $g = \texttt{window\_gate}(x)$ after embedding.
3. Refactor the time loop:
```
for i, xi in enumerate(tokens):
    Ti = ceil(T_max * g[i])
    for t in range(Ti):
        # LIF projection ...
```
4. **Regularization:** Add penalty $\lambda\,\mathbb{E}[T_i]$ to the loss.

1.3 Ablation & Validation

- Plot average T_i vs. token index or return-to-go.
- Compare reward and spike counts to fixed-T.

2 Phase 2: Hybrid Local Plasticity + Surrogate-Gradient

2.1 High-Level Design

Combine global backprop with a local three-factor rule in the output LIF layer:
$$\Delta W_O \propto \underbrace{\sum_t \mathrm{pre}(t)\,\mathrm{post}(t)}_{\text{eligibility}} \times R_t.$$

2.2 Implementation Steps

1. Subclass Norse's `LIFCell` to accumulate eligibility traces.
2. After each trajectory, compute and apply:
$$W_O \;\leftarrow\; W_O + \eta_{\text{local}}\, e_{ij}\, G_t.$$
3. Normalize/clamp the local update to stabilize training.

2.3 Ablation & Validation

- Measure epochs to target reward with/without local plasticity.
- Report training curves for both variants.

3 Phase 3: Spike-Domain Positional Encodings & Routing

3.1 Positional Spiking Codes

Encode position via learned oscillators:

$$\mathrm{pos\_spike}_k(t) = \mathbf{1}\big(\sin(\omega_k t + \phi_k) > 0\big).$$

Learnable parameters: $\{\omega_k, \phi_k\}$.

3.2 Dendritic-Style Routing

After computing H heads' outputs $\{y_i^{(h)}(t)\}$, apply a routing MLP:

$$\alpha = \mathrm{softmax}\big(W_{\mathrm{route}}\,[y_i^{(1)}, \ldots, y_i^{(H)}]\big).$$

Re-weight heads by α.

3.3 Implementation Steps

1. In embedding, generate `phase_spikes` alongside rate spikes.
2. In multi-head wrapper:

```
concat = torch.stack(head_outputs, dim=-1)  # [..., H]
alpha = F.softmax(self.route_mlp(concat), dim=-1)
mixed = (concat * alpha).sum(-1)
```

3.4 Ablation & Validation

- Compare performance with/without phase coding.
- Visualize learned ω_k, ϕ_k.

4 Phase 4: Theoretical Convergence & Expressivity

4.1 Expressivity Theorem

Claim. For any dense attention matrix $A \in \mathbb{R}^{L\times L}$ and $\varepsilon > 0$, there exist spike trains of length $T = O(\log \frac{1}{\varepsilon})$ such that

$$\big\|\mathrm{softmax}(\alpha S) - A\big\|_\infty < \varepsilon.$$

4.2 Convergence Bound

Under Lipschitz surrogate gradients, SNN-DT gradient descent converges to ANN-DT gradients as spike counts increase.

4.3 Implementation Steps

- Formalize assumptions (bounded weights, Lipschitz constant L_σ).
- Write proof sketch in a new `theory.tex` appendix.

4.4 Validation

Empirically plot $\|W_{\text{SNN}} - W_{\text{ANN}}\|$ vs. average spikes T.

5 Phase 5: Scalable, Sparse Spiking Attention

5.1 Locality-Sensitive Hashing (LSH)

Hash accumulated spike vectors $q_i \in \{0,1\}^T$ into buckets; compute attention only within buckets.

5.2 Block-Sparse Attention

Divide the sequence into blocks of size B. Compute full attention within each block and use a global "summary" head across block means.

5.3 Implementation Steps

1. Implement LSH hashing: `bucket = torch.sign(random_proj @ qi)`
2. Refactor double loop:

```
for block in blocks:
    # intra-block attention
# summary head attends across block means
```

5.4 Ablation & Validation

- Measure spikes & latency up to $N = 500$.
- Show returns remain within 95% of dense attention.

Version Control & Workflow

- Use separate feature branches: `adaptive-window`, `local-plasticity`, etc.
- After each merge, run benchmarks to catch regressions.
- Maintain consistent style (e.g. Google Python Style Guide).