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], **Timothée Masquelier**[2]

[1]U of Tehran, Iran. [2]CERCO, CNRS - U Toulouse 3, France.

1. Introduction

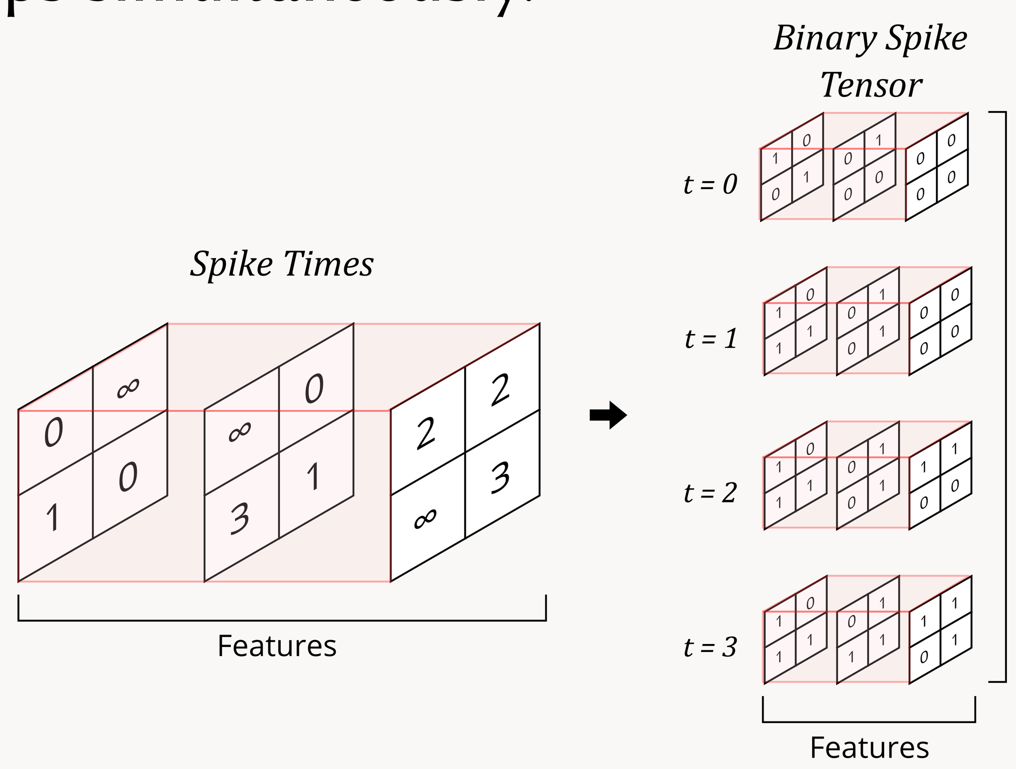
- Spiking neural networks (SNNs) are energy-efficient and hardware-friendly.
- SNNs with time-to-first-spike coding and spike-timing-dependent plasticity (STDP) learning rules have done well in various artificial intelligence (AI) tasks.
- **SpykeTorch** is a simulator for such SNNs that
  - makes use of **PyTorch** highly optimized **tensor operations**, on CPUs and (multi-) GPUs.
  - is fully **compatible** with and **integrated** to **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 $T_{f,r,c}$ 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/`

Method Paper:

`https://arxiv.org/abs/1903.02440`

4. Tutorial

Network Structure

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

```python
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)
```

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.

```python
   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
```

Plasticity

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

```python
   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"])
```

Input Transform

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

```python
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)
```

Execution

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

```python
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)
```

```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)
```

5. Results

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

References

1. Mozafari et al 2019, Pattern Recogn, (in press)
2. Kheradpisheh et al 2018, Neural Networks