SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks With at Most One Spike per Neuron

Milad Mozafari, Mohammad Ganjtabesh, Abbas Nowzari-Dalini, Timothée Masquelier

frontiers in Neuroscience

TECHNOLOGY REPORT
published: 12 July 2019
doi: 10.3389/fnins.2019.00625

SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks With at Most One Spike per Neuron

Milad Mozafari[1,2], *Mohammad Ganjtabesh*[1]*, *Abbas Nowzari-Dalini*[1] and *Timothée Masquelier*[2]

[1] Department of Computer Science, School of Mathematics, Statistics, and Computer Science, University of Tehran, Tehran, Iran, [2] CERCO UMR 5549, CNRS - Université Toulouse 3, Toulouse, France

Application of deep convolutional spiking neural networks (SNNs) to artificial intelligence (AI) tasks has recently gained a lot of interest since SNNs are hardware-friendly and energy-efficient. Unlike the non-spiking counterparts, most of the existing SNN simulation frameworks are not practically efficient enough for large-scale AI tasks. In this paper, we introduce SpykeTorch, an open-source high-speed simulation framework based on PyTorch. This framework simulates convolutional SNNs with at most one spike per neuron and the rank-order encoding scheme. In terms of learning rules, both spike-timing-dependent plasticity (STDP) and reward-modulated STDP (R-STDP) are implemented, but other rules could be implemented easily. Apart from the aforementioned properties, SpykeTorch is highly generic and capable of reproducing the results of various studies. Computations in the proposed framework are tensor-based and totally done by PyTorch functions, which in turn brings the ability of just-in-time optimization for running on CPUs, GPUs, or Multi-GPU platforms.

Keywords: convolutional spiking neural networks, time-to-first-spike coding, one spike per neuron, STDP, reward-modulated STDP, tensor-based computing, GPU acceleration

OPEN ACCESS

Edited by:
Guoqi Li,
Tsinghua University, China

Reviewed by:
Deboleena Roy,
Purdue University, United States
Quansheng Ren,
Peking University, China

***Correspondence:**
Mohammad Ganjtabesh
mgtabesh@ut.ac.ir

Specialty section:
This article was submitted to Neuromorphic Engineering, a section of the journal Frontiers in Neuroscience

Received: *01 March 2019*
Accepted: *31 May 2019*
Published: *12 July 2019*

Citation:
Mozafari M, Ganjtabesh M, Nowzari-Dalini A and Masquelier T (2019) SpykeTorch: Efficient Simulation of Convolutional Spiking Neural Networks With at Most One Spike per Neuron. Front. Neurosci. 13:625. doi: 10.3389/fnins.2019.00625

1. INTRODUCTION

For many years, scientist were trying to bring human-like vision into machines and artificial intelligence (AI). In recent years, with advanced techniques based on deep convolutional neural networks (DCNNs) (Rawat and Wang, 2017; Gu et al., 2018), artificial vision has never been closer to human vision. Although DCNNs have shown outstanding results in many AI fields, they suffer from being data- and energy-hungry. Energy consumption is of vital importance when it comes to hardware implementation for solving real-world problems.

Our brain consumes much less energy than DCNNs, about 20 W (Mink et al., 1981) – roughly the power consumption of an average laptop, for its top-notch intelligence. This feature has convinced researchers to start working on computational models of human cortex for AI purposes. Spiking neural networks (SNNs) are the next generation of neural networks, in which neurons communicate through binary signals known as spikes. SNNs are energy-efficient for hardware implementation, because, spikes bring the opportunity of using event-based hardware as well as simple energy-efficient accumulators instead of complex energy-hungry multiply-accumulators that are usually employed in DCNN hardware (Furber, 2016; Davies et al., 2018).

Spatio-temporal capacity of SNNs makes them potentially stronger than DCNNs, however, harnessing their ultimate power is not straightforward. Various types of SNNs have been proposed for vision tasks which can be categorized based on their specifications such as:

- network structure: shallow (Masquelier and Thorpe, 2007; Yu et al., 2013; Kheradpisheh et al., 2016), and deep (Kheradpisheh et al., 2018; Mozafari et al., 2019),
- topology of connections: convolutional (Cao et al., 2015; Tavanaei and Maida, 2016), and fully connected (Diehl and Cook, 2015),
- information coding: rate (O'Connor et al., 2013; Hussain et al., 2014), and latency (Masquelier and Thorpe, 2007; Diehl and Cook, 2015; Mostafa, 2018),
- learning rule: unsupervised (Diehl and Cook, 2015; Ferré et al., 2018; Thiele et al., 2018), supervised (Diehl et al., 2015; Liu et al., 2017; Bellec et al., 2018; Mostafa, 2018; Shrestha and Orchard, 2018; Wu et al., 2018; Zenke and Ganguli, 2018), and reinforcement (Florian, 2007; Mozafari et al., 2018).

For recent advances in deep learning with SNNs, we refer the readers to reviews by Tavanaei et al. (2018), Pfeiffer and Pfeil (2018), and Neftci et al. (2019).

Deep convolutional SNNs (DCSNNs) with time-to-first-spike information coding and STDP-based learning rule constitute one of those many types of SNNs that carry interesting features. Their deep convolutional structure supports visual cortex and let them extract features hierarchically from simple to complex. Information coding using the earliest spike time, which is proposed based on the rapid visual processing in the brain (Thorpe et al., 1996), needs only a single spike, making them super fast and more energy efficient. These features together with hardware-friendliness of STDP, turn this type of SNNs into the best option for hardware implementation and online on-chip training (Yousefzadeh et al., 2017). Several recent studies have shown the excellence of this type of SNNs in visual object recognition (Kheradpisheh et al., 2018; Mostafa, 2018; Mozafari et al., 2018; Mozafari et al., 2019; Falez et al., 2019; Vaila et al., 2019).

With simulation frameworks such as Tensorflow (Abadi et al., 2016) and PyTorch (Paszke et al., 2017), developing and running DCNNs is fast and efficient. Conversely, DCSNNs suffer from the lack of such frameworks. Existing state-of-the-art SNN simulators have been mostly developed for studying neuronal dynamics and brain functionalities and are not efficient and user-friendly enough for AI purposes. For instance, bio-realistic and detailed SNN simulations are provided by NEST (Gewaltig and Diesmann, 2007), BRIAN (Stimberg et al., 2014), NEURON (Carnevale and Hines, 2006), and ANNarchy (Vitay et al., 2015). These frameworks also enable users to define their own dynamics of neurons and connections. In contrast, frameworks such as Nengo (Bekolay et al., 2014) and NeuCube (Kasabov, 2014) offer high-level simulations focusing on the neural behavior of the network. Recently, BindsNet (Hazan et al., 2018) framework has been proposed as a fast and general SNN simulator based on PyTorch that is mainly developed for conducting AI experiments. A detailed comparison between BindsNet and the other available frameworks can be found in their paper.

In this paper, we propose SpykeTorch, a simulation framework based on PyTorch which is optimized specifically for convolutional SNNs with at most one spike per neuron. SpykeTorch offers utilities for building hierarchical feedforward SNNs with deep or shallow structures and learning rules such as STDP and R-STDP (Gerstner et al., 1996; Bi and Poo, 1998; Frémaux and Gerstner, 2016; Brzosko et al., 2017). SpykeTorch only supports time-to-first-spike information coding and provides a non-leaky integrate and fire neuron model with at most one spike per stimulus. Unlike BindsNet which is flexible and general, the proposed framework is highly restricted to and optimized for this type of SNNs. Although BindsNet is based on PyTorch, its network design language is different. In contrast, SpykeTorch is fully compatible and integrated with PyTorch and obeys the same design language. Therefore, a PyTorch user may only read the documentation to find out the new functionalities. Besides, this integrity makes it possible to utilize almost all of the PyTorch's functionalities either running on a CPU, or (multi-) GPU platform.

The rest of this paper is organized as follows: Section 2 describes how SpykeTorch includes the concept of time in its computations. Section 3 is dedicated to SpykeTorch package structure and its components. In section 4, a brief tutorial on building, training, and evaluating a DCSNN using SpykeTorch is given. Section 6 summarizes the current work and highlights possible future works.

2. TIME DIMENSION

Modules in SpykeTorch are compatible with those in PyTorch and the underlying data-type is simply the PyTorch's tensors. However, since the simulation of SNNs needs the concept of "time," SpykeTorch considers an extra dimension in tensors for representing time. The user may not need to think about this new dimensionality while using SpykeTorch, but, in order to combine it with other PyTorch's functionalities or extracting different kinds of information from SNNs, it is important to have a good grasp of how SpykeTorch deals with time.

SpykeTorch works with time-steps instead of exact time. Since the neurons emit at most one spike per stimulus, it is enough to keep track of the first spike times (in time-step scale) of the neurons. For a particular stimulus, SpykeTorch divides all of the spikes into a pre-defined number of spike bins, where each bin corresponds to a single time-step. More precisely, assume a stimulus is represented by F feature maps, each constitutes a grid of $H \times W$ neurons. Let T_{max} be the maximum possible number of time-steps (or bins) and $T_{f,r,c}$ denote the spike time (or the bin index) of the neuron placed at position (r, c) of the feature map f, where $0 \leq f < F$, $0 \leq r < H$, $0 \leq c < W$, and $T_{f,r,c} \in \{0, 1, ..., T_{max} - 1\} \cup \{\infty\}$. The ∞ symbol stands for no spike. SpykeTorch considers this stimulus as a four-dimensional binary spike-wave tensor S of size $T_{max} \times F \times H \times W$ where:

$$S[t, f, r, c] = \begin{cases} 0 & t < T_{f,r,c}, \\ 1 & otherwise. \end{cases} \tag{1}$$

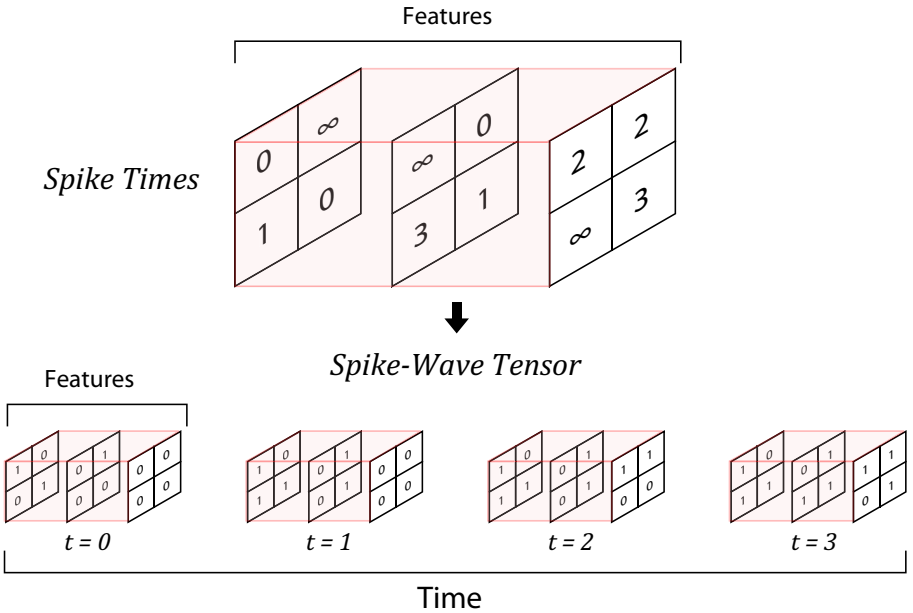

FIGURE 1 | An example of generating spike-wave tensor from spike times. There are three feature maps, each constitutes a 2 × 2 grid of neurons that are going to generate a spike-wave representing a stimulus. If the maximum number of time-steps is 4, then the resulting spike-wave is a four-dimensional tensor of size $4 \times 3 \times 2 \times 2$. If a neuron emits spike at time step $t = i$, the corresponding position in the spike-wave tensor will be set to 1 from time-step $t = i$ to the final time step ($t = 3$).

Note that this way of keeping the spikes (accumulative structure) does not mean that neurons keep firing after their first spikes. Repeating spikes in future time steps increases the memory usage, but makes it possible to process all of the time-steps simultaneously and produce the corresponding outputs, which consequently results in a huge speed-up. **Figure 1** illustrates an example of converting spike times into a SpykeTorch-compatible spike-wave tensor. **Figure 2** shows how accumulative spikes helps simultaneous computations.

3. PACKAGE STRUCTURE

Basically, SpykeTorch consists of four python modules; (1) `snn` which contains multiple classes for creating SNNs, (2) `functional` that implements useful SNNs' functions, (3) `utils` which gathers helpful utilities, and (4) `visualization` which helps to generate graphical data out of SNNs. The following subsections explain these modules.

3.1. snn Module

The `snn` module contains necessary classes to build SNNs. These classes are inherited from the PyTorch's `nn.Module`, enabling them to function inside the PyTorch framework as network modules. Since we do not support error backpropagation, the PyTorch's auto-grad feature is turned off for all of the parameters in `snn` module.

`snn.Convolutional` objects implements spiking convolutional layers with two-dimensional convolution kernel. A `snn.Convolutional` object is built by providing the number of input and output features (or channels), and the size of the convolution kernel. Given the size of the kernel, the corresponding tensor of synaptic weights is randomly initialized using a normal distribution, where the mean and standard deviation can be set for each object, separately.

A `snn.Convolutional` object with kernel size $K_h \times K_w$ performs a valid convolution (with no padding) over an input spike-wave tensor of size $T_{max} \times F_{in} \times H_{in} \times W_{in}$ with stride equals to 1 and produces an output potentials tensor of size $T_{max} \times F_{out} \times H_{out} \times W_{out}$, where:

$$\begin{aligned} H_{out} &= H_{in} - K_h + 1, \\ W_{out} &= W_{in} - K_w + 1, \end{aligned} \tag{2}$$

and F_{in} and F_{out} are the number of input and output features, respectively. Potentials tensors (P) are similar to the binary spike-wave tensors, however $P[t, f, r, c]$ denotes the floating-point potential of a neuron placed at position (r, c) of feature map f, at time-step t. Note that current version of SpykeTorch does not support stride more than 1, however, we are going to implement it in the next major version.

The underlying computation of `snn.Convolutional` is the PyTorch's two-dimensional convolution, where the mini-batch dimension is sacrificed for the time. According to the accumulative structure of spike-wave tensor, the result of applying PyTorch's `conv2D` over this tensor is the accumulative potentials over time-steps.

It is important to mention that simultaneous computation over time dimension improves the efficiency of the framework,

but it has dispelled batch processing in SpykeTorch. We agree that batch processing brings a huge speed-up, however, providing it to the current version of SpykeTorch is not straightforward. Here are some of the important challenges: (1) Due to accumulative format of spike-wave tensors, keeping batch of images increases memory usage even more. (2) Plasticity in batch mode needs new strategies. (3) To get the most out of batch processing, all of the main computations such as plasticity, competitions, and inhibitions should be done on the whole batch at the same time, especially when the model is running on GPUs.

Pooling is an important operation in deep convolutional networks. `snn.Pooling` implements two-dimensional max-pooling operation. Building `snn.Pooling` objects requires providing the pooling window size. The stride is equal to the window size by default, but it is adjustable. Zero padding is also another option which is off by default.

`snn.Pooling` objects are applicable to both spike-wave and potentials tensors. According to the structure of these tensors, if the input is a spike-wave tensor, then the output will contain the earliest spike within each pooling window, while if the input is a potentials tensor, the maximum potential within each pooling window will be extracted. Assume that the input tensor has the shape $T_{max} \times F_{in} \times H_{in} \times W_{in}$, the pooling window has the size $P_h \times P_w$ with stride $R_h \times R_w$, and the padding is (D_h, D_w), then the output tensor will have the size $T_{max} \times F_{out} \times H_{out} \times W_{out}$, where:

$$\begin{aligned} H_{out} &= \lfloor \frac{H_{in} + 2 \times D_h}{R_h} \rfloor, \\ W_{out} &= \lfloor \frac{W_{in} + 2 \times D_w}{R_w} \rfloor. \end{aligned} \tag{3}$$

To apply STDP on a convolutional layer, a `snn.STDP` object should be built by providing the value of required parameters such as learning rates. Since this simulator works with time-to-first-spike coding, the provided implementation of the STDP function is as follows:

$$\Delta \mathcal{W}_{i,j} = \begin{cases} A^+ \times (\mathcal{W}_{i,j} - LB) \times (UB - \mathcal{W}_{i,j}) & if \quad T_j \leq T_i, \\ A^- \times (\mathcal{W}_{i,j} - LB) \times (UB - \mathcal{W}_{i,j}) & if \quad T_j > T_i, \end{cases} \tag{4}$$

where, $\Delta \mathcal{W}_{i,j}$ is the amount of weight change of the synapse connecting the post-synaptic neuron i to the pre-synaptic neuron j, A^+ and A^- are learning rates, and $(\mathcal{W}_{i,j} - LB) \times (UB - \mathcal{W}_{i,j})$ is a stabilizer term which slows down the weight change when the synaptic weight $(\mathcal{W}_{i,j})$ is close to the lower (LB) or upper (UB) bounds.

To apply STDP during the training process, providing the input and output spike-wave, as well as output potentials tensors are necessary. `snn.STDP` objects make use of the potentials tensor to find winners. Winners are selected first based on the earliest spike times, and then based on the maximum potentials. The number of winners is set to 1 by default. `snn.STDP` objects also provide lateral inhibition, by which they completely inhibit the winners' surrounding neurons in all of the feature maps within a specific distance. This increases the chance of learning diverse features. Note that R-STDP can be applied using two `snn.STDP` objects; one for STDP part and the other for anti-STDP part.

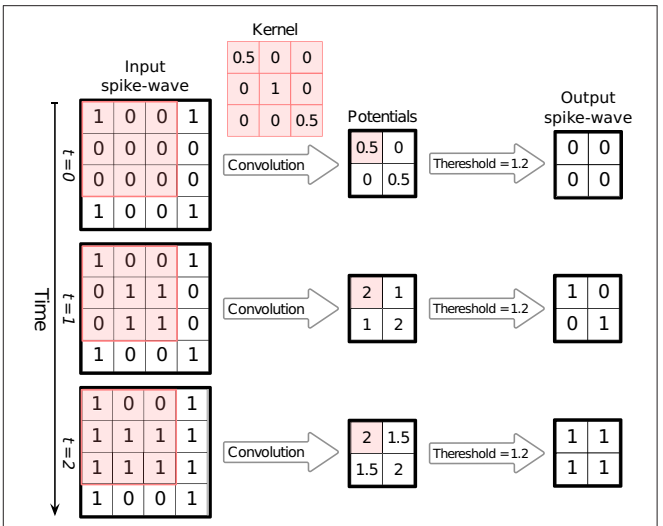

FIGURE 2 | An example of simultaneous processing of spikes over time-steps. Here the input spike-wave tensor has one 5 × 5 channel (feature map) and the spikes are divided into three time-steps. When SpykeTorch applies the convolution kernel of size 3 × 3 (valid mode) simultaneously on all of the time-steps, the resulting tensor will contain potentials in all of the time-steps. Since spikes are stored in accumulative format, then the potentials are accumulative as well. Applying a threshold function over the whole potential tensor generates the corresponding output spike-wave tensor, again in accumulative format.

3.2. `functional` Module

This module contains several useful and popular functions applicable on SNNs. Here we briefly review the most important ones. For the sake of simplicity, we replace the `functional.` with `sf.` for writing the function names.

As mentioned before, `snn.Convolutional` objects give potential tensors as their outputs. `sf.fire` takes a potentials tensor as input and converts it into a spike-wave tensor based on a given threshold. `sf.threshold` function is also available separately that takes a potentials tensor and outputs another potentials tensor in which all of the potentials lower than the given threshold are set to zero. The output of `sf.threshold` is called thresholded potentials.

Lateral inhibition is another vital operation for SNNs specially during the training process. It helps to learn more diverse features and achieve sparse representations in the network. SpykeTorch's `functional` module provides several functions for different kinds of lateral inhibitions.

`sf.feature_inhibition` is useful for complete inhibition of the selected feature maps. This function comes in handy to apply dropout to a layer. `sf.pointwise_inhibition` employs competition among features. In other words, at each location, only the neuron corresponding to the most salient feature will be allowed to emit a spike (the earliest spike with the highest potential). Lateral inhibition is also helpful to be applied on input intensities before conversion to spike-waves. This will decrease the redundancy in each region of the input. To apply this kind of inhibition, `sf.intensity_lateral_inhibition`

is provided. It takes intensities and a lateral inhibition kernel by which it decreases the surrounding intensities (thus increases the latency of the corresponding spike) of each salient point. Local normalization is also provided by `sf.local_normalization` which uses regional mean for normalizing intensity values.

Winners-take-all (WTA) is a popular competition mechanism in SNNs. WTA is usually used for plasticity, however, it can be involved in other functionalities such as decision-making. `sf.get_k_winners` takes the desired number of winners and the thresholded potentials and returns the list of winners. Winners are selected first based on the earliest spike times, and then based on the maximum potentials. Each winner's location is represented by a triplet of the form (*feature*, *row*, *column*).

3.3. `utils` Module

`utils` module provides several utilities to ease the implementation of ideas with SpykeTorch. For example, `utils.generate_inhibition_kernel` generates an inhibition kernel based on a series of inhibition factors in a form that can be properly used by `sf.intensity_lateral_inhibition`.

There exist several transformation utilities that are suitable for filtering inputs and converting them to spike-waves. Current utilities are mostly designed for vision purposes. `utils.LateralIntencityInhibition` objects do the `sf.intensity_lateral_inhibition` as a transform object. `utils.FilterKernel` is a base class to define filter kernel generators. SpykeTorch has already provided `utils.DoGKernel` and `utils.GaborKernel` in order to generate DoG and Gabor filter kernels, respectively. Objects of `utils.FilterKernel` can be packed into a multi-channel filter kernel by `utils.Filter` objects and applied to the inputs.

The most important utility provided by `utils` is `utils.Intensity2Latency`. Objects of `utils.Intensity2Latency` are used as transforms in PyTorch's datasets to transform intensities into latencies, i.e., spike-wave tensors. Usually, `utils.Intensity2Latency` is the final transform applied to inputs.

Since the application of a series of transformations and the conversion to spike-waves can be time-consuming, SpykeTorch provides a wrapper class, called `utils.CacheDataset`, which is inherited from PyTorch's `dataset` class. Objects of `utils.CacheDataset` take a dataset as their input and cache the data after applying all of the transformations. They can cache either on primary memory or secondary storage.

Additionally, `utils` contains two functions `utils.tensor_to_text` and `utils.text_to_tensor`, which handle conversion of tensors to text files and the reverse, respectively. This conversion is helpful to import data from a source or export a tensor for a target software. The format of the text file is as follows: the first line contains comma-separated integers denoting the shape of the tensor. The second line contains comma-separated values indicating the whole tensor's data in row-major order.

3.4. `visualization` Module

The ability to visualize deep networks is of great importance since it gives a better understanding of how the network's components are working. However, visualization is not a straightforward procedure and depends highly on the target problem and the input data.

Due to the fact that SpykeTorch is developed mainly for vision tasks, its `visualization` module contains useful functions to reconstruct the learned visual features. The user should note that these functions are not perfect and cannot be used in every situation. In general, we recommend the user to define his/her own visualization functions to get the most out of the data.

4. TUTORIAL

In this section, we show how to design, build, train, and test a SNN with SpykeTorch in a tutorial format. The network in this tutorial is adopted from the deep convolutional SNN proposed by Mozafari et al. (2019) which recognizes handwritten digits (tested on MNIST dataset). This network has a deep structure and uses both STDP and Reward-Modulated STDP (R-STDP), which makes it a suitable choice for a complete tutorial. In order to make the tutorial as simple as possible, we present code snippets with reduced contents. For the complete source code, please check SpykeTorch's GitHub[1] web page.

4.1. Step 1. Network Design

4.1.1. Structure

The best way to design a SNN is to define a class inherited from `torch.nn.Module`. The network proposed by Mozafari et al. (2019), has an input layer which applies DoG filters to the input image and converts it to spike-wave. After that, there are three convolutional (*S*) and pooling (*C*) layers that are arranged in the form of $S1 \rightarrow C1 \rightarrow S2 \rightarrow C2 \rightarrow S3 \rightarrow C3$ (see **Figure 3**). Therefore, we need to consider three objects for convolutional layers in this model. For the pooling layers, we will use the functional version instead of the objects.

As shown in **Listing 1**, three `snn.Convolutional` objects are created with desired parameters. Two `snn.STDP` objects are built for training *S*1 and *S*2 layers. Since *S*3 is trained by R-STDP, two `snn.STDP` are needed to cover both STDP and anti-STDP parts. To have the effect of anti-STDP, it is enough to negate the signs of the learning rates. Note that the `snn.STDP` objects for `conv3` have two extra parameters where the first one turns off the stabilizer and the second one keeps the weights in range [0.2, 0.8].

Although `snn` objects are compatible with `nn.Sequential` (`nn.Sequential` automates the forward pass given the network modules), we cannot use it at the moment. The reason is that different learning rules may need different kinds of data from each layer, thus accessing each layer during the forward pass is a must.

[1] https://github.com/miladmozafari/SpykeTorch

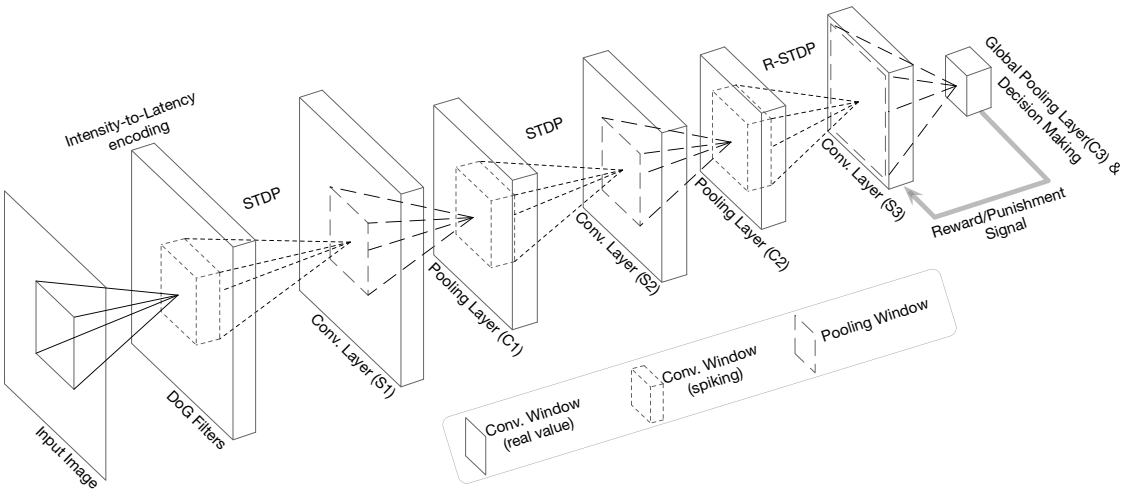

FIGURE 3 | Overall structure of the network in the tutorial (modified figure from the work by Mozafari et al., 2019).

```
import torch.nn as nn
import SpykeTorch.snn as snn
import SpykeTorch.functional as sf
class DCSNN(nn.Module):
   def __init__(self):
      super(DCSNN, self).__init__()

      #(in_channels, out_channels, kernel_size, weight_mean=0.8, weight_std=0.02)
      self.conv1 = snn.Convolution(6, 30, 5, 0.8, 0.05)
      self.conv2 = snn.Convolution(30, 250, 3, 0.8, 0.05)
      self.conv3 = snn.Convolution(250, 200, 5, 0.8, 0.05)

      #(conv_layer, learning_rate, use_stabilizer=True, lower_bound=0, upper_bound=1)
      self.stdp1 = snn.STDP(self.conv1, (0.004, -0.003))
      self.stdp2 = snn.STDP(self.conv2, (0.004, -0.003))
      self.stdp3 = snn.STDP(self.conv3, (0.004, -0.003), False, 0.2, 0.8)
      self.anti_stdp3 = snn.STDP(self.conv3, (-0.004, 0.0005), False, 0.2, 0.8)
```

Listing 1 | Defining the network class.

4.1.2. Forward Pass

Next, we implement the forward pass of the network. To this end, we override the `forward` function in `nn.Module`. If the training is off, then implementing the forward pass will be straightforward. **Listing 2** shows the application of convolutional and pooling layers on an input sample. Note that each input is a spike-wave tensor. We will demonstrate how to convert images into spike-waves later.

As shown in **Listing 2**, the input of each convolutional layer is the padded version of the output of its previous layer, thus, there would be no information loss at the boundaries. Pooling operations are also applied by the corresponding function `sf.pooling`, which is an alternative to `snn.Pooling`. According to Mozafari et al. (2019), their proposed network makes decision based on the maximum potential among the neurons in the last pooling layer. To this end, we use an infinite threshold for the last convolutional layer by omitting its value from `sf.fire_` function. `sf.fire_` is the in-place version of `sf.fire` which modifies the input potentials tensor P_{in} as follows:

$$P_{in}[t,f,r,c] = \begin{cases} 0 & if \;\; t < T_{max} - 1, \\ P_{in}[t,f,r,c] & otherwise. \end{cases} \tag{5}$$

Consequently, the resulting spike-wave will be a tensor in which, all the values are zero except for those non-zero potential values in the last time-step.

Now that we have the potentials of all the neurons in $S3$, we find the only one winner among them. This is the same as doing a global max-pooling and choosing the maximum potential among

```
def forward(self, input):
   input = sf.pad(input, (2,2,2,2))
   if not self.training:
      pot = self.conv1(input)
      spk = sf.fire(pot, 15)
      pot = self.conv2(sf.pad(sf.pooling(spk, 2, 2), (1,1,1,1)))
      spk = sf.fire(pot, 10)
      pot = self.conv3(sf.pad(sf.pooling(spk, 3, 3), (2,2,2,2)))
      # omitting the threshold parameters means infinite threshold
      spk = sf.fire_(pot)
      winners = sf.get_k_winners(pot, 1)
      output = -1
      # each winner is a tuple of form (feature, row, column)
      if len(winners) != 0:
         output = self.decision_map[winners[0][0]]
      return output
```

Listing 2 | Defining the forward pass (during testing process).

```
def save_data(self, input_spk, pot, spk, winners):
   self.ctx["input_spikes"] = input_spk
   self.ctx["potentials"] = pot
   self.ctx["output_spikes"] = spk
   self.ctx["winners"] = winners
```

Listing 3 | Saving required data for plasticity.

them. `decision_map` is a Python list which maps each feature to a class label. Since each winner contains the feature number as its first component, we can easily indicate the decision of the network by putting that into the `decision_map`.

We cannot take advantage of this forward pass during the training process as the STDP and R-STDP need local synaptic data to operate. Therefore, we need to save the required data during the forward pass. We define a Python dictionary (named `ctx`) in our network class and a function which saves the data into that (see **Listing 3**). Since the training process is layer-wise, we update the `forward` function to take another parameter which specifies the layer that is under training. The updated `forward` function is shown in **Listing 4**.

There are several differences with respect to the testing forward pass. First, `sf.fire` is used with an extra parameter value. If the value of this parameter is `True`, the tensor of thresholded potentials will also be returned. Second, `sf.get_k_winners` is called with a new parameter value which controls the radius of lateral inhibition. Third, the forward pass is interrupted by the value of `max_layer`.

4.1.3. Plasticity

Now that we saved the required data for learning, we can define a series of helper functions to apply STDP or anti-STDP. **Listing 5** defines three member functions for this purpose. For each call of `STDP` objects, we need to provide tensors of input spike-wave, output thresholded potentials, output spike-wave, and the list of winners.

4.2. Step 2. Input Layer and Transformations

SNNs work with spikes, thus, we need to transform images into spike-waves before feeding them into the network. PyTorch's datasets accept a function as a transformation which is called automatically on each input sample. We make use of this feature together with the provided transform functions and objects by PyTorch and SpykeTorch. According to the network proposed by Mozafari et al. (2019), each image is convolved by six DoG filters, locally normalized, and transformed into spike-wave. As appeared in **Listing 6**, a new class is defined to handle the required transformations.

Each `InputTransform` object converts the input image into a tensor (line 9), adds an extra dimension for time (line 10), applies provided filters (line 11), applies local normalization (line 12), and generates spike-wave tensor (line 13). To create `utils.Filter` object, six DoG kernels with desired parameters are given to `utils.Filter`'s constructor (lines 15–17) as well as an appropriate padding and threshold value (line 18).

```
    def forward(self, input, max_layer):
        input = sf.pad(input, (2,2,2,2))
        if self.training: #forward pass for train
            pot = self.conv1(input)
            spk, pot = sf.fire(pot, 15, True)
            if max_layer == 1:
                winners = sf.get_k_winners(pot, 5, 3)
                self.save_data(input, pot, spk, winners)
                return spk, pot
            spk_in = sf.pad(sf.pooling(spk, 2, 2), (1,1,1,1))
            pot = self.conv2(spk_in)
            spk, pot = sf.fire(pot, 10, True)
            if max_layer == 2:
                winners = sf.get_k_winners(pot, 8, 2)
                self.save_data(spk_in, pot, spk, winners)
                return spk, pot
            spk_in = sf.pad(sf.pooling(spk, 3, 3), (2,2,2,2))
            pot = self.conv3(spk_in)
            spk = sf.fire_(pot)
            winners = sf.get_k_winners(pot, 1)
            self.save_data(spk_in, pot, spk, winners)
            output = -1
            if len(winners) != 0:
                output = self.decision_map[winners[0][0]]
            return output
        else:
            # forward pass for testing process
```

Listing 4 | Defining the forward pass (during training process).

```
    def stdp(self, layer_idx):
        if layer_idx == 1:
            self.stdp1(self.ctx["input_spikes"], self.ctx["potentials"],
            ↪  self.ctx["output_spikes"], self.ctx["winners"])
        if layer_idx == 2:
            self.stdp2(self.ctx["input_spikes"], self.ctx["potentials"],
            ↪  self.ctx["output_spikes"], self.ctx["winners"])

    def reward(self):
        self.stdp3(self.ctx["input_spikes"], self.ctx["potentials"], self.ctx["output_spikes"],
        ↪  self.ctx["winners"])

    def punish(self):
        self.anti_stdp3(self.ctx["input_spikes"], self.ctx["potentials"],
        ↪  self.ctx["output_spikes"], self.ctx["winners"])
```

Listing 5 | Defining helper functions for plasticity.

4.3. Step 3. Data Preparation

Due to the PyTorch and SpykeTorch compatibility, all of the PyTorch's dataset utilities work here. As illustrated in **Listing 7**, we use `torchvision.datasets.MNIST` to load MNIST dataset with our previously defined `transform`. Moreover, we use SpykeTorch's dataset wrapper, `utils.CacheDataset` to enable caching the transformed data after its first presentation. When the dataset gets ready, we use PyTorch's `DataLoader` to manage data loading.

4.4. Step 4. Training and Testing

4.4.1. Unsupervised Learning (STDP)

To do unsupervised learning on *S*1 and *S*2 layers, we use a helper function as defined in **Listing 8**. This function trains

```
import SpykeTorch.utils as utils
import torchvision.transforms as transforms
class InputTransform:
    def __init__(self, filter):
        self.to_tensor = transforms.ToTensor()
        self.filter = filter
        self.temporal_transform = utils.Intensity2Latency(15, to_spike=True)
    def __call__(self, image):
        image = self.to_tensor(image) * 255
        image.unsqueeze_(0)
        image = self.filter(image)
        image = sf.local_normalization(image, 8)
        return self.temporal_transform(image)

kernels = [ utils.DoGKernel(3,3/9,6/9), utils.DoGKernel(3,6/9,3/9),
          utils.DoGKernel(7,7/9,14/9), utils.DoGKernel(7,14/9,7/9),
          utils.DoGKernel(13,13/9,26/9), utils.DoGKernel(13,26/9,13/9)]
filter = utils.Filter(kernels, padding = 6, thresholds = 50)
transform = InputTransform(filter)
```

Listing 6 | Transforming each input image into spike-wave.

```
from torch.utils.data import DataLoader
from torchvision.datasets import MNIST
MNIST_train = utils.CacheDataset(MNIST(root=data_root, train=True, download=True,
↪  transform=transform))
MNIST_test = utils.CacheDataset(MNIST(root=data_root, train=False, download=True,
↪  transform=transform))
MNIST_loader = DataLoader(MNIST_train, batch_size=1000)
MNIST_test_loader = DataLoader(MNIST_test, batch_size=len(MNIST_test))
```

Listing 7 | Preparing MNIST dataset and the data loader.

```
def train_unsupervised(network, data, layer_idx):
    network.train()
    for i in range(len(data)):
        data_in = data[i].cuda() if use_cuda else data[i]
        network(data_in, layer_idx)
        network.stdp(layer_idx)
```

Listing 8 | Helper function for unsupervised learning.

layer `layer_idx` of `network` on `data` by calling the corresponding `STDP` object. There are two important things in this function: (1) putting the network in train mode by calling. `train` function, and (2) loading the sample on GPU if the global `use_cuda` flag is `True`.

4.4.2. Reinforcement Learning (R-STDP)

To apply R-STDP, it is enough to call previously defined `reward` or `punish` member functions under appropriate conditions. As shown in **Listing 9**, we check the network's decision with the label and call `reward` (or `punish`) if it matches (or mismatches) the target. We also compute the performance by counting correct, wrong, and silent (no decision is made because of receiving no spikes) samples.

4.4.3. Execution

Now that we have the helper functions, we can make an instance of the network and start training and testing it. **Listing 10** illustrates the implementation of this part. Note that the `test` helper function is the same as the `train_rl` function, but it

```python
import numpy as np
def train_rl(network, data, target):
    network.train()
    perf = np.array([0,0,0]) # correct, wrong, silent
    for i in range(len(data)):
        data_in = data[i].cuda() if use_cuda else data[i]
        target_in = target[i].cuda() if use_cuda else target[i]
        d = network(data_in, 3)
        if d != -1:
            if d == target_in:
                perf[0]+=1
                network.reward()
            else:
                perf[1]+=1
                network.punish()
        else:
            perf[2]+=1
    return perf/len(data)
```

Listing 9 | Helper function for reinforcement learning.

```python
net = DCSNN()
if use_cuda:
    net.cuda()
# First Layer
for epoch in range(epochs_1):
    for data,targets in MNIST_loader:
        train_unsupervised(net, data, 1)
# Second Layer
for epoch in range(epochs_2):
    for data,targets in MNIST_loader:
        train_unsupervised(net, data, 2)
# Third Layer
for epoch in range(epochs_3):
    for data,targets in MNIST_loader: # Training
        print(train_rl(net, data, targets))
    for data,targets in MNIST_test_loader: # Testing
        print(test(net, data, targets))
```

Listing 10 | Training and testing the network.

calls `network.eval` instead of `network.train` and it does not call plasticity member functions. Also, invoking `net.cuda`, transfers all the network's parameters to the GPU.

4.5. Source Code

Through this tutorial, we omitted many parts of the actual implementation such as adaptive learning rates, multiplication of learning rates, and saving/loading the best state of the network, for the sake of simplicity and clearance. The complete reimplementation is available on SpykeTorch's GitHub web page. We have also provided scripts for other works (Kheradpisheh et al., 2018; Mozafari et al., 2018) that achieve almost the same results as the main implementations. However, due to technical and computational differences between SpykeTorch and the original versions, tiny differences in performance are expected. A comparison between SpykeTorch and one of the previous implementations is provided in the next section.

5. COMPARISON

We performed a comparison between SpykeTorch and the dedicated C++/CUDA implementations of the network proposed by Mozafari et al. (2019) and measured the training and inference time. Both networks are trained for 686 epochs (2 for the first, 4 for the second, and 680 for the last trainable layer). In each training or inference epoch, the network sees all of the training or testing samples, respectively. Note that during the training

TABLE 1 | Comparison of C++/CUDA and SpykeTorch scripts simulating the network proposed by Mozafari et al. (2019).

Script	Total training	Inference epoch	Accuracy
C++/CUDA	174,120 s (= 2d 00h 22m 20s)	35 s	97.2%
SpykeTorch	121,600 s (= 1d 09h 46m 40s)	20 s	96.9%

Both scripts are executed on a same machine with Intel(R) Xeon(R) CPU E5-2697 (2.70 GHz), 256G Memory, NVIDIA TITAN Xp GPU, PyTorch 1.1.0, and Ubuntu 16.04.

of the last trainable layer, each training epoch is followed by an inference epoch.

As shown in **Table 1**, SpykeTorch script outperformed the original implementation in both training and inference times. The small performance gap is due to some technical differences in functions' implementations and performing a new round of parameter tuning fills this gap. We believe that SpykeTorch has the potentials of even more efficient computations. For example, adding batch processing to SpykeTorch would result in a large amount of speed-up due to the minimization of CPU-GPU interactions.

6. CONCLUSIONS

In recent years, SNNs have gained many interests in AI because of their ability to work in a spatio-temporal domain as well as energy efficiency. Unlike DCNNs, most of the current SNN simulators are not efficient enough to perform large-scale AI tasks. In this paper, we proposed SpykeTorch, an open-source high-speed simulation framework based on PyTorch. The proposed framework is optimized for convolutional SNNs with at most one spike per neuron and time-to-first-spike information coding scheme. SpykeTorch provides STDP and R-STDP learning rules but other rules can be added easily.

The compatibility and integrity of SpykeTorch with PyTorch have simplified its usage specially for the deep learning communities. This integration brings almost all of the PyTorch's features functionalities to SpykeTorch such as the ability of just-in-time optimization for running on CPUs, GPUs, or Multi-GPU platforms. We agree that SpykeTorch has hard limitations on type of SNNs, however, there is always a trade-off between computational efficiency and generalization. Apart from the increase of computational efficiency, this particular type of SNNs are bio-realistic, energy-efficient, and hardware-friendly that are getting more and more popular recently.

We provided a tutorial on how to build, train, and evaluate a DCSNN for digit recognition using SpykeTorch. However, the resources are not limited to this paper and additional scripts and documentations can be found on SpykeTorch's GitHub page. We reimplemented various works (Kheradpisheh et al., 2018; Mozafari et al., 2018; Mozafari et al., 2019) by SpykeTorch and reproduced their results with negligible difference.

Although the current version of SpykeTorch is functional and provides the main modules and utilities for DCSNNs (with at most one spike per neuron), we will not stop here and our plan is to extend and improve it gradually. For example, adding automation utilities would ease programming the network's forward pass resulting a more readable and cleaner code. Due to the variations of training strategies, designing a general automation platform is challenging. Another feature that improves SpykeTorch's speed is batch processing. Enabling batch mode might be easy for operations like convolution or pooling, however, implementing batch learning algorithms that can be run with none or a few CPU-GPU interactions is hard. Finally, implementing features to support models for other modalities such as the auditory system makes SpykeTorch a multi-modal SNN framework.

DATA AVAILABILITY

The dataset analyzed for this study can be found in this link http://yann.lecun.com/exdb/mnist/.

AUTHOR CONTRIBUTIONS

MM, MG, AN-D, and TM sketched the overall structure of SpykeTorch, revised, and finalized the manuscript. MM implemented the whole SpykeTorch package and wrote the first version of the manuscript.

FUNDING

This research was partially supported by the Iranian Cognitive Sciences and Technologies Council (Grant no. 5898) and by the French Agence Nationale de la Recherche (grant: Beating Roger Federer ANR-16-CE28-0017-01).

ACKNOWLEDGMENTS

The authors would like to thank Dr. Jean-Pierre Jaffrézou for proofreading this manuscript and NVIDIA GPU Grant Program for supporting computations by providing a high-tech GPU.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). "Tensorflow: a system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA), 265–283.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., et al. (2014). Nengo: a python tool for building large-scale functional brain models. *Front. Neuroinformatics* 7:48. doi: 10.3389/fninf.2013.00048

Bellec, G., Salaj, D., Subramoney, A., Legenstein, R., and Maass, W. (2018). "Long short-term memory and learning-to-learn in networks of spiking neurons," in *Advances in Neural Information Processing Systems* (Montréal, QC), 795–805.

Bi, G. Q., and Poo, M. M. (1998). Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *J. Neurosci.* 18, 10464–10472. doi: 10.1523/JNEUROSCI.18-24-10464.1998

Brzosko, Z., Zannone, S., Schultz, W., Clopath, C., and Paulsen, O. (2017). Sequential neuromodulation of hebbian plasticity offers mechanism for effective reward-based navigation. *eLife* 6, 1–18. doi: 10.7554/eLife.27756

Cao, Y., Chen, Y., and Khosla, D. (2015). Spiking deep convolutional neural networks for energy-efficient object recognition. *Int. J. Comput. Vis.* 113, 54–66. doi: 10.1007/s11263-014-0788-3

Carnevale, N. T., and Hines, M. L. (2006). *The NEURON Book*. Cambridge, UK: Cambridge University Press.

Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Diehl, P. U., and Cook, M. (2015). Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Front. Comput. Neurosci.* 9:99. doi: 10.3389/fncom.2015.00099

Diehl, P. U., Neil, D., Binas, J., Cook, M., Liu, S.-C., and Pfeiffer, M. (2015). "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *Neural Networks (IJCNN), 2015 International Joint Conference on* (Killarney: IEEE), 1–8.

Falez, P., Tirilly, P., Bilasco, I. M., Devienne, P., and Boulet, P. (2019). Multi-layered spiking neural network with target timestamp threshold adaptation and stdp. *arXiv: 1904.01908*.

Ferré, P., Mamalet, F., and Thorpe, S. J. (2018). Unsupervised feature learning with winner-takes-all based stdp. *Front. Comput. Neurosci.* 12:24. doi: 10.3389/fncom.2018.00024

Florian, R. V. (2007). Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Comput.* 19, 1468–1502. doi: 10.1162/neco.2007.19.6.1468

Frémaux, N., and Gerstner, W. (2016). Neuromodulated spike-timing-dependent plasticity, and theory of three-factor learning rules. *Front. Neural Circuits* 9:85. doi: 10.3389/fncir.2015.00085

Furber, S. (2016). Large-scale neuromorphic computing systems. *J. Neural Eng.* 13:051001. doi: 10.1088/1741-2560/13/5/051001

Gerstner, W., Kempter, R., van Hemmen, J. L., and Wagner, H. (1996). A neuronal learning rule for sub-millisecond temporal coding. *Nature* 383:76. doi: 10.1038/383076a0

Gewaltig, M.-O., and Diesmann, M. (2007). Nest (neural simulation tool). *Scholarpedia* 2:1430. doi: 10.4249/scholarpedia.1430

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., et al. (2018). Recent advances in convolutional neural networks. *Patt. Recogn.* 77, 354–377. doi: 10.1016/j.patcog.2017.10.013

Hazan, H., Saunders, D. J., Khan, H., Sanghavi, D. T., Siegelmann, H. T., and Kozma, R. (2018). Bindsnet: a machine learning-oriented spiking neural networks library in python. *Front. Neuroinformatics* 12:89. doi: 10.3389/fninf.2018.00089

Hussain, S., Liu, S.-C., and Basu, A. (2014). "Improved margin multi-class classification using dendritic neurons with morphological learning," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on* (Melbourne, VIC: IEEE), 2640–2643.

Kasabov, N. K. (2014). Neucube: a spiking neural network architecture for mapping, learning and understanding of spatio-temporal brain data. *Neural Netw.* 52, 62–76. doi: 10.1016/j.neunet.2014.01.006

Kheradpisheh, S. R., Ganjtabesh, M., and Masquelier, T. (2016). Bio-inspired unsupervised learning of visual features leads to robust invariant object recognition. *Neurocomputing* 205, 382–392. doi: 10.1016/j.neucom.2016.04.029

Kheradpisheh, S. R., Ganjtabesh, M., Thorpe, S. J., and Masquelier, T. (2018). Stdp-based spiking deep convolutional neural networks for object recognition. *Neural Netw.* 99, 56–67. doi: 10.1016/j.neunet.2017.12.005

Liu, T., Liu, Z., Lin, F., Jin, Y., Quan, G., and Wen, W. (2017). "Mt-spike: a multilayer time-based spiking neuromorphic architecture with temporal error backpropagation," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (Irvine, CA), 450–457.

Masquelier, T., and Thorpe, S. J. (2007). Unsupervised learning of visual features through spike timing dependent plasticity. *PLoS Comput. Biol.* 3:e31. doi: 10.1371/journal.pcbi.0030031

Mink, J. W., Blumenschine, R. J., and Adams, D. B. (1981). Ratio of central nervous system to body metabolism in vertebrates: its constancy and functional basis. *Am. J. Physiol. Regul. Integr. Compar. Physiol.* 241, R203–R212. doi: 10.1152/ajpregu.1981.241.3.R203

Mostafa, H. (2018). Supervised learning based on temporal coding in spiking neural networks. *IEEE Trans. Neural Netw. Learn. Syst.* 29, 3227–3235. doi: 10.1109/TNNLS.2017.2726060

Mozafari, M., Ganjtabesh, M., Nowzari-Dalini, A., Thorpe, S. J., and Masquelier, T. (2019). Bio-inspired digit recognition using reward-modulated spike-timing-dependent plasticity in deep convolutional networks. *Patt. Recogn.* 94, 87–95. doi: 10.1016/j.patcog.2019.05.015

Mozafari, M., Kheradpisheh, S. R., Masquelier, T., Nowzari-Dalini, A., and Ganjtabesh, M. (2018). First-spike-based visual categorization using reward-modulated stdp. *IEEE Trans. Neural Netw. Learn. Syst.* 29, 6178–6190. doi: 10.1109/TNNLS.2018.2826721

Neftci, E. O., Mostafa, H., and Zenke, F. (2019). Surrogate gradient learning in spiking neural networks. *arXiv: 1901.09948*.

O'Connor, P., Neil, D., Liu, S. C., Delbruck, T., and Pfeiffer, M. (2013). Real-time classification and sensor fusion with a spiking deep belief network. *Front. Neurosci.* 7:178. doi: 10.3389/fnins.2013.00178

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., et al. (2017). "Automatic differentiation in pytorch," in *NIPS-W* (Long Beach, CA).

Pfeiffer, M., and Pfeil, T. (2018). Deep learning with spiking neurons: opportunities and challenges. *Front. Neurosci.* 12:774. doi: 10.3389/fnins.2018.00774

Rawat, W., and Wang, Z. (2017). Deep convolutional neural networks for image classification: a comprehensive review. *Neural Comput.* 29, 2352–2449. doi: 10.1162/neco_a_00990

Shrestha, S. B., and Orchard, G. (2018). "Slayer: spike layer error reassignment in time," in *Advances in Neural Information Processing Systems* (Montréal, QC), 1419–1428.

Stimberg, M., Goodman, D. F., Benichoux, V., and Brette, R. (2014). Equation-oriented specification of neural models for simulations. *Front. Neuroinformatics* 8:6. doi: 10.3389/fninf.2014.00006

Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., and Maida, A. (2018). Deep learning in spiking neural networks. *Neural Netw.* 111, 47–63. doi: 10.1016/j.neunet.2018.12.002

Tavanaei, A., and Maida, A. S. (2016). Bio-inspired spiking convolutional neural network using layer-wise sparse coding and STDP learning. *arXiv: 1611.03000*.

Thiele, J. C., Bichler, O., and Dupret, A. (2018). Event-based, timescale invariant unsupervised online deep learning with stdp. *Front. Comput. Neurosci.* 12:46. doi: 10.3389/fncom.2018.00046

Thorpe, S., Fize, D., and Marlot, C. (1996). Speed of processing in the human visual system. *Nature* 381:520. doi: 10.1038/381520a0

Vaila, R., Chiasson, J., and Saxena, V. (2019). Deep convolutional spiking neural networks for image classification. *arXiv: 1903.12272*.

Vitay, J., Dinkelbach, H. Ü., and Hamker, F. H. (2015). Annarchy: a code generation approach to neural simulations on parallel hardware. *Front. Neuroinformatics* 9:19. doi: 10.3389/fninf.2015.00019

Wu, Y., Deng, L., Li, G., Zhu, J., and Shi, L. (2018). Spatio-temporal backpropagation for training high-performance spiking neural networks. *Front. Neurosci.* 12:331. doi: 10.3389/fnins.2018.00331

Yousefzadeh, A., Masquelier, T., Serrano-Gotarredona, T., and Linares-Barranco, B. (2017). "Hardware implementation of convolutional stdp for on-line visual feature learning," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on* (Baltimore, MD: IEEE), 1–4.

Yu, Q., Tang, H., Tan, K. C., and Li, H. (2013). Rapid feedforward computation by temporal encoding and learning with spiking neurons. *IEEE Trans. Neural Netw. Learn. Syst.* 24, 1539–1552. doi: 10.1109/TNNLS.2013.2245677

Zenke, F., and Ganguli, S. (2018). Superspike: supervised learning in multilayer spiking neural networks. *Neural Comput.* 30, 1514–1541. doi: 10.1162/neco_a_01086

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.