SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks with at most one Spike per Neuron

1. Introduction

- Spiking neural networks (SNNs) are energy-efficient and hardware-friendly.
- SNNs with time-to-first-spike coding and spike-time-dependent plasticity (STDP) learning rules have done well in various artificial intelligence (AI) tasks.

**SpykeTorch** is a simulator for such SNNs that makes use of **PyTorch** highly optimized tensor operations, on CPUs and (multi-) GPUs.

- It is fully compatible with and integrated into PyTorch, and it is user-friendly and easy to learn specially for deep learning community.

2. Time Dimension

- **SpykeTorch** considers an extra dimension in tensors for representing time.
- **SpykeTorch** divides all of the spikes of a particular stimulus into a pre-defined number of spike bins, where each bin corresponds to a single time-step.

If $T_{f,r}$ spike time of the neuron placed at position $(r, c)$ of the feature map $f$, then the SpykeTorch’s corresponding tensor $S$ will be:

$$S[t,f,r,c] = \begin{cases} 0 & t < T_{f,r,c}, \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

- Keeping spikes in accumulative format increases the memory usage, but let us perform network operations on all time steps simultaneously.

3. Resources

- **Source Code**: https://github.com/miladmozafari/SpykeTorch
- **Documentation**: http://cnrl.ut.ac.ir/SpykeTorch/doc/

4. Tutorial

**Network Structure**

We implement the network proposed in [1] with seven layers: DoG, Conv1, Pool1, Conv2, Pool2, Conv3, Decision.

**Forward Pass**

- Override the forward function in nn.Module
- Apply convolutions.
- Apply pooling by functional module.
- Saving data for plasticity (input, output, winners).
- Testing pass is the same but without saving plasticity data.

**Plasticity**

STDP for the first two Conv layers, and Reward-Modulated STDP (R-STDP) for the last Conv layer.

**Input Transform**

- Prepare data (MNIST here).
- Use CacheDataset to cache the pre-processed data.
- Read image.
- Apply DoG filters.
- Apply intensity-to-latency.

**Execution**

- Create network instance.
- Run unsupervised learning on Conv1 and Conv2.
- Run reinforcement learning on Conv3.
- After each epoch, evaluate network on testing set.
- The test function is similar to train.r1, but without calling plasticity functions.

5. Results

- Almost the same results as original implementations.
- 97% [1], and 98.4% [2] on MNIST.

6. References

2. Kheradpisheh et al 2018, Neural Networks